SAL-Hashing: A Self-Adaptive Linear Hashing Index for SSDs

Peiquan Jin, *Member, IEEE*, Chengcheng Yang, Xiaoliang Wang, Lihua Yue, and Dezhi Zhang

Abstract—Flash memory based solid state drives (SSDs) have emerged as a new alternative to replace magnetic disks due to their high performance and low power consumption. However, random writes on SSDs are much slower than SSD reads. Therefore, traditional index structures, which are designed based on the symmetrical I/O property of magnetic disks, cannot completely exert the high performance of SSDs. In this paper, we propose an SSD-optimized linear hashing index called *Self-Adaptive Linear Hashing* (*SAL-hashing*) to reduce small random-writes to SSDs that are caused by index operations. The contributions of our work are manifold. First, we propose to organize the buckets of a linear hashing index into *groups* and *sets* to facilitate coarse-grained writes and adaptivity to access patterns. A group consisting of a fixed number of buckets is proposed to transform small random writes to buckets into coarse-grained writes and in turn improve write performance of the index. A set consists of a number of groups, and we propose to employ different split strategies for each set. With this mechanism, *SAL-hashing* is able to adapt to the changes of access patterns. Second, we attach a log region to each set, and amortize the cost of reads and writes by committing updates to the log region in batch. Third, in order to reduce search cost, each log region is equipped with Bloom filters to index update logs. We devise a cost-based online algorithm to adaptively merge the log region with the corresponding set when the set becomes search-intensive. Fourth, we propose a new technique called virtual split to optimize the search performance of *SAL-hashing*. Finally, we propose a new scheme for the management of the log buffer. We conduct extensive experiments on real SSDs. The results suggest that our proposal is self-adaptive according to the change of access patterns, and outperforms several competitors under various workloads.

Index Terms—Linear hashing, solid state drives, self-adaptive

1 Introduction

Hash indexes have high efficiency for point queries owing to the constant search cost of the hashing techniques. Compared with static hashing, dynamic hashing is more suitable for database indexes because it can adjust its hash-table size according to the data size [1]. Dynamic hashing has two kinds of implementations: extendible hashing and linear hashing. Extendible hashing doubles the bucket directory in case of a bucket overflow, while linear hashing increases its bucket number linearly. Linear hashing offers a better trade-off between space efficiency and performance, and thus has been commonly used in database systems.

Traditional indexes are commonly designed for magnetic disks or main memory, and are not optimized for flash memory based solid state drives (SSDs) [2], [3], [4], [5]. To adapt hash indexes to SSDs, various hash index structures [2], [6], [7], [8], [9] have been proposed. However, they have the following problems:

(1) Existing approaches are mainly designed for reducing random writes to SSDs, and thus are suitable for update-intensive access patterns. However, they cannot adapt to the changes of access patterns. As access patterns in real applications are very likely to change with time, it is necessary to devise self-adaptive hash indexes that can provide high efficiency for both update and search.

(2) Existing SSD-aware hash indexes do not take into account the internal parallelism of modern SSDs [10], [11], which means that one I/O operation with a large granularity of consecutive logical pages is usually divided into several simultaneous read/write operations to the internal flash-memory chips. We have found that this mechanism has a significant impact on the overall bandwidth of SSDs [12]. Thus, it is helpful to integrate the internal parallelism of SSDs with hash indexes.

To address these problems, we propose a new hash index for SSDs called *Self-Adaptive Linear Hashing* (*SAL-hashing*). The key idea of *SAL-hashing* is to (1) adapt its structure according to the change of access patterns, and (2) utilize the internal parallelisim of SSDs to improve the overall performance. *SAL-hashing* aims to achieve high update performance by reducing inferior small random writes and prevent degradation of search performance at the same time. For this purpose, we propose a complete set of techniques, including new data structures and algorithms, to make the hashing index more efficient for SSDs.

• *The authors are with the School of Computer Science and Technology, Key Laboratory of Electromagnetic Space Information, University of Science and Technology of China, Chinese Academy of Sciences, 96 Jinzhai Road, Hefei 230027, China.*
E-mail: {jpq, ychengc, wxl147, zdz}@mail.ustc.edu.cn, llyue@ustc.edu.cn.

Manuscript received 12 June 2017; revised 7 Aug. 2018; accepted 19 Nov. 2018. Date of publication 6 Dec. 2018; date of current version 4 Feb. 2020.
(Corresponding author: Peiquan Jin.)
Recommended for acceptance by W. Wang.
Digital Object Identifier no. 10.1109/TKDE.2018.2884714

Overall, the technical contributions of this paper fall into the following aspects:

(1) We propose to group small random writes caused by split operations of the index into coarse-grained writes in the *SAL-hashing* index. For this purpose, we define a new terminology called *group*. A group is a fixed number of buckets, and small random writes to SSDs are transformed into group-based writes. This design can well utilize the internal parallelism of SSDs and we demonstrate that it is able to improve the write performance of *SAL-hashing*.

(2) We propose to organize groups in the *SAL-hashing* index into *sets*. A set consists of several groups. The key point is that each set has different access patterns. Thus, we can employ different split strategies for each set, e.g., lazy-split for update-intensive sets and eager-split for search-intensive sets, to make *SAL-hashing* adaptive to access patterns.

(3) We devise an online cost-based algorithm to dynamically determine whether a set is search-intensive. We prove that this problem is similar to the Ski-Rental problem. We further present a randomized algorithm to determine the access pattern of a set. The correctness and efficiency of the algorithm is ensured by a theoretical analysis.

(4) We theoretically analyze the relationship between the number of overflow pages and the split factor in the *SAL-hashing* index. Based on the analysis, we propose a memory efficient structure to reduce search costs on the *SAL-hashing* index, particularly on overflow buckets. As a result, we demonstrate that every search only causes one page read.

(5) We conduct extensive experiments on two real SSDs, and the results suggest the efficiency of our proposal.

This paper substantially extends our previous work [12]. First, we clarify the research problems and the technical contributions of the paper (Section 1). Second, we make a new literature survey and update the related work (Section 2). Third, we improve the presentation of the motivation and idea of group and set (Section 3.1), and propose a memory efficient structure to reduce search costs on the *SAL-hashing* index (Section 3.4). Fourth, we devise a new scheme for the log-buffer management in *SAL-hashing* (Section 3.5). Fifth, we extend the *SAL-hashing* index to support transactions and crash recovery (Section 3.6). Finally, we conduct extensive experiments to evaluate our proposal (Section 4).

2 Related Work

In recent years, to adapt hash indexes to flash memory based SSDs, various hash index structures have been proposed, and many of them are based on the extendible hashing and the linear hashing. While some studies are based on the extendible hashing [6], [7], [9], the drawbacks of the extendible hashing [1], such as low space utilization and high updating cost for doubling the bucket directory, make it hard to achieve balance between performance and space efficiency.

Therefore, some researchers proposed to optimize the linear hashing for flash memory. For example, the Lazy-Split hash [2] was proposed to optimize the linear hashing by employing a lazy split strategy to reduce intermediate writes to flash memory. But it has poor performance when processing search-intensive workloads. Generally, we need to make a tradeoff between search efficiency and update efficiency when access patterns change with time.

Another problem of existing flash memory based hash indexes is that they do not take into account the internal parallelism of SSDs [10], [11]. Modern SSDs often employ parallel controls to maximize read/write speeds of devices. Our previous study [12] showed that a larger logical-page size leads to a higher I/O bandwidth of the SSD, because more parallel internal reads or writes are incured inside the SSD. Thus, we can see that this parallelism is helpful to improve the overall performance of SSDs.

The *SAL-hashing* proposed in this paper has benefitted from the delta-log based technique [4], [13], [14], [15], [16], [17], which is to enable appending delta-logs for recording updates, and then are merged together later. This technique has been an effective compression of pages across many versions, and can avoid redundantly storing the unchanged portion of the page for every single update [16]. BFTL [4] handles updates to B-trees over the FTL (flash translation layer). It has a node translation table to separate the logical and physical location of a page. It also has a reservation buffer for delta-logs that accumulate recent updates. When the reservation buffer is full, it has a process to store deltas continuously on flash memory. Accordingly, page write cost is amortized over multiple updates. But, on the other hand, searching a node needs to first inspect the node translation table, and then read a series of pages to reconstruct a node. The Bw-tree [14] is similar to the BFTL, but differs in concurrency control and management of the storage layer. The Bw-tree achieves high performance via a latch-free approach, which is implemented by using the atomic compare and swap (CAS) instruction. Moreover, to ensure data persistence, it specifically designs a log structured store [15], which blurs the distinction between a page and a record store to facilitate storage efficiency and fast sequential writes. A major disadvantage of the Bw-Tree is that each search on a page has to traverse the delta chain. A record-level indirection technique [16], which distinguishes between a physical row-identifier (RID) and a logical record identifier (LID), is proposed to avoid updating indexes on unaffected attributes in multi-version databases. The indirection level is persistent on SSDs with much higher IOPS than hard disks, and can dramatically reduce the amount of disk I/Os that are needed for index updates. The In-Place Appends (IPA) approach [17] makes use of the low-level Incremental Step Pulse Programming (ISPP) technique to performe in-place appends on the original flash pages. Compared with the IPL storage model [13], IPA does not need to read additional pages to reconstruct the up-to-date page. However, IPA cannot work directly on operating systems and SSDs.

3 Self-adaptive Linear Hashing

In this section, we present the design of *SAL-hashing*. The notations used throughout the paper are summarized in Table 1.

3.1 Group and Set

The *SAL-hashing* index is based on two new terminologies called *group* and *set*.

