Learned Buffer Replacement for Database Systems

Yigui Yuan*
School of Computer Science and Technology, University of Science and Technology of China, China

Peiquan Jin
School of Computer Science and Technology, University of Science and Technology of China, China

ABSTRACT

Most current database buffering schemes adopt an empirical design, which cannot adapt to the change of workloads. In this paper, we show how we can use machine learning to help design a new buffer replacement policy for database systems. We name the new policy LBR (Learned Buffer Replacement). The key idea of LBR is to use machine learning models to periodically learn the access pattern from historical requests to make the buffer replacement adaptive to the workload change. Particularly, we present two ways to learn the access pattern. One is a classifier named LBR-c, which can distinguish hot pages from cold ones based on the training on historical requests; the other is a regressor called LBR-r, which can predict the future replacement behavior according to historical accesses. We implement the proposed LBR-c and LBR-r and compare them to a number of existing schemes, including the theoretically optimal Belady's algorithm, three traditional algorithms (LRU, 2Q, and ARC), and LeCaR, which is a recently-proposed adaptive buffer scheme. The results show that our algorithms achieve a higher hit ratio than LRU, ARC, 2Q, and LeCaR. In addition, both LBR-c and LBR-r can adapt to workload changes, which is better than LRU, 2Q, ARC, and LeCaR. Overall, our proposal achieves comparable performance with the optimal buffer replacement algorithm, advancing the state-of-the-art in the well-studied area of buffer management in DBMSs.

CCS CONCEPTS

• **Information systems**; • **record and buffer management**;

KEYWORDS

Learned buffer management, Deep neural networks

ACM Reference Format:
Yigui Yuan and Peiquan Jin. 2022. Learned Buffer Replacement for Database Systems. In *2022 the 5th International Conference on Data Storage and Data Engineering (DSDE) (DSDE 2022), February 25–27, 2022, Sanya, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3528114.3528118

1 INTRODUCTION

Buffer management is a key module in database systems to improve query performance [3]. 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 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) [3]. 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. Other widely-used buffering algorithms such as CLOCK [3] and ARC [11] adopt a similar idea. In a word, traditional buffer replacement algorithms use an empirical way to select the victim for replacement. 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 [6], CCF-LRU[8], and LeCaR [16], their performance relies on the empirical setting of parameters, which are still empirical solutions.

Recently, machine learning has been demonstrated as an efficient tool in many prediction tasks. For example, in machine translation [10], a machine learning model can learn the distribution of not only every single word but also word phrases, which is used to predict the result of sentence or document translation. Inspired by many successes of integrating machine learning with databases, e.g., learned indices [2, 8], this paper proposes integrating machine learning with buffer management in DBMSs. This study aims to answer the following two questions:

- Can machine learning be helpful for improving the performance of database buffer replacement?
- How to implement a learned buffer manager that can leverage the training cost and procedure of machine learning?

Aiming to address the above two questions, in this paper, we show how we can use machine learning to help design a new Learned Buffer Replacement (LBR) policy for database systems. The key idea of LBR is to use machine learning models to periodically learn the access pattern from historical requests to make the buffer replacement adaptive to the workload change. Particularly, we present two ways to learn the access pattern, including a classification-based way (LBR-c) and a regression-based approach

*yuan1gui@mail.ustc.edu.cn

DSDE 2022, February 25–27, 2022, Sanya, China

ACM ISBN 978-1-4503-9572-4/22/02…$15.00
https://doi.org/10.1145/3528114.3528118

(LBR-r). We experimentally demonstrate that our proposal is more efficient than a couple of traditional buffering schemes. Briefly, we make the following contributions in this paper.

- We propose to use machine learning to improve the performance of buffer management and present a new learned buffer replacement scheme based on a Transformer model. To the best of our knowledge, this study is the first one that addresses the learned buffer management in the database community. Although there are a number of machine learning models, Transformer has been demonstrated as an efficient machine learning model that performs well in time-series prediction tasks. Thus, we believe that Transformer is appropriate to model and predict the access pattern in buffer management.
- We present an embedding approach to capture the features of buffer requests and propose a data-oriented and a behavior-oriented replacement scheme based on feature learning. For the data-oriented replacement, we develop a classifier-based algorithm called LBR-c that classifies pages into hot and cold ones according to historical requests. Based on the classification result, we evict the coldest page in the buffer as the victim. For the behavior-oriented replacement, we implement a regression-based algorithm called LBR-r that can predict the reuse distance of each page when it is accessed. Then, we select the page with the longest distance as the victim.
- We implement the proposed LBR-c and LBR-r algorithms and compare them to a number of existing schemes, including the theoretically optimal Belady's algorithm, three traditional algorithms (LRU, 2Q, and ARC), and one recently proposed adaptive buffer scheme named LeCaR. The results show that our algorithms achieve a higher hit ratio than all competitors and achieve comparable performance with the optimal buffer replacement algorithm.

