The Impact of Solid State Drive on Search Engine Cache Management

Jianguo Wang Eric Lo Man Lung Yiu
Department of Computing
The Hong Kong Polytechnic University
{csjgwang, ericlo, csmlyiu}@comp.polyu.edu.hk

Jiancong Tong Gang Wang Xiaoguang Liu
Nankai-Baidu Joint Lab
Nankai University
{lingfenghx, wgzwpzy, liuxguang}@gmail.com

ABSTRACT

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 paper, we carry out a series of empirical experiments to study the impact of SSD on search engine cache management. The results give insights to practitioners and researchers on how to adapt the infrastructure and how to redesign the caching policies for SSD-based search engines.

Categories and Subject Descriptors

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

General Terms

Experimentation, Measurement, Performance

Keywords

Search Engine, Solid State Drive, Cache, Query Processing

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 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12].

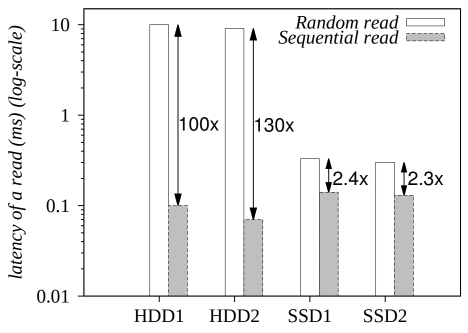


Figure 1: Read latency on two HDDs and two SSDs (for commercial reasons, we do not disclose their brands). Each read fetches 4KB. The OS buffer is by-passed.

The primary goal of caching is to improve *query latency* (reduce the query response time). To that end, search engines commonly dedicate portions of servers' memory to cache certain *query results* [1, 5, 7], *posting lists* [2, 3, 6], *documents* [4] and *snippets* [12] in order to avoid 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 [13, 14, 15, 16]. In addition, SSD is much more energy efficient than HDD (around 2.5% of HDD energy consumption [17, 18]). Now, SSD is getting cheaper and cheaper (e.g., it dropped from $40/GB in 2007 to $1/GB in 2012 [19]). Therefore, SSD has been employed in many industrial settings including MySpace [20], Facebook [21], and Microsoft Azure [22]. Baidu, the largest search engine in China, has used SSD to completely replace HDD as its main storage since 2011 [23].

The growing trend of using SSD to replace HDD has raised an interesting research question specific to our 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 SSD and two brands of HDD. 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, in traditional, 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, 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, has changed if SSD is used to replace HDD. Specifically, Figure 1 shows that the cost of a random read is only about two times 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 both the saving of the random read and the saving of the 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 [24, 25] and snippet generations [26, 27]), 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.

In this paper, we carry out a large-scale experimental study to evaluate the impact of SSD on the effectiveness of various caching policies, on all types of cache 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 policy can 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 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 real replays of a million of queries on an SSD-enabled search engine architecture and **our findings are reported based on actual query latency**.

To the best of our knowledge, this is the first study to evaluate the impact of SSD on search engine cache management. The experimental results here can bring the message "*it is time to rethink about your caching management*" to practitioners who have used or are planning to use SSD to replace HDD in their infrastructures. Furthermore, the results can give the research community insights into the redesign of caching policies for SSD-based search engine infrastructures.

The rest of this paper is organized as follows. Section 2 provides an overview of contemporary search engine architecture and the various types of cache involved. Section 3 presents the set of evaluation experiments carried out and the evaluation results. Section 4 discusses some related studies of this paper. Section 5 summarizes the main findings of this study.

2. BACKGROUND

In this section, we first give an overview of the architecture of the state-of-the-art Web search engines and the cache types involved (Section 2.1). Then we give a review of the various 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 [28, 29, 30] is shown in Figure 2. A search engine is typically composed by three sets of servers: *Web Servers* (WS), *Index Servers* (IS), and *Document Servers* (DS).

Web Servers The web servers are the front-end for interacting with end users and coordinating the whole process of query evaluation. Upon receiving a user's query q with n terms $t_1, t_2, ..., t_n$ [STEP ①], the web server that is in charge of q checks whether q is in its in-memory *Query Result Cache* (QRC) [1, 11, 31, 3, 5, 7, 9, 10]. 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 the same size and a query result consists of (i) the title, (ii) the URL, and (iii) the snippet [26, 27] (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 [31, 32, 12, 10]). If the result of q is not found in the QRC (i.e., a cache miss), the query is forwarded to an index server [STEP ②].

Index Servers The index servers are responsible for the computation of the top-k ranked result related to a query q. An index server works by: [STEP IS1] retrieving the corresponding *posting list* $PL_i = [d_1, d_2, ...]$ of each term t_i in q, [STEP IS2] intersecting all the retrieved posting lists $PL_1, PL_2, ..., PL_n$ to obtain the 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 $d_1, d_2, ..., d_k$ 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) [11, 3, 33] is employed to cache the posting lists of some terms. Upon receiving a query $q(t_1, t_2, ..., t_n)$ 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 $d_1, d_2, ..., d_k$ from the index server, the web server forwards the list and the query q to a document server for further processing [STEP ④]. The document server is responsible for the generating the final result. The final result is a Web page 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 — it is the 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) [12] and *Document Cache* (DC) [4]. 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 Web page 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