TABLE 1
Notations Used in This Paper

Notation	Description
w	Average cost of writing a flash page
w/n	Average cost of a coarse-grained flash-page write
lf	Current load factor
l	Threshold (upper bound) of the load factor
B	Capacity of a bucket page
k_n	Count of indexed keys
b	Count of buckets
g	Count of fixed buckets contained in a group
z	Page size (bytes)
r	Record size (bytes)
m	Log buffer size (pages)
f	False positive probability of Bloom filters
s	Current count of log pages in the log region
d	Max count of log pages in the log region
sc	Count of primary buckets to be split in a split cycle
sf	The proportion of split primary buckets
$bucket_j$	The jth bucket
$group_i$	The ith group
$set_{k,c}$	The kth set, c is the count of bits used to generate the set
$Bit(x)$	Function to compute the bit number of the binary value of x
$h(x,j)$	Function to get the last j bits of x
$h(K)$	Function to get the hash value of key K, and $h(K) < b$

Group. Basically, hash indexes need to be SSD-friendly when they are implemented for SSDs, meaning that they need to consider the intrinsic properties of SSDs. This is similar with designing cache-friendly hash indexes [18] for in-memory databases. In the traditional linear hash index, a bucket typically consists of one logical page, and each write to SSDs is actually a page write. However, this is not friendly to SSDs. Our previous work [12] shows, the internal parallelism of modern SSDs has a significant impact on the overall performance of SSDs. More specifically, we found that a larger granularity of I/O leads to a higher bandwidth of SSDs. Thus, in order to reduce small random writes to SSDs, it is beneficial to put small random writes into a group, and write the group to SSDs by coarse-grained writes. With this mechanism, we can reduce the number of small random writes to SSDs and in turn improve the overall performance of the hashing index on SSDs. This leads to the idea of group in the *SAL-hashing* index.

Definition 1 (group). *A group consists of a fixed number of buckets in the linear hash index.*

When performing coarse-grained writes, we assume that the granularity of coarse-grained writes is g. For the convenience of performing coarse-grained writes, we let g be the power of 2, e.g., 8, 16 or 32. Then, we define the *i*th group $group_i$ by Equation (1).

$$group_i = \left\{bucket_j | \frac{j}{g} = i, 0 \le j \le b\right\}. \tag{1}$$

In summary, the introduction of group in the *SAL-hashing* index is to utilize the internal parallelism of SSDs to transform a great number of small random page-writes incurred by split operations into fewer coarse-grained writes. Thus, *SAL-hashing* is expected to be SSD-friendly.

Set. The introduction of the terminology *set* is to realize the self-adaptivity of the *SAL-hashing* index. As access patterns may change with time, and more importantly, different data pages may have different access patterns. Thus, in order to make the *SAL-hashing* index adaptive to access patterns, we propose to dynamically classify data into different sets. Then, we use different split strategies for sets. For example, for an update-intensive set, we can use a lazy-split strategy [2], while for a search-intensive set, we can use an eager-split strategy. Since we have organized the buckets into groups, we define a *set* as a set of groups.

Definition 2 (set). *A set consists of several associated groups.*

All the groups in the *SAL-hashing* index are partitioned into sets. Let c be the count of bits, the kth set in the hash index, namely $set_{k,c}$ is defined by

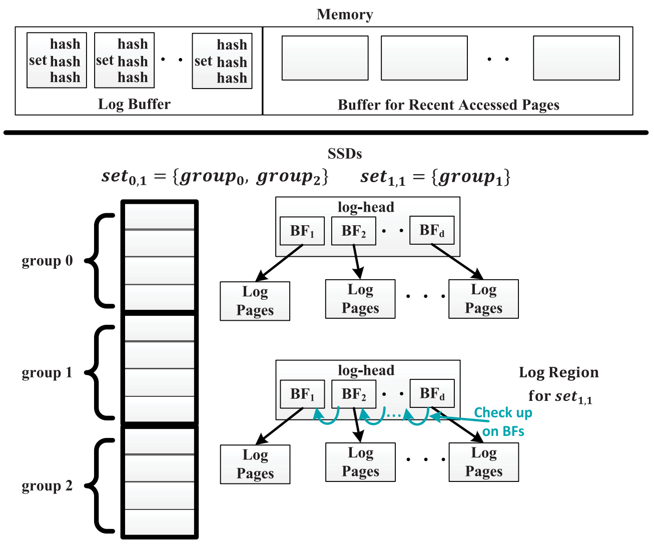
$$set_{k,c} = \left\{group_i | i \in \left\{x | x\%2^c = k, k \le x \le \frac{b-1}{g}\right\}\right\}. \tag{2}$$

The association between *groups* and *sets* is determined by parameter k and c. The insert procedure in Section 3.2 explains how to classify groups into sets, and also describes how to set *k* and *c*.

Partitioning all groups into sets has two aspects of advantages. First, different sets have different access patterns, so we can employ specific split strategies for sets. For each set, all the split pages are redistributed inside the set, because these pages are supposed to have the same access pattern. This design can make *SAL-hashing* adaptive to access patterns. Second, this design is helpful for improving the efficiency of piggybacking updates. Piggybacking update, i.e., grouping multiple write operations as one using a log buffer, has been widely used to amortize write costs on flash-optimized tree indexes. However, this strategy is not suitable for flash-optimized hash indexes because keys in a hash index are randomly distributed in buckets. The introduction of set is able to solve this problem. We gather all updates of each set and flush them later. For example, let the group size $g = 16$, if a write-intensive set contains 3 groups, all updates to 48 buckets are gathered and flushed together. Thus, the amortized read/write cost of each update operation is 1/48 of the strategy that gathers and flushes updates within each bucket. The detailed cost analysis can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2018.2884714.

3.2 Overview of *SAL-Hashing*

Fig. 1 shows the structure of *SAL-hashing*. *SAL-hashing* consists of one structure residing in main memory and another structure in SSDs. A fixed number of adjacent buckets are assembled as a group. Moreover, groups are classified into sets. The log buffer is used to cache data updates caused by insertions, deletions, and update operations. When the log buffer is full, some selected elements in the log buffer are flushed in batch to the log region of the associated set, which can reduce random writes to the log region in SSDs.



Memory
hash set hash hash
Log Buffer
Buffer for Recent Accessed Pages
SSDs
$set_{0,1} = \{group_0, group_2\}$
$set_{1,1} = \{group_1\}$
group 0
group 1
group 2
log-head
BF_1 BF_2 BF_d
Log Pages
Log Region for $set_{1,1}$
Check up on BFs

Fig. 1. Overview of *SAL-hashing*.

Each log region of a set is equipped with a *log-head*, which maintains Bloom filters of all the log pages.

Log Buffer. Piggybacking update, i.e., grouping multiple write operations as one using a log buffer, has been widely used to amortize write costs on flash-optimized tree indexes [4], [19], [20]. However, this strategy is not used in flash-optimized hash indexes since keys are randomly distributed in all buckets. We adopt the piggybacking update strategy because we need to gather updates to one set. The piggybacking update strategy in terms of the set granularity helps to improve the efficiency of caching compared with that in terms of the bucket granularity. A theoretical analysis of the piggybacking update strategy is presented in Appendix A, available in the online supplemental material. Each time an update operation arrives at the log buffer, we just add the operation log to the list of its corresponding set. In addition, we divide the list of each set into several smaller lists using a simple hashing technique. Through this way, we can improve the performance of searching the logs associated with a set in the log buffer.

Log Region. The log region of a set consists of a *log-head* page and several log pages. Each time after allocating 2^i bucket pages, $\frac{2^i}{g}$ pages are reserved as *log-head* pages, which are contiguously stored after the bucket pages. The *log-head* page maintains Bloom filters [21] and pointers to all the log pages. The operation logs from the log buffer are written to the log pages in an append-only mode. After writing the log pages, the corresponding Bloom filters in the *log-head* are also updated. When searching the log region to evaluate a point query, the Bloom filters are checked in a reverse order. We find out that the false positives in Bloom filters have limited effects on page reads if we set a small false-positive probability. The detailed analysis is put in Appendix A, available in the online supplemental material.

Insertion. The insert operation is described in Algorithm 1. When lf exceeds l, we add a new group which contains g buckets. After inserting the ⟨key, value, i⟩ log into the log buffer, if the log buffer is full, we flush all the operation logs of the selected victim set in batch to the log region on SSDs.

In order to classify groups into sets, we first classify all groups into 3 categories: namely *empty-groups*, *lazy-split-groups*, and *split-groups* [12]. The newly-added group is considered as an empty-group. During the inception phase, there are only two groups, namely $group_0$ and $group_1$. Both of them are regarded as *split-groups*. In addition, if a set with multiple groups is split and therefore all the key-values among the groups are redistributed in the set, then all these groups are transformed into split-groups. When a group is added, a routine *AddGroup()* is invoked to put it into its corresponding set. As shown in Algorithm 2, a split-group will be transformed into a lazy-split-group if we do not redistribute key-value pairs in the split-group to the newly added group immediately. In our implementation, each group has a metadata of two bytes: two bits describe the category of the group, five bits make up the *bitnum* field used in Algorithm 2, and eight bits record the number of lookups in the log region of the corresponding set. In addition, we reserve a bit for each group to indicate whether a log region is attached to its corresponding set. Only lazy-split-groups and split-groups use the reserved bit and the eight bits recording lookups.

Algorithm 1. *Insert*

```
   Input: ⟨key, value⟩: the key and value to be inserted;
 1 $k_n$ ++;
 2 if $lf > l$ then
 3     $gr \leftarrow b/g$;
 4     AddGroup(gr); /* Add a new group                  */
 5     $b \leftarrow b + g$;
 6 end
 7 $gr \leftarrow h(key)/g$;
 8 $s \leftarrow$ find the corresponding set of $gr$;
 9 Add an operation log ⟨key, value, i⟩ to the list of the set $s$;
10 if the log buffer is full then
11     Select a set with maximum operation logs as a victim;
12     Flush the logs of the set to the log region;
13 end
```

Algorithm 2. *AddGroup*

```
   Input: gr: the group which is ready to be added;
 1 $bit\_number \leftarrow Bit(gr) - 1$;
 2 $cor\_group \leftarrow gr \% 2^{bit\_number}$;
 3 if cor_group is a split-group then
 4     Transform cor_group from a split-group into a lazy-split-
       group ;
 5     $cor\_group.bitnum \leftarrow bit\_number$ ;
 6     $gr.bitnum \leftarrow bit\_number$ ;
 7 else /* cor_group is a lazy-split-group or an
   empty-group                                          */
 8     $gr.bitnum \leftarrow cor\_group.bitnum$ ;
 9 end
10 Set gr as an empty-group ;
```

Based on the three kinds of groups, we can infer that there are two types of sets in *SAL-hashing*: (1) *lazy-split-set* consists of a lazy-split-group and one or more empty-groups; (2) *split-set* consists of only one split-group. There are two parameters in our set definition, namely k and c. For a split-set, k equals the group number of its split-group, and c is a meaningless parameter. For a lazy-split-set, k equals the group number of the lazy-split-group contained in it, and c equals the value stored in the lazy-split-group's *bitnum* field. When a lazy-split-set is split, we can get all the groups through Equation (2), and redistribute key-value pairs among these groups. During the split process, we always conduct coarse-grained writes to write groups to SSDs. Through this way, we can avoid small random writes caused by split operations. After a split operation is completed, all the groups in the set are marked as split-groups, meaning that a lazy-split-set is split into multiple split-sets.