The rest of the paper is structured as follows. Section 2 introduces the background and related work. Section 3 details the design of the learned buffer replacement scheme. Section 4 reports the experimental results, and finally, in Section 5, we conclude the paper and discuss future work.

2 BACKGROUND AND RELATED WORK

2.1 Buffer Management in DBMSs

In the past two decades, many well-known buffering policies have been proposed, e.g., LRU, ARC [11], and LIRS [5]. Most of these algorithms have been well explained in textbooks. The literature [3] presents a good survey on traditional buffer management policies.

Least Recently Used (LRU) [3]. LRU 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 in the case of reference patterns having high temporal locality, i.e., currently referenced pages have a high re-reference probability shortly. But LRU also has severe disadvantages. First, it only considers the recency of page references and does not exploit the frequency of references. Second, LRU is not scan-resistant, i.e., a sequential scan operation pollutes the buffer with one-time referenced pages and possibly evicts pages with higher re-reference probability.

Adaptive Replacement Cache (ARC) [11]. ARC 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 cannot capture the full frequency distribution of the workload and perform well for LFU-friendly workloads. For a scan workload, new items that go through T1 will protect frequent items previously inserted into T2. However, for churn workloads, ARC's inability to distinguish between equally important items leads to continuous cache replacement [13].

Low Interference Recency Set (LIRS) [5]. LIRS is a state-of-the-art caching algorithm based on reuse distance. LIRS handles scan workloads well by routing one-time accesses via its short filtering list Q. However, LIRS's ability to adapt is compromised because of its use of a fixed-length Q. In particular, if reuse distances exceed the 1% length, LIRS is unable to recognize reuse quickly enough for items with low overall reuse. And similar to ARC, LIRS does not have access to the full frequency distribution of accessed items which limits its effectiveness for LFU-friendly workloads.

2Q [7]. The 2Q algorithm introduces parallel buffers and supporting queues to manage the buffer. Instead of considering just recency as a factor, 2Q also considers access frequency while making the decision to ensure the page that is really warm gets a place in the LRU cache. It admits only hot pages to the main buffer and tests every page for a second reference. The golden rule that 2Q is based on is - Just because a page is accessed once does not entitle it to stay in the buffer. Instead, it should be decided to keep it in the buffer if it is re-accessed.

2.2 Machine Learning for Cache Replacement

So far, to the best of our knowledge, there is no study focusing on the learned buffer replacement for DBMSs. However, in other areas, such as computer architecture, some related studies focus on improving the CPU cache efficiency by introducing machine learning models.

The recent LeCaR algorithm [16] is a unique 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 [16]. 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, including the state-of-the-art methods like ARC, which was designed over 18 years ago [11]. The LeCaR algorithm is an online reinforcement learning algorithm that relies on only two fundamental cache replacement policies typically taught in an introductory Operating Systems class, namely the LRU policy and the Least Frequently Used (LFU) policy. LeCaR assumes that the best strategy at any given time is a probabilistic mix of the two policies and attempts to "learn" the optimal mix using a regret