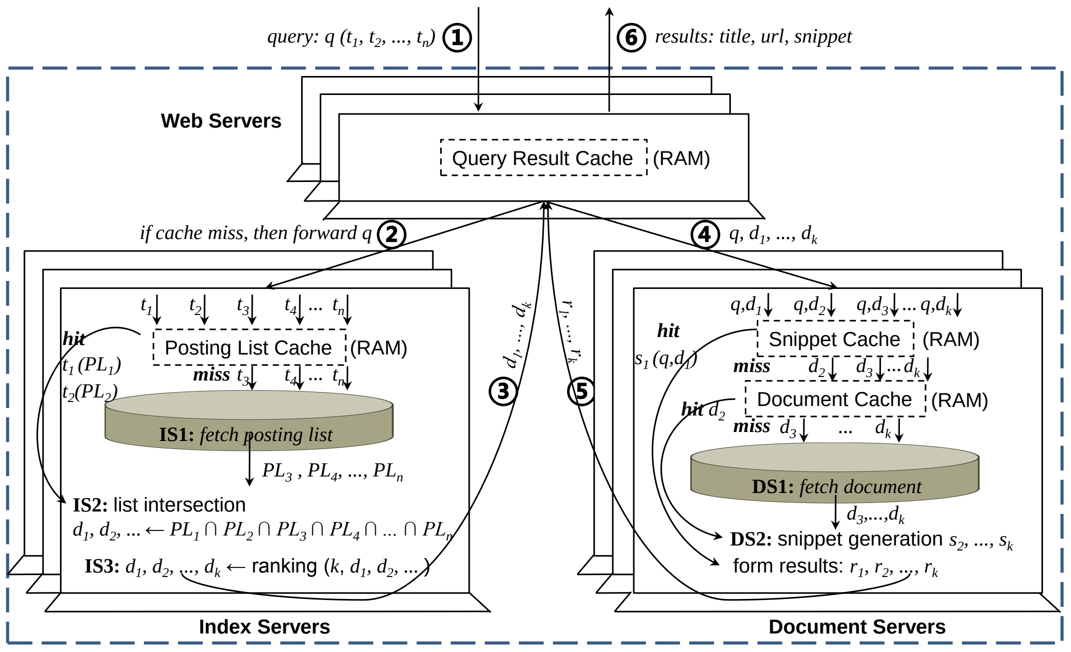


Figure 2: Web search engine architecture

may cache the result in the QRC and then pass the result back to the end user [STEP ⑥].

2.2 Caching Policy

Caching is a widely-used optimization technique to improve system performance [1, 34, 35, 36]. Web search engines use caches in different types of servers to reduce the query response time [11]. 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 [37, 38, 39]. Static caching is less common but does exist in search engines [1, 5, 3, 6, 8, 9, 10]. 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 items at run-time. The same cache content continuously serves the requests and its entire content is refreshed in a periodic manner [1]. Static caching can avoid the situations of having long-lasting popular items being evicted by the admission of many momentarily popular items as in dynamic caching.

2.2.1 Cache in Web Servers

Query Result Cache (QRC) is the cache used in the 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 Markatos [1] was the first to discuss about the use of dynamic query result cache (D-QRC) in search engines. By analyzing the query logs of Excite search engine, the authors observed a significant temporal locality in the queries. Therefore, they proposed the use of query result cache. Two classic replacement policies were used there: least-recently-used (LRU) and the least-frequently-used (LFU). In this paper, we refer them as D-QRC-LRU and D-QRC-LFU, respectively.

Recently, Gan and Suel [7] have developed a feature-based replacement policy for D-QRC. In this paper, we refer this policy as D-QRC-FB. That policy applies machine learning techniques offline to learn the values of a set of query features from the query log. The cache replacement is then tuned using those values.

Static Query Result Cache Markatos [1] also discussed about the potential use of static caching in search engines. 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 paper, we refer this policy as S-QRC-Freq.

Ozcan et al. [5] reported that some query results cached by S-QRC-Freq policy are not useful because there is a significant number of frequent queries quickly lose their popularities but still reside in the static cache before the next periodic cache refresh. Consequently, they proposed another admission policy that selects queries with high *frequency stability*. Queries with high frequency stability are those frequent queries and the logs show that they remain frequent over a time period. In this paper, we refer to this as S-QRC-FreqStab. Later on, Ismail et al. [8, 9, 10] developed a cost-aware replacement policy for S-QRC. In addition to traditional factors such as the popularity of terms, it also takes into account the latency of a query, as different queries may have different processing time. Nonetheless, its policy has not considered the cost of other expensive steps (e.g., snippet generation) in the whole query processing stack. In this paper, we refer that as S-QRC-CA.

Remark: According to [7], D-QRC-FB has the highest effectiveness among all D-QRC policies. Nonetheless, it requires a lot of empirical tuning on the various parameters used by the machine learning techniques. According to [8, 9, 10], S-QRC-CA is the most stable and effective policy in static query result cache. D-QRC and S-QRC can co-exist and share the main memory of a web server. In [31], the empirically suggested ratios for D-QRC to S-QRC are 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. [11] were the first to discuss the effectiveness of dynamic posting list caching in index servers. In that work, the simple LRU policy was used. In this paper, we refer this as D-PLC-LRU. In [3, 6], the authors also evaluated another common used policy, LFU, in dynamic PLC, in which we refer this as D-PLC-LFU in this paper. In addition, the authors observed that a popular term tends to have a longer posting list. Therefore, to balance the tradeoff 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 paper, we refer this policy as D-PLC-FreqSize.