Deletion and Update. The deletion and update are similar to the insertion. We just insert an operation log in the log buffer, and flush it to the log region in batch later.

Search. *SAL-hashing* has a different searching process compared with the linear hashing. *SAL-hashing* maintains all key-value pairs in three sites: log buffer, log regions, and buckets, which are evaluated in order during a search process. Once we get a result in one place, the search process is interrupted. This is because all update operations are always first put in the log buffer, and then flushed to a log region, and finally merged to the buckets on SSDs. If we find a related log in the log region, two pages reads are necessary (one for the *log-head*, and the other for the log page containing the related log). Otherwise, we have to perform at least two pages reads, because we not only read the *log-head* page to determine whether there is a related log in the log region, but also need to search a bucket.

Merge Operation. When a log region is full or the online algorithm (see Section 3.3) decides to merge the log region to buckets, a merge operation is invoked. During the merge procedure, not only the update logs in the log region, but also the update logs in the log buffer (if exist), are merged together. In addition, when merging the log region in a lazy-split-set, a split operation is in passing. This means that we do not merge all the update logs to the set's lazy-split-group, but merge these logs and redistribute key-value pairs in the lazy-split-group to the proper groups according to the up-to-date $h(key)$ function. Through this way, the write cost is amortized by a merge operation and a split operation. Since all the bucket pages in each group are continuously stored, coarse-grained writes are applied in the procedure of merge and split to avoid small random writes.

3.3 Self-Adaptive Method

In this section, we present a cost-based online algorithm that dynamically determines whether to merge the log region to buckets.

In Appendix A, available in the online supplemental material, we will show that the expected additional page reads (denoted as *read-penalty*) are $1 + s \cdot f$, if we have to search the log region. As shown in Table 1, s represents the count of the log pages in the log region, i.e., the number of valid Bloom filters in the *log-head*, and f is the false positive probability. Thus, each time a log page is added to the log region (s increase by one), the *read-penalty* increases by f. On the other side, we have to read one more page to perform the merge operation, i.e., our *merge-cost* increase by one. Since f is far less than 1, the *merge-cost* to *read-penalty* ratio increases monotonically along with the parameter s. The problem scenario of whether to merge the log region to buckets is similar to the *ski-rental* problem [22], [23], [24]. The only difference is that the *rent* to *buy* ratio keeps constant in the *ski-rental* problem.

The *ski-rental* problem can be formally described as follows. Assume that renting skis costs 1 per day and buying skis costs p units. Every day $x \geq 1$ you have to decide, in an online fashion, whether you will continue renting skis for one more day or buy a pair of skis. One would like to apply an optimal online algorithm to minimize the cost of skiing.

Studies on the *ski-rental* problem can be divided into two classes: deterministic and randomized algorithms. Competitive analysis, which aims at comparing the performance of on-line algorithms with that of an optimal off-line algorithm, is usually used to evaluate the expense of algorithms. The break-even [22] algorithm has been proven to be the best deterministic algorithm with a strong competitive ratio of 2. It advises you to rent for $p-1$ days, and buy skis on the morning of day p if you still intend to ski. However, randomized algorithms can do better than deterministic algorithms. The main idea is to flip coins to make decisions. In the morning of day i, the probability of buying skis is π_i. Karlin et al. [24] first presented this algorithm with distribution π_i, as shown in Equation 3. It achieves strongly competitive ratios approaching $\frac{e}{e-1}$, and no other randomized algorithm can do better than it.

$$\pi_i(p) = \begin{cases} \frac{1}{p\cdot\left(\left(1+\frac{1}{p}\right)^p-1\right)} \cdot \left(\frac{p+1}{p}\right)^{(i-1)} & (i = 1 \ldots p) \\ 0 & (otherwise) \end{cases}. \quad (3)$$

Though the problem scenario of our proposal has a bit difference from the *ski-rental* problem, we find that the randomized algorithm designed for the *ski-rental* problem is applicable to the problem scenario of ours. This implies that each time before we search the log region of a set, we decide to merge the log region to buckets at a certain probability, which is determined by the number of lookups occurred in the log region. Next, we prove the feasibility of our findings. The detailed proof can be found in our previous work [12].

Lemma 1. *$\pi_i(p)$ is a decreasing function of p $(p > 1)$.*

Theorem 1. *The randomized algorithm designed for the ski-rental problem is applicable to the problem scenario of ours.*

3.4 Improving Search Performance of SAL-Hashing

Probability of Overflow. The linear hash index splits its primary buckets in a round-robin mode: first bucket 0, then bucket 1, and so on. During a split cycle, a split point indicates which bucket is the next to be split. A full kth ($k \geq 1$) split cycle splits all the 2^k primary buckets, and then the number of total buckets doubles. When the kth split cycle ends up, the split point is set to zero and the $(k+1)$th split cycle begins. Given b buckets, the number of primary buckets to be split (denoted as sc in Table 1) is a constant value of $2^{\lfloor log_2(b-1)\rfloor}$ during a split cycle.

We assume that keys are uniformly and independently hashed, and the number of records in each bucket follows a Poisson distribution. Then given a bucket with μ expected records, the probability of overflow is as follows:

$$P(\mu) = \sum_{i=B+1}^{k_n} \frac{\mu^i}{i!} \cdot e^{-\mu} \approx 1 - \sum_{i=0}^{B} \frac{\mu^i}{i!} \cdot e^{-\mu}. \quad (4)$$

The value of μ is closely related to the split factor sf, which refers to the proportion of primary buckets having

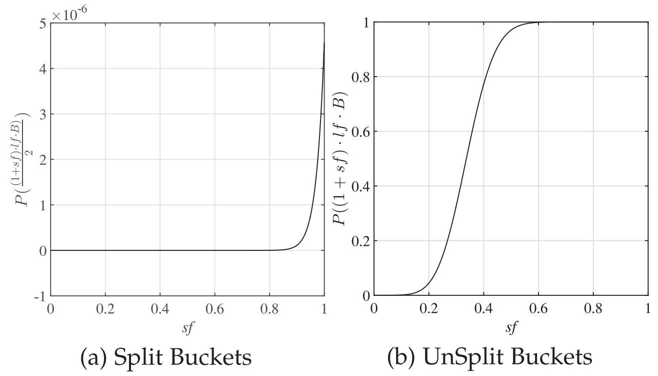


Fig. 2. Probability of overflows versus sf.

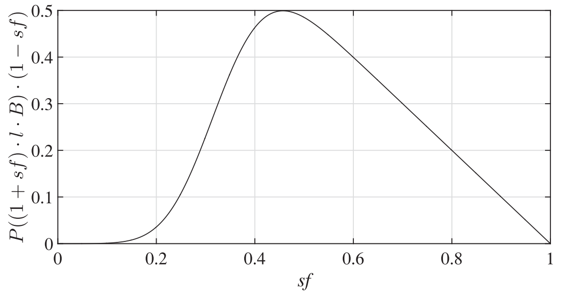


Fig. 3. Probability of a record residing in a bucket with overflow pages.

been split. The expected number of records in an unsplit bucket is $(1 + sf) \cdot lf \cdot B$, while for a split bucket, the value is $\frac{(1+sf)\cdot lf\cdot B}{2}$. Given a deterministic page size z and record size r, we can get the capacity B of a bucket page, namely $B = \frac{z}{r}$. Fig. 2 shows the correlation between $P(\mu)$ and sf with $z = 4096(bytes)$, $r = 16(bytes)$, and $lf \approx l = 0.75$. The unsplit buckets are more likely to overflow as sf increases, and the probability approaches 1 after sf exceeds 0.5. On the other hand, the split buckets almost never overflow regardless of sf. Then, given a record, the probability it resides in a bucket with overflow pages is approximately equal to $P((1 + sf) \cdot l \cdot B) \cdot (1 - sf)$. As shown in Fig. 3, nearly 50 percent of the searches hit in overflowed buckets in the worst case. In addition, searching these buckets needs to access the overflow pages, especially for unsuccessful searches, which results in the degradation of search performance.

Virtual Splits for Improving Search Performance. To improve the search performance, we focus on optimizing searches in unsplit buckets since split buckets are not likely to overflow. We aim to reduce search costs to one page-read when searching unsplit buckets.

When an unsplit $bucket_j$ overflows, an intuitive way is to split it and redistribute records in $bucket_j$ and $bucket_{j+sc}$. However, the bucket number of $bucket_{j+sc}$ exceeds the maximal range $(b - 1)$ of the *SAL-hashing* index; thus its physical storage cannot be allocated. To address this problem, we propose a technique called *virtual split*. During a virtual split of $bucket_j$, we allocate an overflow page for $bucket_j$ and take this overflow page as $bucket_{j+sc}$. Then, we redistribute records to $bucket_j$ and $bucket_{j+sc}$. After that, if we maintain the record-bucket index in memory for $bucket_{j+sc}$, we can ensure that each access to the records in $bucket_{j+sc}$ only costs one page read. Here, the key idea is that $bucket_j$ and $bucket_{j+sc}$ will keep unfull during the split cycle.

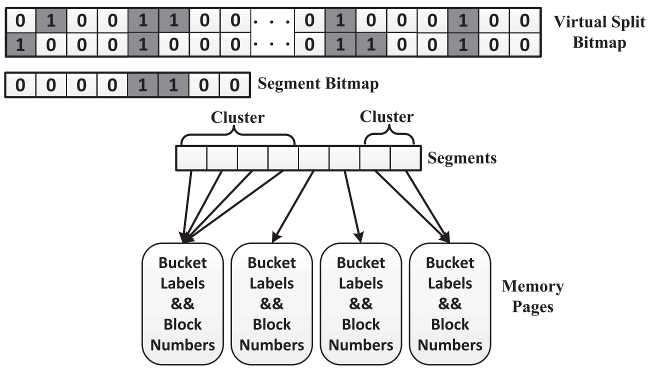


Fig. 4. Memory efficient structure to index overflow pages.

