2012 IEEE 14th International Conference on High Performance Computing and Communications

Efficient Online Index Maintenance for SSD-based Information Retrieval Systems

Ruixuan Li, Xuefan Chen, Chengzhou Li, Xiwu Gu and Kunmei Wen
Intelligent and Distributed Computing Laboratory
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan 430074, P. R. China
Email: rxli@hust.edu.cn, {xoquoxxotil, chengzhouli}@smail.hust.edu.cn, {guxiwu, kmwen}@hust.edu.cn

***Abstract*—Solid state disks (SSDs) can potentially eliminate the I/O bottleneck for many conventional applications. However, they have a very unique characteristic of erase-before-write, which probably makes existing index maintenance methods inapplicable to SSDs. In this paper, we propose Hybrid Merge, a new online index maintenance strategy for information retrieval systems, which applies SSDs instead of hard disk drives (HDDs) to store inverted indexes. We analyze the existing indexing methods through experiments, and design a new merge-based indexing method with no random writes. We try to take the full advantage of the SSD's fast random reads to overcome the defects of existing methods. Experimental results show that the proposed method improves indexing and query performance with extremely low write traffic compare to existing approaches.**

I. Introduction

Inverted Index is the core data structure for Information Retrieval (IR) systems, which are often applied to manage large-scale digital contents in the fields like libraries and e-mail services. An inverted index contains a list of terms and their posting lists. A term's posting list is a sequence of pointers. Each pointer contains the ID of the document, which containing the term and its positions in the document. To process the queries, IR systems also maintain a lexicon, which contains the mapping between the terms and the disk address of their posting lists. When a query request arrives, the IR system scans the lexicon to find the position of the posting lists for each term. Zobbel and Moffat [1] give an introduction of the inverted index based query mode.

Due to the massive size of the inverted indexes, they have to be stored on hard disk drives as inverted files. Although hard disks have the merits of high capacity and low cost, compared to the dramatic increase of CPU and memory speeds, the performance of hard disks improves slowly in the past decade. Because of their mechanic nature, this trend is unlikely to change in the nearest future. Therefore, many applications with substantial I/O operations, such as IR systems, become disk-bound. Fortunately, the Flash-based Solid State Drive (SSD) as a new storage device with no mechanical latency provides extraordinary high I/O performance and other advantages such as low power consumption and shock resistance. In fact, SSDs are currently considered to be the most possible option to replace hard disks. As a result, it is inevitable to transplant IR systems from hard disks to SSDs.

However, data access on SSDs is significantly different from that on hard disks. Existing index maintenance strategies that are designed toward hard disks may not work as efficient as before. Our experimental analyses show that these methods are no longer suitable for the SSDs in some different ways. The in-place approach keeps postings sequential on disk by random writes, which is inefficient and almost impossible to achieve on SSDs because of the Flash Translation Layer (FTL) inside. Although the merge-based approach performs all disk operation sequentially, its merge event causes heavy write traffic that could be harmful to SSDs. This is because flash memories in SSDs have a very unique characteristic of erase-before-write, which constrains the lifespan of SSDs as a single block of flash memory that can be erased only a limited number of times between 10,000 and 100,000. After those times it worn out and cannot store data anymore [2]. At present, SSDs are considered less reliable than hard disks under stressful workloads.

In this paper, we firstly analyze the existing indexing strategies on SSDs, and then propose Hybrid Merge, an SSD-based online indexing method, which follows the basic idea of merge-based update to eliminate random writes and takes advantage of the highly random access performance of SSDs to relieve write traffic on SSDs by reducing unnecessary merge operations. In Hybrid Merge approach, terms are classified into long and short based on the size of their postings. Only the long term's postings are maintained by merging as it is relatively frequent in queries. For short terms, their postings is much lesser and SSD's fast random access can still ensure the efficiency of reading them even if they are scattered. Therefore, there is no need to merge postings of short terms frequently.

The rest of this paper is organized as follows. Section 2 gives a brief introduction of SSD and index maintenance strategies. Section 3 analyzes the existing indexing strategies under the circumstance that index files are stored on SSDs. Section 4 describes our new online index maintenance approach for SSDs. Section 5 presents the evaluation results. Finally, Section 6 concludes this paper.

II. Background and Related Work

A. Flash Memory Based SSD

The data access of flash memories are quite different from hard disks. In flash memories, the basic unit of read and write is a page while that for erase operations is a block,

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.43

which consists of several pages. If the data in one page need to be modified, the total block that contains this page must be erased first. There are two major approaches to apply flash memories. One is using a system software called flash translation layer (FTL) to emulate them as hard disks by exposing an array of logic blocks to the host, so that flash memories can easily adapt to various types of file systems. Flash memory based SSDs follow this way. FTL also provides the functionalities of wear-leveling and garbage collection toward the feature of erase-before-write, which levels the erases of each blocks as evenly as possible and erases the certain blocks marked as unavailable. A survey [3] summarizes the basic FTL techniques. The second way is to directly apply flash memory without any intermediate layers. This method can be seen in flash file systems, such as YAFFS [4] and some specially designed DBMSs [5]. We focus on the first approach since the indexes in existing IR systems are organized as inverted files and managed by file systems.