Static Posting List Cache Static posting list caching was first studied in [2] and the admission policy was based on selecting posting lists with high access frequency. In this paper, we refer to this as S-PLC-Freq. In [3, 6], the static cache version of D-PLC-FreqSize was also discussed. We refer that as S-PLC-FreqSize here.

Remark: According to [3, 6], 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 is no corresponding static caching technique yet.

Document Cache In [4], the authors observed that the time of reading a document from disk dominates the snippet generation process. So, they proposed to cache some documents in the document servers's memory so as to reduce the snippet generation time. In that work, the simple LRU replacement policy was used and here we refer it as DC-LRU. According to [4], document caching is able to significantly reduce the time for generating snippets.

Snippet Cache In [12], the authors 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 to cache some generated snippets in the document server's main memory. In this paper, we refer this as SC-LRU because it uses the LRU policy as replacement policy.

3. THE IMPACT OF SSD

In this study, we use the typical search engine architecture (Figure 2) to evaluate the impact of SSD on the cache management of the index servers (Section 3.1), document servers (Section 3.2), and the web servers (Section 3.3). We focus on a pure SSD-based architecture like Baidu [23]. Many studies predict that SSD will soon completely replace HDD in all layers [40, 41, 42]. 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 [43].

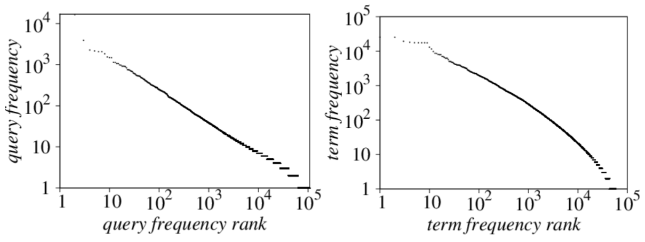
Experimental Setting We deploy a Lucene[1] based search engine implementation on a sandbox infrastructure consisting of one index server, one document server, and one web server. All servers are Intel commodity machines (2.5GHz CPU, 8GB RAM), with Windows 7 installed. We have carried out our experiments 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.

We use a real collection of 5.3 million Web pages[2] crawled in the middle of 2008, by Sogou (`sogou.com`), a famous commercial search engine in China. The entire data set takes over 5TB. To accommodate our experimental setup, we draw a random sample of 100GB data. It is a reasonable setting because large-scale Web search engines shard their indexes and data across clusters of machines [23]. As a reference, the sampled data sets in some recent works are 15GB [6], 37GB in [10], and 2.78 million Web pages in [3]. Table 1 shows the characteristics of our sampled data.

Number of documents	11,970,265
Average document size	8KB
Total data size	100GB
Inverted index size	10GB

Table 1: Statistics of Web data

As mentioned in the introduction, the cache hit ratio alone is inadequate to evaluate the effectiveness of caching techniques. So, the experiments are done by a replay of real queries found in Sogou search engine in June 2008[3], and the average end-to-end query latency is reported in tandem with cache hit ratio. The log contains 51.5 million queries. To ensure the 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 [10]). The query frequency and term frequency of our sampled query set follow power-law distribution with skew-factor $\alpha = 0.85$ and 1.48, respectively (see Figure 3). As a reference, the query frequencies in some related work like [11], [7], and [6] follow power-law distribution with α equals 0.59, 0.82, and 0.83, respectively; and the term frequencies in some recent work like [6] follow power-law distribution with $\alpha = 1.06$.



(a) Query frequency distribution (b) Term frequency distribution

Figure 3: Frequency distribution of our query set

In addition, our sampled query set also exhibits the property that the temporal locality of terms is higher than that of queries [11, 3, 6] by observing that the skew-factor α of the query frequency is lower than that of the term frequency. Table 2 shows the characteristics of our query set.

We divide the real query log into two parts: 50% of queries are used for warming the cache and the other 50% are for the replay, following the usual settings [1, 8, 10, 32, 9].

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

[2]`http://www.sogou.com/labs/dl/t-e.html`

[3]`http://www.sogou.com/labs/dl/q-e.html`

Number of queries	1,000,000
Number of distinct queries	200,444
Number of terms	1,940,671
Number of distinct terms	82,503
Total size of the posting lists of the distinct terms	3.89GB
Power-law skew-factor α for query frequency	0.85
Power-law skew-factor α for term frequency	1.48

Table 2: Statistics of query log data

In the experiments, we follow some standard settings found in recent work. Specifically, we follow [31, 32, 12, 10] to retrieve the top-10 most relevant documents. We follow [11] 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 [44] in the experiments. The page (block) size in the system is 4KB [45, 46, 47]. When evaluating the caching policy on one type of cache, we disable all other caches in the sandbox. For example, when evaluating the query result cache on the web server, the posting list cache, snippet cache, and document cache are all disabled.

3.1 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 our 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.