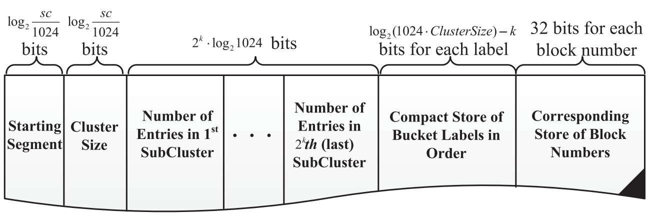

$\log_2 \frac{sc}{1024}$ bits
$\log_2 \frac{sc}{1024}$ bits
$2^k \cdot \log_2 1024$ bits
$\log_2(1024\cdot ClusterSize)-k$ bits for each label
32 bits for each block number
Starting Segment
Cluster Size
Number of Entries in 1st SubCluster
Number of Entries in 2^kth (last) SubCluster
Compact Store of Bucket Labels in Order
Corresponding Store of Block Numbers

Fig. 5. Data organization of a shared memory page.

For maintaining the record-bucket index in memory, we propose a memory-efficient bitmap structure to index overflow pages.

As Fig. 4 shows, the virtual split bitmap marks whether an unsplit bucket overflows. We note that each unsplit bucket has different overflow pages, and more importantly there are few overflow pages for unsplit buckets at the beginning of index construction. Thus, for reducing memory costs, we partition the unsplit buckets into *segments*, and each segment consists of a few unsplit buckets. We let several segments (called a *cluster*) share one memory page to store index entries for the virtual splits in the segments. Here, the key point is that there is only one memory page associated with a segment to store the virtual splits; thus we can ensure that only one memory-page-read is needed when searching a segment. With the increasing of the split factor as well as the number of virtual splits, clusters are split to ensure that one segment is associated with one memory page at most. The segment bitmap in Fig. 4 marks whether a segment occupies a memory page exclusively.

Data Organizaiton of a Share Memory Page. Fig. 5 shows the compact data organization of a shared memory page occupied by a cluster. The *starting segment* indicates the first segment of the cluster. Note that as cluster size becomes smaller and smaller due to the splits of memory pages, less bits are needed to represent a bucket label in that cluster. Moreover, to improve the memory utilization and store more index entries in a memory page, the cluster is further divided into 2^k $(k \geq 1)$ sub-clusters, and then the bits required to represent a bucket label is reduced by k.

In general, memory pages managed by OS is 4KB, and 4 bytes (32 bits) are adequate for representing the block number of an index page, then a memory page can at most store 1024 index entries for consecutive buckets. Therefore, we set the size of a segment to 1024, and when a segment occupies a memory page exclusively, there is no need to store the bucket label, we can just regard the index page as an array. Given the parameters above, the capacity of a shared memory page is:

$$B(k) = \frac{z \cdot 8 - 2 \cdot log_2 \frac{sc}{1024} - 2^k \cdot 10}{log_2(1024 \cdot ClusterSize) + 32 - k}. \tag{5}$$

Furthermore, we can get Lemma 2.

Lemma 2. *$B(k)$ monotonically increases when $k \leq 6$, and monotonically decreases when $k \geq 7$.*

On the other hand, when locating an index entry, we need to first check $\frac{2^k+1}{2}$ counts of entries in each sub-cluster on average. Therefore, a large k leads to more memory accesses and CPU cycles. In our implementation, we set k to 3 to make a tradeoff between the memory utilization and the memory access.

To sum up, we propose the virtual split technique to improve search performance of *SAL-hashing*. We focus on reducing page reads incurred by overflow buckets. This is implemented by introducing memory-efficient bitmaps for maintaining virtual splits for unsplit buckets, which are organized as segments and clusters. Through the split of clusters, we ensure that each segment has at most one memory page storing the virtual splits. Thus, when searching on unsplit buckets that are very likely to have overflow pages, we need only one page read since the addresses of all overflow pages have been maintained in memory pages.

3.5 Log Buffer Management

In order to further reduce writes to SSD, in this section we explore the management policy for the log buffer, as it has a significant impact on the write performance of *SAL-hashing*. Next, we first present a solution to limiting the write cost in *SAL-hashing*, based on which we propose an efficient victim selection policy for the log buffer.

Write Amplification of Processing Read-Intensive Requests in SAL-hashing. SAL-hashing may incur write amplification problems when processing read-intensive workloads. For read-intensive workloads, the log region of some sets in *SAL-hashing* is likely to have few operational logs. However, frequent search operations in read-intensive workloads will trigger merge operations to avoid inspecting the log region frequently. Such merge operations will substantially increase write operations and in turn result in the write amplification in *SAL-hashing*. In the worst case, each log-flushing operation will be accompanied by a series of lookups and a merge operation.

Bounding the Write Cost by Resizing the Log Buffer. According to Appendix A, available in the online supplemental material, the amortized write cost of an update is given by Equation (6), where $g_n = \frac{k_n}{B \cdot l \cdot g}$, and c_l represents the number of logs in the log region when a merge operation is invoked.

$$U_a = \frac{2 \cdot w \cdot (r+4) \cdot \frac{k_n}{B \cdot l \cdot g}}{m \cdot z - 2 \cdot \frac{k_n}{B \cdot l \cdot g}} + \frac{g \cdot \frac{w}{n} + \frac{1-sf}{1+sf} \cdot P((1+sf) \cdot l \cdot B) \cdot g \cdot w}{c_l}. \tag{6}$$

Given the capacity of a bucket page $B = \frac{z}{r}$ and the load factor l, we can get the upper bound of $\frac{1-sf}{1+sf} \cdot P((1+sf) \cdot l \cdot B)$, which is named as p_u in the following text. Then, the amortized write cost of an update can be estimated by

$$U_e(m, c_l) = \frac{2 \cdot w \cdot (r+4) \cdot g_n}{m \cdot z - 2 \cdot g_n} + \frac{g \cdot \frac{w}{n} + p_u \cdot g \cdot w}{c_l} \geq U_a. \tag{7}$$

In the worst case, when a merge operation is triggered, the log region has been flushed only once. Thus, we have $c_l = \frac{m \cdot z - 2 \cdot g_n}{(r+4) \cdot g_n}$. As a result, the update cost in the worst case can be defined by

$$U_{worst} = \frac{w \cdot (r+4) \cdot g_n}{m \cdot z - 2 \cdot g_n} \cdot \Big(2 + \frac{g}{n} + p_u \cdot g\Big). \tag{8}$$

Equation (8) shows that U_{worst} is proportional to g_n, and inversely proportional to the log-buffer size m. Therefore, we propose to dynamically adjust m to ensure that U_{worst} is below a given threshold U_t. Let U_t be the threshold of write cost, it is easy to get the log buffer threshold m_t satisfying the inequality of $U_{worst} \leq U_t$.

Whenever a merge operation is triggered, the estimated write cost U_e can be figured out by Equation (7). If U_e exceeds the threshold U_t, some memory pages are reclaimed from the page buffer. These memory pages will be used as additional spaces of the log buffer. Moreover, each time the number of memory pages reclaimed is at most $\lceil \frac{m_t - m}{2} \rceil$, where m is the size of current log buffer. Even in the worst case, that is, each merged log region has been flushed only once, we can bound the amortized write cost to the given threshold U_t after $\lceil log_2(m_t - m) \rceil$ adjustments. In other words, the growing trend of the log buffer towards m_t is similar to the curve of an exponential distribution $Exp(ln2)$.

On the other hand, when the workload is write-intensive, there are many operational logs residing in the log region, then the amortized write cost of a merge operation is quite low and the write cost U_a is dominated by the flushing cost. As the overall performance of a write-intensive workload is significantly impacted by write costs, we set an upper bound for the amortized flushing cost U_f. Each time after flushing, if U_f exceeds the imposed upper bound $\alpha \cdot U_t (0 < \alpha < 1)$, we enlarge the log buffer gradually.

SAL-hashing dynamically adjusts the log-buffer size by continuously monitoring workload changes. If $U_e < U_t$ and $U_f < \alpha \cdot U_t$ hold for a time period under current access pattern, we reduce the size of the log buffer and the reduced memory is added to the page buffer. The reduction procedure is terminated when any of the following conditions is satisfied: (i) U_f approaches the threshold value $\alpha \cdot U_t$; (ii) $U_e(m, c_l^{max})$ approaches the threshold value U_t, where c_l^{max} is the maximum value of c_l during this period; and (iii) the log buffer reaches its initial size.

In summary, the write cost is approximately limited to $\alpha \cdot U_t$ for write-intensive workloads, while for read-intensive workloads the imposed upper bound is set to U_t. The parameters of α and U_t can be set according to specific application scenarios, e.g., maximum data size, available memory spaces, and so on. Moreover, as analyzed in Appendix A, available in the online supplemental material, the log buffer of *SAL-hashing* is memory efficient since it has high efficiency for piggybacking updates. Taking write-intensive workloads as an example, the log buffer size is at most $\frac{2\cdot w\cdot(r+4)\cdot g_n}{\alpha\cdot U_t\cdot z}+\frac{2\cdot g_n}{z}$ when imposing an upper bound $\alpha\cdot U_t$ on U_f, while the index size is at least $g_n\cdot g$ pages, then the ratio of the log-buffer size to the index size is at most $\frac{2\cdot w\cdot(r+4)}{\alpha\cdot U_t\cdot z\cdot g}+\frac{2}{z\cdot g}$. Based on the parameters used in the experiment section, i.e., $z=4096(bytes)$, $r=16(bytes)$, $g=16$, $\alpha=0.5$, $U_t=0.5w$, the ratio is at most 0.247 percent.

Victim Selection Policy for the Log Buffer. The victim selection policy aims to schedule the committing sequence of the update requests so as to reduce the write cost. When the log buffer is full, we always select a set with log regions on SSD as the victim because U_e decreases with c_l. In addition, only when the log number of a set exceeds the average number, (i.e.,$\frac{\#capacity\_of\_log\_buffer}{\#number\_of\_sets\_in\_log\_buffer}$), the set can be selected as a victim.

When we have to select a set without log regions on SSD as a victim, we choose the set which is unlikely to trigger a merge operation immediately after flushing. Further, even if that happens, we aim to make the write cost U_e below the threshold U_t.