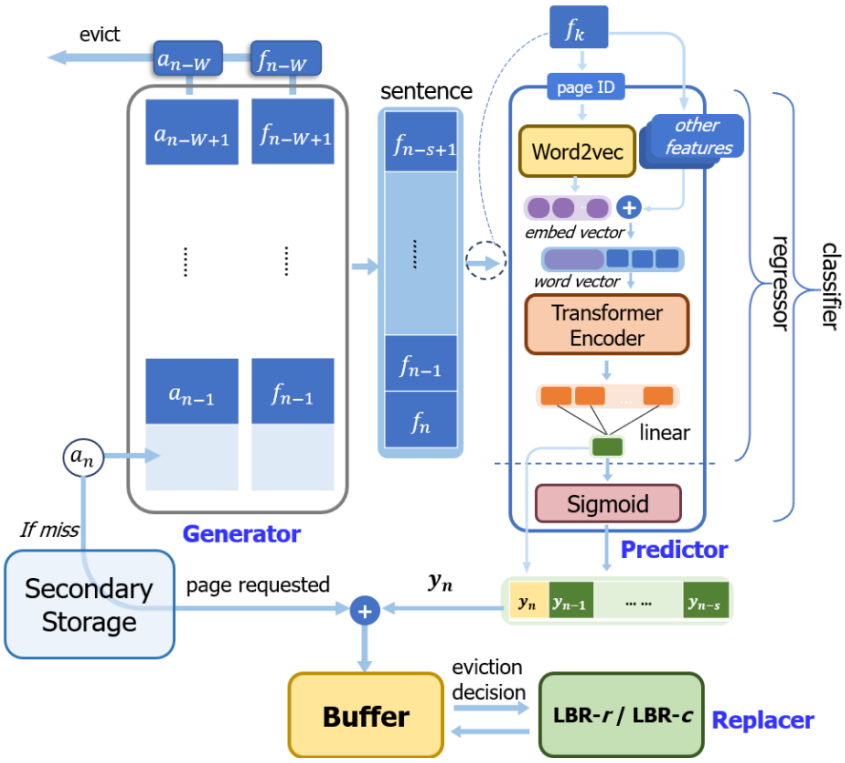

Figure 1: Architecture of the Learned Buffer Replacement.

minimization strategy. Following the idea of LeCaR, there are some recent studies that also concentrate on optimizing the CPU cache management by using machine learning techniques [12, 14, 15]. For example, a recent work [17] used machine learning to improve the cache efficiency in LSM-tree. However, all the previous works were either towards CPU cache management or LSM-tree caching issues, which is different from the research focus of this study, i.e., learned buffer replacement for database systems.

However, due to the successful adoption of LeCaR in CPU cache replacement, in this paper, we also consider transforming LeCaR to database buffer management. Therefore, we re-implement LeCaR to make it run as a database buffer and take it as a competitor in our experiment. Our experimental results will show that our proposal outperforms LeCaR. The details will be discussed in the experiment section.

3 LEARNED BUFFER REPLACEMENT (LBR)

3.1 General Idea

Buffer replacement is essential for predicting the future workload. We suppose that the prediction about page accesses should not be merely based on the past access behavior of the page itself but also the past accesses of other pages coming along with it. This is a reasonable supposition because the usage of a page depends on higher-level tasks, and these tasks usually involve more than one page. This resembles a word prediction problem in NLP (Natural Language Processing), where the words are not independent, and information is contained in the combination of words. Our motivation is to adopt Transformer, a network widely used in Translation and word embedding problems.

The transformer is a sequence-to-sequence structure with an encoder and a decoder. For regression of classification purposes, we only need the encoder part. The encoder is a stack of N identical encoder layers. Each layer processes an input sequence and feeds its output to the subsequent layer or the decoder if it is the last layer. Unlike other sequence-to-sequence models such as recurrent neural networks (RNN) that process a sequence word by word, the encoder layer of Transformer uses a self-attention network to process the sequence in parallel. This speeds up the calculation of the network.

As shown in Figure 1, our model consists of a Generator, a Predictor, and a Replacer. The Generator transforms accesses to vectors of features. It keeps a window that stores W most recent accesses and their features. When a current access a_n comes it uses the past information to calculate the feature f_n for this new access. The Generator outputs a sequence of features for every access. This will be the input of the predictor. Though the whole sequence is processed at the same time, to demonstrate the process inside the predictor better, we focus on the changes that happened to one single feature vector fk. fk contains a categorical feature page ID and three other numerical features. The page ID will first get embedded by a word2vec model, producing an embedded vector of size E. The other three features are then appended to the embedded vector and become a word vector for each access. The word vector

goes through a 6-layer Transformer encoder and becomes an output vector of size *E+ 3*. We try to replace the Transformer with a Long Short-term Memory (LSTM) network, which does not affect the predictor much. We only choose Transformer over LSTM for the faster training and predicting process. After the Transformer layer, we use a linear mapping to downsize the output to a float point value. If we are using the classifier model, the output will go through an extra sigmoid layer to be scaled to [0, 1]. Now every access in the sequence has a prediction. If the current access is a miss, as in Fig.1, then a page will be read from the second storage and goes into the buffer labeled by y_n. If it is a hit, then the label of the page accessed is n updated. And the replacer will interpret the prediction, use it as the criteria, and finally decide eviction.

