Access-Pattern-Aware Personalized Buffer Management for Database Systems

Yigui Yuan, Zhaole Chu, Peiquan Jin, Shouhong Wan
[1]*School of Computer Science and Technology, University of Science and Technology of China, Hefei, China*
[2]*Key Lab. of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei, China*
jpq@ustc.edu.cn

***Abstract*—Buffer management is an essential technology for database management systems. Traditional buffer management employs an empirical approach based on access recency or frequency which fails to adapt to access-pattern changes in various database applications. In this paper, we present a new access-pattern-aware buffer manager called PBM (*Personalized Buffer Manager*), which can detect the access patterns for each database file and use a specific buffering policy for each database file. In particular, we propose a workload classifier to detect the access pattern of a database file. Then, we partition the buffer into various zones, set different sizes for each zone, and select the most suitable buffering scheme for each zone. With such a mechanism, each zone is responsible for caching a specific database file, and we can realize a personalized buffer manager for different database files, which can improve the buffer efficiency and reduce the page I/Os of the buffer manager. We compare PBM with three existing buffering algorithms, including LRU, LFU, and LeCaR, on two workloads, namely a regular workload and a shifting workload, which are composed of different access patterns. The results show that PBM outperforms the three competitors in terms of hit ratio and page I/Os. As a consequence, PBM achieves 1.66x, 2.03x, and 1.39x hit-ratio improvements compared to LRU, LFU, and LeCaR, respectively, on the regular workload. While on the shifting workload, PBM achieves 1.90x, 1.55x, and 1.49x higher hit ratios than LRU, LFU, and LeCaR, respectively.**

***Keywords*—Access pattern, Buffer management, Personalized buffer manager, Classification**

I. Introduction

Buffer management is a key module in database systems to improve query performance [1]. The optimal buffer manager can always maintain the pages that will be requested in the future in the buffer so that future requests can hit in the buffer. Generally, as the buffer size is usually limited, we need to use a replacement algorithm to evict some pages out of the buffer when the buffer is full. Therefore, most of the previous works on buffer management focus on the study of buffer replacement schemes, which highly determine the performance of buffer management in database management systems (DBMSs).

A buffer replacement policy aims to evict the most useless pages out of the buffer by predicting the future usage of pages. As future accesses are hard to be predicted, traditional DBMSs employ some empirical algorithms to perform buffer replacement. The most well-known algorithm is LRU (Least Recently Used) [1]. It assumes that the least recently used page is least likely to be requested in the future. Thus, it always selects the least recently used page for a replacement. In one word, traditional buffer replacement algorithms use an empirical way to select the victim for replacement. However, a critical problem of such a mechanism is that they cannot adapt to workload changes. As a result, they may perform well under some kinds of workloads but show poor performance under other workloads. For example, the LRU policy works well under the workloads with high time locality, but has poor performance when the workload involves periodical scans (known as the scan nonresistance problem of LRU). Although a few works studied the adaptivity of LRU, such as AD-LRU [2] and LeCaR [3], their performance relies on the empirical setting of parameters, which are still empirical solutions.

This paper proposes a new idea to improve the efficiency of buffer management in database systems. Differing from the traditional empirical approaches, we present an access-pattern-aware personalized buffer manager called PBM (*Personalized Buffer Manager*). The main contributions of PBM can be summarized as follows.

(1) PBM proposes to use multiple sub-buffers (called zones), each of which is responsible for caching a specific database file. Such a design enables us to use different buffering policies for different database files. As a result, PBM is equipped with multiple buffering policies rather than a single policy in traditional buffer management.

(2) PBM can detect the access pattern of each database file according to the access sequence. A workload classifier is proposed for the detection of the access pattern. Based on the detected pattern, PBM sets different zone sizes and enables different buffering policies for database files.

(3) We compare PBM with three existing buffering algorithms, including LRU, LFU, and LeCaR, on two workloads, namely a regular workload and a shifting workload, which are composed of five access patterns. The results show that PBM outperforms the three competitors in terms of hit ratio and page I/Os.