For each set residing in the log buffer without log regions, we record the number of lookups on the set to simulate the probability of triggering a merge operation if the update logs are flushed to the log region at first. According to Equation (3), the probability of a log region remaining stable after i lookups can be estimated roughly by

$$P_{stable}(i)=1-\sum_{j=1}^{i}\pi_j(p). \quad (9)$$

By setting up an appropriate threshold p_t, the upper bound of lookups l_t can be figured out in advance. For instance, let $p_t=0.2$, if there are more than l_t lookups on a set, the estimated probability of triggering a merge operation is over 80 percent. As a consequence, the procedure of selecting a victim without log regions is as follows:

(1) If the update-log number of a set is over $\frac{m_t\cdot z-2\cdot g_n}{(r+4)\cdot g_n}$, we evict the set as a victim.

(2) If the update-log number of a set is between the average number and $\frac{m_t\cdot z-2\cdot g_n}{(r+4)\cdot g_n}$, we consider two cases. If the number of lookups in a set is below the threshold l_t, we evict it as a victim. Otherwise, the set gets a second chance, and the number of lookups is decayed by a factor of 0.5.

(3) If the update-log number of a set is less than the average number, we decay the lookup count of the set and skip the set.

3.6 Transaction Support and Crash Recovery

In this section, we discuss how the *SAL-hashing* can be augmented to support transaction and crash recovery. As *SAL-hashing* logs updates temporarily in main memory, data-loss will occur after the system crashes, and the system is left in an inconsistent state. The traditional DBMS is usually equipped with write-ahead-logging (WAL) and recovery methods to ensure consistency [25]. As the ARIES recovery method [26] is elegantly designed to work with a steal, no-force buffer management and fuzzy check-points, it is widely used by IBM DB2, Microsoft SQL Server, and many other DBMS. The recovery of ARIES works in three phases: analysis, redo, and undo. Next, we only describe the minimal modifications of the ARIES to make *SAL-hashing* support transaction and recovery.

TABLE 2
Transaction Logs for *SAL-Hashing*

Type	Description
Update Log	$<$ LSN, Operation Type (Update), TransID, PrevLSN, SetID, IndexID & & Key-Value $>$
Flush Log	$<$ LSN, Begin Flush, FlushID, IndexID $>$ $<$ Logs similar to traditional DBMS logs, which keep track of redo and undo info of each updated page $>$ $<$ LSN, End Flush, FlushID, IndexID $>$
CRL (Roll_Log_Buffer)	$<$ LSN, Operation Type (Compensation && Roll_Log_Buffer), TransID, UndoNextLSN, SetID, IndexID && Undo Info $>$
CRL (Roll_Flushed_Set)	$<$ LSN, Operation Type (Compensation && Roll_Flushed_Set), TransID, UndoNextLSN, SetID, IndexID && Undo Info $>$

Additional Data Structure. In ARIES, each log record, which corresponds to a specific write of some transaction, has a log sequence number (LSN) that is monotonically increasing. Along with critical redo and undo logs, two tables have to be maintained for efficient recovery, i.e., the transaction table and the dirty page table. The transaction table keeps the LSN of most recent log record for each active transaction, while the dirty page table contains an entry for each dirty page in the buffer, which includes the LSN corresponds to the earliest update to that page.

Furthermore, these two tables are flushed to the log file when checkpointing. Note that each *set* in the log buffer can be regarded as a dirty page in the page buffer, and adding an operation log to a set corresponds an update to a page. Similarly, an update log is written when updating a set, as shown in Table 2. Moreover, similar to the dirty page table, we also maintain a dirty set table that contains LSNs corresponds to the earliest update to each set, and flush the dirty set table when checkpointing. During the analysis phase, we restore all the three tables as they were at the time of crash by running through the log file from the last checkpoint. And then we determine the point in the log file at which to start the redo phase by finding the minimal LSN in the dirty page table and dirty set table.

Atomicity of Flushing a Set. Since writing a dirty page to stable storage is atomic in traditional DBMS, we need to ensure the atomicity of flushing a set because a set is regarded as a dirty page. Borrowing ideas from the logs of a transaction that also has to guarantee atomicity, we introduce flush logs (see Table 2) that include begin and end log to state clearly whether a flush operation is finished. When system restarts after a crash, the rollback of an unfinished flush operation is the same as an uncommitted transaction.

Transaction Abort. When a transaction is aborted, changes made by the transaction should be undone. During the undo phase of a recovery or while rolling back an aborted transaction, a compensation log record (CLR) is written for each action that is undone. If a transaction is aborted, while the update operations generated by this transaction have been flushed to SSDs in batch, then there is a conflict between the transaction abort and the atomicity of a flush operation, because we can't roll back a flushed set due to the atomicity of flushing a set. We devise the following solutions to resolve the conflict. If the update operation to be rolled back is still in the log buffer, we just remove it from the log buffer, and write a CRL of *Roll_Log_Buffer*, as shown in Table 2. Otherwise, to roll back a flushed update operation, we add an opposite operation log in the log buffer to eliminate the changes made by that update operation, and meanwhile write a CRL of *Roll_Flushed_Set*.

Adaption to Snapshot-Based Recovery. Storage snapshot is an important feature of data storage service, it allows instantaneous snapshots of any volume that can be run adhoc or regularly scheduled via the console. Since taking a snapshot requires zero CPU resource on the hosts and initially requires very minimal storage space, it has become popular to make use of storage snapshot to provide swift backup and recovery on databases [27], [28], [29]. However, as storage snapshot is just a single point in time, it is not possible to roll forward to the newest time point. Besides, storage snapshot simply bases on volume, and it has no knowledge of the transaction states. Thus it is not suitable as replacement for a full recovery strategy. Take the Oracle database as an example, to perform a complete database recovery, it should only restore DATA and INDX volumes from the snapshot without overwriting the online redo logs, the undo tablespace, and the control files. Then, it needs to apply the active and current redo logs to roll forward to the newest time point, and roll back the uncommitted transactions through the undo tablespace and the transaction table. As mentioned before, operations to each set are tactfully converted to general operations of the database, thus we argue that the *SAL-hashing* can also be augmented to work with other recovery algorithms. For example, when coped with the Oracle database that separates the redo log and undo log, updating a set is treated as writing a page, then it needs to add a redo log record in the redo log file and update the transaction table in the undo tablespace. Flushing a set is regarded as a special transaction, it needs to records the changes of data pages in the redo log file, and stores the before images in the undo tablespace. Therefore, *SAL-hashing* can be adapted to recovery algorithms based on redo and undo logs. What's more, adopting the most recent storage snapshot technique to further improve the recovery performance is also applicable.

4 Experimental Results

4.1 Settings

Experimental Setup. We ran experiments on a PC powered by a quad-core Intel Core i5 processor on Linux with 4 GB main memory, and two SSDs are used for index files: an OCZ Core Series OCZSSD2-1C64G with a high asymmetry of read/write latencies, which is regarded as a low-end SSD; the other is an Intel SSD 520 Series SDSC2CW240A310 with a narrowing gap of read/write latencies, and we regard it as a high-end SSD. All experiments were conducted using the direct I/O mode.

We set the page size to 4 KB, and set the buffer size to 64 MB. The *Buffer_Index_Ratio* (i.e., the buffer pool size to the index size) ranges from 2 to 32 percent under various traces. An LRU buffer manager was implemented for caching recently read and written pages. Moreover, we devoted part of the buffer as the log buffer. Specially, 4 MB memory was allocated for the log buffer initially. The record in the experiment has an 8 byte key and an 8 byte value. In addition, the load factor (lf) for each algorithm is 0.75. For *SAL-hashing*, the parameter g is set to 16, U_t is set to $0.5w$, and α is set to 0.5. The adapted log buffer size does not exceed 15 percent of the overall buffer memory.

Indexes for Comparison. We mainly compare *SAL-hashing* with disk/flash oriented hash indexes. Due to the popularity of B-tree in database systems, we also compare *SAL-hashing* with B-Tree-like indexes. The Bw-tree [14] is a B-tree style index with a latch-free mechanism and CPU cache optimization through delta updates. It is currently deployed in Microsoft's main-memory database engine Hekaton [30], where latches and cache coherence overhead are paramount. To coordinate with transaction management and ensure consistency and fast recovery, the Bw-tree proposes several write optimizations, such as incremental flushing and the log-structured store. However, the implementation details of the Bw-tree are not clearly stated in [14]. Thus, in order to evaluate the trade-off between read and write I/O costs when storing the delta chains on flash memory, we refer to an open source implementation of the in-memory Bw-tree[1] and integrate following write optimization techniques, including blind writes, incremental flushing with contiguous deltas, and the log-structured store, which were originally proposed by the Bw-tree. In following texts and figures, we term this implementation as B^w_Δ-tree, i.e., B-tree with delta chains and write optimizations. Note that this study focuses on the asymmetrical I/O costs of flash memory, therefore our experiments are mainly designed to test the number of read and write I/Os, and runtime on SSDs.

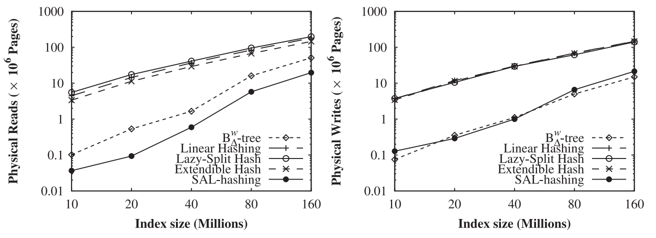
Traces. The traces consist of a sequence of lookups and updates with different read-write ratios. We make use of the *Zipf's law* [31] to generate skewed traces with different reference localities, and a locality of "80-20" means that eighty percent of the references deal with the most active twenty percent of the buckets.

4.2 Insertion and Search Performance

In this section, we evaluate the insert and search performance on various index sizes to validate the scalability of *SAL-hashing*. Three traces were used in the experiment. The first trace, *Trace A*, consists of random insertions to five index files, and the volume of insertions ranges from 10 to 160 million. The second trace, *Trace B*, consists of 10 million random search operations on each index file created by Trace A. The third trace, *Trace C*, consists of 10 million random search operations with various localities on an index file with 80 million records.