3.2 LBR-c: Classifier-based Replacement

First, we propose to view a buffer replacement problem as a binary classification. The classification criteria come from Hawkeye [4]. According to these criteria, every access is either to cause a hit, that is, the requested page will be hit in the future, or to cause a miss. We label a "1" for the accesses that are expected to cause a hit and a "0" for those causing a miss. Note that the binary classification depends on future requests. It is a prediction of future requests but not a classification of historical requests. For instance, in Fig. 2, access 4 will be labeled "0", indicating it will cause a miss in the future, even though it is a hit under the Belady's policy. As mentioned before, the label will be attached to the page accessed, until the page is accessed again or the page leaves the buffer. Note that the event of hit or miss is not only decided by the buffer size and the workload but also the replacement policy. In Fig. 2, for instance, if we are using the Belady's policy, then access 4 is a hit. Thus, access 1 is causing a hit and will be labeled "1". Instead, if we are using LRU, access 4 will be a miss, and access 1 will be labeled "0". For every policy, we can get a labeling function shown in Eq. 1. Here, π is a replacement policy, a is an access, and $\omega = \{a1, a2, \ldots, an\}$ is the workload. $a \in \omega$.

$$\delta : \pi \rightarrow Label\pi(a|Buffer, \omega) \tag{1}$$

According to the definition, a page with the label "1" will not be replaced before it is accessed again. If it is accessed, but the label remains "1", it will not be replaced until the next access. As a result, pages labeled "1" will never be evicted out of the buffer. Moreover, page eviction always happens among the pages labeled "0". We can say that the label function indirectly reflects the behavior of the replacement policy π.

Based on the labels generated, a new replacement policy $\pi*$ is defined as follows: Given a workload ω and a trained label set $l = (Label\pi(a1), Lable\pi(a2), ..., Label\pi(an))$, we keep two lists, L_1 and L_0, for the buffer. An accessed page is put into L_k if the estimated label of its access in l is k. The evictions always happen in L_0, unless L_0 is empty. In $\pi*$, the order of evictions can be arbitrary. Thus, though both *pi* and *pi*$*$ evicts 0-labeled pages before 1-labeled pages, they may evict them in a different order.

Theorem 3.1. *If two buffers B and B$*$ have the same size and content, π is a replacement policy, Labelπ is the labeling function, l is the label list of the workload ω, then at any time i, the* $1-$*labeled list in B under π equals the* $1-$*labeled list in B$*$under $\pi*$.*

Proof. We can prove the above conclusion in a weaker condition. Instead of asking for the same content, we only need the L_1 to be the same at time 0. Suppose that a_1 is a hit and is labeled "1", then the two buffers' contents remain the same, and so do the two lists of L_1. If a is a hit and is labeled "0", then both lists of L_1 lose the same page to the corresponding lists of L. If a_1 is a miss and is labeled "0", both lists of L_1 are unchanged. If a_1 is a miss and labeled "1", the same page is added to both lists of L_1. Therefore, both lists of L_1 remain the same at time 1. By iterating the time from 2, we can conclude that at any time i, both lists of L_1 are equivalent.

Theorem3.2. Given the same conditions of Lemma3.1. The number of hits under $\pi*$ is no less than that under π.

Proof. We only need to prove that for each 1-labeled page in the buffer, it will be hit under $\pi*$. If there is one page in L_1* evicted before hit, since it is not accessed before eviction, it is always labeled "1". The only condition that $\pi*$ will evict a page from L_1* is when $L0*$ is empty. So, the access a_1 causing eviction happens when L_0* is empty. This means $B* = L_1*$. Since B and $B*$ share the same size and by Lemma 3.1, $L_1 = L_1*$ is always true, we know that $B = L_1$ as well. When the buffer is full of 1-labeled pages, accesses will always be hits. Thus, a_1 is a hit and cannot cause an eviction (contradiction).