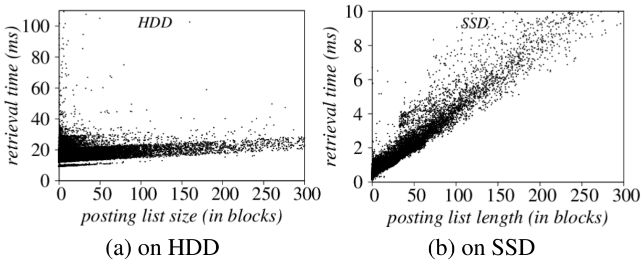


Figure 4: Read access latency of posting lists of varying lengths

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 [3, 6], 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.

To verify our claim, we carried out experiments to re-evaluate the static and dynamic posting list caching policies on our SSD 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 architecture is beyond the scope of this paper.

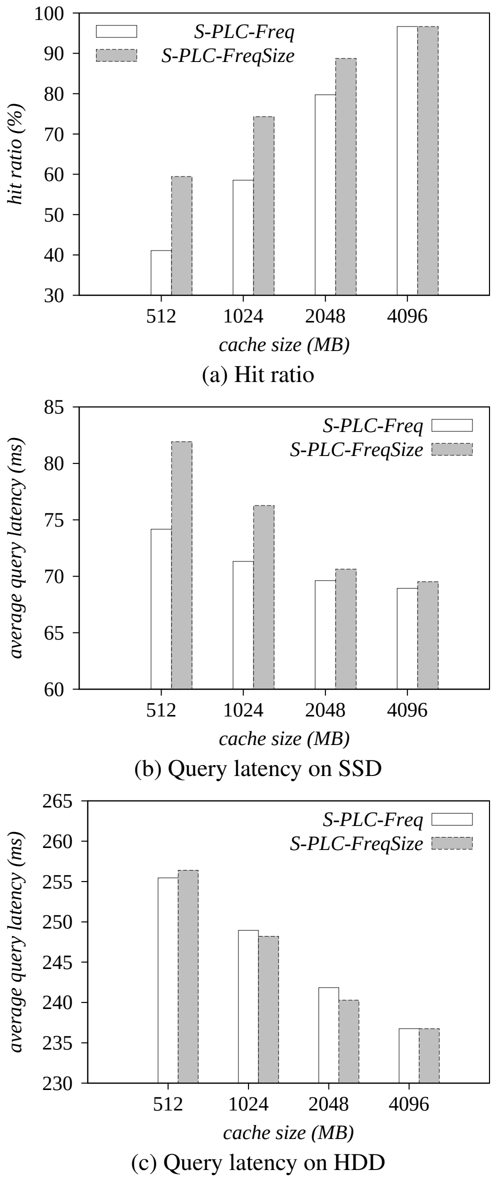


Figure 5: [Index Server] Effectiveness of static posting list caching policies

Reevaluation of static posting list caching policy on SSD We begin with presenting the evaluation results of the two existing static query result 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 result in [3, 6], Figure 5(a) shows 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 98% 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 (3.89GB; Table 2) in the cache warming phase. The 2% cache miss is attributed to the difference between the terms found in the training queries and the terms found in the replay queries.

Although having a higher cache hit ratio, Figure 5(b) shows that the average query latency of S-PLC-FreqSize is actually longer than 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 the above, the experimental results above are real examples that illustrate cache hit ratio is not a reliable metric for caching management in SSD-based search engine architectures.

Figure 5(c) shows the average query latency on HDD. We see a surprising result: even on HDD cache hit ratio is not always reliable! Specifically, we see 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 size of the posting lists participated in all cache hits (i.e., whenever there is a hit in the PLC, we record its size and report the average).

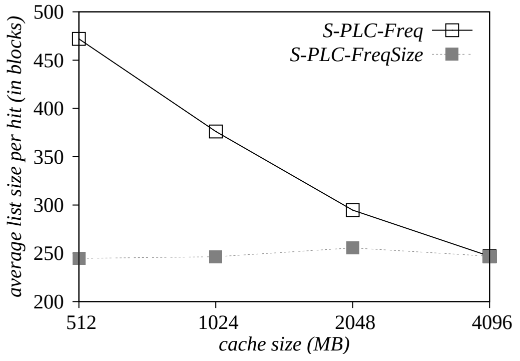
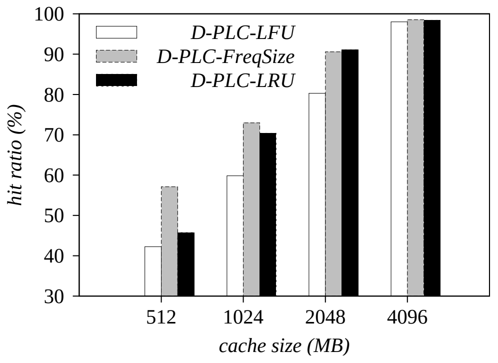


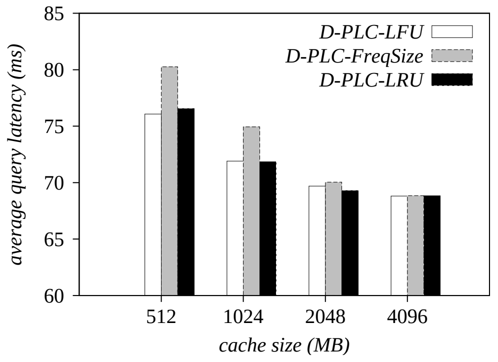
Figure 6: Average list size (in blocks) of all hits in static posting list cache

We see that when the cache memory is small (512MB), S-PLC-Freq keeps the longest (most frequent) posting lists in the cache. In contrast, S-PLC-FreqSize consistently keeps short lists in the cache. At 512MB cache memory, a cache hit under S-PLC-Freq policy can save $470 - 250 = 220$ more sequential reads than S-PLC-FreqSize. Even though sequential reads are cheap on HDD, a total of 220 sequential reads are actually as expensive as two random seeks (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 two extra random seeks saving 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.

