Information Systems 49 (2015) 25–39

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Inverted index maintenance strategy for flashSSDs: Revitalization of in-place index update strategy

Wonmook Jung[a,1], Hongchan Roh[a,b,1], Mincheol Shin[a], Sanghyun Park[a,*]

[a] *Department of Computer Science, Yonsei University, South Korea*
[b] *ICT R&D Division, SK Telecom, South Korea*

ARTICLE INFO

Article history:
Received 8 May 2013
Received in revised form
10 July 2014
Accepted 7 November 2014
Available online 15 November 2014

Keywords:
FlashSSDs
Inverted index
Index maintenance
Information retrieval
Search engines

ABSTRACT

An inverted index is a core data structure of Information Retrieval systems, especially in search engines. Since the search environments have become more dynamic, many on-line index maintenance strategies have been proposed. Previous strategies were designed for HDDs. Consequently, in order to avoid expensive random access cost, Merge-based strategies have been preferred to In-place index update strategies on HDDs. However, flashSSDs have become solid alternatives to HDDs. FlashSSDs currently are adopted in a wide range of areas due to their superior features such as the short access latency, energy efficiency, and high bandwidth. In this article, we first reexamined potentials of In-place index update strategies on flashSSDs. Thanks to the insignificant access latency of flashSSDs, we discovered that In-place index update strategies outperform Merge-based strategies, since In-place index update strategies generate much less amount of I/O than Merge-based strategies despite inducing frequent random accesses. Based on this discovery, we suggest a new inverted index maintenance strategy based on an In-place index update strategy for flashSSDs, called Multipath Flash In-place Strategy (MFIS). To enhance the index maintenance performance, MFIS stores the posting list of each term non-contiguously and exploits the internal parallelism of flashSSDs. Thus, MFIS not only induces the minimum amount of I/O but also utilizes the maximum bandwidth of flashSSDs. Furthermore, MFIS is designed to show high query processing performance by utilizing the internal parallelism of flashSSDs even though the posting list of each term is stored non-contiguously. In our experiments, the index maintenance performance of MFIS was considerably better than other previous maintenance strategies. The index maintenance performance was up to 14.93, 4.04, 5.12, and 2.33 times higher than Merge-based strategies such as Immediate Merge, Geometric Partitioning, Hybrid, and SSD-aware Hybrid, respectively. The query processing performance of MFIS was up to 1.62 times higher than non-contiguous In-place. In addition, MFIS showed almost the best query processing performance as Merge-based strategies did. In conclusion, MFIS is the best on-line inverted index maintenance strategy on flashSSDs in terms of both index maintenance and query processing performance.

1. Introduction

In Information Retrieval (IR) systems, an inverted index is used as a core data structure. It consists of a list of terms (vocabulary) and a list of postings for each term (posting list). Each posting contains term's information such as document ID and term frequency in the document. Many studies have focused on maintaining the inverted index. It is

* Corresponding author. Tel.: +82 2 2123 5714; fax: +82 2 365 2579.
E-mail addresses: jngwnmk@cs.yonsei.ac.kr (W. Jung), hongchan.roh@sk.com (H. Roh), smanioso@cs.yonsei.ac.kr (M. Shin), sanghyun@cs.yonsei.ac.kr (S. Park).

[1] These authors have contributed equally to this paper.

http://dx.doi.org/10.1016/j.is.2014.11.004
0306-4379/© 2014 Elsevier Ltd. All rights reserved.

essentially classified into off-line indexing strategies and on-line indexing strategies. Since search environments have become more dynamic, on-line indexing strategies have been brought to more attention. Consequently, many on-line indexing algorithms have been proposed in the recent past. Classically, Re-build [29], Immediate Merge [27–29], and In-place index update [28,45] strategies have been introduced. More specifically, In-place index update strategies can be generally classified into the contiguous In-place and non-contiguous In-place index update strategies based on how they store the posting list of each term. New Merge-based strategies such as Logarithmic Merge [6], Geometric Partitioning [26], and Horizontal Partitioning [17] have emerged, which control periods of the merge operation. These enhance the index maintenance performance by using trade-offs with query processing performance. Additionally, Hybrid strategies [7–9] were also proposed that blend Merge and In-place index update strategies. The above strategies focus on monotonically growing text collections that only allow document insertions. In this article, we also focus on maintaining the inverted index for document insertions, which is a quite common restriction as shown in previous studies [9,26,29,42,43,45].

The most important thing that we need to know about the above algorithms is that they have been designed only considering HDDs as storage for the inverted index. For this reason, these algorithms have tried to gather posting lists of each term in order to minimize the number of random accesses to HDDs because of the long access latency of HDDs. Merge-based strategies are considered the most efficient algorithms for on-line indexing even though these strategies have to read and write posting lists of all terms regardless of whether they are required to be updated or not. In addition, multiple inverted indexes need to be maintained. That is, Merge-based strategies that prefer many sequential I/Os to few random I/Os are suitable for HDDs.

However, In-place index update strategies only update the posting lists of terms that exist in the in-memory onto the on-disk inverted index. Therefore, In-place index update strategies require considerably less amount of I/O than Merge-based strategies require. Nevertheless, it has been considered that In-place index update strategies are not suitable for massive update of posting lists because of numerous random accesses to HDDs. Therefore, very few attempts have been made to develop an inverted index maintenance strategy only based on In-place index update strategies in recent years although Hybrid strategies combine In-place index update strategies with Merge-based strategies.

In the last few years, flashSSDs have been adopted as main storage devices in a wide range of areas from laptop computers to enterprise servers, thanks to their short access latency, high bandwidth, and low power consumption. Therefore, inverted index maintenance strategy also needs to be redesigned according to the basic characteristics of flashSSDs. To fully utilize flashSSDs, there are key principles that have to be considered to design inverted index maintenance strategy on flashSSDs.

First, inverted index maintenance strategy needs to be designed to minimize the total amount of I/O without the consideration of minimizing random access counts. The most important feature of flashSSDs is their insignificant access latency which is considerably shorter than that of HDDs as presented in Table 1. This is because flashSSDs have no mechanical part to move. Thus, the access latency can be considered insignificant in I/O processing.

Table 1
FlashSSD, HDD specification [36,18].

	FlashSSD (P300)	HDD (Deskstar 7K1000.D)
Average access latency	0.154 ms (read) 0.424 ms (write)	12.37 ms (read) 13.37 ms (write)
Sustained transfer rate	360 MB/s (read) 255 MB/s (write)	189 MB/s

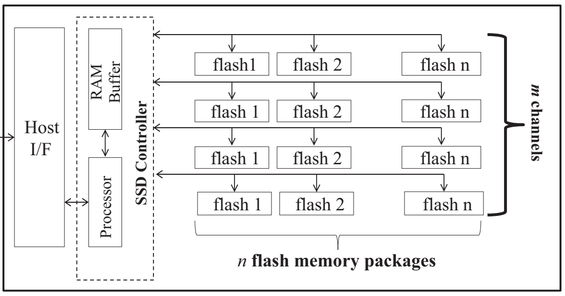

Fig. 1. An illustration of internal architecture of flashSSDs.

Second, inverted index maintenance strategy does not need to avoid random writes to flashSSDs thanks to the advanced FTL mapping technologies. FlashSSDs include FTL (Flash Translation Layer) that maps LBAs (logical block addresses) of the host to physical addresses of flash memory[2]. State of the art FTL mapping technologies processes random writes very efficiently with little write amplification inside flashSSDs while early flashSSDs could not handle random-write requests efficiently, thereby amplifying the amount of writes actually written into flash memory. This further emphasize that the inverted index design does not need to be constrained by the I/O pattern.

Moreover, the internal I/O parallelism of flashSSDs has to be exploited for better design of inverted indexes on flashSSDs. flashSSDs have a new unique feature, the internal parallelism [13], which has not been observed in HDDs. A flashSSD embeds multiple flash memory packages each of which is connected to each other through multiple channels as depicted in Fig. 1. Due to this internal parallelism, multiple I/O requests can be simultaneously processed spreading over multiple flash memory packages inside the flashSSD. The maximum bandwidth of the flashSSD is nearly equal to the total sum of the bandwidths of the embedded flash memory packages. Therefore, unless a number of I/O requests are delivered to the flashSSD at the same time, the full bandwidth of the flashSSD cannot be exploited.

[2] The mapping is required because write operations to flashSSDs are internally processed with out-place write operations (i.e. no direct overwrites to flash memory pages) to the flash memory packages, due to the erase-before-write feature of flash memory.

Therefore, general principles to design I/O bound algorithms for flashSSDs are as follows. First, minimize the total amount of I/O rather than minimize the total number of random accesses. Due to the insignificant access latency and the advance of FTL technologies, the amount of I/O determines the performance of the I/O bound algorithms. Second, exploit the internal parallelism by delivering a number of I/O requests to flashSSDs simultaneously [41].

In this respect, In-place index update strategies have more potential on flashSSDs than on HDDs because they need considerably less amount of I/O than Merge-based strategies during indexing.

In this article, we first verify our conjecture that In-place index update strategies will be better than Merge-based strategies on flashSSDs. Then, we propose a novel inverted index maintenance strategy, called Multipath Flash In-place Strategy (MFIS) which not only requires the minimum amount of I/O but also exploits the internal parallelism of flashSSDs. When the in-memory inverted index is combined with the on-disk inverted index, MFIS reads the last blocks of postings lists into memory from a flashSSD simultaneously rather than reads one by one. Then, updated posting lists are written back to the flashSSD again from memory at the same time. The posting list of each term is allowed to be stored non-contiguously to minimize the amount of I/O. In the query processing of MFIS, non-contiguously stored posting list is read at once to increase the number of I/O requests to flashSSDs.

To verify our conjecture, we experimentally evaluated the performance of In-place index update strategies with a representative Merge-based strategy on a HDD [18] and a flashSSD (ioDrive [16]). In the index maintenance performance, while Geometric Partitioning was extremely faster than contiguous In-place and non-contiguous In-place (27.47 and 26.74 times, respectively) on the HDD, In-place index update strategies were faster than Geometric Partitioning (1.27 and 1.49 times, respectively) on the flashSSD. Specifically, non-contiguous In-place was 1.17 times faster than contiguous In-place in the index maintenance performance. In contrast, non-contiguous In-place was 3.88 times slower than contiguous In-place in the query processing performance on the flashSSD. This is because non-contiguous In-place reads scattered postings without exploiting the internal parallelism of flashSSDs.