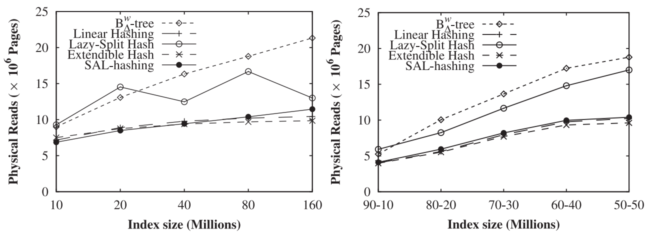
Fig. 6 shows SSD reads and writes of different indexes on Trace A. Though the Lazy-Split hash applies a lazy-split strategy, it globally controls split operations, and may forcibly split some buckets which have not many records. In addition, a global lazy-split control may cause some frequently inserted buckets to scan a longer bucket list to find free space for new insertions. As Fig. 6a shows, the Lazy-Split hash has about 9-20 percent more SSD reads than the linear hashing, and 36-54 percent more than the extendible hashing. What's worse, the increase of SSD reads decreases the hit ratio, and it

1. https://github.com/wangziqi2013/BwTree



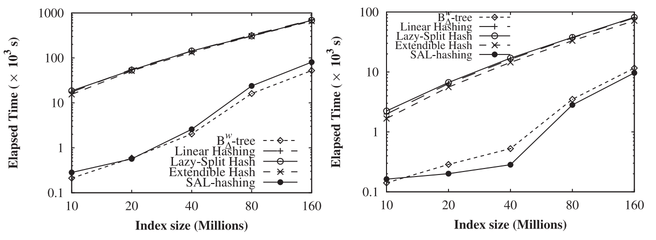
(a) SSD Reads (b) SSD Writes

Fig. 6. I/O count on trace A.



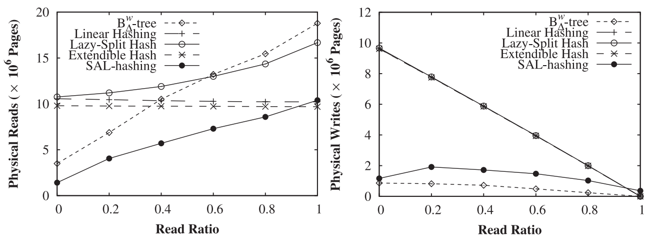
(a) On Trace B (b) On Trace C

Fig. 8. SSD reads on trace B and trace C.



(a) On OCZ SSD (b) On Intel SSD

Fig. 7. Run time on trace A and two SSDs.



(a) SSD Reads (b) SSD Writes

Fig. 9. I/O count on Trace D.

may further decrease the number of write hits in the buffer [32]. As a consequence, the Lazy-Split hash only reduces SSD writes by 4 percent on average compared with the linear hashing. The results of SSD writes show the benefits of write optimizations implemented in *SAL-hashing* and the B^{w}_{Δ}-tree. The B^{w}_{Δ}-tree has comparable SSD writes with *SAL-hashing*, mainly because of its delta-updating, incremental batch-flushing policy, and page store as a compact string, which result in high storage efficiency and effectivity in reducing write amplification. In combination with fast sequential writes or coarse-grained writes, the B^{w}_{Δ}-tree and *SAL-hashing* are over an order magnitude faster than other indexes in most cases, as shown in Fig. 7. On the other hand, each page consolidation of the B^{w}_{Δ}-tree needs to traverse the delta chains and the base page, and thus results in around 2× more SSD reads than *SAL-hashing*. Thus, in general *SAL-hashing* should have better overall performance when running on SSDs with a close gap between read and write latency, which has been a key feature of modern SSDs. The run-time results in Fig. 7b has supported this claim, where the overall run-time of the *SAL-hashing* is 23 percent less than that of the B^{w}_{Δ}-tree on Intel SSD. Since Trace A is a write-intensive trace, we can reasonably imagine that the *SAL-hashing* should be more efficient than the B^{w}_{Δ}-tree when running on read/write-mixed or read-intensive traces.

Fig. 8 shows SSD reads of different indexes on Trace B and C. Since both traces only consist of searches, SSD reads can reveal the search performance of each index. As the Lazy-Split hash triggers a split operation when the total bucket number is four times of the Split-cursor, the search performance changes regularly with the index size. If the ratio of the total bucket number to Split Cursor is small, which indicates that all lazy-split buckets have just been split, and most searches can be accomplished by inspecting one bucket, yielding high search performance. Otherwise, there are many lazy-split buckets, and searches related to these lazy-split buckets should consider multiple buckets that may contain the searched key. The search performance of *SAL-hashing* is closely related to the index size and access localities. When dealing with relatively small datasets or better localities, *SAL-hashing* detects read-intensive sets quickly and effectively, and merge their log regions to the corresponding buckets on SSDs. Consequently, it has similar search performance compared with the extendible hashing. In addition, even in large datasets with uniform access patterns, *SAL-hashing* avoids scanning bucket lists through the virtual splits mechanism, and therefore it has comparable performance with the linear hashing. The B^{w}_{Δ}-tree has 25-86 percent more reads than *SAL-hashing*, because it has to read multiple fragments to get the delta updates and base value of a page when processing search requests. Especially under large datasets with uniform access patterns, the delta updates are likely to spread on many flushing pages, which will incur a number of SSD reads.

4.3 Impact of Parameter Settings

Search Ratio. As *SAL-hashing* optimizes SSD writes and makes a tradeoff between update and search performance, it is important to explore traces with different search/update ratios. In this section, we take into account both the search/update ratio and access locality. We first generated an index file containing 80 million records, and used two traces in the experiment. The first trace, *Trace D*, consists of 6 traces, and each trace consists of 10 million search/update operations with uniform access patterns, but has various search ratios. The second trace, *Trace E*, is similar to the Trace D, except that each trace has an "80-20" access locality.

Fig. 9 shows I/O count of different indexes on Trace D. The write amplification gap between *SAL-hashing* and the B^{w}_{Δ}-tree increases under search/update mixed workloads. This is because the B^{w}_{Δ}-tree makes a page consolidation only when the delta chain exceeds the threshold, while *SAL-hashing* monitors the search/update tendency of each sets and triggers merge operations when necessary. But, fortunately, SSD writes that caused by merge operations are performed in coarse-grained granularities, yielding a high bandwidth. On the other side, the log structured store of the B^{w}_{Δ}-tree has a side effect of read amplification. As Fig. 11 shows, it leads to worse time performance in read-intensive workloads. The design

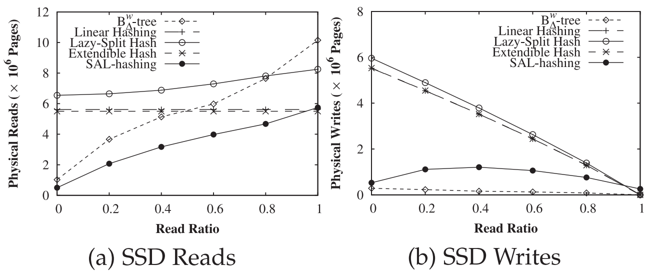


Fig. 10. I/O count on Trace E.

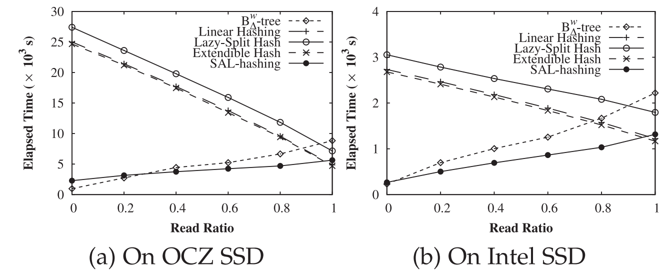


Fig. 12. Run time on Trace E and two SSDs.

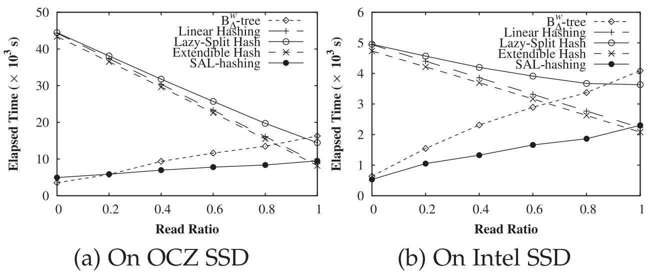


Fig. 11. Run time on Trace D and two SSDs.

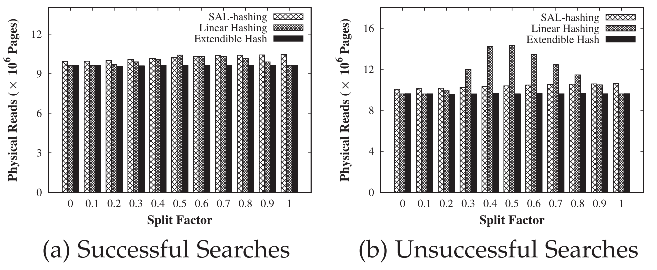


Fig. 13. Search performance versus sf.

trade-off between read and write I/O cost is crucial to the overall performance, especially for modern SSDs with a narrowing gap of read/write speeds. Although our online algorithm detecting the change of access patterns dully on uniform access patterns, *SAL-hashing* specifically optimizes reads for overflows, and it has comparable performance to the linear hashing even when the search ratio equals 1.

When handling traces with "80-20" access localities, as shown in Fig. 10, the linear hashing has about the same SSD reads as the extendible hashing since most of the overflow reads hit in the buffer. The read amplification gap between the B^w_Δ-tree and *SAL-hashing* is smaller compared with uniform access patterns (Trace D). There are mainly two reasons: one is better localities lead to more contiguous deltas stored in flash memory, and this result in less SSD reads when traversing the delta chain; the other is more searches hit in the delta updates in memory, which can be regarded as record-level read cache. The experimental results of run time, as shown in Figs. 11 and 12, indicate that B^w_Δ-tree is more time-efficient under better localities.

In summary, *SAL-hashing* not only optimize write performance through high efficiency of piggybacking updates and high bandwidths of coarse-grained writes, but also adapt itself to make a trade-off between read and write I/O cost. Therefore, it achieves balanced performance under various search ratios and access patterns. As shown in Figs. 11 and 12, when running on Trace D and OCZ SSD, *SAL-hashing* reduces runtime of the linear hashing, the Lazy-Split hash, the extendible hashing, and the B^w_Δ-tree by 60, 69, 57 and 16 percent on average. On Trace D and Intel SSD, *SAL-hashing* reduces runtime by 52, 63, 49 and 36 percent on average. When running on Trace E and OCZ SSD, the reduction is 60, 69, 59 and 9 percent on average, and on Intel SSD, the reduction is 53, 64, 52 and 25 percent on average.