Our replacer in LBR-c can be thought of as *Belady*$*$. We simulate the Belady's policy and generate a label list l for historical accesses, which is used as the label set for training. After training the Transformer network for classification, we take the Transformer's output as the label. Then, the replacer can manage the buffer according to the labels. The general process is described in Algorithm 1

Algorithm 1 LBR-c Replacer

Input : Access a, page ID k of page it accesses and the label of the access l.

```
if k in L1 then
    L1.pop(k);
end
else if k in L0 then
    L0.pop(k);
end
else if B if full then
    /* A miss occurs */
    if L0 is not empty then
        L0.deleteLRU();
    end
     else
        L1.deleteLRU();
  end
end
if l == 1 then
    L1.append(k);
end
else
    L0.append(k);
end
```

3.3 LBR-r: Regressor-based Replacement

Compared to LBR-c, LBR-r algorithm is much more straightforward. The predictor is trained to predict the next visit time for each page. And the replacer chooses the page with the largest next visit time as the victim. But the next visit time grows with the time. For example, in a periodic workload $w = \{a, b, c, a, b, c, a\}$, access 1 and access 4 is exactly the same in meaning, but the next visit time of them are 4 and 7. Though the relative position in the access stream can indicate this difference, we lose this information when we cut the access into small sentences. Instead of predicting the next access time directly, the LBR-r predictor predicts the distance D from current access to the next access. Actually, to be compatible with the normalized feature that will be discussed in the next section, we use "normalized" distance as the goal. That is $d = \frac{D}{W}$, where W is the length of a sliding W window used in feature generation. And we keep both the access time t and the predicted distance for each page in the buffer. The next visit time can be calculated as $T = W \cdot d + t$. The replacer of LBR-r is shown in Algorithm 2

Algorithm 2 LBR-r Replacer

Input : Access *a*, *page ID k* of the page it accesses and the label of the access *l*.
Set the queue L to empty and count = 0;
/* nxt is the next visit time of a */
nxt = 1 * W + count;
if *k in B* **then**
    /* A hit occurs */
    B.pop(k);
end
else if *B is full* **then**
    /* Evict page with the largest nxt */
    victim = B.FindLargestKey();
    B.pop(victim);
    B.insert(k, nxt);
    count ++;
end

3.4 Feature Learning

In a Transformer network, the input sequences contain the same amount of word vectors. Here, our sentence is formed of accesses. In addition, we need to find a vector representation of the access.

We consider several features that either reflect the access pattern of a certain page or represent the higher-level meaning of the access. The detailed features are described as follows:

- page ID. This identifies the page it accesses. If there are page_num pages in the second storage, then each page can be represented by an integer from 0 to *page_num*−1. This feature is useful in a scenario that the content of a page is mostly fixed, so that there is a stable mapping between page ID and page content. We do not expect the network to learn the map itself, but we hope it to discover the relationship among different pages, demonstrated by the relationship among different page IDs. For example, access to page *a* (*a* is the page ID, an integer) could indicate future accesses to pages *b*, *c* with similar content. The network will constantly see a, b, c showing up in short distances and therefore learn that these page IDs are closely related. However, in a database where the content of a page is dynamically changing, the relationship among page IDs would be elusive and unpredictable. Some work uses the delta of addresses of consecutive pages instead of straight page ID to cope with this. But that is not the main concern of this paper.
- frequency. The access frequency of the page accessed is the feature LFU focuses on. A rise in the frequency of one page could mean more visits to it and its related pages described above. When a page is accessed, we count the times it gets visited in the last w accesses. Here, w is the size of the window for calculating frequency.
- reuse distance. We call it *rd* for short. Now *rd* is the number of accesses between current access and the last access to the same page. This is the feature LRU focuses on. When a page is accessed, we search for the nearest visit in the last w of it and calculate the distance. Here, w is the size of the window for calculating rd. If no preceding visit to the page is found in the window, the rd of the page is set as W.
- penultimate reuse distance (*prd* for short). *prd* is used in 2Q and its modifications. This feature has been used for a buffer to acquire scan resistance. We simply use the *rd* of the former access of the same page in the window as the *prd* of current access. If no former access of the same page is found, the *prd* is set as 0.

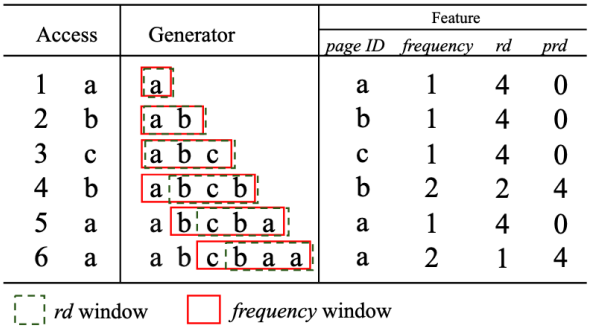

Figure 2: Features for access sequences ($W = 4, w = 3$).