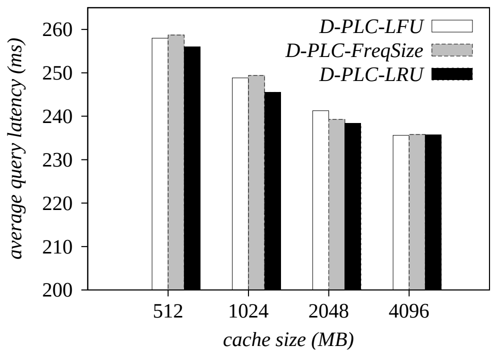
Reevaluation of dynamic posting list caching policy on SSD We next present the evaluation results of the three existing dynamic query result caching policies, (1) D-PLC-LRU, (2) D-PLC-LFU, and (3) D-PLC-FreqSize, mentioned in Section 2.2.2. Figure 7 shows the cache hit ratio and the average query latency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



(a) Hit ratio



(b) Query latency on SSD



(c) Query latency on HDD

Figure 7: [Index Server] Effectiveness of dynamic posting list cache policies

First, we also see that while D-PLC-FreqSize has a better cache hit ratio than D-PLC-LFU (Figure 7(a)), its query latency is actually longer than D-PLC-LFU (Figure 7(b)) in SSD-based architecture. This further supports that the claim of favoring terms with high frequency over length ratio no longer sustains in SSD-based search engine architecture. Also, this gives yet another example of the fact that cache hit ratio is not a reliable measure in SSD caching management.

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 architecture. As is shown in [3, 6], the posting list length generally increases with the term frequency, therefore D-PLC-LFU, which favors terms with high frequency, also favors terms with long posting lists. 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.

Figure 7(c) shows the average query latency on HDD. First, we once again see that cache hit ratio is not reliable even on HDD-based architecture. 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 an HDD-based architecture. That is due to the same reason that we explained in static caching — Figure 8 shows 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 Figure 8 shows 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.

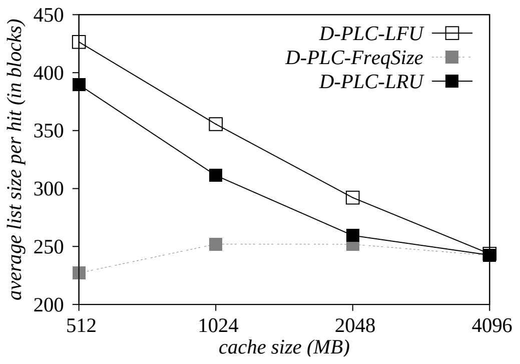


Figure 8: Average list size (in blocks) of all hits in dynamic posting list cache

3.2 The Impact of SSD on Document Servers

We next evaluate the impact of SSD on the cache management of 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 $\{d_1, d_2, \ldots, d_k\}$ of the k most relevant documents of q 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 d_i is in found the document cache, the retrieval of d_i from the disk is skipped. In our Web data (Table 1), a document is about 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 architecture 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 the k documents from the HDD (see Figure 9(a)). 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

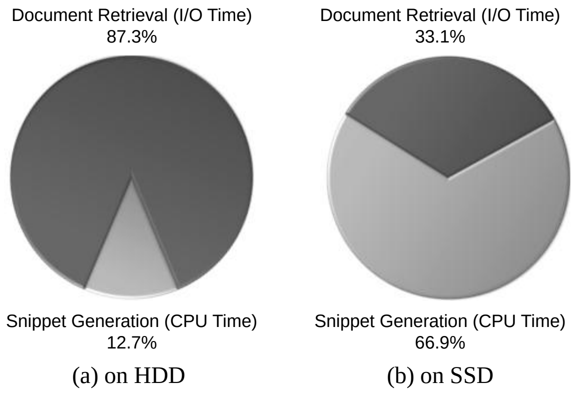
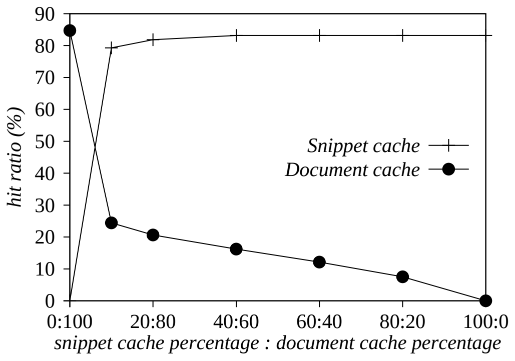


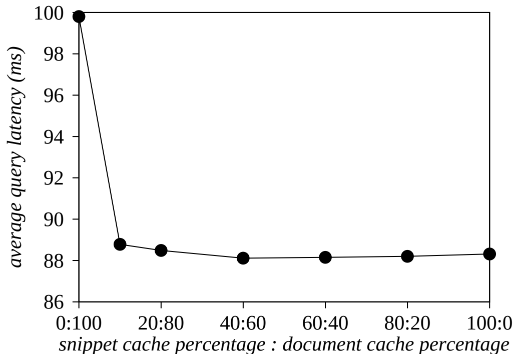
Figure 9: Document retrieval time vs. snippet generation in a document server

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 [26, 27] according to a specific ranking function [27] is indeed CPU-intensive. Figure 9(b) shows that the CPU time spent on snippet generation becomes two times 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 [26], we believe the importance of document cache in SSD-based search engine architectures is largely diminished. In contrast, we believe the snippet cache is far more 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; and (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 8KB in our Web data). That implies a double-benefit of the use of snippet cache over document cache: the same amount of cache memory in a web server can cache many more snippets than documents, and the benefit of a snippet cache hit is much more significant than the benefit of a document cache hit.