The remainder of the paper is structured as follows. Section II summarizes the related work. Section III details the structure and key technologies of PBM. Section IV reports the experimental results, and finally, Section V concludes the whole paper.

DOI reference number: 10.18293/SEKE2022-141

II. Related Work

In the past two decades, many well-known buffering policies have been proposed, e.g., LRU [1], LFU [1], ARC [4], 2Q [5], and LeCaR [3]. Most of these algorithms have been well explained in textbooks. The literature [1] presents a good survey on traditional buffer management policies.

LRU (Least Recently Used) [1] always evicts the least-recently-used page from an LRU queue used to organize the buffer pages, which are ordered by time of their last reference. It always selects as a victim the page found at the LRU position. The most important advantage of LRU is its constant runtime complexity. Furthermore, LRU is known for its good performance on workloads having high temporal locality. However, LRU does not exploit the frequency of references. Also, LRU is not scan-resistant.

LFU (Least Frequently Used) [1] removes the least frequently used page whenever the buffer is overflowed. The simplest method to employ an LFU algorithm is to assign a counter to every page that is loaded into the buffer. Each time a reference is made to that page, the counter is increased by one. When the buffer reaches the maximum capacity and a new page is waiting to be inserted, the buffer will search for the page with the lowest counter and remove it from the buffer.

ARC (Adaptive Replacement Cache) [4] is an adaptive caching algorithm that is designed to recognize both recency and frequency of access. ARC divides the cache into two LRU lists, T1 and T2. T1 holds items accessed once while T2 keeps items accessed more than once since admission. Since ARC uses an LRU list for T2, it is unable to capture the full frequency distribution of the workload and perform well for LFU-friendly workloads. For a scan workload, new items go through T1, protecting frequent items previously inserted into T2. However, for churn workloads, ARC's inability to distinguish between items that are equally important leads to continuous cache replacement.

The recent LeCaR algorithm [3] is an outstanding cache replacement algorithm that is based on reinforcement learning and regret minimization. The algorithm accepts a stream of requests for memory pages and decides which page to evict from a cache when a new item is to be stored in the cache following a "cache miss". LeCaR has been shown to be among the best performing cache replacement algorithms in practice [3]. Experiments have shown that it is competitive with the best cache replacement algorithms for large cache sizes and is significantly better than its nearest competitor for small cache sizes like ARC [4].

III. PBM: Personalized Buffer Manager

In this section, we detail the design of PBM. We first analyze the different access patterns and demonstrate that different buffering schemes are suitable for different access patterns, which motivates this study. Then, in Section III-B we present the workload classifier. Finally, in Section III-C we discuss the architecture and algorithms of PBM.

A. Analysis of Access Patterns

In the literature [6], the Turing Prize Winner, Michael Stonebraker, has summarized four types of common workloads: sequential accesses to blocks that are not seen nor re-visited, sequential accesses to blocks repeatedly visited, random accesses to blocks not seen nor re-visited, and random accesses to blocks where some blocks have a non-zero probability of reference. However, this definition of types is not quite fit for the purpose of choosing a replacement policy. For example, it makes no difference for LRU or LFU if the accesses are sequential or random. Therefore, in this paper, based on Stonebraker's definition, we define five access patterns, including *random*, *scan*, *skewed*, *cyclic*, and *vary*. The *random* pattern corresponds to the third type in Stonebraker's definition, where all pages in the file follow uniform distribution. The *scan* pattern corresponds to the first type. In the *skewed* pattern, some pages are accessed with a larger probability than the others, and in the *cyclic* patterns, a certain set of pages are cyclically accessed. The last pattern is a supplement to the former four patterns. It represents the access pattern where the hot region of the file varies over time.