We empirically evaluated the index maintenance and query processing performance of MFIS. The experiments were conducted on three different flashSSDs (ioDrive [16], Vertex3 [37], and P300 [36]). The result showed that the index maintenance of MFIS was up to 14.93, 4.04, 5.12, and 2.33 times faster than Merge-based strategies such as Immediate Merge, Geometric Partitioning, Hybrid, and SSD-aware Hybrid, respectively. MFIS was up to 3.78 and 2.27 times faster on indexing than traditional In-place index update strategies such as contiguous In-place and non-contiguous In-place, respectively. MFIS showed up to 1.71 times higher query processing performance than non-contiguous In-place. Furthermore, it showed almost the best query processing performance as Immediate Merge did. In addition, it demonstrated slightly faster query processing performance than Geometric Partitioning, Hybrid, and SSD-aware Hybrid (on average 1.05, 1.07, and 1.09 times, respectively).

There are three contributions of this article. First, we have made a discovery that In-place index update strategies can be better than Merge-based strategies on flashSSDs, which is contrary to the long held belief that In-place index update strategies show poor index maintenance performance in most cases. Second, we proposed the best inverted index maintenance strategy for flashSSDs. Our method not only considerably outperformed the existing strategies in the index maintenance performance, but also it showed nearly the best query processing performance which was slightly better or nearly the same as the performance of existing strategies such as Immediate Merge, Geometric Partitioning, Hybrid, SSD-aware Hybrid, and In-place index update strategies on flashSSDs. To the best of our knowledge, MFIS is the first inverted index maintenance strategy based on In-place index update strategy for flashSSDs.

This article is organized as follows. Section 2 describes the advance of FTL technologies with its implication in write amplification factor, the importance of exploiting internal parallelism of flashSSDs, and also provide background on diverse inverted index maintenance strategies. In Section 3, we revisit In-place index update strategies. We introduce our new inverted index maintenance strategy for flashSSDs in Section 4. We evaluate our algorithm through experiments with three flashSSDs in Section 5. We describe related work in Section 6. Section 7 concludes our work.

2. Background

2.1. FlashSSD architecture

Fig. 1 illustrates the architecture of a flashSSD, which is composed of multiple flash memory packages. More precisely, n flash memory packages are gathered into a channel and m channels are connected to a SSD controller, which includes a CPU processor and a RAM buffer. Since flashSSDs have no mechanical part to move contrarily to HDDs, no mechanical movement overheads such as seek or rotational delays exist. Therefore, access latencies of flashSSDs are considerably shorter than those of HDDs. For the example of devices used in our empirical study, the average access latencies of P300 flashSSD is 80.32 (read) and 31.53 (write) times shorter than those of the Deskstar 7K1000.D HDD (refer to Table 1 for the details).

2.2. Advance of FTL technologies and write amplification

FlashSSDs include FTL (Flash Translation Layer) that maps LBAs (logical block addresses) of the host to physical addresses of flash memory. The mapping is required because write operations to flashSSDs are internally processed with out-place write operations (i.e. no direct overwrites to flash memory pages) to the flash memory packages, due to the erase-before-write feature of flash memory. FTL mapping methods and associated background operations such as garbage collection and wear leveling have been a major research topic by previous studies [21–24,32,33,38] for the last decade. Since FTL mapping methods and associated background algorithms have been key factors of flashSSD performance, manufacturers were reluctant to reveal their methods and algorithms. However, numerous FTL mapping algorithms have been proposed in the research domain and manufacturers have adopted proposed methods and made them more sophisticated to commercialize them. One of the major performance measures in FTL mapping methods is *write amplification factor* (WAF) which is the ratio of the data written by the host to the data written to the flash memory. This is a crucial factor for the FTL performance since it can determine the lifetime of flashSSDs. The flash memory has the limited number of erases for each block. Therefore, the higher WAF, the shorter lifetime of the flashSSD. FTL mapping methods have evolved by reducing mapping granularity from block-level mapping to page-level mapping, thereby reducing the write amplification factor. Due to the large mapping granularity, block level mapping was prone to small-sized random writes, causing very high WAF values. Then, hybrid mapping methods such as BAST [22] and FAST [23] were proposed, which commonly adopts log-blocks that temporarily hold the data of updated pages. With the advent of hybrid mapping methods, the internal write overhead caused by small sized random writes was alleviated, and thus WAF values could be significantly reduced. Later, many variants of these hybrid mapping methods such as Superblock FTL [21], and SAST [38], LAST [24] were proposed. Recently, a feasible page-level mapping method [33] was proposed which was considered to be the smallest mapping granularity. Most recently, object-level mapping method OFTL [32], which has even less mapping granularity than the page-level mapping, has been proposed. OFTL claims to have 0.19∼0.89 WAF values. The FTL technologies have been substantially enhanced in both research domain and industry. Recent benchmark results [3,4] based on SMART [2] logs reported that recent commercial flashSSDs such as Samsung 840 EVO and Intel SSD 520 have WAF values that range from 0.17 to 2.9.

2.3. *Importance of exploiting internal parallelism of flashSSDs*

Because of the embedded multiple flash memory packages, flashSSDs have the internal parallelism. Multiple I/O requests can be simultaneously performed on the m flash memory packages through multiple channels. This is referred to as channel-level parallelism [13]. Additionally, multiple I/O data transfers can be interleaved even in a single channel among the n flash memory packages ganged into the channel. While some of the flash memory packages are busy with their own I/O processing, I/O data transfers to the other flash memory packages can be done in the shared channel. The pages of the flash memory packages that share the same channel are usually striped, thus making a larger I/O unit. This is referred to as package-level parallelism [13]. Therefore, a flashSSD can theoretically have the maximum bandwidth as high as $m{\bullet}n$ times the bandwidth of a flash memory package.

In order to utilize the maximum bandwidth of a flashSSD, it is required to exploit the internal parallelism. Unless I/O requests with large amount of I/O are delivered to flashSSDs at the same time, some of the flash memory packages can become idle. There are two general principles to exploit the internal parallelism of flashSSDs [41]. First, in order to exploit the channel-level parallelism, request multiple I/O requests (especially random I/Os) at once. In this way, multiple channels can deliver at once the I/O requests to multiple flash memory packages connected to the channels. Second, in order to exploit the package-level parallelism, request I/Os with large granularity (large I/O sizes). This can make the flash memory packages, which shares the same channel, busy. Moreover, this can make the designated blocks for the I/O requests spread over more flash memory packages across the channels.

In order to apply the first principle, it is needed to request multiple I/Os at once. A recent study [41] suggested a new I/O request method called *Psync* I/O (Parallel Synchronous I/O), which can deliver multiple random I/Os at once to a flashSSD with a single request in a single process (or a thread). *Psync* I/O is similar to traditional synchronous I/O except that *Psync* I/O uses an array of I/O requests as a unit of its operation. It delivers the array of I/O requests to the flashSSDs and retrieves the request results at once. Another array of I/O requests can be submitted in sequence only after the results of the previous I/O requests are retrieved.

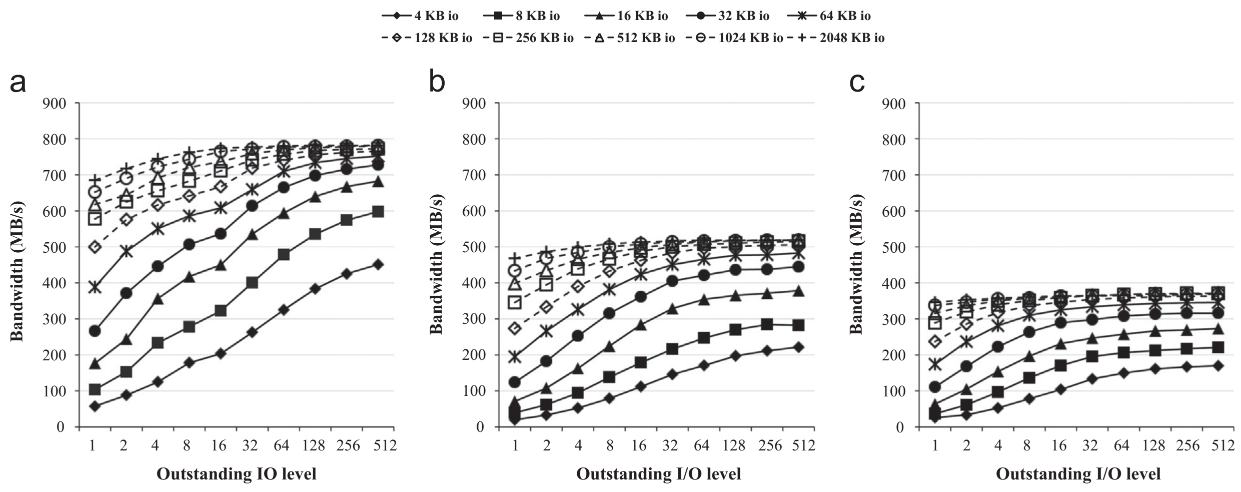

Fig. 2. Benchmark results of the read performance with *Psync* I/O on three flashSSDs (a) ioDrive (b) Vertex3 (c) P300.

To investigate the performance of *Psync* I/O, we conducted benchmarks to measure read bandwidths of *Psync* I/O on three flashSSDs (ioDrive [16], Vertex3 [37], and P300 [36]). The devices used for the benchmarks were described in Section 5.1. *Psync* I/O was implemented by using the Linux-native asynchronous API (libaio) [5] as described in [41]. We measured the read bandwidths increasing the number of random I/Os requested at once through *Psync* I/O (outstanding I/O level) from 1 to 512 with the I/O size fixed. We repeated the same benchmark, increasing the I/O size from 4KB to 2048KB. In order to compare the performance of *Psync* I/O with that of synchronous I/O, when the outstanding I/O level was 1, the I/O was requested using synchronous I/O with the corresponding I/O size. Fig. 2 presents this result. Higher outstanding I/O level created higher bandwidths. In addition, larger granularity of I/O generated higher bandwidths. It is worth noting that even with small granularity the bandwidths of flashSSDs converged close to the maximum bandwidths when the outstanding I/O level was high enough (e.g. when the outstanding I/O level was 512 with the I/O size of 16KB on ioDrive). In general, when *Psync* I/O was performed with *N* outstanding I/O level and *S* KB size, it generated almost the same bandwidth as a single I/O with the I/O size of *N*•*S* KB. For example, when the outstanding I/O level was 8 with the I/O size of 128KB, the bandwidth (336.52MB/s) was almost the same as the bandwidth (337.22MB/s) of a single synchronous I/O with the I/O size of 1024KB on Vertex3.