Both methods have to deal with the same problem of random writes, which can be harmful to the flash memories. The random writes trigger much more data copying and erasing than sequential writes, which lead to the effect known as write-amplification [6]. Therefore, the random writes are inefficient and can further shorten the lifetime of flash memories in SSDs. Although FTL inside SSDs can lessen the impact of random writes, non-sequential workloads still trigger more writes and erases, which significantly affects the efficiency of some conventional hard disk based applications with large number of random disk accesses.

B. Different Applications on SSD

Since the unique characteristic of erase-before-write of flash memories, the most concerned for applications on flash memories or SSDs is the random write problem as we outlined above. Lee and Moon [7] analyzed the data update in DBMS and figured out that they are usually small and random, which could be quite harmful to flash memories. Then they proposed an In-page logging (IPL) method which translates many small, random updates into few large, sequential ones by maintaining some log pages inside each block where small updates are firstly recorded in. Na and Moon et al further design a B-Tree index based on IPL [8]. Here we must point out that the IPL index, along with some other flash-aware tree structured index [9] are applied on raw flash memories with no firmwares like FTL. However, in commodity SSDs, the FTL are already employed inside and SSDs can be only accessed as conventional hard disks. Therefore, in this paper, we design our indexing approach according to the features of flash memories themselves while ignoring the detailed data management of FTL inside SSDs.

On the other hand, Kim and Ramachandran [10] collected the write traces of Peer-to-Peer (P2P) downloading from various P2P file sharing programs on SSDs and found the write operations of P2P applications are extremely random. They implemented an user-level library for P2P sharing applications which changes most of the random writes into sequential ones according to the basic idea of the log-structured file system [11]. To the best of our knowledge, there are no research work concerning on the management of inverted index on SSDs for IR systems. As the existing index maintenance strategies are all based on hard disks, we believe that they could also cause the same problem as databases and P2P systems.

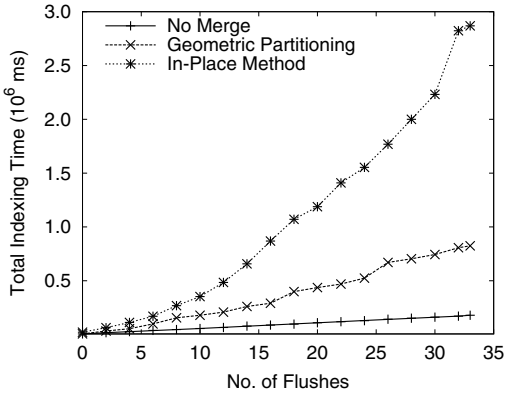
C. Online Index Maintenance

Index Maintenance techniques are categorized into offline and online. The offline approach deals with static dataset while the online one is implemented to handle dynamic document collections, which needs to update existing indexes frequently in order to serve query requests. During indexing, new documents are first parsed into terms with postings and temporarily maintained in memory, which form the in-memory index. While the memory has been exhausted, this in-memory index will be flushed into the on-disk index. There are two basic approaches to update the on-disk index: In-place method and Merge-based method.

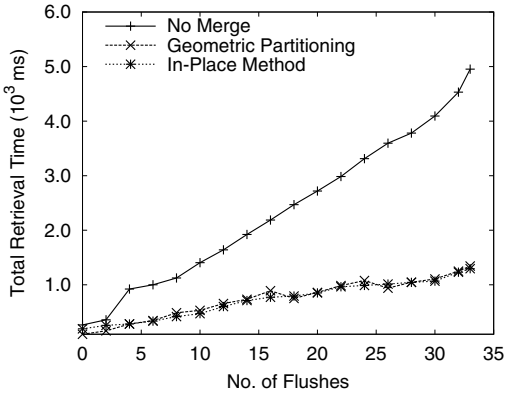
In-place Update. The in-place update approach appends the new postings of the term to the end of its existing ones on disk if there is enough space, or the combined posting list of the term is copied to a new location and a new free space is allocated to place the incoming postings. Although this over-allocation of space may wastes some disk space, it avoids the frequent relocation of the term's on-disk postings. The assignment of free space is a critical part of the in-place approaches. Cutting and Pedersen [12] keep the short lists within the vocabulary structure and the long lists on disk to save time and space. Shoens et al. [13] put the short lists in the fixed-size "bucket" structures and the long lists in the single long list area. Then predictive over-allocation strategy was also proposed [14] to further improve the performance of in-place update approach.

Merge-based Update. In merge-based update, the postings from memory are merged with existing ones and then written to a new location. The simplest merge-based approach is Re-merge (or Immediate Merge), which are described in [15]. Re-merge only maintains one single, contiguous inverted file on disk and the postings in memory are merged with it immediately. The re-merge is inefficient in index maintenance since the cost of merge event is extremely high as it is triggered whenever the in-memory index is to be written. The multiple partition strategies such as Logarithmic Merge [15] and Geometric Partitioning [16] are proposed, which maintain several sub-indexes on disk. Each sub-index contains only part of the on-disk index. Multiple partition strategies gain a higher performance of indexing as they require fewer merge event. Though they scatter the postings of the term into several sub-indexes which could lower the speed of query, multiple partition strategies maintain good balance between indexing and query performance.