We first generate workloads following the five access patterns and test the performance of several existing buffering policies on these access patterns. The results are shown in Fig. 1. We can see that no buffering scheme can maintain the highest hit ratio on all workloads. Also, different algorithms are suitable for different patterns. For example, LFU achieves the best hit ratio on the *cyclic* pattern but gets the worst performance on the *vary* workload. Therefore, a better way is to choose the best policy for a specific access pattern. Moreover, as the objects within a database may have different access patterns, it is better to use different policies for different database objects, which motivates the design of PBM.

B. Workload Classifier

We use two statistical features to distinguish the access pattern, namely *Correlation List* and *Page Coverage*. Both features are calculated based on the frequency histogram of the access. Let $H = \{a_1, a_2, ..., a_n\}$ be an access sequence, where a_k is an access, and $F = \{p_1, p_2, ..., p_m\}$ is the file it accesses. We first cut the sequence into segments, each with length S. Then, we calculate the frequency histogram for each segment. The frequency histogram is an m-length vector. The ith element of the vector is the frequency of page p_i in the segment. This vector conveys the page distribution information of the access. The reason why we segment the sequence is to recognize the change of distribution. Type A and B have stable distribution, while Type C's distribution varies.

(1) *Correlation List*. By comparing between segment histogram, we can figure out if the distribution has changed or not. This brings out our first feature: *Correlation List*. We use cosine correlation as the metric of the similarity between segments. Suppose the sequence has been cut into subsequences $\{s_1, s_2, ..., s_t\}$, where s_i is a segment, and $\{h_1, h_2, ..., h_t\}$ is the corresponding histogram list. Then the correlation list is $\{corr(h_1, h_2), corr(h_2, h_3), ..., corr(h_{n-1}, h_n)\}$. Figure 2

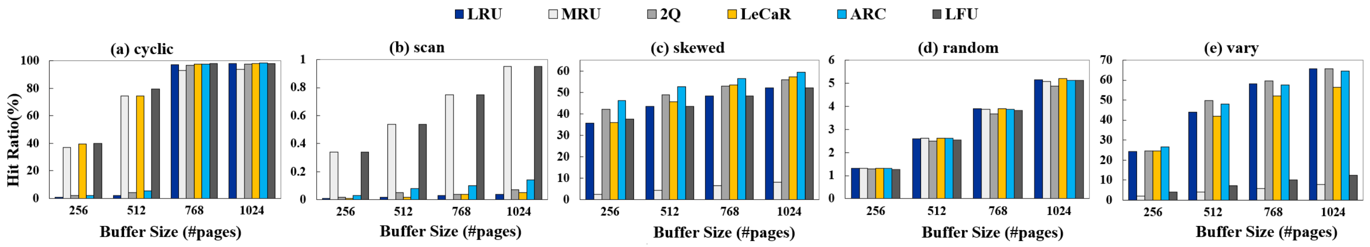

Fig. 1. The Performance Varying of Different Buffering Schemes on Various Access Patterns

shows the correlation list of the five access patterns. We can see that the correlation lists of *cyclic* and *skewed* are close to 1, showing that strong consistency exists between adjacent segments. Though the *random* and the *scan* pattern has a constant distribution, their correlation lists are close to or even below 0. For the *vary* pattern, the case is a little more complicated. Figure 2 shows when the segment length is 2000 and the phase changing period for *vary* is 5000, the correlation between two adjacent segments drops sharply every five segments.