2.4. On-line maintenance strategies

Prior to explanation of indexing strategies, we first introduce the process of how new documents are inverted into an index. The IR system tokenizes the input documents forming a lexicographically ordered list of terms called the vocabulary and a list of postings (called posting list) corresponding to each term. Each posting contains term's information such as document ID, which is the unique number of the document containing the term, term frequency in the document, and a position list of the term within the document. An in-memory inverted index is updated in main memory until it becomes full. Once the main memory is fully filled, the in-memory inverted index is moved to HDDs, and then combined with the on-disk inverted index. Query requests can be served during the index maintenance process in on-line indexing strategies.

2.4.1. Merge-based maintenance

Merge-based strategies maintain the on-disk inverted index by merging the in-memory posting lists with the on-disk posting lists. During the merge process, the in-memory posting lists are checked to determine whether they have to be updated with existing on-disk posting lists. By checking both the on-disk terms and the in-memory terms, the posting lists of terms that only exist in memory are written to the new on-disk index and the posting lists of terms that only exist on the on-disk index are read from the old on-disk index and re-written to the new on-disk index. The posting lists of terms that exist both on the on-disk and in-memory index are merged and re-written to the new on-disk index. This strategy also requires keeping a copy of the old on-disk index to deal with query processing during the merge process.

Several studies have been made to improve on-line indexing strategies based on the Merge strategy. No-Merge [6] performs no merge operation when the in-memory index is moved to HDDs. Instead, it creates a new on-disk index called sub-index. Therefore, it shows high index maintenance performance and poor query processing performance since it requires at most n disk seeks to read a posting list for a given term, where n is the number of sub-indexes. Immediate Merge [27–29] is an extreme case optimized for query processing performance that performs merge operation whenever the in-memory index is moved to HDDs. Immediate Merge shows high query processing performance because only one on-disk index is maintained, and the posting list of each term is stored contiguously. However, the index maintenance performance worsens as the size of index grows since the entire index has to be read and the whole updated index needs to be written for every merge. To balance index maintenance and query processing performance, several algorithms have been proposed. For example, both Logarithmic Merge [6] and Geometric Partitioning [26] aim to increase index maintenance performance by allowing multiple on-disk sub-indexes and periodically performing the merge operation. Instead of merging whenever the in-memory index is moved to HDDs, these strategies perform the merge operation at their specific conditions. In Geometric Partitioning, each partition includes a sub-index. Each partition only contains no more than the defined number of maximum postings. Parameter r is used to define the capacity of postings on each partition. If the in-memory index can hold b postings, $(r-1)r^{(k-1)}b$ postings can be included in the k^{th} partition. When a newly generated index leads to more than $(r-1)r^{(k-1)}b$ postings in k^{th} partition, the new index is merged with the existing old index and moved to the $k+1^{\text{th}}$ partition. This is iterated until there are no more excess of postings on the corresponding partition. Logarithmic Merge introduces the concept of "generation" instead of "partition". If there are two sub-indexes on the same generation g, they are merged, and then moved to generation $g+1$. This merge process is repeated until no more collisions occur on generations. Despite the enhanced index maintenance performance, these methods still have the drawback of seeking multiple sub-indexes in query processing.

2.4.2. In-place index update

In-place index update strategies [28,42,45] do not read posting lists of terms, which do not exist in the in-memory index. When the in-memory index is combined with the on-disk index, the posting list of each term is appended to the free space for the posting list of the corresponding term. Unless the corresponding term exists on the on-disk index, the term is inserted into the vocabulary, and posting list is appended to the end of the inverted index file. Since In-place index update strategies require no read and write for unrelated posting lists, the total amount of I/O for the index maintenance is lower than for other approaches. However, since updates of posting lists are conducted term by term, it leads to numerous random I/Os, which are very slow on HDDs because of the expensive access cost of HDDs.

In-place index update strategies can be generally classified into the contiguous In-place [28] and non-contiguous In-place [45] strategies according to how the posting list of each term is stored on-disk index. To contiguously store the posting list, when the free space of a term is insufficient for new postings of the term, relocation is occurred (contiguous In-place). Relocation reads all the old postings of a term from the on-disk index, appends new postings, and writes the integrated posting list to the free space of the on-disk index. When the integrated posting list is written, the free space of this term is over-allocated for future incoming postings. Several over-allocation policies [28,42,45] were proposed. Proportional over-allocation [45] allocates $k \cdot s$ space for a posting list where k is the proportional over-allocation factor and s is the size of an integrated posting list. Consequently, contiguous In-place has overheads such as additional seek time, I/Os for relocation, and the wasted space for over-allocation. On the other hand, the posting list can be stored non-contiguously without relocation (non-contiguous In-place). Instead of relocating the old posting list, only new postings are written to the end of the on-disk index. Since non-contiguous In-place does not need relocation, it requires less amount of I/O than contiguous In-place. Therefore, it has the advantage over contiguous In-place in the index maintenance performance. However, since the postings of each term are scattered in the inverted index, it leads to numerous disk seeks for even a single query, which only contains a single term as a keyword. Consequently, it causes the degradation of query processing performance.

There are some variations of In-place index update strategy such as including short posting lists within the vocabulary [15], predictive over-allocation for long posting lists to alleviate relocation [42], and using a “bucket” structure to handle short posting lists while long posting lists are maintained using various criteria [43,45].

2.4.3. *Hybrid*

Hybrid strategies [7–9] are a mixture of Merge and In-place index update strategies. The idea is based on separating short posting lists and long posting lists. Short posting lists are updated according to Merge-based strategies, whereas long posting lists are updated with In-place index update strategies. This is because In-place index update strategies show better index maintenance performance when the long posting list is updated, whereas Merge-based strategies show better index maintenance performance when the short posting list is updated [7–9]. This difference comes from HDDs' random access latency. If numerous postings are read to be updated with a single random access, the random access latency is a relatively small cost compared to the transfer time needed for the large set of postings. In contrast, random accesses are considered a relatively heavy overhead when only few postings are read from the on-disk index to be updated. This is because the random access latency dominates the transfer time needed for the small set of postings. Therefore, maintaining long posting lists by In-place index update strategies is preferred even though In-place index update strategies cause a random access for reading and writing a posting list of each term. On the other hand, maintaining short posting lists by Merge-based strategies is preferred, since Merge-based strategies are not negatively influenced by the random access.

3. Rediscovery of in-place index update strategy

In-place index update strategies have been regarded as the most inefficient maintenance approach when the HDDs are used for main storage of the inverted index [29]. This is because numerous random accesses occur during indexing since it has to first find the location of the posting list where new postings are appended for each term. These random accesses can cause performance degradation on HDDs because of long access latency of HDDs (refer to Table 1 in Section 1). For this reason, most of the index maintenance strategies are developed based on the Merge-based strategy in order to avoid random accesses.

However, on flashSSDs, the overhead that results from random access can be ignored since the access latency of flashSSDs is insignificant. Therefore, In-place index update strategies can be the most suitable maintenance strategy on flashSSDs. In-place index update strategies have advantages over other strategies. First of all, In-place index update strategies only consider posting lists of terms that exist in the in-memory inverted index. In other words, it eliminates the overhead that is caused by unnecessary read and write of posting lists that do not reside in the in-memory index. However, In-place index update strategies cause a random read each time when the posting list of the term is loaded to memory and a random write when the updated posting list is written again to the on-disk index. For this reason, In-place index update strategies are not considered an appropriate method to deal with large volume documents on HDDs. However, since the cost of random access is insignificant on flashSSDs, random accesses to update posting list are not a significant overhead cost for flashSSDs.

To verify our conjecture, we evaluated the performance of In-place index update strategies with a representative Merge-based strategy on a HDD (7200 rpm Hitachi Deskstar 7K1000.D [18]) and a flashSSD (ioDrive [16]). We measured the total indexing time when the in-memory inverted indexes are moved to the HDD or flashSSD. Each in-memory inverted index was built with 1,000 documents. Once an in-memory index was combined with the on-disk index, 10,000 queries were performed to measure query processing performance. During the experiment, a total of 100 in-memory indexes were moved to the HDD or flashSSD. In other words, 100,000 documents were indexed into the inverted index. More details of the experimental settings can be found in Section 5.1. As described in Section 2.4.2, depending on the manner of maintaining posting lists, In-place index update strategies were implemented in two ways [45]: 1) contiguous In-place (*In-place_C*) with $k = 1.25$ (the proportional over-allocation factor described in Section 2.4.2) and 2) non-contiguous In-place (*In-place_NC*). Non-contiguous structure introduced in [45] as an extreme allocation policy for long posting lists optimized for index maintenance performance. In this article, to demonstrate the superiority of non-contiguous structure in the index