Both methods keep the postings of the terms sequential on disk as much as possible in different ways. However, in general, the merge-based methods outperform the in-place methods [17] since the in-place update requires large expense



(a) Index Maintenance Performance



(b) Query Performance

Fig. 1. Performance of in-place and merge-based indexing methods. (a) Index maintenance performance. Time represents the total time taken to update in-memory index into SSD. (b) Query performance. Time represents the total time taken to complete a query request.

on free space management and updates postings by inefficient random disk operations on hard disks.

III. Analysis of Existing Index Maintenance Strategies

In this section, we analyze two types of existing index maintenance strategies respectively by experiments and then indicate why they are not suitable to the new storage devices. We use a Windows Server 2003 PC equipped with one Intel Dual E2180 2.0GHz CPU, 2GB of RAM, one 320GB sized, 7200RPM SATA hard disk and one 40GB sized Intel X25-M SSD. The document collection for experiments is collected from Wikipedia [18]. The total size of this collection is about 106GB, which includes approximately 8 million HTML documents. The maximum size of the in-memory index is set to about 100MB. Write traces are collected by DiskMon [19].

A. In-place Update Methods

The in-place update method that we use is based on the existing file systems. Since the important optimizations for the in-place indexing method, such as allocating free space for

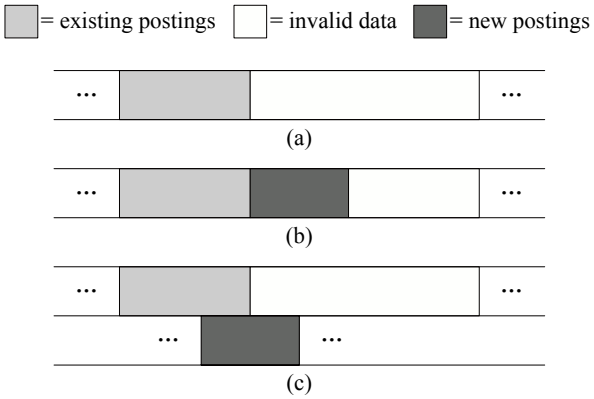


Fig. 2. In-place update on different devices. (a) represents the state of the term's on-disk postings and over-allocation is performed by writing some invalid data into disks. (b) represents an update of existing postings on hard disks. (c) represents an update of existing postings on SSD with out-of-place write, and new postings haven't been written at the end of the existing ones to avoid erases.

incoming postings, have already been used in file systems for a long time, we believe that the implementation of a modern file system is no worse than a custom implementation of the in-place index maintenance approach. Furthermore, some other research works are also based on this assumption [20]. Therefore, in our in-place update method, each term has its own inverted file to record the postings, and the postings of the term on SSD are updated by appending new ones from in-memory index to the term's existing inverted file on SSD.

The basic idea of the in-place update method is appending new postings to the existing ones, which mainly keeps the unchanged postings untouched. On hard disks, the in-place update is usually implemented by filling a disk space with the term's postings and a certain amount of invalid data. As a result, the over-allocated part could be used as a free space. During the in-place update, new postings of the term are recorded in this free space by covering these invalid data in the free space, part (a) and part (b) of Figure 2 give an illustration of the in-place update on hard disks. Obviously this procedure leads to random disk access during indexing, which is the main reason that in-place method performs indexing inefficiently on hard disks.

Moreover, since the limited erase count of flash memories, the FTL of SSDs will try to prevent write requests from modifying existing data in blocks, which further results in erasing blocks by assigning empty or other available blocks to record the updates of the existing data. This process is called out-of-place write (or relocate-on-write) [6], which makes the in-place update of postings almost impossible to achieve on SSD since the new postings of the term will actually not be written right after its existing ones. Figure 2(c) illustrates how out-of-place write can affect in-place update of postings. The extra management of in-place update by FTL also lower the speed of indexing. From Figure 1(a) we can see that comparing to the geometric partitioning the total indexing

time for in-place approach is 3.5 times longer while the query performance is nearly the same.

B. Merge-based Update Methods

Merge-based update approaches perform disk operations sequentially while indexing since this type of approaches update postings of the term by merging the existing postings with new ones and then writing them to a new inverted file on disk. However, during merge event the existing postings have been copied into memory and written to disk again. This process would generate massive size of extra writes when the document collection grows larger and larger.

To measure the total size of data that have been written to SSDs during index maintenance for the same data collection, we implement three variants of merge-based index maintenance strategies: logarithmic merge (LM), geometric partitioning (GP) and no-merge (NM) that creates a new inverted file for each flush of in-memory index. From Figure 3(a) we can see that, compared to the NM strategy, in multiple partition methods like LM and GP, the amount of writes increases exponentially while in no merge method the growth is linear. In the end, the size of data that have been written in LM and GP are about 5.0 times and 4.8 times larger than no merge approach respectively. This is also harmful to SSDs as they are less reliable than hard disks under heavy write traffic due to their limited number of erase operations. In fact, the SSD manufacturers often advise users to apply high-end SSDs in order to deal with applications that undergo intensive write traffic [21].