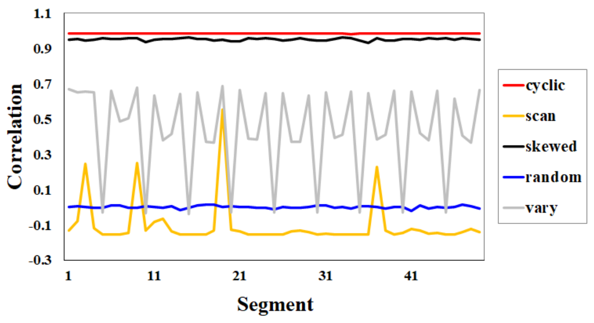

Fig. 2. The Correlation of the Five Access Patterns

(2) *Page Coverage*. For a segment of the workload, the *Page Coverage* is calculated as:

$$\frac{\#pages\ visited\ twice\ or\ more}{\#pages\ visited\ once\ or\ more}$$

The access in *vary* is concentrated on hot pages, while a *random* access is dispersed. Thus, there would be much more pages visited in *random* than in *vary*. However, most pages visited in *vary* will be visited twice or more. In *random*, on the contrary, only a small fraction of pages will be visited more than once. Suppose we have a file of size F, and two workloads with the *random* pattern and the *vary* pattern, respectively. The *vary* workload has a hot zone of size αF, and the probability of visiting its hot zone is β. If we pick a segment length larger than $2\alpha F$ but smaller than F, then the average visiting number of the pages in the hot zone of *vary* is:

$$\frac{average\ visiting\ number\ of\ the\ hot\ zone}{\#pages\ in\ the\ hot\ zone} \geq 2\beta,$$

while the average visiting number of the pages in *random* is:

$$\frac{\#visits\ in\ the\ segment}{\#pages\ in\ the\ file} \leq 1.$$

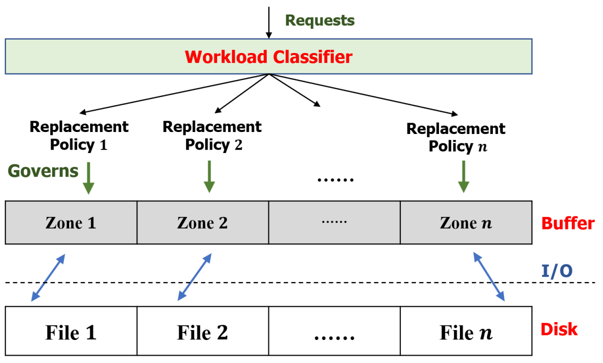

Fig. 3. The Structure of PBM.

As a result, the workload classifier based on *Correlation List* and *Page Coverage* is shown in Algorithm 1.

Algorithm 1: Workload Classifier

```
Input : seg_len: the segment length; $\epsilon$: the threshold for
        correlation; $\gamma$: the threshold for page coverage; $k$:
        the threshold for number of correlations below the
        threshold; $H$: the access sequence to be classified;
Output: the type of $H$
segment_list = segment($H$, seg_len);
page_coverage = the average page-coverage of all segments;
count = 0;
for $s_i$ in segment_list do
    if correlation($s_i$, $s_{i+1}$) $<\epsilon$ then
        count += 1;
    end
end
if count $> k$ then
    if page_coverage $>\gamma$ then
        return $C$;
    else
        return $A$;
    end
else
    return $B$;
end
```

C. Architecture and Algorithms of PBM

Figure 3 shows the architecture of PBM. First, we divide the buffer into zones. Each zone is responsible for a database file in the disk and is governed by its own buffer replacement policy. The choice of a replacement policy depends on the access pattern of the file. During the run time, the size allocated to each zone is variable. We group the five access patterns into three types. Type A includes *random* and *scan*. Type B consists of *cyclic* and *skewed*. Type C contains only the *vary* pattern. For a database file with Type A, since it will not contribute to the hit ratio under any policy, we shrink the size of its buffer zone. Thus, it will not pollute the buffer zone for other files. In our implementation, we make the zones for Type B managed by LFU and the Type C zones governed by ARC. The size of zones with Type B and C is dynamic. We use an evicting queue for each file to assess the buffer size it needs. The detailed PBM algorithm is as in Algorithm 2.