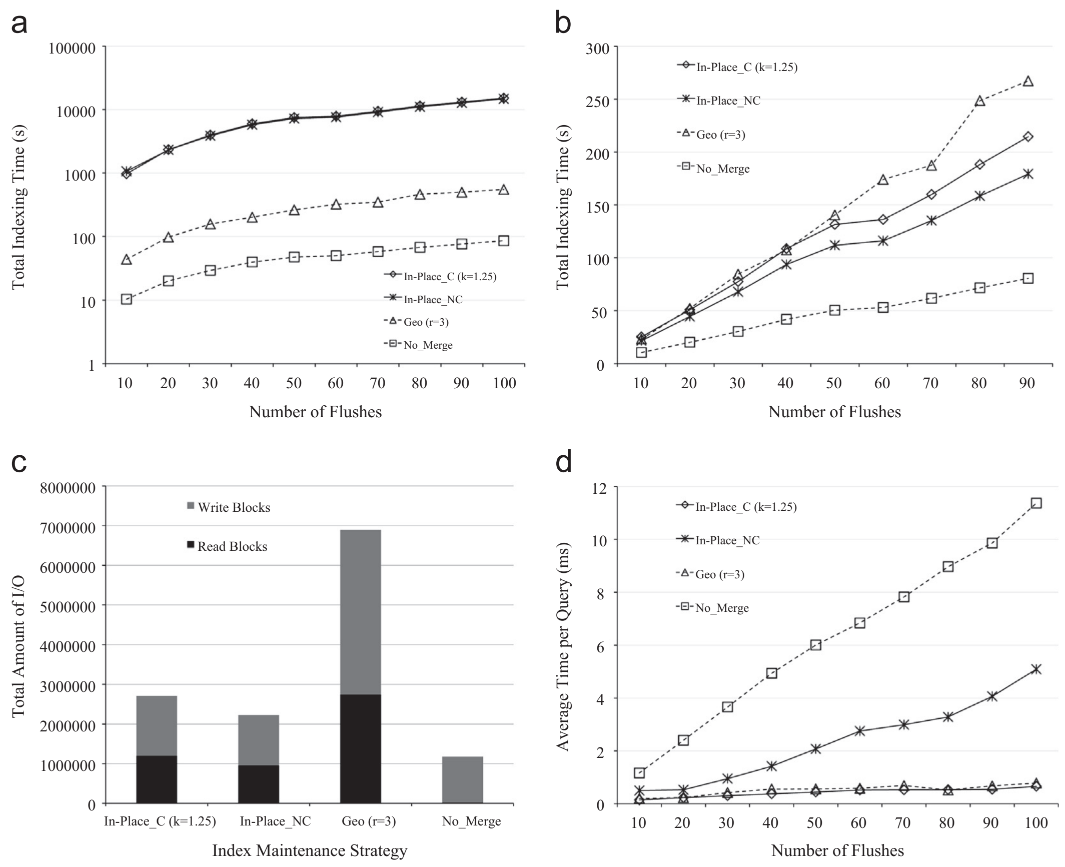

Fig. 3. Comparisons of In-place update strategies performance on the HDD and flashSSD (a) Indexing time on the HDD (b) Indexing time on the flashSSD (c) Amount of I/O (d) Average time per query on the flashSSD.

maintenance performance, the posting list of each term is stored non-contiguously regardless of length of posting lists. When the space of a posting list is not enough for new postings, non-contiguous In-place stores new postings at the end of the on-disk index instead of relocation. As a representative Merge-based strategy, Geometric Partitioning (*Geo*) was used, which is one of the most suitable Merge-based maintenance strategies for HDDs. *Geo* was implemented based on the descriptions in their paper [26]. According to their paper, 3 was chosen for parameter *r*. As a lower boundary of indexing time, No-Merge (*No_Merge*) [3] that creates a new index for each flush was chosen.

Fig. 3(a) presents the total indexing time on the HDD. Geo and *No_Merge* showed extremely better index maintenance performance than both In-place index update strategies. *Geo* was 26.74X and 27.47X faster than *In-place_NC* and *In-place_C*, respectively. *No-Merge* was 172.04X and 176.73X faster than *In-place_NC* and *In-place_C*. As depicted in Fig. 3(b), however, In-place index update strategies showed better index maintenance performance on the flashSSD. *In-place_NC* and *In-place_C* was 1.49X and 1.27X faster than *Geo*. The gap of performance among *In-place_NC*, *In-place_C*, and *Geo* was increased as the size of the index grows. *No_Merge* was 2.22X and 2.68X faster than *In-place_NC* and *In-place_C*, respectively. Although *No_Merge* still showed the fastest update performance, this result contradicts the common wisdom that the In-place index update strategy is the slowest approach among the inverted index maintenance strategies.

This phenomenon is due to the minimal access latency of flashSSDs. Fig. 3(c) presents the total amount of I/O of each strategy, suggesting how the access cost affects index maintenance performance. As shown in Fig. 3(c), In-place index update strategies required less amount of I/O than *Geo* during indexing. Especially the amount of written data of *In-place_NC* is about four times less than that of *Geo*. This is because In-place index update strategies only updated terms that existed in the in-memory index. Nevertheless, it took more indexing time on the HDD because of the long access latency of the HDD (refer to Table 1 in Section 1). Since posting lists of terms to be updated were scattered in the on-disk index, numerous random accesses were occurred. However, on flashSSDs, thanks to the negligible access latency of flashSSDs, In-place index update strategies outperformed *Geo*. Specifically, *In-place_NC* showed better index maintenance performance than *In-place_C*. This is because *In-place_C* has the relocation overhead as described in Section 2.4.2. Although the relocation did not occur many times with small dataset (100,000 documents), the significant overhead of relocation can be examined as the size of index grows (refer to the experiment with large dataset in Section 5.1). However, considering the high gap between the amount of I/O of *Geo* and *In-place_NC* (about a factor of 3), the indexing time gap between *Geo* and *In-place_NC* is relatively small (about a factor of 1.5). This can be explained by the assumption that random I/Os created by *In-place_NC* could not utilize the internal parallelism of flashSSDs enough and could worsen the write amplification factor due to random writes.

Fig. 3(d) presents query processing performance on the flashSSD. *In-place_NC* showed considerably longer query time than *Geo* and *In-place_C*. *In-place_NC* was on average 2.80X ($_{\text{Min}}$: 1.54X, Max: 3.85X) and 3.88X ($_{\text{Min}}$: 2.20X, Max: 5.13X) slower than *Geo* and *In-place_C*, respectively. The performance gap was steadily increased. Since the posting list of a term is not stored contiguously, a number of random reads are required for even a single query, which only contains a single term as a keyword. Since the random reads are requested one by one with small I/O sizes in *In-place_NC*, the full bandwidth of flashSSDs cannot be utilized since the internal parallelism can be exploited only if there is enough amount of I/O requested at once as we mentioned in Section 2.3. In contrast, since *In-place_C* and *Geo* store the posting list of each term contiguously, the whole postings of each term are read with larger I/O sizes, thereby utilizing higher bandwidth of flashSSDs than *In-place_NC*. *No_Merge* showed extremely slow query performance compared to other strategies. *No_Merge* was on average 13.73X ($_{\text{Min}}$: 8.12X, Max: 18.06X), 11.37X ($_{\text{Min}}$: 5.77X, Max: 17.15X) and 2.96X ($_{\text{Min}}$: 2.23X, Max: 4.55X) slower than *In-place_C*, *Geo*, and *In-place_NC*. Since *No_Merge* approach do not merge with existing on-disk indexes while *In-place_NC* appends new postings to the end of the existing posting if pre-allocated space has enough space, *No_Merge* generates more random accesses for query processing than *In-place_NC* does. Therefore, *No_Merge* cannot be an appropriate update strategy even though it shows the best update performance.

In conclusion, we discovered that In-place index update strategies have the potential to be better update strategy than Merge-based strategies in the index maintenance performance on flashSSDs especially when the posting list of each term is stored non-contiguously. Due to the insignificant access latency of flashSSDs and the minimum amount of I/O written during indexing, non-contiguous In-place showed the best index maintenance performance. Nevertheless, it still does not utilize the maximum bandwidth of flashSSD in indexing and query processing, due to no consideration of exploiting the internal parallelism of flashSSDs and the overhead of write amplification worsened by random writes. In the experiments, this problem was more clearly revealed in query processing. Since query processing performance of non-contiguous In-place was much worse than the others, non-contiguous In-place cannot be a choice for one of the practical solutions on flashSSDs without enhancing query processing performance.

To address this problem and maximize the performance of index maintenance and query processing, we design a new inverted index maintenance strategy that exploits the internal parallelism of flashSSDs, thereby utilizing the maximum bandwidth of flashSSDs in both indexing and query processing. For the write amplification problem worsened by random writes, we rely on FTL mapping technologies that have evolved significantly. The state of the art FTL mapping technologies alleviate our concerns since they have already achieved to sustain the write amplification factors under 1 even with an extreme workload including only random writes. The benchmark results of recent commercial flashSSDs (WAFs ranging from 0.17 to 2.9) demonstrate this as mentioned in Section 2.3. Note that the write amplification in the host system for *Geo*, a factor of 4, dominates the write amplification inside a flashSSD for *In-place NC*, a factor of 0.17 to 2.9, since the amount of written data of *Geo* is about four times more than that of *In-place_NC*.

4. Multipath flash in-place strategy (MFIS)

We propose a novel inverted index maintenance strategy for flashSSDs, called Multipath Flash In-place Strategy (MFIS). There are two key ideas. One is to non-contiguously maintain a posting list of each term in order to minimize the total amount of I/O when the in-memory index is combined with the on-disk index. The other is to utilize *Psync* I/O (refer to Section 2.3) in order to request multiple I/Os at the same time when the posting lists are read and written on flashSSDs, thereby exploiting the internal parallelism of flashSSDs.

4.1. Data structure

In this article, a unit of the on-disk inverted index is a block. The block contains multiple postings. A set of sequentially located blocks composes a chunk. As shown in Fig. 4, there are postings of “Term 1” on Block1 and Block2. A chunk of “Term 1” consists of Block1 and Block2. Postings are stored in ascending order of document ID within blocks. We assume that each document ID is assigned in order of entry into the IR system. This order is maintained even after posting lists are updated. In this article, the block size of MFIS is fixed at 8KB that can contain 1022 postings at most. As shown in Fig. 2(Section 2.3), larger granularity of I/O generated higher bandwidths. If the size of block is too small, it is hard to utilize the maximum bandwidth of flashSSDs. On the other hand, if the size of block is too large, it leads to the internal fragmentation in the block, which only has small number of postings. Therefore, 8KB is not only enough to generate the maximum bandwidth of flashSSDs through *Psync* I/O and also to minimize the internal fragmentation in the block.

4.2. Index maintenance of MFIS

The index maintenance process of MFIS is composed of three phases: Scan, Migration, and Write.