Meanwhile, keep postings sequential on disk by merge event are no longer extremely important since the SSD provides much faster random reads than hard disks. In fact, a large amount of merge events are unnecessary. Figure 3(b) shows that the average retrieval time of GP and NM on both hard disk drives (HDDs) and SSDs, we observe that though NM is nearly 5.9 times slower than that of GP on HDDs, it is even a bit faster on SSDs than GP on HDDs, and the gap between GP and NM on SSDs is much smaller. On SSDs GP is only 2.1 times faster than NM, which means even the postings of the term are separated into several inverted files, the efficiency of SSDs' random read still provides acceptable speed of read.

From the above analysis we can see that both in-place and merge-based update indexing methods have shortcomings while applied on SSDs. They are no longer applicable because of the unique characteristic of writes and erases of flash memories in SSDs, which is not considered in either approaches. Therefore, it is critical to design new online indexing approaches based on the features of SSDs.

IV. A Hybrid Merge Strategy for SSD-based Index Maintenance

We now present our solution to SSD-based Indexing. In our design we set two objectives:

1) Maintain balance between high index maintenance and query performance.
2) Avoid random disk access and decrease total amount of data to be written on SSD under the constraint above.

We consider these objectives based on the requirement of the IR system under dynamic environments which need to perform both indexing and query in highly-efficiency, and the unique structure of flash memories inside SSDs which makes SSDs vulnerable under stressful workloads such as index maintenance. Therefore, special treatment should be applied to SSD-based indexing method in order to prevent SSDs from worn out soon.

A. Basic Concept

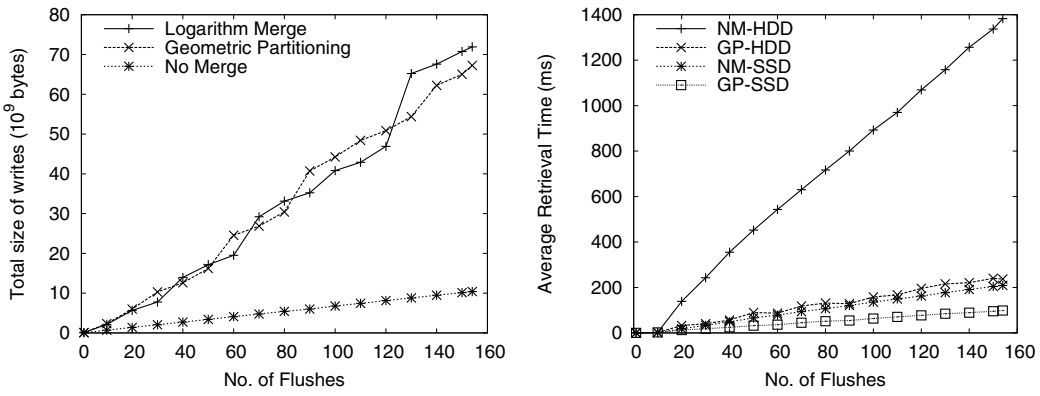
From the analysis of existing indexing approaches above, we believe that the in-place update actually can not avoid random writes on SSDs. Thus, for SSD-based index maintenance the way with merge-based update should be applied. Even though merge-based method has its own defect of producing too much writes while copying the old postings to new locations, this disadvantage could be improved based on the fact that the SSDs perform random reads much faster than hard disks.

This assumption can be verified by our experiment above. From Figure 3(b) we can see that no-merge method, once considered impractical on hard disks, has been significantly improved by the new storage devices of SSDs. Its query performance is much closer to that of multiple partition method. On hard disks no-merge is inefficient for query since it doesn't merge any on-disk inverted files at all. As a result, the postings of the term could be separated into several files which means, to fetch the entire postings of a term, hard disks have to perform the read operationss many times instead of fewer times. However, the SSDs perform random reads much faster than hard disks, which means a large part of merge event during indexing is actually unnecessary. Although on SSDs the multiple partition method still outperforms no-merge, we can combine it with other merge-based method to organize a hybrid merge method.

Therefore, in order to design an efficient index maintenance approach for SSDs we make the following decisions:

1) Avoid in-place updates to eliminate random writes during indexing.
2) Categorize terms into short and long according to the size of their postings.
3) For two types of terms we apply different merge-based methods respectively. For the postings of long terms, the indexing method is active as it triggers merge event frequently, while the way for the postings of short terms is infrequent.