An example of feature generation is illustrated in Figure 2

For the first access to page a, there is no preceding accesses. So, the frequency is 1, the $rd = W = 4$, and the *prd* is 0. For the last access to page *a*, one preceding visit to a exists. Thus, the frequency count is 2. The *rd* is 1, and the *prd* is the same as the *rd* of the preceding access, which is 4.

Normalization is needed before we put all these features into the Transformer network. For $frequency$, rd, prd, this can be done by dividing them by their window sizes, respectively. But for *page ID*, we need to take another method because this is not a numerical feature and lacks some sort of continuity. Naturally, we think about applying word embedding. There are many ways to implement embeddings, like one-hot or word2vec. One-hot is adequate for the purpose of normalization, but the length of the embedding vector grows with the category number, here the *page_num*. Word2vec generates embedding vectors that reflect the relationship among

the words. Also, the embedding size of word2vec is much smaller than the category number.

As shown in Figure 2, suppose we choose to use one-hot embedding, then the input vector will be (1, 0, 0, 0.5, 0.25, 1). The former three is the embedding, and the latter three is normalized $frequency$, rd, prd.

4 PERFORMANCE EVALUATION

4.1 Experiment Setup

4.1.1 Workloads. We evaluate LBR on several handmade workloads. The total page number is 10000, and each workload contains 1 million accesses. We use the former half to train our predictor and the latter half as the test set. The workloads are formed by several fragments. There are four types of fragments, pattern, zipf, scan and random. A pattern fragment is a fixed-length sequence of accesses. These accesses follow a uniform distribution and are selected before the generation of workload. Similarly, a zipf fragment is of preassigned length, but its accesses follow a zipf' s distribution and are selected during the generation of workload. A scan fragment is a fixed length of serial accesses starting from a random position. The length and the content of a random fragment are both decided during the generation of a workload. The length of a random fragment follows a Normal distribution. When generating a workload, we choose fragments by certain probability repeatedly until the total amount of accesses reaches 1 million. We list the probability and the length of the fragments for each workload below:

- *period.* In this workload, a 1500-accesses pattern fragment will appear with probability 0.9. The pattern length is slightly bigger than the largest buffer size in this experiment, which is 1024 pages, because a very long pattern is basically the same with total random accesses, thus is unpredictable. Also, a pattern shorter than the buffer size will only cause hit, so it is not fit for the test either. To make the workload more real, we also add some irregular scans and random accesses. Apart from the pattern fragment, scan of length 1100 and random with average length 512 appear with the probability 0.05.
- *multi.* This workload has two pattern fragments, of length 1100 and 1050, respectively. These two fragments have shared former 600 accesses. The 1100 pattern is assigned probability 0.6, and the 1050 one is assigned 0.3.
- *dynamic.* This workload also contains two pattern fragments of length 1500. But this time, the probability of the two fragments varies with time. If the current length of generated workload is x. then the probability of pattern 1 is $0.495(1 + \sin(\frac{x}{2\pi T}))$, and the probability of pattern 2 is $0.495(1 - \sin(\frac{x}{2\pi T}))$. The period T of the probability is around 10000, which means in train set $2\pi T$ the predictor can see this changing about 5 times.
- *unshuffled.* This workload has no pattern fragment. Instead, it has a 1500-accesses zipf fragment with probability 0.9. In this zipf fragment, page k appears with probability $\frac{1}{k^\alpha}/Z$, $Z = \sum_{i=1}^{10000} \frac{1}{i^\alpha}$ and $\alpha = 0.9495$. This α is the one that makes 80% of the accesses happen on 20% of the pages.
- *shuffled.* The only difference between this workload and the previous one is that this zipf fragment is shuffled. The pages are rearranged with a permutation τ. Page k's probability is $\frac{1}{\tau(k)^\alpha}/Z$.

4.1.2 Baselines. We evaluate LBR-r and LBR-c against the following replacement policies, including the theoretically optimal Belady's algorithm [1], LRU, 2Q [7], ARC [11], and LeCaR [16]. The optimal Belady's algorithm is to select the theoretical optimal page as the victim. It is also known as OPT or the Clairvoyant replacement algorithm. It works as follows: when a page needs to be replaced, the DBMS evicts the page whose next use will occur farthest in the future. Note that the optimal algorithm cannot be implemented in real DBMSs because it is not possible to know how long it will be reused before a page is going to be used. However, the optimal algorithm can be regarded as the up-bound of buffer replacement.