First, in the Scan phase (lines 13–18 of Algorithm 1), terms built in the in-memory inverted index and terms stored on the on-disk inverted index are scanned. Postings of terms that only exist in the in-memory inverted index are moved to the output buffer (line 16). Meanwhile, the last blocks of posting lists to be updated are read to the input buffer from the on-disk inverted index. To read many blocks at once, the read operations for the multiple desired

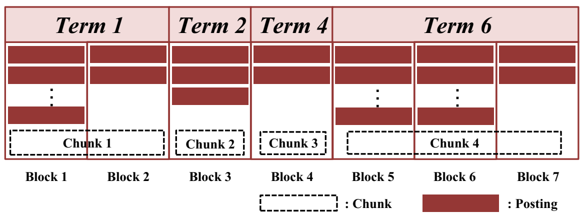

Fig. 4. Internal data structure of the inverted index.

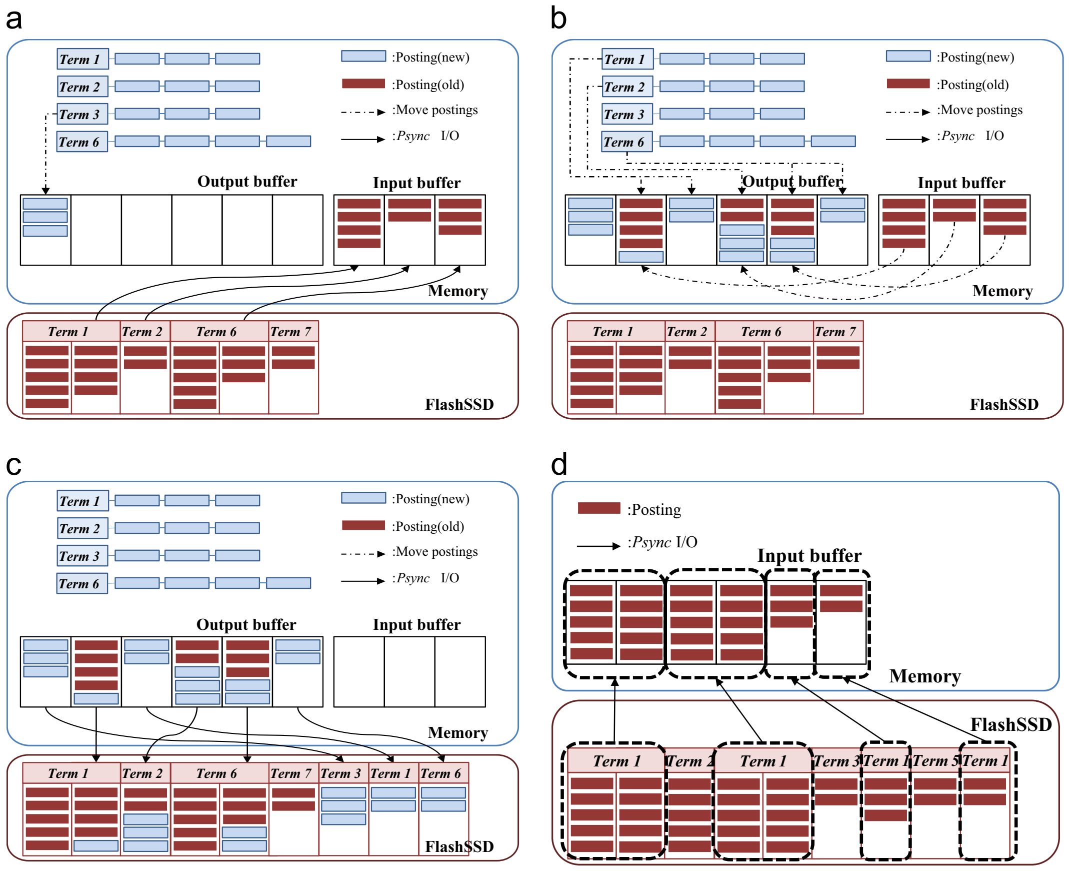

Fig. 5. An example of MFIS (a) Scan (b) Migration (c) Write (d) Query Processing.

blocks are performed through a single *Psync* I/O request (lines 4, 9). By using this method, we can utilize the maximum read bandwidth of flashSSDs. As described in Section 2.3, requesting multiple I/O at the same time generates higher read bandwidth in flashSSDs. Therefore, Scan phase increases the performance the algorithm rather than reading blocks term by term. For the example of Fig. 5 (a), postings of "Term 3" are moved to the output buffer since "Term 3" only exists in the in-memory index. On the other hand, the last blocks of "Term 1", "Term 2", and "Term 6" are loaded to the input buffer simultaneously through *Psync* I/O. Scanning the terms continues until the input buffer is fully filled with blocks at the defined number of maximum blocks (*threshold*) of the input buffer.

Algorithm 1:. Index Maintenance of MFIS

```
Procedure MFIS_UPDATE(V,P,InputMax)
Input: V(vocabulary), P(set of posting lists), InputMax(the threshold
    of InputBuffer)
1:  for each term t in V do
2:    Scan(t, P(t), t.lbp)
3:    if (r_point.size >= QUOTE InputMax) then
        /*r_point[] : array of locations of blocks to be read from
      flashSSDs*/
4:    PsyncRead(InputBuffer[], r_point[])
5:    for each block ib in InputBuffer[] do
6:        Migration(ib, ib.term, ib.obp)
7:      Write()
8:  if(r_point.size !=0) then
9:    PsyncRead(InputBuffer[], r_point[])
10:   for each block ib in InputBuffer[] do
11:     Migration(ib, ib.term, ib.obp)
12:   Write()
function Scan(t, P(t), t.lbp)
Input : t (term to be scanned) , P(t) (posting list of term t) , t.lbp (the
  location of last block on flashSSDs of term t)
13:  if t exists both on-disk and in-memory index then
14:      r_point[i_index++] := t.lbp
15:  else if t only exists in-memory then
16:     N := move(P(t), OutputBuffer[o_index])
17:     for i := 0 to N-1 step 1 do
           /*w_point[] : array of locations of blocks to be written to
  flashSSDs, eof : the location of the end of file*/
18:        w_point[o_index++] :=eof+i
function Migration(ib, ib.term, ib.obp)
Input: ib (a block in InputBuffer[]) , ib.term (term of block ib), ib.obp
  (ib's original block location on flashSSDs)
19:  move(ib, OutputBuffer[o_index])
20:  posting_list := getPostingList(ib.term)
21:  size := append(posting_list, OutputBuffer[o_index])
22:  for i := 0 to size-1 step 1 do
23:    if i = 0 then
24:        w_point[o_index++] :=ib.obp
25:    else
26:        w_point[o_index++] :=eof+i
function Write()
27:  PsyncWrite(w_point[])
28:  Reset r_point[], w_point[] , o_index := i_index := 0
```

Second, in the Migration phase (lines 19–26), new postings are appended to empty spaces of the corresponding loaded blocks (line 21) after the postings in the input buffer are migrated to the output buffer (line 19). New blocks are allocated for new postings in case loaded blocks do not have enough space to hold the postings. Fig. 5(b) shows how old postings of “Term 1”, ”Term 2”, and “Term 6” are combined with new postings. In this example, we assume each block contains at most five postings. For this reason, new blocks are allocated for “Term 1” and “Term 6” at 3rd and 6th of the output buffer, respectively.

Finally, after the whole postings of terms are appended, loaded blocks are re-written simultaneously to flashSSDs where the blocks have existed, in the Write phase (lines 27, 28). When new blocks are allocated, they are written to the end of the inverted index file. These write operations are conducted at once through *Psync* I/O (line 27). Similar to Scan phase, writing numerous blocks at the same time also makes it possible to exploit the internal parallelism of flashSSDs. As described in Section 2.3, since flashSSDs have the embedded multiple flash memory packages, requesting multiple I/O increases update performance of the algorithm rather than writing blocks term by term. As shown in Fig. 5(c), the 1st, 3rd, and 6th blocks of the output buffer are written to the end of the on-disk index file. In contrast, the 2nd, 4th, and 5th blocks are re-written to the locations where they came from. To write the blocks from the output buffer at once, *Psync* I/O is used.

Algorithm 2:. Query Processing of MFIS

```
Procedure MFIS_SEARCH(Term, InputMax)
Input:   Term(a term of query), InputMax(the threshold of
    InputBuffer)
    //CP[] : array of location and size of chunks
1:  CP[] := Vocabulary.find(Term)
2:  i_index := 0
3:  for each chunk c in CP[] do
      /*r_point[] : array of locations of blocks to be read from
    flashSSDs*/
4:      if(r_point.size >= InputMax) then
5:        PsyncRead(r_point[], InputBuffer[])
6:        i_index := 0
      /*c.numBlks : the number of blocks in a chunk c, c.pos : the start
    location of the chunk c */
7:    for i := 0 to c.numBlks-1 step 1 do
8:      r_point[i_index++] := c.pos + i
9:if(r_point.size != 0) then
10:     PsyncRead(r_point[],InputBuffer[])
```

In summary, MFIS begins from the Scan phase to find the last blocks of terms to be updated (line 2). Once the number of the found last blocks reaches the *threshold* of the input buffer (line 3), MFIS reads these blocks at the same time through *Psync* I/O (line 4). After reading the last blocks, MFIS conducts the Migration process to merge the old postings, which are read from the on-disk index, with new postings (lines 5, 6), and then performs the Write process to update the on-disk index (line 7). These processes are repeated until all terms are scanned from the on-disk and the in-memory index.

4.3. *Query processing of MFIS*

MFIS maintains a non-contiguous posting list for each term in an on-disk inverted index. In other words, the blocks that contain the posting list of the term are not contiguously stored in the on-disk index. For example, as shown in Fig. 5(c), chunks of blocks are stored non-contiguously. The 1st block of “Term1” is adjacent to the 2nd block of “Term 1” whereas the 3rd block is located away from the first two blocks. Even though this approach has the benefit to index maintenance performance, it induces random reads with the block size (8KB in this article) during query processing. Although, random reads on flashSSDs are considerably faster than HDDs due to the insignificant access latencies, small sized reads cannot utilize the full bandwidth of flashSSDs. This is because the I/O pattern that synchronously request random reads with small I/O sizes one by one cannot provide enough amount of I/O for flashSSDs to exploit the internal parallelism as we described in Section 2.3. Therefore without a more advanced query processing method, the query processing performance will be worse than other inverted index maintenance strategies that store the posting list contiguously. For example, the query processing of naïve non-contiguous In-place was considerably slower than that of the Merge-based approach in Section 3.