The reasons of these decisions are based on two aspects. First, the access frequency of terms differs greatly that follows the Zipf-like distribution and the terms with larger size of postings are relatively frequent in queries and will continue to accumulate postings while short terms with fewer postings perform oppositely. Therefore, the speed of query processing will not be heavily affected by the speed of reading postings of short terms. Second, even if the inactive merge-based method



(a) Total Size of Writes

(b) Query Performance

Fig. 3. (a) Amount of data that have been written during the process of indexing for three merge-based methods. Y-axis represents the total size of data that written to the SSD and X-axis represents the times of in-memory index flushed to the disk. A swift growth of writes indicates a merge event happens. (b) Query processing performance for no-merge (NM) and geometric partitioning (GP) on both HDD and SSD respectively. Y-axis represents the average time taken to read the entire posting list after each memory flush.

makes the postings of short terms scattered on SSDs, the efficient random reads of SSDs still ensures the speed of fetching entire postings for the short terms as their size of postings are relatively small. On the contrary, for long terms with large number of postings, their postings are maintained by indexing methods with far more merge event to guarantee that they are stored sequentially as the speed of random reads still cannot equals that of sequential reads for massive size of data.

B. Hybrid Merge Strategy

In our proposed hybrid merge strategy, the first step is the same as existing methods. New documents are parsed into terms with their postings and placed in memory as in-memory index. When the size of in-memory index exceeds certain size we will split it into long term index and short term index first and then achieve our hybrid merge process. As for the selection of active and inactive merge-based methods, in this paper we follow a straightforward way of applying the no-merge method to maintain the postings of short terms, while for long terms we update their postings by the immediate merge method. The main reason we make this decision is that this combination is simple without costly pretreatment during index maintenance and in practice it also performs well. Figure 4 gives the overview of our hybrid merge strategy. During a flush of in-memory index postings that belong to short terms share one inverted file on SSDs and will never be merged. On the other hand, for long terms with large size of postings we create an inverted file on SSDs for each of them and use immediate merge method to maintain these files, which means, during the flush of the term's postings if there already exists an inverted file of the term, they are merged into a new inverted file immediately.

Actually, the hybrid merge can also be applied on conventional merge-based indexing methods without partitioning the in-memory index into blocks as Figure 4 shows. In other words, they can follow the old way which creates a single large inverted file for a flush of postings in memory and then apply hybrid merge. However, without the step of partitioning in-memory index, the postings of a long term could be recorded in multiple files, even all inverted files. In this case, the cost of the merge event for the term depends on the size of entire on-SSD index as Figure 5(a) shows. To solve this we break down the in-memory postings by terms and create inverted files for each long term. As a result, merge postings at cost that only depends on the size of the term's postings instead of the size of the whole index files as shown in Figure 5(b).

Our approach reduces the cost of merge event of the short term's postings in order to lower write traffic on SSDs, which also brings a high index maintenance performance as the total amount of merge events drops. Moreover, we make use of the saved cost of merge event and apply immediate merge to maintain the postings of long terms, which always keeps the postings of the long term sequential on SSDs. Therefore, the speed of fetching complete postings of the long term could be even faster than the multiple partition methods as they reduce the frequency of merge event to maintain an acceptable indexing performance as we explained before. Although the query processing of the short term's postings may be a bit slower, it doesn't affect the overall query performance since it is quite infrequent in queries.

C. Selective In-memory Postings Flush

Other than the hybrid merge, we also introduce selective postings flush when the in-memory index reaches certain size and need to be updated into on-SSD index. Instead of a complete flush of in-memory postings, during each flush of memory, we only flush the fix-sized postings and determine whether to flush postings of short terms or long terms. Selective postings flush is applied since in our hybrid merge strategy postings of the long term are maintained by immediate merge, which results in extremely high write traffic since it always

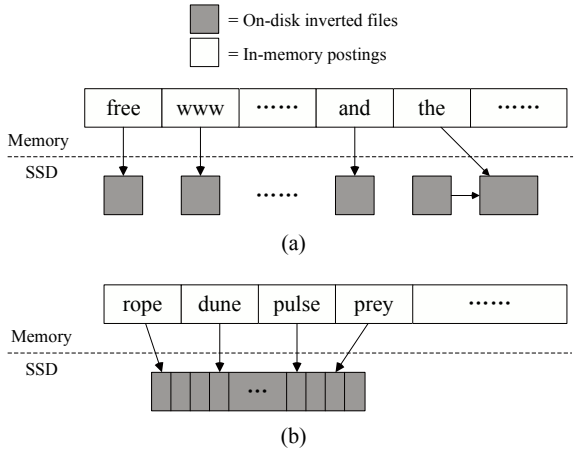


Fig. 4. (a) The posting flush of long terms. We create an inverted file for each long terms to place their postings. If an inverted file of a term already exists, it will be merged with postings in memory immediately. (b) The posting flush of short terms. During one flush of posting lists of short terms they will share one single inverted file on the disk.

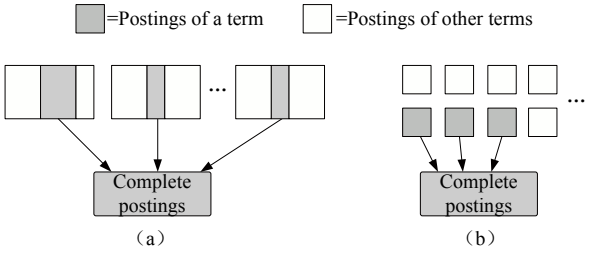


Fig. 5. (a) Fetching the entire postings of a term in existing merge-based indexing methods. (b) Fetching the entire postings of a term in hybrid merge.

triggers merge event and discard existing files. Therefore, we need some policy to delay the flush of the long term's postings which may further results in a merge event.