Split Factor. We next evaluate the search performance of *SAL-hashing* under various split factors since it specifically optimizes reads for overflows. We first generate 11 index files by different volumes of insertions: 50.33 million (2^{18} buckets, $sf = 0$), 55.36 million ($sf = 0.1$), etc. And then we use two traces (denoted as *Trace F*) that separately contain 10 million successful and unsuccessful searches. In addition, we keep the *Buffer_Index_Ratio* constant (set as 4 percent) across all index files. We eliminate the result of the lazy-split hash since its search performance has little to do with the split factor, and set the extendible hashing as baseline.

As shown in Fig. 13, the search performance of the linear hashing basically agrees with Fig. 3, especially for unsuccessful searches. This is because many successful searches hit in bucket pages, and there is no need to scan overflow pages. The performance of *SAL-hashing* keeps stable under both traces, because it just needs one read for all searches after transformations, as well as the extendible hashing. *SAL-hashing* cannot adapt its structure timely under full random access patterns, but thanks to its read optimization for overflows, it has comparable SSD reads to the linear hashing when sf ranges from 0.3 to 0.8 under successful searches. Moreover, it has a better search performance than the linear hashing most of time under unsuccessful searches.

We also test the memory footprints of the in-memory structure proposed to index overflow pages. When there are lots of inserted key-value pairs in the log region, of course, the corresponding buckets hardly overflow, and thus the in-memory structure occupies less memory. We only consider the situation where the in-memory structure takes up maximal memory space, i.e., when all log regions are merged to corresponding buckets. Fig. 14 shows the memory usage when index size increases from 50.33 million to 100.66 million (denoted as *Trace G*), which also means sf increases from 0 to 1. According to Fig. 3, the optimal upper bound of expected memory needed is $\frac{2^{18}\cdot 0.5\cdot 4}{1024} = 512\ \mathrm{KB}$. Our proposed memory-efficient structure only has a little more memory space than the optimal strategy. As Fig. 14 shows, when sf is around 0.4, there is a peak memory usage. This is because there are lots of index entries stored in shared memory pages, where we have to carve out some space for bucket labels. However, when sf exceeds 0.5, nearly all memory pages are occupied by segments exclusively, and there is no need to waste space storing bucket

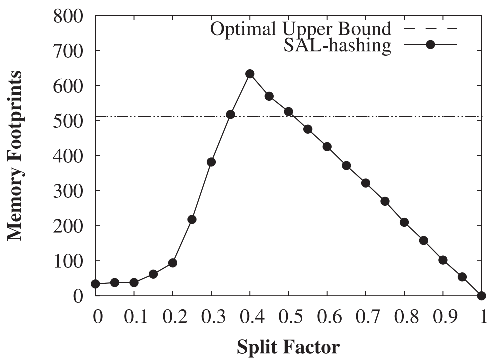


Fig. 14. Memory footprints

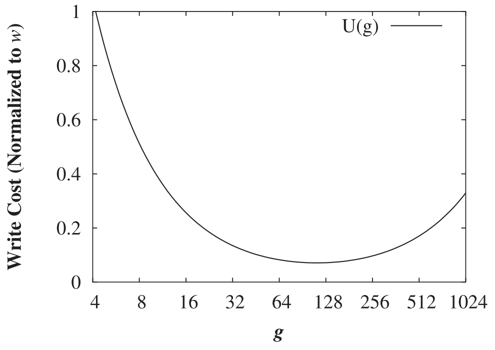


Fig. 15. Write cost.

labels, which result in high space utilization. Moreover, the curve basically matches the analytic curve in Fig. 3.

Number of Buckets in a Group. As discussed in Appendix A, available in the online supplemental material, the write cost U_a is highly related to g. Given an index with 80 million records, i.e., $k_n = 8 \times 10^7$, U_a varies according to g. We set $\frac{1-sf}{1+sf} \cdot P((1+sf) \cdot l \cdot B)$ to its upper bound, i.e., 0.35, and as Fig. 15 shows, U_a declines sharply when g increases from 4 to 32. Then, it falls slowly when g varies from 32 to 128. Finally, it grows with g gradually.

To evaluate the impact of g over the performance of *SAL-hashing*, we first set the log buffer fixed, and generated an index file with 80 million records. Then we used a synthetic *Trace H* which consists of 10 million search/update operations with a uniform access pattern, and the search ratio is 0.5. We ran Trace H multiple times with different values of g. As shown in Fig. 16, the tendency of run time is roughly consistent with the curve in Fig. 15. Note that run time decreases more than expected when g is higher than 32, this is because a larger g can also amortize the read cost of each update better. We set the parameter g to 16 or 32. The first reason for this setting is that the performance is not significantly improved as before when g exceeds 32. The second reason is that a larger g means a high-latency of performing a merge operation. This may cause an unacceptable spike of wait time to subsequent lookups.

4.4 Impact of Buffer Management Policies

Finally, we perform experimental evaluations on the impact of flash-aware buffer management policies. We choose the CFDC [33] and the FD-buffer [34] as comparisons, and only use Trace E as datasets since Trace D has no data locality.

The results of I/O count are depicted in Fig. 17. We notice that *SAL-hashing* is more friendly to CFDC than the linear

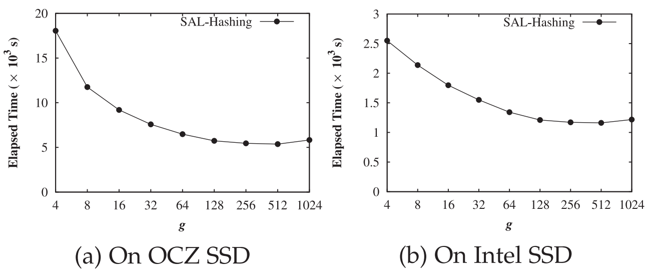


Fig. 16. Run time on Trace H and two SSDs versus parameter g.

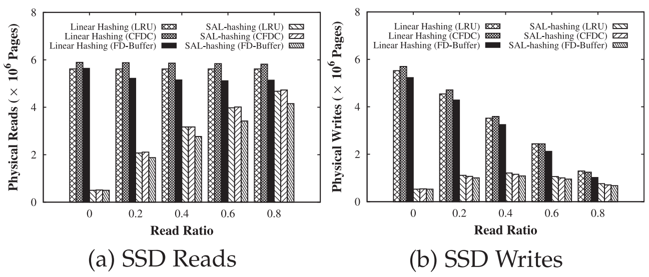


Fig. 17. I/O count of various buffer management policies on Trace E.

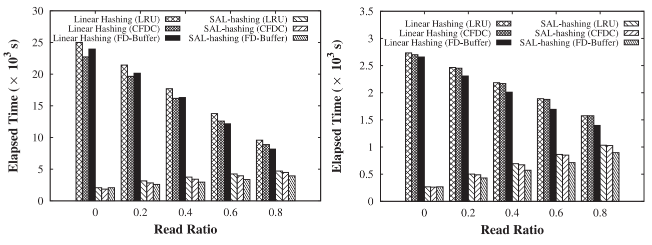
hashing. Compared with *SAL-hashing* (LRU), *SAL-hashing* (CFDC) has less SSD writes but hardly leads to more SSD reads. While for the linear hashing (CFDC), it has more SSD reads and writes than the linear hashing (LRU). This is mainly due to the CFDC's page clustering mechanism, which always tries to evict dirty pages with strong spatial locality. But it sometimes has to evict some relatively hot dirty pages, and this decreases the read hits and write hits. On the other side, *SAL-hashing* always clusters updates to the log region by delta-logs, which results in "local writes" and reduces the need of flushing hot dirty pages. For the FD-buffer, it can reduce the I/O count of both indexes with its cost-based adaptation. Note that although FD-buffer gives priority to dirty pages, its two-stack based victim selection policy can evict cold pages faster than LRU under skewed access patterns, because both them have shorter stack length than LRU. This has the similar effect of midpoint LRU insertion strategy, which avoids the buffer pollution caused by cold data, thus results in high hit ratio and less SSD reads.

The result in Fig. 18 shows that *SAL-hashing* can benefit from flash-aware buffer management policies. Note that the CFDC does not work well under the Intel SSD, possibly because of its internal controller's optimization for writes. In summary, when using CFDC, *SAL-hashing* reduces runtime of the linear hashing from 49 to 92 percent on OCZ SSD, and from 35 to 90 percent on Intel SSD. For the FD-buffer, *SAL-hashing* reduces runtime of the linear hashing from 52 to 91 percent on OCZ SSD, and from 36 to 90 percent on Intel SSD.

5 Further Discussions

Integration with Dynamic Page Replication. Page replication [35] is another effective way to exploit the multi-chip parallelism of SSDs, it was originally designed to resolve the impact of SSDs' costly internal activities.

The package-level parallelism is mainly implemented by striping pages with consecutive LBA (Logical Block Address), and cannot handle the blocking problem of read/write requests caused by internal activities [11]. However, dynamic



(a) On OCZ SSD (b) On Intel SSD

Fig. 18. Runtime of various buffer management policies on Trace E.

page replication monitors the states of all chips inside an SSD, and replicates the pages that are much likely to be blocked by internal activities across chips.

The dynamic page replication can be combined with *SAL-hashing*, e.g., by utilizing the software library of replication manager [35], to further improve performance. We can divide the SSD into multiple equal-sized partitions, and each partition is simulated as a chip for replication. The replication manager monitors the read/write access load, and dynamically determines the set of pages to be replicated. For example, if a partition has many write-intensive sets mixed with some read-intensive sets, some frequently accessed read-intensive sets can be copied to another partition that has more idle periods. Then, searches on these sets can be redirected to replicas according to the in-memory replication map, and therefor page reads are less likely to be blocked. Further, when dealing with bursty and skewed accesses, we can invoke replicated writes, which put a copy of the written pages in other partitions, to reduce the response time, because we can return success to upper-level applications when any of the replicated writes is finished.

Adaption to Non-Volatile Memory. Recently, with the advances of the 3D Xpoint non-volatile memory (NVM) developed by Intel and Micron, NVM has emerged as a new revolution in the area of memory and storage.

To address the needs of utilizing PCIe-based block devices, Intel proposed to implement 3D Xpoint SSDs in congjuction with the high-performance scalable NVMe host controller, which is commonly called NVMe SSDs. One example is the Intel OptaneTM Memory Series. Due to the substantial latency overhead added by the PCIe and NVMe protocol, NVMe SSDs usually reveal a read/write latency that is approximately 100$\times$ slower than DRAM. In addition, the read/write asymmetry still exists in NVMe SSDs. For this kind of block devices, *SAL-hashing* can also be effective. The piggybacking updates in *SAL-hashing* at the granularity of sets can effectively reduce the amortized read and write costs, and the lazy-split strategy can avoid relative slow writes of intermediate results. Furthermore, the introduction of the *log-head* and cost-based online algorithm makes a tradeoff between search efficiency and update efficiency.