To address this problem, MFIS reads all chunks of the posting list for a given term through *Psync* I/O simultaneously. It searches the vocabulary using a term as a key in the given query. The vocabulary returns a set of locations of chunks for the given term (line 1 of Algorithm 2). By using the set of locations, MFIS reads desired chunks at the same time rather than reads the chunks one by one (lines 5, 10). Since MFIS can read the desired chunks through a single *Psync* I/O request, MFIS can exploit the maximum bandwidth of flashSSDs in query processing as well.

Fig. 5(d) shows an example of query processing of MFIS. The example assumes that the on-disk index is now a state after several updates were performed. It also assumes that “Term 1” is included in a user query. Because chunks of “Term 1” are not stored contiguously, four random reads are needed to read four chunks for the query. The chunks of “Term 1” are read at once through *Psync* I/O instead of reading them chunk by chunk.

As a variation of MFIS_SEARCH, it can be modified to operate with a set of terms instead of a single term. When a user query contains multiple terms, modified MFIS_-SEARCH can read chunks of multiple terms simultaneously instead of term by term.

5. Experimental results

5.1. *Experimental setting*

System setup: All experiments were performed on a Linux machine with a 6-core 3.2 GHz CPU and 16 GB DDR3 RAM. In addition, direct I/O mode was adopted for the I/O requests to bypass the file system cache.

Devices: We carefully chose three recent flashSSDs; Fusion-io ioDrive high-priced enterprised-level SSD [16], OCZ Vertex3 max IOPS a consumer-level SSD [37], and

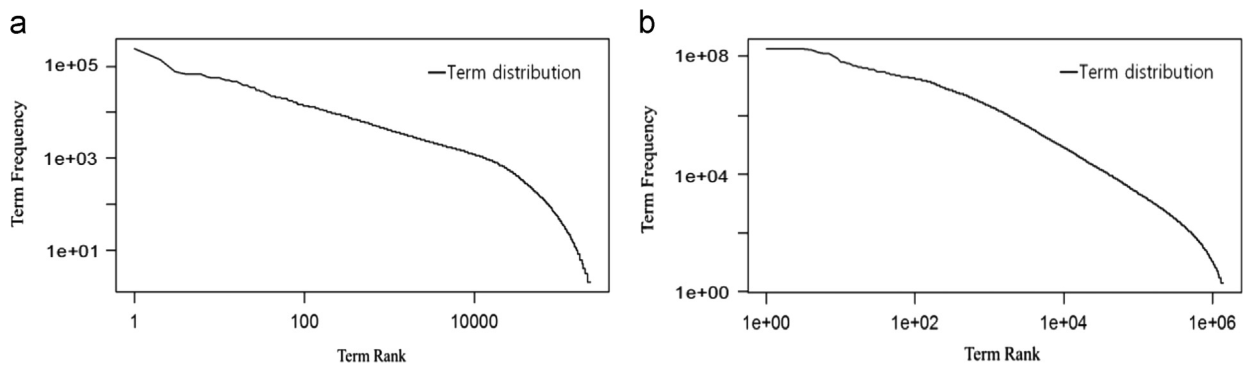

Fig. 6. Rank/frequency curves for documents of experiments in Section 3 and Section 5. (a) 100,000 documents (Section3) (b) 2,000,000 documents (Section 5).

micron P300 an enterprise-level SSD [36] to demonstrate that our algorithm generates performance gains for both index maintenance and query processing regardless of the type of flashSSDs.

Datasets: To evaluate the index maintenance performance, the Wikipedia Static HTML Dump [46] was used as a data set. It consists of 6.5 million documents with a total size of 105GB. The collection is free and is often used as the dataset to measure the inverted index maintenance performance since it is comparable in size to TREC GOV2. The term distribution of 100,000 documents that were used in Section 3 is shown in Fig. 6(a). Similarly, Fig. 6(b) also shows term distribution that follows the Zipf distribution. In other words, the small number of terms contains the majority of postings in the inverted index. To simulate the real world query, we used the AOL query log [1,40]. In Section 3, we only randomly picked up 10,000 queries from the AOL query log. On the other hand, in Section 5, we composed 10,000 queries with the ratio of 70%/30% (long posting list/short posting list) to measure average time per query on SSD-aware Hybrid strategy [30]. We also conducted query processing for each strategy by using the 10,000 queries, which were used for SSD-aware Hybrid strategy.

Implementation details: Throughout all experiments in Section 5, the size of block that contains postings, which is described in Section 4.1, was set at 8KB that can contain 1022 postings at most. To measure indexing time and query processing time simultaneously, we created a sequence of about 20,000 documents insertions and 10,000 queries. Once an in-memory inverted index was built with about 20,000 documents and combined with the on-disk inverted index, query processing was performed. In total, 100 in-memory indexes were moved to flashSSDs in this experiment. In other words, about two million documents were inverted into the index during one million queries.

5.2. *Index maintenance performance*

5.2.1. *Total indexing time*

In this section, we evaluate the total indexing time of MFIS on three flashSSDs. In this experiment, the defined number of maximum blocks (*threshold*) of the input buffer was configured to be 1024. MFIS was evaluated with Immediate Merge (*IM*) [29], Geometric Partitioning (*Geo*) [26], In-place (contiguous In-place: *In-place_C*, non-contiguous In-place: *In-place_NC*) [45], Hybrid (*HLM_NC*) [9], No-Merge (*No_Merge*)[3], and SSD-aware Hybrid (*SSD_Hybrid*) [30].*IM* was implemented by reading entire on-disk index and merging with new in-memory index for every flush. To implement *In-place_C*, the proportional over-allocation factor *k*, which is described in Section 2.4.2, was set at 1.25. *In-place_NC* was used to describe the superiority of non-contiguous structure in the index maintenance performance. *In-place_NC* stores new postings at the end of the on-disk index without relocation if the space of a posting list is not enough for new postings. We chose *Geo* as a representative Merge-based strategy. During the experiment, the parameter *r*, which is used to define the capacity of postings on each partition, was initialized to 3. *HLM_NC* was implemented by blending Logarithmic Merge and non-contiguous In-place, which shows better index maintenance performance than other family of Hybrid strategies. According to the paper of *HLM_NC*, we chose the long posting list threshold values T as 1,000,000. *SSD_Hybrid* is an SSD-aware Merge-based update strategy, which applies *No_Merge* for short posting lists and *IM* for long posting lists. We regarded as the long posting lists that have more than 10,000 postings during each flush of in-memory index. *No_Merge*, which creates a new index for each flush, was used as a lower boundary of indexing time.

Fig. 7 presents the total indexing time of the inverted indexes with different strategies. The y-axis indicates the total time taken to combine the *N* in-memory inverted indexes with the existing on-disk inverted index where *N* is the number of flushes (x-axis) of the in-memory index. Table 2 indicates how many times MFIS was faster than other strategies. MFIS always outperformed the other strategies in the index maintenance performance on three flashSSDs except *No_Merge*, which is the lower boundary of the indexing time.

There are several reasons for this result. First, this is because MFIS requires less amount of I/O than *IM*, *Geo*, *HLM_NC*, *SSD_Hybrid*, and *In-place_C*. Since access latency is insignificant on flashSSDs, the amount of I/O is an important factor of the index maintenance performance. MFIS only handles posting lists, which are required to be updated, while Merge-based strategies such as *IM*, *Geo*, *HLM_NC*, and *SSD_Hybrid* read and re-write unrelated posting lists of terms to maintain contiguous posting lists regardless of whether terms need to be updated or not. Although *In-place_C* also only handles posting lists, which need to

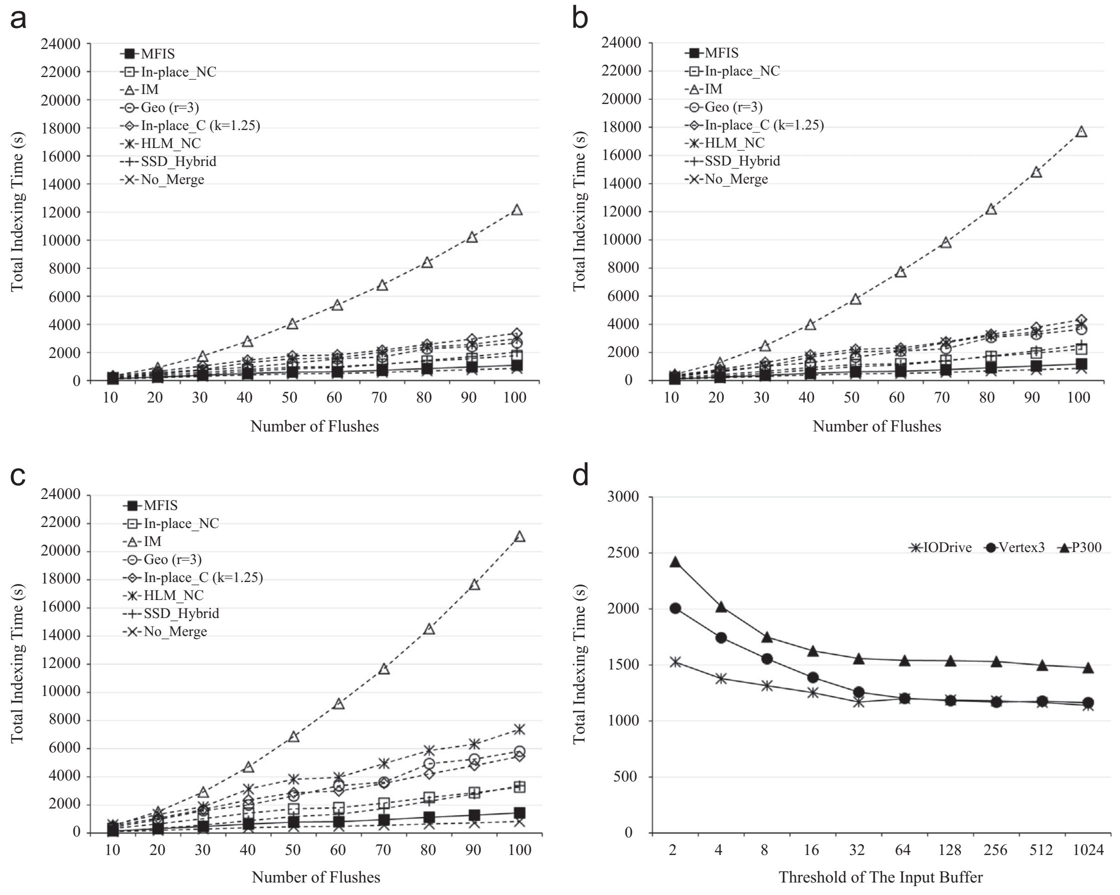

Fig. 7. Index maintenance performance on three flashSSDs (a) ioDrive (b) Vertex3 (c) P300 (d)Varying Threshold.