First we define two parameters, S_p represents the least postings size of the long term, which means if the size of a terms postings reaches S_p, it will be considered as a long term. P_f represents the least size of flushed postings for one flush from in-memory index to SSDs. In our approach, whether the terms is long or short is not determined from the beginning but during each flush of in-memory index. That is, a term may be treated as a long term in this flush. However, if in next flush its size of postings doesn't reach S_p, it will be treated as a short term. The merit of this policy is that if a term is not frequent anymore the maintenance method for its postings could change quickly, which avoid meaningless merge event.

Before the memory flush, we organize the in-memory index into two tables and sort short and long terms respectively by the size of their posting lists. After that we firstly scan the table of short terms and calculate the total size of its content. If the size of all short terms' postings can reach P_f, we begin the flush from the postings of the short term with the least sized postings, and then remove them from the short terms table afterwards. This is because the term with small sized postings

Algorithm 1 Selective Postings Flush.

Input: index in memory & on SSD
Output: updated inverted files on SSD

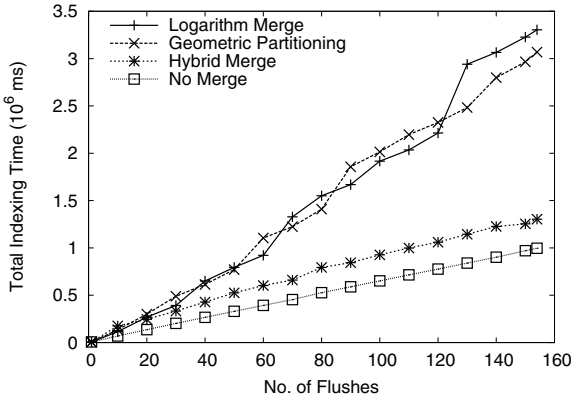
```
1: Calculate the total size of all short terms' postings
2: if The total size of all short terms' postings ⩾ P_f then
3:    Sort the short term list by the size of their postings in
      ascending order
4:    Create an new inverted file F_share
5:    while size of flushed postings < P_f do
6:       T_least = the short term with least postings
7:       Flush postings of T_least to F_share and remove T_least
         from short term list
8:    end while
9: else
10:   Sort the long term list by the size of their postings in
      descending order
11:   while size of flushed postings < P_f do
12:      T_most = the long term with most postings
13:      if T_most has a inverted file already then
14:         Merge new postings to the inverted file on disk and
            remove T_most
15:      else
16:         Create an new inverted file F_t and flush postings
            of T_most
17:         Remove T_most from long term list
18:      end if
19:   end while
20: end if
```

is less possible to scatter in many inverted files on disk. Thus, we always choose the one with the shortest posting list first. This process continues until the size of flushed postings reaches P_f, a new inverted file is created to record these data as we described above.

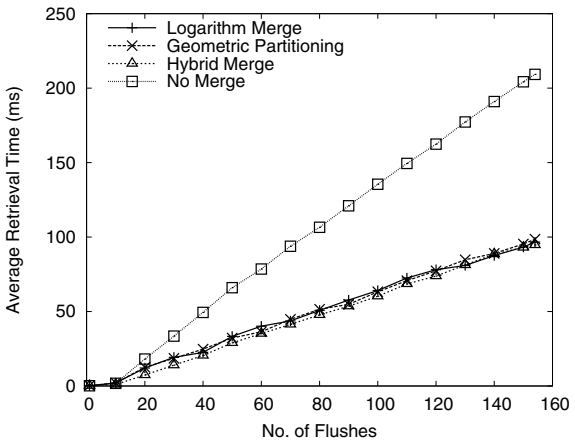
Otherwise, if the postings of all short terms are not enough for one flush, we need to flush the postings of long terms, which are distinct from the selection of short terms. Here we begin the flush from the long terms with the maximum sized postings. The difference is that in this case each long term forms an inverted file on disk, if an inverted file for a term already exists, then we merge them right away and written to a new file. The reason that we choose the one with the longest posting list is that it would take much longer for this term to been chosen to flushed again, by which can delay a merge event with significant writes as possible as we can. Algorithm 1 shows the detail procedure of selective postings flush in Hybrid Merge.

V. Performance Evaluation

The experimentation environment and dataset are the same with those described in Section 3. In these experiments, we evaluate the performance of proposed strategy of hybrid merge (HM) along with the geometric partitioning (GP), logarithmic merge (LM) and no-merge (NM). During the experiments for hybrid merge we set the parameters S_p and P_f to 1MB and



(a) Index Maintenance Performance



(b) Query Performance

Fig. 6. Performance of in-place and merge-based indexing methods. X-axis represents the times of in-memory index flush to SSDs. (a) Index maintenance performance. Time represents the total time taken to update in-memory index into SSDs. (b) Query performance. Time represents the average time taken to complete a query request.