Algorithm 2: PBM

```
Input: B: the buffer; p: a page request;
1  adding_zone = B.find_zone(p);
2  if $p \notin$ adding_zone then
3      if $p \in$ adding_zone.evict_list then
4          if adding_zone is not Type A then
5              adding_zone.evict_list.remove(p);
6              adjust_size = True;
7          end
8      end
9      if B is full then
10         if adding_zone is full then
11             adding_zone.evict();
12         else
13             evicting_zone = find_full_zone();
14             evicting_zone.evict();
15         end
16     end
17 end
18 adding_zone.request(p);
19 if adjust_size then
20     B.adjust_size(adding_zone);
21 end
```

Each zone with Type B or C has an evict list and a weight. The evict list keeps the metadata of the page evicted from the zone, and the total number of pages contained in the zone and its evict list is the total buffer size. At lines 11 and 14 of Algorithm 2, the *evict()* function evicts the page from the zone, records it at the head of the evict list, and evicts the tail element in the evict list if the total size exceeds the upper bound. When a missing page is in the evict list, we increase the weight of the zone, thus increasing its size. This is a better way to allocate space than only considering the file size or the number of visits. Because even though a file is large and more frequently accessed, the working set of it may be small, which means it requires less buffer space. But finding the missing page in the evict list shows that it is possible for the added space to be used for working set. Also, when we generate a new size allocation, we simply reset the size of each zone but do not make eviction immediately. When a miss occurs, and the buffer is full, we find a zone that is full or has overflowed to make eviction. The function $find_full_zone$ returns a zone with Type A first. The *AdjustSize* function invoked by Algorithm 2 is shown in Algorithm 3.

Algorithm 3: AdjustSize

```
Input: $B$: the buffer; $f_k$: the size of file $k$; $F$: the total size
       of the database; $w_k$: the weight of the zone $k$; $S$: the
       available size for zones of Type B and C; $s_k$: the size
       of zone $k$; $i$: the id of the adding_zone;
1  for every zone k with Type B or C do
2      $w_k$ *= $\sum w_j/(\sum w_j + 1)$;
3  end
4  $w_i$ += $\sum w_k/(\sum w_k + 1)$;
5  for every zone k of Type B or C and $k \neq i$ do
6      $s_k = (w_k * S) / \sum w_k$;
7  end
8  $s_i = S - \sum s_k (k \neq i)$;
```

IV. Performance Evaluation

In this section, we compare PBM to several replacement policies, including LRU, LFU, and LeCaR. We mainly focus on two metrics: hit ratios and page I/Os.

A. Setting

We run all experiments on a database consisting of ten files, each of which contains pages whose page number ranges from 0 to 100,000 (100k). Therefore, the database consists of 100k pages totally. To simulate the different access patterns of the files, we manually make each file have different access patterns. Each file contains 5k or 15k pages. Table I shows the details of each file used in the experiment. Note that each access pattern is associated with two files, with one file containing 15k pages and another file containing 5k pages. The default buffer size is set to 1,024 pages.

We generate 2500k page accesses for all the files. Further, we prepare two workloads based on the 2500k requests, namely a regular workload and a shifting workload.

(1) *Regular Workload.* In this workload, we distribute the page accesses to each file uniformly, i.e., each file receives 250k page requests. The requests to each file follow the access pattern of the file. For example, the 250k requests to file 1 satisfy the cyclic access pattern.

(2) *Shifting Workload.* In this workload, the page accesses to each file are skewed. To be more specific, we first participate the total 250k page requests into two parts, each of which contains 125k requests. Then, we let the 80% of the first half accesses focus on files 1 to 5 and the remaining 20% on files 6 to 10. Thus, in the first half requests, files 1 to 5 will be heavily accessed, but files 6 to 10 are not. For the second half 125k requests, we use the opposite setting, which is to make the 80% of the second half accesses focus on files 6 to 10 and the remaining 20% on files 1 to 5.

B. Performance on the Regular Workload

In this experiment, we compare the proposed PBM with existing buffering policies on the regular workload. Figure 4(a) shows the comparison of the hit ratios of PBM and other three existing buffering algorithms, including LRU, LFU, and LeCaR. Figure 4(b) shows the I/O comparison among all the

TABLE I

DESCRIPTION OF THE SYNTHETIC WORKLOAD.