To verify our claims, we carried out experiments to vary the memory ratio between the snippet cache and the document cache in our SSD-based sandbox infrastructure. The caching policies in document cache and snippet cache are DC-LRU [4] and SC-LRU [12], respectively. Figure 10 shows the query latency and the hit ratios of the snippet cache and document cache when varying the memory allocation ratio between snippet cache and document cache of 2GB cache memory. Figure 10(a) shows the cache hit ratio of snippet cache increases when we allocate more memory to the snippet cache, but when we continue to allocate more than 2GB $\times$ 40% = 800MB of memory to the snippet cache, the snippet cache hit ratio stays flat because the memory footprint of a snippet is so small that all snippets generated in the warming phase can be resided in the cache. When we go on allocating more memory to the snippet cache even after all snippets could be cached in the memory, it only gives less memory to the document cache. Therefore, the document cache hit ratio in Figure 10(a) continues to drop even when the snippet cache hit ratio has plateaued out. Figure 10(b) shows that when we allocate more memory to the snippet cache (i.e., the document cache has less memory allocated), the query latency drops monotonically before all snippets are in the cache. Furthermore, after all snippets are in the cache and we allocate less memory to the document cache, the query latency almost stays flat. The results indicate that snippet caching is far more ef-



(a) Hit ratio



(b) Query latency

Figure 10: [Document Server] Effectiveness of snippet cache and document cache on SSD

fective than document caching. The observations above persist for cache memory of size 512MB, 1GB, and 4GB (figures are omitted for space reasons).

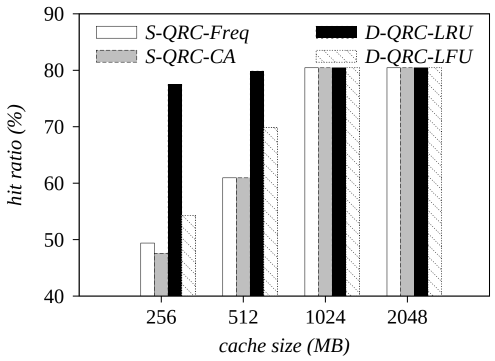
3.3 The Impact of SSD on Web Servers

We believe the use of SSD has no impact on the cache management in the web servers because the storage media does not play a major role in web servers (see Figure 2). Therefore, we believe that the use of SSD has no impact on the effectiveness of the caching policies either. Figure 11 shows the cache hit ratio and query latency of four selected query result caching policies: (1) D-QRC-LRU, (2) D-QRC-LFU, and (3) S-QRC-Freq, (4) S-QRC-CA. We see that the tradition holds here: when one policy has a higher cache hit ratio than the others, its query latency is also shorter than the others.[4]

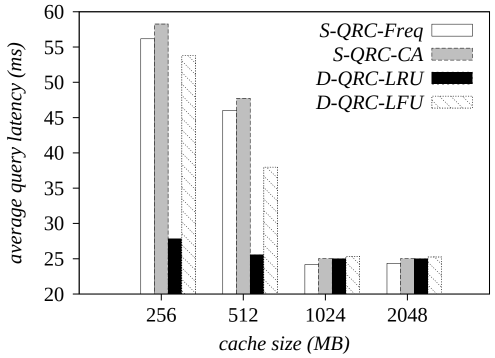
4. RELATED STUDIES

SSD is expected to gradually replace HDD as the primary permanent storage media in both consumer computing and enterprise computing [41, 42, 14, 47, 40] because of its 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.

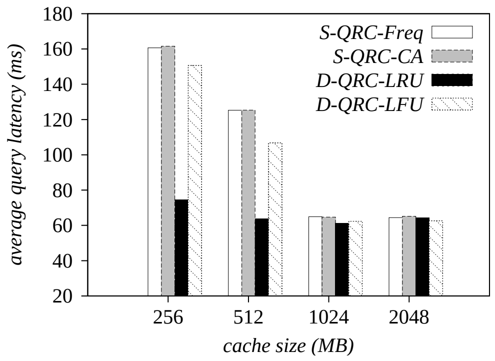
[4]In Figure 11(a), the cache hit ratio does not further increase with more than 1024MB of cache memory simply because all query results generated in the warming phase can reside in the cache. Although we claim that SSD has no impact on the cache management in the web servers, Figure 11(b) and Figure 11(c) still show differences in query latency — that is merely due to the other steps (e.g., the posting list retrieval step) in the index server and document server are also sped up by the use of SSD in this experiment.