4.1.3 Training. In this section, we describe the training of models. The learning rate is adjusted following the schedule in [9] and the maximum training epoch is 28. The regularization is 0.02. To transform the float point value output to 0/1 label, we need to set a threshold for the classifier. Since the training set can be unbalanced, we choose a threshold that maximizes the F-score. F-score is a weighted average of recall and precision of the classifier and is often be used to evaluate a classifier's performance on an unbalanced set. We calculate the F-score by Eq. 2.

$$f = \frac{1 + \beta \cdot p \cdot r}{\beta \cdot p + r} \tag{2}$$

p is precision, r is recall, and $\beta = negative/positive$. Parameters for baselines. The Kin and Kout of 2Q are set 25% and 50% of the buffer size as suggested in [7]. For LeCaR, the initial learning rate is 0.45, and the initial discounting rate is $0.005^{1/\ buffer\_size}$ as suggested in [16].

4.2 Comparison of Hit Rates

Figure 3 shows the hit rates of LBR-r, LBR-c, and all baselines when varying the buffer size from 256 to 1024 pages. When our LBR policy achieves comparable performance with the optimal Belady's policy, we can see that. Both LBR-c and LBR-r outperform traditional buffering schemes, including LRU, 2Q, and ARC. In addition, they also perform better than LeCaR. The high hit rate of LBR on the workload dynamic shows that LBR can adapt to the workload change, which answers the first question we present in the introduction part, that is, machine learning does help to improve the hit rate of the database buffer manager.

Figure 3 also shows that LBR-c is more sensitive to the buffer size than LBR-r. This is because LBR-c adopts a classification model, which is highly impacted by the imbalance of the training set. On the other hand, LBR-r is less affected by the buffer size because it uses a regression model. Overall, LBR-r performs better than LBR-c on most workloads. Especially when running on the workload pattern, LBR-r achieves a hit rate that reaches over 70% of the hit rate of the optimal Belady's algorithm.

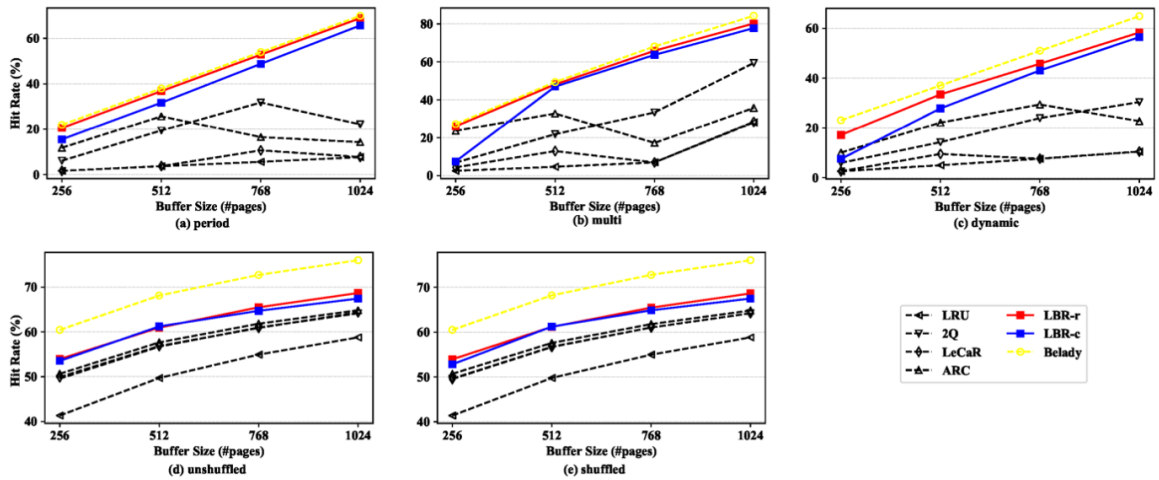

Figure 3: Comparison of the hit rate of LBR and baselines.