	Page-ID Range	File Size (pages)	Access Pattern
File 1	0-15k	15k	cyclic
File 2	15k-20k	5k	cyclic
File 3	20k-35K	15k	scan
File 4	35k-40k	5k	scan
File 5	40k-55k	15k	skewed
File 6	55k-60k	5k	skewed
File 7	60k-75k	15k	random
File 8	75k-80k	5k	random
File 9	80k-95k	15k	vary
File 10	95k-100k	5k	vary

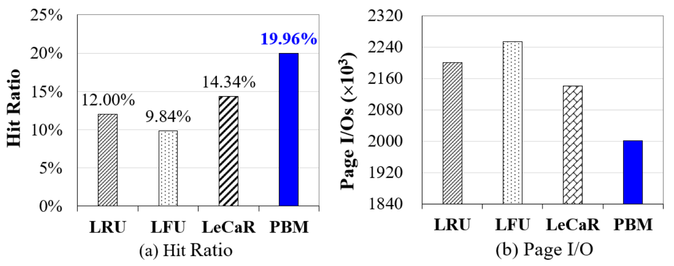

Fig. 4. Performance Comparison on the Regular Workload

compared buffering schemes. We can see that our proposed PBM achieves the highest hit ratio and the lowest page I/Os, owing to its dynamical algorithm selection according to access patterns. Particularly, PBM achieves 1.66x, 2.03x, and 1.39x hit-ratio improvements compared to LRU, LFU, and LeCaR, respectively. In addition, PBM reduces up to 11% page I/Os compared to its competitors. Note that LeCaR also shows good performance because it can adapt to access patterns. However, it only considers the recency and frequency of accesses and cannot detect other access patterns like *cyclic* and *vary*.

C. Performance on the Shifting Workload

In this experiment, we compare the proposed PBM with existing buffering policies on the shifting workload. Figure 5(a) shows the comparison of the hit ratios of PBM and other three buffering algorithms. Figure 5(b) shows the I/O comparison among all the compared buffering schemes. Compared with the experimental results on the regular workload, we can see that PBM achieves much more improvements over the three existing schemes. In particular, PBM achieves 1.90x, 1.55x, and 1.49x hit-ratio improvements compared to LRU, LFU, and LeCaR, respectively. Also, PBM reduces up to 14% page I/Os compared to its competitors. The higher performance of PBM on the shifting workload than on the regular workload is owing to its adaptivity, which can change the buffering scheme of each file according to the workload change.

To demonstrate the adaptivity of PBM more clearly, we calculate the average number of buffered pages for each file when performing the 250k page requests. As the first 125k requests in the shifting workload are focused on files 1 to 5, we expect that PBM can cache more pages of files 1 to 5 in the buffer. On the other hand, when running the second 125k requests that are focused on files 6 to 10, we expect that PBM can quickly adapt to the change of the access pattern and maintain more pages of files 6 to 10 in the buffer. As shown in Fig. 6, we can see that PBM can always keep more hot pages in the buffer when the workload changes with time.