40MB, for GP the r is set to 3. To collect the sample queries, we parse the AOL query log and then chose the sample 10000 queries.

The graphs in Figure 6(a) show that total index maintenance time of HM only about 1.3 times higher than NM, which defines the lower boundary time of index construction that creates a new inverted file for each flush. While the multiple partition method of LM and GP cost nearly 3.3 and 3.1 times higher time than NM. The proposed hybrid merge method provides an excellent indexing performance. As we apply hybrid merge, only part of the postings are involved in merge operations. While for multiple partition methods like GP, all postings have the opportunity to be updated by merging, which triggers merge events frequently. Furthermore, we apply selective postings flush in which the postings of the short term is chosen to be flushed as possible as we can. Therefore, in our method, large parts of memory flushes trigger no-merge event, which is similar to no-merge.

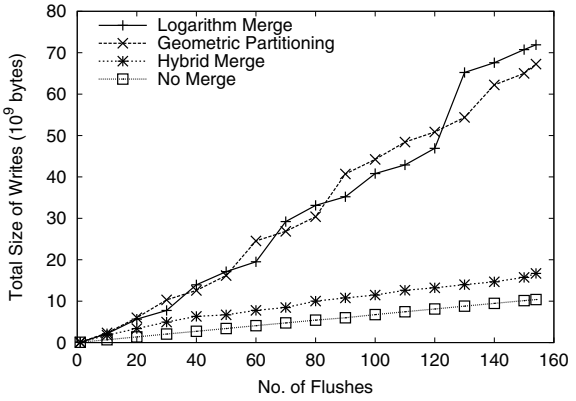


Fig. 7. Total size of data that have been written to SSD after each flush for each method.

Hybrid merge also provides good query performance. From Figure 6(b) we can see that, compared to LM and GP, HM requires only about 98.0% and 96.0% of time to complete a query request. NM have the lowest query processing speed with no doubts. The speed of query processing for HM can equal that of GP and LM. The reason lies in two aspects. First, the postings of the long term are maintained by immediate merge, which means they are always stored on SSDs sequentially, which in fact improves the speed of fetching time, while in LM and GP, they are always kept in several inverted files. Second, the size of the postings of the short term are relatively small. The SSDs can still read them in high speed even if they are scattered. Moreover, in the query log that is collected from real environment, the long terms appear more frequently than short terms. Thus, the speed of query is mainly affected by the speed of reading postings of the long terms.

Fewer merge events means lower write traffic on SSDs. In this section, we measure the quantity of writes produced by each method during indexing. From Figure 7 we can see that the write traffic of LM and GP is much heavier than NM. On the contrary, at the end of indexing HM, the size of data writes only 1.6 times larger than NM and only writes 23.2% and 24.9% of data compared to LM and GP. This also owes to the selective flush of postings, in which short terms are preferred with no-merge event triggered.

In the rest of the section, we examine the sensitivity of the query time and amount of write traffic of hybrid merge with different parameter values. From Figure 8(a) and 8(b) we can see that the query performance degrades and the total size of writes drops with the increase of S_p. This is because parameter S_p affects the categorization of terms into short or long. Bigger value of S_p leads to fewer long term, which decreases the number of merge event and further reduces write traffic, but with lower query performance. In order to make balance between these two factors we set S_p to 1MB.

On the other hand, parameter P_f determines at least how much postings are needed to be flushed. From Figure 8(c) and 8(d) we observe that the query performance drops slightly with

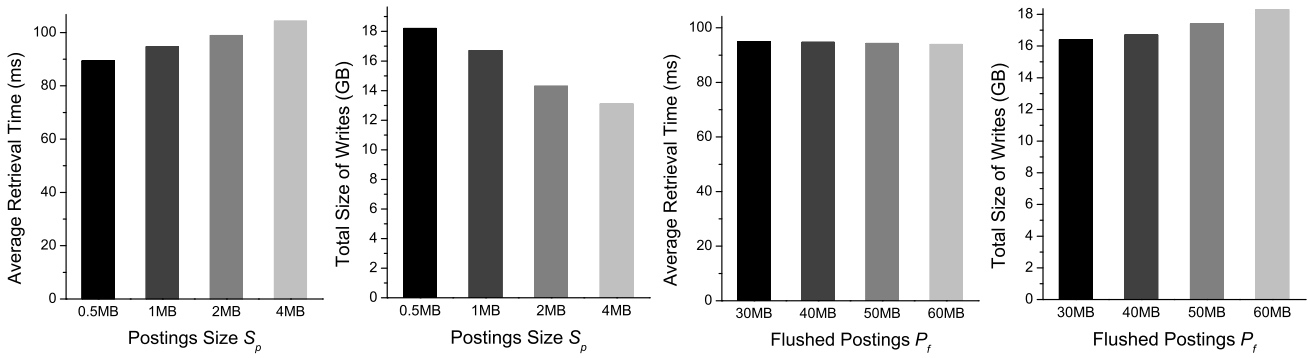


Fig. 8. Query performance and amount of write traffic of hybrid merge under different value of each parameters. (a) Query performance under different value of S_p. (b) Amount of write traffic under different value of S_p. (c) Query performance for different value under P_f (d) Amount of write traffic under different value of P_f.

the increasing value of P_f while amount of write traffic raises significantly. This owes to the fact that with more data to be written, the chance of the immediate merge event occurrence gets higher, which results in heavier write traffic. However, with small value of P_f, hybrid merge cannot get sufficient free space for new postings to accumulate, which makes the indexing inefficient. Therefore, in our experiment, the value of P_f is 40MB.