(a) Hit ratio



(b) Query latency (using SSD in Web Server)



(c) Query latency (using HDD in Web Server)

Figure 11: [Web Server] Effectiveness of query result caching policies on SSD and HDD

The discussion of the impacts of SSD on computer systems had started as early as in 1995 [48], in which an SSD-aware file system was proposed. Later on, more SSD-aware file systems have been designed, e.g., JFFS [49], YAFFS [50]. After that, the discussion has extended to other components in computer systems, e.g., virtual machine [51, 52], buffer manager [53, 54], I/O scheduler [55], and RAID [56, 57].

The discussion of the impacts of SSD on database systems, our sister field, had started as early as in 2007 [58]. Since then, SSD has become an active topic in database research. The discussions cover the impacts of SSD on database architecture [58, 14, 59], query processing [47, 60], index structures [61, 62, 63] and algorithms [45], data placement and migration [64, 65], transaction management [14, 66, 67] and buffer management [68, 69]. Most concern about the asymmetric performance between slow random write and fast sequential write on SSD [70, 61, 62, 63, 58, 14, 59, 45, 64, 67], some concern about the asymmetric fast read and slow write [68, 14, 69, 45, 66], and some concern about the comparable performance between fast random reads and fast sequential reads [47, 60, 45] like what we do in this paper. The recent trend is to exploit the energy efficiency of SSD [71, 72] and its parallelism [73, 74] to fully leverage the power of SSD in large-scale data processing systems.

Baidu first announced their SSD-based search engine infrastructure in 2010 [23], but they did not investigate the impact of SSD on their cache management. The issue of allocating indexes on SSD was recently studied in 2011 [75]. Last year, the issue of maintaining inverted index resided on SSD was also discussed [76]. In [43], the use of SSD as a layer between the RAM and HDD was discussed. Our work focuses on the possibly the next generation of search engine architecture in which SSD completely replaces HDD.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we 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. This study contributes the following messages to our community:

1. Traditional cache hit ratio is no longer a reliable measure of the effectiveness of caching policies and we shall use query latency as the major evaluation metric in SSD-based search engine cache management.

2. The previous known best caching policy in index servers, S-PLC-FreqSize [3, 6], 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.

3. 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 [4], 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.

4. 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.

As future work, we will next focus on the new bottleneck of query processing in SSD-based web search engine architecture and consider the potential of other cache types (e.g., intersection cache [77]) in the study.

6. ACKNOWLEDGMENTS

This work is partially supported by the Research Grants Council of Hong Kong (GRF PolyU 525009, 521012, 5302/12E), NSFC of China (60903028, 61070014), Key Projects in the Tianjin Science & Technology Pillar Program (11ZCKFGX01100).

7. REFERENCES

[1] Evangelos P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[2] Ricardo Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, pages 56–65, 2003.

[3] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190, 2007.

[4] Andrew Turpin, Yohannes Tsegay, David Hawking, and Hugh E. Williams. Fast generation of result snippets in web search. In *SIGIR*, pages 127–134, 2007.

[5] Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. Static query result caching revisited. In *WWW*, pages 1169–1170, 2008.

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

[7] Qingqing Gan and Torsten Suel. Improved techniques for result caching in web search engines. In *WWW*, pages 431–440, 2009.

[8] Ismail Sengor Altingovde, Rifat Ozcan, and Özgür Ulusoy. A cost-aware strategy for query result caching in web search engines. In *ECIR*, pages 628–636, 2009.

[9] Rifat Ozcan, I. Sengor Altingovde, B. Barla Cambazoglu, Flavio P. Junqueira, and Özgür Ulusoy. A five-level static cache architecture for web search engines. *IPM*, 48(5):828–840, 2011.

[10] Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web*, 5:1–25, 2011.

[11] Paricia Correia Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *SIGIR*, pages 51–58, 2001.

[12] Diego Ceccarelli, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Caching query-biased snippets for efficient retrieval. In *EDBT*, pages 93–104, 2011.

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

[14] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *SIGMOD*, pages 1075–1086, 2008.

[15] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS*, pages 181–192, 2009.

[16] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys*, pages 145–158, 2009.

[17] Euiseong Seo, Seon Yeong Park, and Bhuvan Urgaonkar. Empirical analysis on energy efficiency of flash-based SSDs. In *HotPower*, 2008.

[18] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR*, pages 21–31, 2011.

[19] Flexstar SSD test market analysis. `http://info.flexstar.com/Portals/161365/docs/SSD_Testing_Market_Analysis.pdf`.

[20] MySpace uses fusion powered I/O to drive greener and better data centers. `http://www.fusionio.com/case-studies/myspace-case-study.pdf`.

[21] Releasing flashcache. `http://www.facebook.com/note.php?note_id=388112370932`, 2010.

[22] Microsoft azure to use OCZ SSDs. `http://www.storagelook.com/microsoft-azure-ocz-ssds/`, 2012.

[23] Ruyue Ma. Baidu distributed database. In *SACC*, 2010.

[24] Howard Turtle and James Flood. Query evaluation: strategies and optimizations. *IPM*, 31(6):831–850, 1995.

[25] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[26] Andrew Turpin, Yohannes Tsegay, David Hawking, and Hugh E. Williams. Fast generation of result snippets in web search. In *SIGIR*, pages 127–134, 2007.

[27] Anastasios Tombros and Mark Sanderson. Advantages of query biased summaries in information retrieval. In *SIGIR*, pages 2–10, 1998.

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

[29] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, 2009.