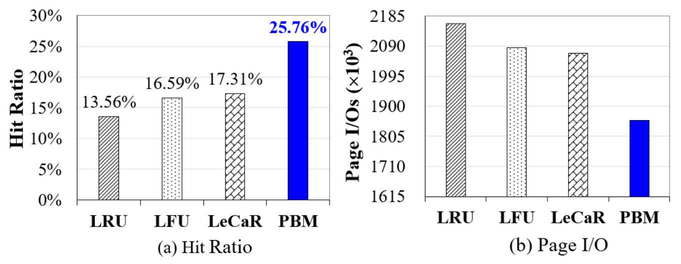

Fig. 5. Performance Comparison on the Shifting Workload

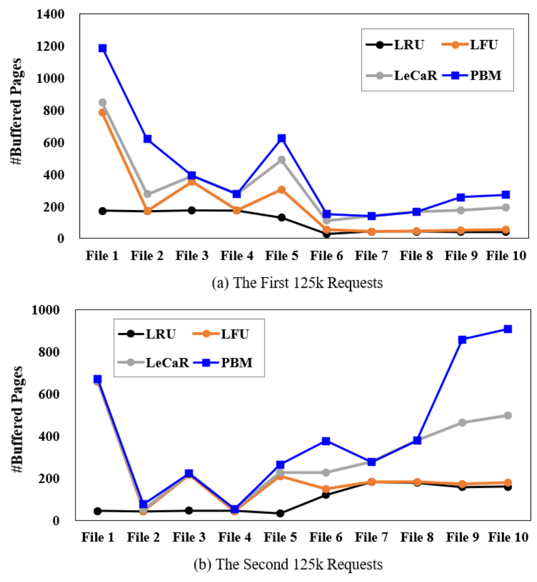

Fig. 6. The Change of the Buffered Pages When the Workload Shifts.

D. Impact of the Segment Length

Segment length is a very important parameter for the classifier. The longer the segment is, the closer the histogram vector is to the distribution of the workload. However, a long segment can be bad for detecting the *vary* pattern, because when we calculate the histogram of several phases, the frequency of the hot zone is amortized, and the overall distribution resembles the *random* pattern. In this experiment, we test the influence of the segment length on the classifier. As shown in Fig 7, when we choose a long segment length, the gap between Type A and Type B is widened, meaning that the histogram of a long segment length can reflect the distribution better. In addition, a long segment length does not impede the separation of Type C, because both the *vary* pattern and the *random* pattern have a small correlation.

E. Comparison of Similarity Measures

The page classifier used in PBM employs the correlation-based approach. In this experiment, we consider other possible