Another approach for realizing NVM is to deploy NVM in DRAM-style DIMM modules, such as the Intel Apache Pass. As NVM like phase change memory (PCM) has many attractive properties such as non-volatility and byte-addressability, it is regarded as an alternative of next-generation memory. Recent researches on indexes for NVM [36], [37], [38], [39], [40] mainly focused on improving traditional logging or shadowing techniques to reduce write amplification overheads in NVM, or on ensuring persistence, robust performance, and fast recovery based on the atomic writes of NVM. The *SAL-hashing* proposed in this paper is mainly designed to work on block devices, and cannot be directly applied to the DIMM-style NVMs. Bascially, extending *SAL-hashing* for DIMM-style NVM is orthogonal to this paper, and we will investigate this issue in future work.

6 Conclusion

In this paper, we proposed an SSD-aware hash index called *SAL-hashing*. Previous flash-oriented hash indexes mainly focused on reducing writes, but cannot balance between update efficiency and search efficiency. *SAL-hashing* has several advantages. First, it is able to reduce random writes by using a lazy-split strategy and piggybacking update mechanism. Second, it can adapt to various access patterns. In particular, it optimizes searches on buckets with overflow pages, and has comparable search performance with the extendible hash on search intensive sets. Consequently, *SAL-hashing* can achieve high update efficiency and search efficiency at the same time. Furthermore, it exploits the internal parallelism of SSDs by transforming small random writes caused by split operations to coarse-grained writes.

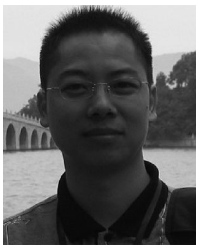
Acknowledgments

The authors would like to thank the editor and anonymous reviewers for their valuable suggestions and comments. This work is partially supported by the National Science Foundation of China (61672479 and 61472376).

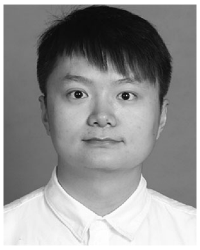
References

[1] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database System Implementation*, vol. 654, Englewood Cliffs, NJ, USA: Prentice Hall, 2000.
[2] X. Li, Z. Da, and X. Meng, "A new dynamic hash index for flash-based storage," in *Proc. 9th Int. Conf. Web-Age Inf. Manag.*, 2008, pp. 93–98.
[3] P. Jin, C. Yang, C. S. Jensen, et al., "Read/write-optimized tree indexing for solid-state drives," *Int. J. Very Large Data Bases*, vol. 25, no. 5, pp. 695–717, 2016.
[4] C. Wu, T. Kuo, and L. Chang, "An efficient B-tree layer implementation for flash-memory storage systems," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, 2007, Art. no. 19.
[5] Y. Li, B. He, R. J. Yang, et al., "Tree indexing on solid state drives," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 1195–1206, 2010.
[6] C. W. Yang, K. Y. Lee, M. H. Kim, et al., "An efficient dynamic hash index structure for NAND flash memory," *IEICE Trans. Fundamentals Electron. Commun. Comput. Sci.*, vol. 92, no. 7, pp. 1716–1719, 2009.
[7] L. Wang and H. Wang, "A new self-adaptive extendible hash index for flash-based DBMS," in *Proc. IEEE Int. Conf. Inf. Autom.*, 2010, pp. 2519–2524.
[8] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, et al., "Microhash: An efficient index structure for flash-based sensor devices," in *Proc. 4th USENIX Conf. File Storage Technol.*, 2005, pp. 3–3.
[9] B. K. Kim, S. W. Lee, and D. H. Lee, "H-Hash: A hash index structure for flash-based solid state drives," *J. Circuits Syst. Comput.*, vol. 24, no. 09, 2015, Art. no. 1550128.
[10] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 266–277.
[11] H. Roh, S. Park, S. Kim, et al., "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 286–297, 2011.
[12] C. Yang, P. Jin, L. Yue, et al., "Self-adaptive linear hashing for solid state drives," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 433–444.

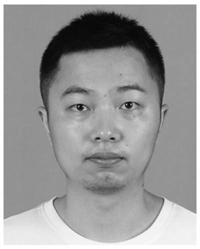
[13] S. W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD Conf.*, 2007, pp. 55–66.
[14] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," in *Proc. IEEE Int. Conf. Data Eng.*, 2013, pp. 302–313.
[15] J. J. Levandoski, D. Lomet, and S. Sengupta, "LLAMA: A cache/storage subsystem for modern hardware," *Proc. VLDB Endowment*, vol. 6, no. 10, pp. 877–888, 2013.
[16] M. Sadoghi, K. A. Ross, M. Canim, et al., "Exploiting SSDs in operational multiversion databases," *VLDB J.*, vol. 25, no. 5, pp. 651–672, 2016.
[17] S. Hardock, I. Petrov, R. Gottstein, et al., "From in-place updates to in-place appends: Revisiting out-of-place updates on flash," in *Proc. ACM Int. Conf. Manag. Data*, 2017, pp. 1571–1586.
[18] Y. Wang, L. Zhang, J. Tan, et al., "HydraDB: A resilient RDMA-driven key-value middleware for in-memory cluster computing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, Art. no. 22.
[19] A. K. S. Nath, "FlashDB: Dynamic self-tuning database for nand flash," in *Proc. 6th Int. Conf. Inf. Process. Sensor Netw.*, 2007, pp. 410–419.
[20] M. Sarwat, M. F. Mokbel, X. Zhou, et al., "Fast: A generic framework for flash-aware spatial trees," in *Proc. Int. Symp. Spatial Temporal Databases*, 2011, pp. 149–167.
[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[22] A. R. Karlin, M. S. Manasse, L. Rudolph, et al., "Competitive snoopy caching," *Algorithmica*, vol. 3, no. 1, pp. 79–119, 1988.
[23] S. S. Seiden, "A guessing game and randomized online algorithms," in *Proc. Annu. ACM Symp. Theory Comput.*, 2000, pp. 592–601.
[24] A. R. Karlin, M. S. Manasse, L. A. McGeoch, et al., "Competitive randomized algorithms for nonuniform problems," in *Proc. 1st Annu. ACM-SIAM Symp. Discrete Algorithms*, 1990, pp. 301–309.
[25] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surveys*, vol. 15, no. 4, pp. 287–317, 1983.
[26] C. Mohan, D. Haderle, B. Lindsay, et al., "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992.
[27] W. Xiao, Q. Yang, J. Ren, et al.,"Design and analysis of block-level snapshots for data protection and recovery," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1615–1625, Dec. 2009.
[28] M. MC, K. Kapa, and W. Chen, "Oracle database backup and recovery using dell storage snapshot technologies," [Online]. Available: http://en.community.dell.com/techcenter/enterprise-solutions/m/oracle_d b_gallery/20336712.
[29] PureStorage, "Using flasharrays for oracle backup and recovery," Tech. Rep. AR-160601-v01, (2016, Jul.). [Online]. Available: https://support.purestorage.com/@api/deki/files/2743
[30] E. I. C. Diaconu, C. Freedman, et al., "Hekaton: SQL server's memory-optimized OLTP engine," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 1243–1254.
[31] D. Knuth, *The Art of Computer Programming*, Sorting and Searching, Addison-Wesley, vol. 3, 1973.
[32] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2002, pp. 31–42.
[33] Y. Ou, t. Härder, and P. Jin, "CFDC: A flash-aware replacement policy for database buffer management," in *Proc. 5th Int. Workshop Data Manag. New Hardware*, 2009, pp. 15–20.
[34] S. T. On, S. Gao, B. He, et al.,"FD-Buffer: A cost-based adaptive buffer replacement algorithm for flashmemory devices," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2288–2301, Sep. 2014.
[35] B. He, J. Yu, and A. C. Zhou, "Improving update-intensive workloads on flash disks through exploiting multi-chip parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 152–162, 2015.
[36] L. Li, P. Jin, C. Yang, et al., "Optimizing B+-Tree for PCM-based hybrid memory," in *Proc. Int. Conf. Extending Database Technol.*, 2016, pp. 662–663.
[37] J. Yang, Q. Wei, C. Chen, et al.,"NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, vol. 15, pp. 167–181.
[38] I. Oukid, J. Lasperas, A. Nica, et al., "Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory," in *Proc. Int. Conf. Manag. Data*, 2016, pp. 371–386.
[39] F. Xia, D. Jiang, J. Xiong, et al., "HiKV: A hybrid index key-value store for DRAM-NVM memory systems," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 349–362.
[40] B. Debnath, A. Haghdoost, A. Kadav, et al., "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating Syst. Rev.*, vol. 49, no. 2, pp. 18–26, 2016.



Peiquan Jin received the PhD degree in computer science from the University of Science and Technology of China (USTC), in 2003. He is now an associate professor with the School of Computer Science and Technology at USTC. His research interests focus on databases on new hardware, spatiotemporal databases, and Web information retrieval. He is a senior member of CCF and a member of the IEEE and ACM.

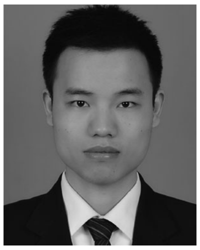


Chengcheng Yang is currently working toward the PhD degree in the School of Computer Science and Technology at the University of Science and Technology of China (USTC). His research interests include flash-based databases and massive data processing.



Xiaoliang Wang is currently working toward the PhD degree in the School of Computer Science and Technology, University of Science and Technology of China (USTC). His research interests include databases on new hardware including flash memory and phase change memory.

Lihua Yue received the bachelor's and master's degrees in computer science from the University of Science and Technology of China (USTC). She is a full professor with the School of Computer Science and Technology, USTC. Her research interests include flash-based databases, spatiotemporal databases, information retrieval, and image processing. She is a senior member of the CCF and a member of the ACM.



Dezhi Zhang is a currently working toward the master degree in the School of Computer Science and Technology, University of Science and Technology of China (USTC). His research interests include flash-based databases and data management on hybrid storage.