[30] Ricardo Baeza-yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. Challenges on distributed web retrieval. In *ICDE*, pages 6–20, 2007.

[31] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *TOIS*, 24:51–78, 2006.

[32] I. Sengor Altingovde, Rifat Ozcan, B. Barla Cambazoglu, and Özgür Ulusoy. Second chance: a hybrid approach for dynamic result caching in search engines. In *ECIR*, pages 510–516, 2011.

[33] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[34] Enric Herrero, José González, and Ramon Canal. Distributed cooperative caching. In *PACT*, pages 134–143, 2008.

[35] Memcached – a distributed memory object caching system. `http://memcached.org/`.

[36] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. *TODS*, 9(4):503–525, 1984.

[37] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[38] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, pages 31–42, 2002.

[39] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *CUSR*, 35(4):374–398, 2003.

[40] Storage market outlook to 2015. `http://www.storagesearch.com/5year-2009.html`.

[41] Jim Gray. 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.

[42] Ari Geir Hauksson and Sverrir Íćsmundsson. Data storage technologies. `http://olafurandri.com/nyti/papers2007/DST.pdf`, 2007.

[43] Ruixuan Li, Chengzhou Li, Weijun Xiao, Hai Jin, Heng He, Xiwu Gu, Kunmei Wen, and Zhiyong Xu. An efficient SSD-based hybrid storage architecture for large-scale search engines. In *ICPP*, pages 450–459, 2012.

[44] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR*, pages 222–229, 2002.

[45] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *CSUR*, 37:138–163, 2005.

[46] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High throughput persistent key-value store. In *VLDB*, pages 1414–1425, 2010.

[47] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *SIGMOD*, pages 59–72, 2009.

[48] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *ATC*, pages 155–164, 1995.

[49] Red Hat Corporation. JFFS2: The journalling flash file system. `http://sources.redhat.com/jffs2/jffs2.pdf`.

[50] Charles Manning. YAFFS: Yet another flash file system. `http://www.aleph1.co.uk/yaffs`.

[51] Han-Lin Li, Chia-Lin Yang, and Hung-Wei Tseng. Energy-aware flash memory management in virtual memory system. *TVLSI*, 16(8):952–964, 2008.

[52] Mohit Saxena and Michael M. Swift. FlashVM: revisiting the virtual memory hierarchy. In *HotOS*, 2009.

[53] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *CASES*, pages 234–241, 2006.

[54] Hyojun Kim and Seongjun Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST*, pages 1–14, 2008.

[55] Stan Park and Kai Shen. FIOS: a fair, efficient flash I/O scheduler. In *FAST*, 2012.

[56] Cagdas Dirik and Bruce Jacob. The performance of pc solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *ISCA*, pages 279–289, 2009.

[57] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD reliability. *TOS*, 6(2):1–22, 2010.

[58] Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD*, pages 55–66, 2007.

[59] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD*, pages 863–870, 2009.

[60] Mehul A. Shah, Stavros Harizopoulos, Janet L. Wiener, and Goetz Graefe. Fast scans and joins using flash drives. In *DaMoN*, pages 17–24, 2008.

[61] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: an optimized index structure for flash devices. *PVLDB*, 2(1):361–372, 2009.

[62] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *PVLDB*, 3(1-2):1195–1206, September 2010.

[63] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient R-tree implementation over flash-memory storage systems. In *GIS*, pages 17–24, 2003.

[64] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *PVLDB*, 1:514–525, 2008.

[65] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. An object placement advisor for DB2 using solid state storage. *PVLDB*, 2(2):1318–1329, August 2009.

[66] Sai Tung On, Jianliang Xu, Byron Choi, Haibo Hu, and Bingsheng He. Flag commit: Supporting efficient transaction recovery in flash-based dbmss. *TKDE*, 24(9):1624–1639, 2012.

[67] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder – a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[68] Sai Tung On, Yinan Li, Bingsheng He, Ming Wu, Qiong Luo, and Jianliang Xu. FD-buffer: a buffer manager for databases on flash disks. In *CIKM*, pages 1297–1300, 2010.

[69] Yanfei Lv, Bin Cui, Bingsheng He, and Xuexuan Chen. Operation-aware buffer management in flash-based systems. In *SIGMOD*, pages 13–24, 2011.

[70] Chin hsien Wu, Li pin Chang, and Tei wei Kuo. An efficient B-tree layer for flash-memory storage systems. In *RTCSA*, pages 17–24, 2003.

[71] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. Analyzing the energy efficiency of a database server. In *SIGMOD*, pages 231–242, 2010.

[72] Theo Härder, Volker Hudlet, Yi Ou, and Daniel Schall. Energy efficiency is not enough, energy proportionality is needed! In *DASFAA*, pages 226–239, 2011.

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

[74] Risi Thonangi, Shivnath Babu, and Jun Yang. A practical concurrent index for solid-state drives. In *CIKM*, pages 1332–1341, 2012.

[75] Bojun Huang and Zenglin Xia. Allocating inverted index into flash memory for search engines. In *WWW*, pages 61–62, 2011.

[76] Ruixuan Li, Xuefan Chen, Chengzhou Li, Xiwu Gu, and Kunmei Wen. Efficient online index maintenance for SSD-based information retrieval systems. In *HPCC*, pages 262–269, 2012.

[77] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, pages 257–266, 2005.