Table 2
Index maintenance performance gain ratios.

	ioDrive	Vertex3	P300
In-place_NC	1.63	1.88	2.27
IM	11.20	14.93	14.65
Geo (r=3)	2.47	3.07	4.04
In-place_C (k=1.25)	3.10	3.65	3.78
HLM_NC	2.75	3.38	5.12
SSD_Hybrid	1.86	2.13	2.33
No_Merge	0.80	0.73	0.57

be updated, *In-place_C* generates more amount of I/O than MFIS because of relocation. Second, since MFIS can utilize the internal parallelism of flashSSDs, it outperforms *In-place_NC*. Although *In-place_NC* also requires same amount of I/O with MFIS, *In-place_NC* updates posting lists term by term. However, MFIS reads and writes posting lists of terms at the same time through *Psync* I/O. As described in Section 2.3, requesting multiple I/Os simultaneously to flashSSDs can exploit considerably higher bandwidth of flashSSDs than requesting one by one. Although *SSD_Hybrid* showed higher performance compared to traditional strategies such as *IM*, *Geo*, and *HLM_NC*, which are not designed for flashSSDs, MFIS was faster than *SSD_Hybrid*. This is because *SSD_Hybrid* treats long posting list by *IM*, which increases total number of I/O.

5.2.2. Experiment for the threshold of the input buffer

To evaluate the effect of the *threshold* of the input buffer on MFIS, we measured the total indexing time while 100 in-memory indexes moved to flashSSDs varying the *threshold* of the input buffer from 2 to 1024 on the ioDrive, Vertex3, and P300 flashSSDs.

As shown in Fig. 7(d), there were similar tendencies regardless of type of flashSSDs. As the *threshold* of the input buffer increased, the total indexing time steadily decreased. This is because MFIS could read and write more blocks at the same time through *Psync* I/O as the *threshold* value was increased. Fig. 7(d) also showed that the index maintenance performance was very little changed from when the *threshold* value was greater than 32. This indicates that the 32 *threshold* is sufficient to utilize the maximum bandwidth of the flashSSDs regardless of type of flashSSDs.

5.3. Query processing performance

In this section, we evaluate the query processing performance of MFIS with a set of 10,000 queries from the AOL query log. Each query set was performed after an in-memory inverted index was combined with the on-disk inverted index. We measured the average time per query to retrieve posting lists of given terms of 10,000 queries.

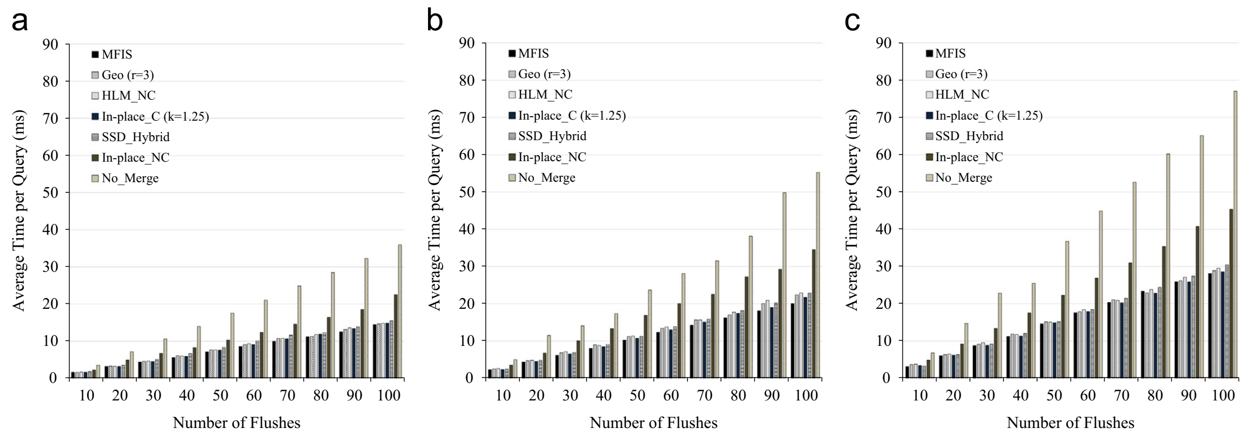

Fig. 8. Average times per query of different maintenance strategies on three flashSSDs. (a) ioDrive (b) Vertex3 (c) P300.

Since postings of each term are stored contiguously in *IM* and *In-place_C*, we only show the query processing result of *In-place_C*. *No_Merge* was used as an upper boundary of query processing. Although some caching [34,35] techniques have been proposed to increase query performance, we did not consider caching effects in experiments. Additionally, we followed standard ranking techniques that return documents containing the identical terms with queries without query expansion [10,12].

As shown in Fig. 8, the average time per query of all the methods was increased as the size of index increased. *No_Merge* and *In-place_NC* showed slow query processing performance on three flashSSDs. On the other hand, MFIS showed similar query times compared to *In-place_C*, *Geo*, *HLM_NC*, and *SSD_Hybrid*. The query processing of MFIS was on average 1.47X, 1.62X, and 1.54X faster than that of *In-place_NC* on the ioDrive, Vertex3, and P300 flashSSDs, respectively. Meanwhile, the query processing of MFIS was on average 1.05X, 1.07X, and 1.09X faster than that of *Geo*, *HLM_NC*, and *SSD_Hybrid* on three flashSSDs, respectively.

MFIS showed considerably enhanced query processing performance compared to *In-place_NC*. This is because MFIS reads a number of non-contiguous chunks for a posting list at the same time through *Psync* I/O while *In-place_NC* reads chunk by chunk with synchronous I/O. According to the benchmark result in Section 2.3, we confirmed why reading chunks simultaneously was faster than chunk by chunk. As shown in Fig. 2, flashSSDs showed higher read bandwidths with higher outstanding I/O levels. In other words, MFIS generated higher outstanding I/O levels so that MFIS could search more quickly utilizing higher read bandwidths.

It is worth noting that MFIS shows almost the same query processing performance compared to *In-place_C*. Additionally, MFIS was slightly faster than *Geo*, *HLM_NC*, and *SSD_Hybrid*. The results have a connection with the following reason.

MFIS uses granularity of I/O that is large enough to exploit the internal parallelism of flashSSDs by binding contiguous blocks into a chunk. It affects to the read bandwidth of *Psync* I/O. As shown in Fig. 2, the larger granularity of I/O (large I/O sizes) was used, the bandwidth reached to the maximum bandwidth with less number of I/Os requested at the same time (less outstanding I/O level). However, in case of very small granularity of I/O, the bandwidth could not reach to the maximum bandwidth even with a very high outstanding I/O level. When MFIS allocates new blocks of postings for a term during the Migration phase, those blocks were stored at the end of index contiguously during the Write phase. For example, the size of a chunk for terms, which were frequent on the query set, on average was 120KB, which contains 15 blocks. Therefore, MFIS used larger granularity for *Pysnc* I/O, thus enabling MFIS to exploit the maximum bandwidth of flashSSDs.

6. Related work

Recently, several studies tried to use a flash memory as storage for the inverted index. An early study [44] designed a search system suitable for small devices using the flash memory as its storage. Additionally, a similar study [11] proposed an inverted index structure for NAND flash memory, which is usually used for small mobile devices. Aside from these studies, Bojun et al. [20] have suggested allocating the flash memory space as much as possible of the DRAM portion for the inverted index. For the flashSSDs, a study [30] suggested a merge-based hybrid inverted index maintenance strategy by applying No-Merge for short posting lists and Immediate Merge for long posting lists. To the best of our knowledge, there has been no study to design flashSSDs-aware inverted index maintenance strategy based on In-place index update strategy.

Data management methods for flashSSDs are more actively studied by the database community. An early study [25] focused on addressing random write problems associated with the first generation of flashSSDs. The authors focused on reducing the number of write operations to flashSSDs. Several flashSSD-oriented index structures were proposed [31,41]. FD-tree [31] focused on exploiting the high bandwidth of flashSSDs by converting random I/Os into sequential I/Os. PIO B-tree [41] focused on exploiting the internal parallelism of flashSSDs.

While early studies paid less attention to the parallel architectures of flashSSDs, a recent study [13] claimed that exploiting the internal parallelism of flashSSDs can considerably enhance the performance of I/O bound applications. Other studies [19,39] tried to improve the internal architecture for nourishing more I/O parallelism inside flashSSDs. In the studies, they noted that channel-level parallelism is a core of the I/O parallelism inside flashSSDs. The most recent study [41] suggested an efficient I/O request method (*Psync* I/O) for submitting random I/Os at once in a single process, which focused on exploiting the channel-level parallelism.

7. Conclusion and discussion

In this article, we rediscovered the potential of In-place index update strategies on flashSSDs. Especially, non-contiguous In-place strategy outperformed Merge-based strategies and even contiguous In-place strategy by eliminating overheads such as re-location and proportional over-allocation. However, non-contiguous In-place strategy does not fully utilize the maximum bandwidth of flashSSDs. More importantly, the query processing performance can be degraded because of the non-contiguously stored posting list. To fully utilize the maximum bandwidth of flashSSDs, and address the degraded query processing performance, we proposed a novel inverted index maintenance strategy that exploits the internal parallelism of flashSSDs, namely Multipath Flash In-place Strategy (MFIS). MFIS enhances the index maintenance performance by reading and writing posting lists, which need to be updated, at the same time through *Psync* I/O. In addition, MFIS addresses the structural problem of non-contiguous posting lists by exploiting the internal parallelism of flashSSDs. In the experimental result, MFIS was up to 14.93, 4.04, 5.12, and 2.33 times faster than Immediate Merge, Geometric Partitioning, Hybrid, *and* SSD-aware Hybrid, respectively. Furthermore, MFIS was up to 3.78 and 2.27 times faster than contiguous In-place and non-contiguous In-place, respectively. MFIS showed up to 1.62 times higher query processing performance than non-contiguous In-place. MFIS even showed on average 1.05, 1.07, and 1.09 times faster query processing performance compared to Geometric Partitioning, Hybrid, and SSD-aware Hybrid, respectively.