VI. Conclusions and Future Work

In this paper we first analyze the defects of existing index maintenance approaches on SSDs and then propose hybrid merge, a new index maintenance strategy towards the SSDs. In our design, we avoid in-place updates and reduce the amount of merge events based on the fast random data access feature of SSDs. Hybrid merge performs well in the experiments. Its comprehensive query performance could be similar with the multiple partition merge-based methods, while its index maintenance performance is nearly 3 times faster and also prevents the heavy write traffic which could lower the lifetime of SSDs. In our future work, we plan to test our method on various type of SSDs and apply automatic adjustment of its configuration parameters towards different datasets and SSDs.

Acknowledgments

This research is partially supported by National Natural Science Foundation of China under grants 61173170 and 60873225, Innovation Fund of Huazhong University of Science and Technology under grants 2011TS135 and 2010MS068, and CCF Opening Project of Chinese Information Processing.

References

[1] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," *ACM Comput. Surv.*, vol. 38, no. 2, pp. 6–61, 2006.
[2] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, 2005.
[3] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A Survey of Flash Translation Layer," *Journal of Systems Architecture - Embedded Systems Design*, vol. 55, no. 5-6, pp. 332–343, 2009.
[4] S. Baek, H. Park, and J. Choi, "On Improving the Reliability and Performance of the YAFFS Flash File System," *IEICE Transactions*, vol. 94-D, no. 12, pp. 2528–2532, 2011.
[5] G.-J. Kim, S.-C. Baek, H.-S. Lee, H.-D. Lee, and M. J. Joe, "LGeDBMS: A Small DBMS for Embedded System with Flash Memory," in *VLDB*, 2006, pp. 1255–1258.
[6] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-Based Solid State Drives," in *SYSTOR*, 2009, pp. 1–9.
[7] S.-W. Lee and B. Moon, "Design of Flash-Based DBMS: An In-Page Logging Approach," in *SIGMOD Conference*, 2007, pp. 55–66.
[8] G.-J. Na, B. Moon, and S.-W. Lee, "In-Page Logging B-Tree for Flash Memory," in *DASFAA*, 2009, pp. 755–758.
[9] J. Pei, M. K. M. Lau, and P. S. Yu, "TS-Trees: A Non-Alterable Search Tree Index for Trustworthy Databases on Write-Once-Read-Many (WORM) Storage," in *AINA*, 2007, pp. 54–61.
[10] H. Kim and U. Ramachandran, "FlashLite: A User-Level Library to Enhance Durability of SSD for P2P File Sharing," in *ICDCS*, 2009, pp. 534–541.
[11] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.
[12] D. R. Cutting and J. O. Pedersen, "Optimizations for Dynamic Inverted Index Maintenance," in *SIGIR*, 1990, pp. 405–411.
[13] K. A. Shoens, A. Tomasic, and H. Garcia-Molina, "Synthetic Workload Performance Analysis of Incremental Updates," in *SIGIR*, 1994, pp. 329–338.
[14] A. Tomasic, H. Garcia-Molina, and K. A. Shoens, "Incremental Updates of Inverted Lists for Text Document Retrieval," in *SIGMOD Conference*, 1994, pp. 289–300.
[15] S. Büttcher and C. L. A. Clarke, "Indexing Time vs. Query Time: Trade-offs in Dynamic Information Retrieval Systems," in *CIKM*, 2005, pp. 317–318.
[16] N. Lester, A. Moffat, and J. Zobel, "Efficient Online Index Construction for Text Databases," *ACM Trans. Database Syst.*, vol. 33, no. 3, 2008.
[17] N. Lester, J. Zobel, and H. E. Williams, "Efficient Online Index Maintenance for Contiguous Inverted Lists," *Inf. Process. Manage.*, vol. 42, no. 4, pp. 916–933, 2006.
[18] "Wikipedia Static HTML Dumps, http://static.wikipedia.org/." [Online]. Available: http://static.wikipedia.org/
[19] M. Russinovich, "DiskMon for Windows v2.01." [Online]. Available: http://technet.microsoft.com/en-us/sysinternals/bb896646/
[20] S. Büttcher and C. L. A. Clarke, "A Hybrid Approach to Index Maintenance in Dynamic Text Retrieval Systems," in *ECIR*, 2006, pp. 229–240.
[21] F. Chen, T. Luo, and X. Zhang, "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives," in *FAST*, 2011, pp. 77–90.