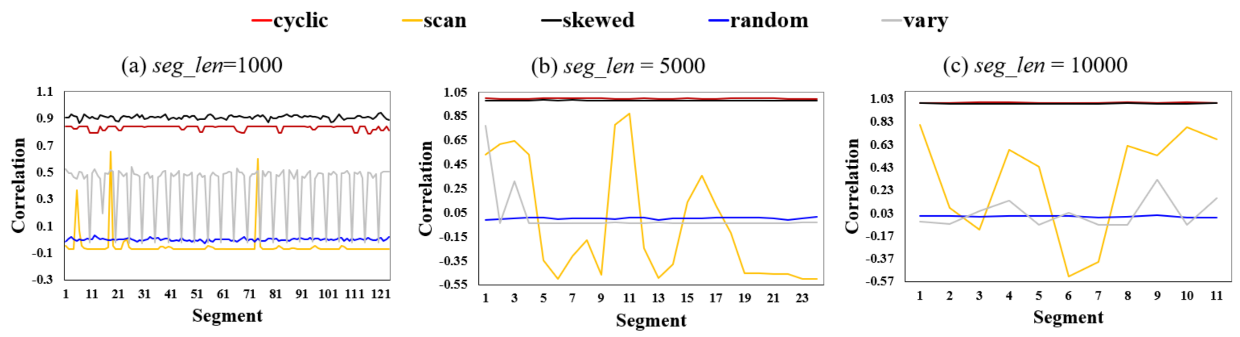

Fig. 7. The Impact of the Segment Length

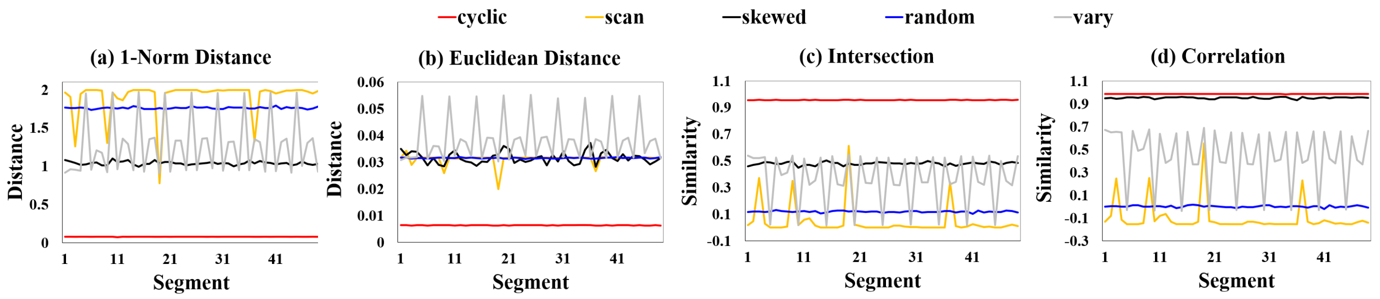

Fig. 8. Comparison of Different Similarity Measures

classifying metrics, aiming to show the superiority of the correlation-based classifier.

In addition to the correlation approach, we implement other three methods, which as listed as follows:

(1) 1-*Norm Distance*. This refers to the distance based on the 1-norm in a linear space.

(2) *Euclidean Distance*. This is the Euclidean distance between two vectors.

(3) *Intersection*. Given two vectors, the intersection between the vectors is defined as the sum of the smaller value for each element in the vectors, as shown in Equation 1.

$$Intersection(h_1, h_2) = \sum_{i}^{n} min(h_{1i}, h_{2i}) \tag{1}$$

Figure 8 shows the similarity of each metric when used for classifying the five access patterns. We can see that the 1-Norm Distance, Euclidean Distance, and Intersection all fail to classify the five access patterns clearly. Compared to the Correlation method, all the three methods cannot distinguish the *cyclic* from the other four patterns, which shows the superiority of the correlation approach proposed in PBM.

V. Conclusions

In this paper, we presented a new access-pattern-aware buffer manager called PBM (Personalized Buffer Manager). PBM can detect the access patterns for each database file and use different buffering policies for database files. We proposed a workload classifier to detect the access pattern and partitioned the buffer into different zones for different files. Each zone uses its own buffering policy for a database file, yielding a personalized buffer manager. We implemented PBM and compared it with three existing buffering algorithms on two workloads, including LRU, LFU, and LeCaR. The results suggested the efficiency of PBM.

In the future, we will investigate personalized buffer management for flash memory [7], [8] and use machine learning models to optimize the buffer management [9], [10].

Acknowledgements

This work was supported by the National Science Foundation of China (62072419). Peiquan Jin is the corresponding author.

References

[1] W. Effelsberg and T. Härder, "Principles of database buffer management," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 560–595, 1984.

[2] P. Jin, Y. Ou, T. Härder, and Z. Li, "AD-LRU: an efficient buffer replacement algorithm for flash-based databases," *Data & Knowledge Engineering*, vol. 72, pp. 83–102, 2012.

[3] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving cache replacement with ML-based LeCaR," in *HotStorage*, 2018.

[4] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *FAST*, 2003.

[5] T. Johnson and D. E. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of VLDB*, 1994, pp. 439–450.

[6] M. Stonebraker, "Operating system support for database management," *Communications of ACM*, vol. 24, no. 7, pp. 412–418, 1981.

[7] Y. Ou, T. Härder, and P. Jin, "CFDC: a flash-aware replacement policy for database buffer management," in *DaMoN*, 2009, pp. 15–20.

[8] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, "CCF-LRU: a new buffer replacement algorithm for flash memory," *IEEE Trans. Consumer Electron.*, vol. 55, no. 3, pp. 1351–1359, 2009.

[9] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in *HPCA*, 2021, pp. 291–303.

[10] Y. Yuan and P. Jin, "Learned buffer management: a new frontier: work-in-progress," in *CODES/ISSS*, 2021, pp. 25–26.