Although we only consider the situation when the postings list is sorted by increasing document ID, MFIS can be extended to handle to maintain other particular orders such as term frequency and impact. In Scanning Phase, which is explained in Section 4, MFIS can be modified to read an entire posting list of the corresponding term rather than the last block. Even though this modified MFIS becomes slower than original MFIS, which only reads the last block of posting lists, it has still advantage over Merge-based strategy in terms of update performance thanks to less amount of I/O. However, since both term frequency-ordered and impact-ordered indexes have disadvantages on Boolean queries and index updates [47], documents-ordered index is preferred depending on the purpose of applications. In our experiments, we restricted to scenario that is only adding new documents without deletion and modification of previous documents, which is common restriction in this fields [9,26,29,42,43,45]. Even though this assumption does not fully reflect the reality of environment of search engines, some garbage collection policies such as [6,14] can be combined to support deletion and modification of documents.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP)(NRF-2012R1A2A1A01010775).

References

[1] AOL Query Dataset 2006. ⟨http://research.aol.com⟩ (inactive), ⟨http://www.gregsadetsky.com/aol-data/⟩ (an alternative data source link).
[2] ATA/ATAPI Command Set (ATA8-ACS). ⟨ftp://ftp.t10.org/t13/docs2004/D1699-ATA8-ACS.pdf⟩.
[3] Endurance Testing the Samsung 840 EVO SSD. ⟨http://ssdendurancetest.com/ssd-endurance-test-report/Samsung-840-EVO-120⟩.
[4] Intel SSD 520 Review. ⟨http://www.tomshardware.com/reviews/ssd-520-sandforce-review-benchmark,3124-11.html⟩.
[5] libaio. ⟨http://lse.sourceforge.net/io/aio.html⟩.
[6] S. Büttcher, C.L.A. Clarke, Indexing time vs. query time: trade-offs in dynamic information retrieval systems (Citeseer), Conference on Information and Knowledge Management: Proceedings of the Fourteenth ACM International Conference on Information and Knowledge Management (2005) 317–318.
[7] S. Büttcher, C.L.A. Clarke, A hybrid approach to index maintenance in dynamic text retrieval systems, Proceedings of the Twenty eighth European conference on Advances in Information Retrieval (2006) 229–240.
[8] S. Büttcher, C.L.A. Clarke, Hybrid index maintenance for contiguous inverted lists, Inform. Retrieval 11 (3) (2008) 175–207.
[9] S. Büttcher, C.L.A. Clarke, B. Lushman, Hybrid index maintenance for growing text collections, Proceedings of the Twenty nineth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM (2006) 356–363.
[10] B. Billerbeck, J. Zobel, Efficient query expansion with auxiliary data structures, Inform. Syst 31 (7) (2006) 573–584.
[11] Z. Cao, S. Zhou, K. Li, Y. Liu, Flashsearch: document searching in small mobile device, in: Business and Information Management, 2008. ISBIM'08, International Seminar on, IEEE (2008) 79–82.
[12] C. Carpineto, R.D. Mori, G. Romano, B. Bigi, An information-theoretic approach to automatic query expansion, ACM T. Inform. Syst 19 (1) (2001) 1–27.
[13] F. Chen, R. Lee, X. Zhang, Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing, in: High Performance Computer Architecture (HPCA), 2011 IEEE Seventeenth International Symposium on, IEEE (2011) 266–277.
[14] T.-C. Chiueh, L. Huang, Efficient real-time index updates in text retrieval systems, Technical report, Experimental Computer Systems Lab, Department of Computer Science, State University of New, Citeseer, 1999.
[15] D. Cutting, J. Pedersen, Optimization for dynamic inverted index maintenance, Proceedings of the Thirteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM (1989) 405–411.
[16] Fusion-Io, ioDrive Specification (2013). ⟨http://www.fusionio.com/load/-media-/1ufytn/docsLibrary/FIO_DS_ioDrive.pdf⟩.
[17] S. Gurajada, On-line index maintenance using horizontal partitioning, Proceedings of the Eighteenth ACM Conference on Information and Knowledge Management, ACM (2009) 435–444.
[18] Hitachi, Hitachi. Deskstart 7K1000.D. ⟨http://www.hgst.com/tech/techlib.nsf/techdocs/93DC192B5F7C2A6188257913006C2535/$file/7K1000.D_OEMspec_v1.0.pdf⟩.
[19] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, S. Zhang, Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity, Proceedings of the International Conference on Supercomputing, ACM (2011) 96–107.
[20] B. Huang, Z. Xia, Allocating inverted index into flash memory for search engines (pp), Proceedings of the Twentieth International Conference Companion on World Wide Web, ACM (2011) 61–62.
[21] J.-U. Kang, H. Jo, J.-S. Kim, J. Lee, A superblock-based flash translation layer for NAND flash memory, Proceedings of the Sixth ACM & IEEE International Conference on Embedded Software, ACM (2006) 161–170.
[22] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, Y. Cho, A space-efficient flash translation layer for compactflash systems, IEEE T. Consum. Electr. 48 (2) (2002) 366–375.

[23] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, H.-J. Song, A log buffer-based flash translation layer using fully-associative sector translation, ACM T. Embedded Comput. Syst. (TECS) 6 (3) (2007) 18.
[24] S. Lee, D. Shin, Y.-J. Kim, J. Kim, LAST: locality-aware sector translation for NAND flash memory-based storage systems, ACM SIGOPS Operating Syst. Rev 42 (6) (2008) 36–42.
[25] S.W. Lee, B. Moon, Design of flash-based DBMS: an in-page logging approach, International Conference on Management of Data: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (2007) 55–66.
[26] N. Lester, A. Moffat, J. Zobel, Fast on-line index construction by geometric partitioning, Conference on Information and Knowledge Management: Proceedings of the Fourteenth ACM International Conference on Information and Knowledge Management (2005) 776–783.
[27] N. Lester, A. Moffat, J. Zobel, Efficient online index construction for text databases, ACM T. Database Syst 33 (3) (2008) 1–33.
[28] N. Lester, J. Zobel, H. Williams, Efficient online index maintenance for contiguous inverted lists, Inform. Process. Manag 42 (4) (2006) 916–933.
[29] N. Lester, J. Zobel, H.E. Williams, In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems, Proceedings of the Twenty Seventh Australasian Conference on Computer Science-Volume 26 (2004) 15–23. (Australian Computer Society, Inc.).
[30] R. Li, X. Chen, C. Li, X. Gu, K. Wen, Efficient online index maintenance for SSD-based information retrieval systems, High Performance Computing and Communication & 2012 IEEE Nineth International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on, IEEE (2012) 262–269.
[31] Y. Li, B. He, R.J. Yang, Q. Luo, K. Yi, Tree indexing on solid state drives, P. VLDB Endow 3 (1-2) (2010) 1195–1206.
[32] Y. Lu, J. Shu, W. Zheng, S. Li, Extending the lifetime of flash-based storage through reducing write amplification from file systems (pp), Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2013) 257–270.
[33] D. Ma, J. Feng, G. Li, LazyFTL: a page-level flash translation layer optimized for NAND flash memory, Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, ACM (2011) 1–12.
[34] T.P. Martin, I.A. Macleod, J.I. Russell, K. Leese, B. Foster, A case study of caching strategies for a distributed full text retrieval system, Inform. Process. Manag 26 (2) (1990) 227–247.
[35] T.P. Martin, J.I. Russell, Data caching strategies for distributed full text retrieval systems, Inform. Syst 16 (1) (1991) 1–11.
[36] Micron, P300. ⟨http://www.micron.com/~/media/Documents/Products/Data%20Sheet/SSD/p300_2_5.pdf⟩.
[37] Ocz, Vertex3 Max IOPS. ⟨http://www.ocztechnology.com/res/manuals/OCZ_Vertex3_MAX_IOPS_Product_sheet.pdf⟩.
[38] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, J.-S. Kim, A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications, ACM T. Embedded Comput. Syst. (TECS) 7 (4) (2008) 38.
[39] S. Park, E. Seo, J.Y. Shin, S. Maeng, J. Lee, Exploiting internal parallelism of flash-based SSDs, Comput. Archit. Lett 9 (1) (2010) 9–12.
[40] G. Pass, A. Chowdhury, C. Torgeson, A picture of search (Citeseer), Proceedings of the First International Conference on Scalable Information Systems (2006) 1.
[41] H. Roh, S. Park, S. Kim, M. Shin, S.-W. Lee, B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives, P. VLDB Endow 5 (4) (2011) 286–297.
[42] W.Y. Shieh, C.P. Chung, A statistics-based approach to incrementally update inverted files, Inform. Process. Manag 41 (2) (2005) 275–288.
[43] K. Shoens, A. Tomasic, H. Garcia-Molina, Synthetic workload performance analysis of incremental updates (Springer-Verlag New York, Inc.), Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (1994) 329–338.
[44] C. Tan, B. Sheng, H. Wang, Q. Li, Microsearch: when search engines meet small devices, Proceedings of the Sixth International Conference, Pervasive Comput. (2008) 93–110.
[45] A. Tomasic, H. Garcia-Molina, K. Shoens, Incremental updates of inverted lists for text document retrieval, in: Proceedings of the 1994 ACM SIGMOD International Conference on Management of data, ACM, 1994, pp. 289–300.
[46] Wikipedia, *Static Data set*. dumps.wikimedia.org.
[47] J. Zobel, A. Moffat, Inverted files for text search engines, ACM Comput. Surv. (CSUR) 38 (2) (2006) 6.