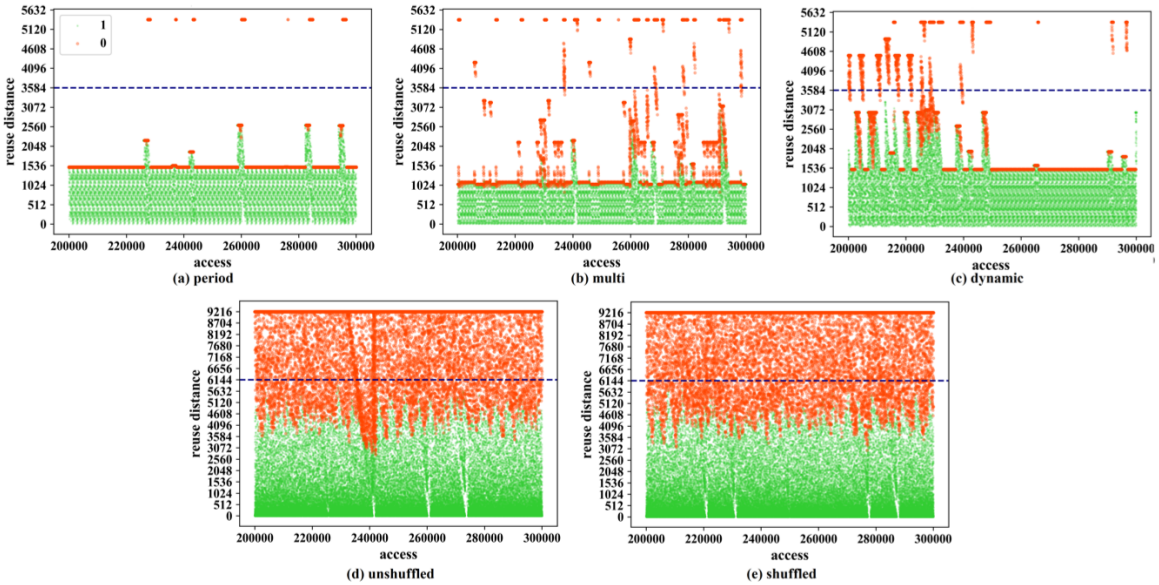

Figure 4: Reuse distance under various workloads (buffer_size = 512).

4.3 Reuse Distance

The Generator in our framework shown in Figure 1 has two parameters, namely the window size W for rd and the window size w for frequency. These two parameters affect the accuracy of the features. For example, rd larger than W will not be distinguished by the predictor. If W is too small, the feature rd will fail to capture the difference among pages with large reuse distances. Luckily, what we discussed in Section 3.3 has indicated that the replacing order of 0-labeled pages will not deteriorate the hit rate. So, we do not need to distinguish two pages if their rds are larger than some upper bound so that they will both cause misses. In other words, all accesses labeled 1 has rd smaller than this upper bound. This upper bound depends on the size of the buffer and the workload. A larger buffer is likely to contain pages for a longer time. And if the workload has too many repeated accesses, the upper bound can be larger.

Figure 4 shows the reuse distance (rd) of 1-labeled accesses, which is marked green in the figure. We can see that for all workloads, the rd of a 1-labeled access will not exceed 12 times the buffer size (for workloads with fewer repeating pages such as *period, multi* and *dynamic*, this upper bound can be smaller). Therefore, we set W as 12 times the buffer size by default in all experiments.

In addition, since there is no need to distinguish pages with rd larger than W, we might as well apply a cutoff when generating the training labels for $LBR - r$ This can benefit the training in two ways. First,, the label is closer to the feature, making the learning

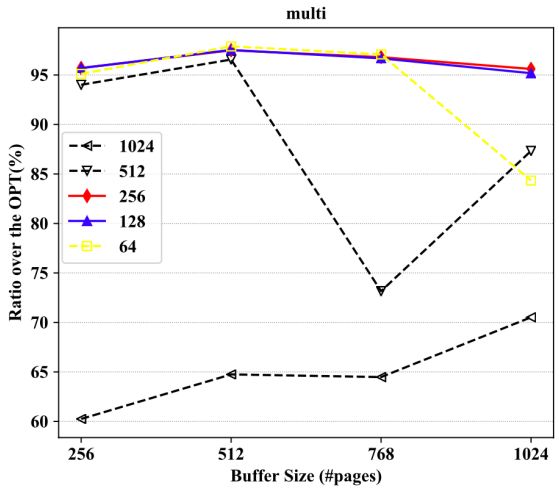

Figure 5: Performance of LBR-r with different input sizes.

progress easier; Second, it frees the predictor from fitting those sharp spikes and makes the difference between pages with small rds more significant. Unlike the rd window size, the frequency window size w can be chosen smaller. Because large w will average the changes of frequency, thus neglecting the changes of the workload distribution. According to the experiment, w does not greatly affect the result, and we use $w = buffer\_size$ for all experiments.

4.4 Impact of the Input Size

The input size of the transformer encoder corresponds to the embedding size of the word2vec layer. The embedding size is related to the vocabulary size, i.e., the total page number. It also reflects the complexity of the relationship among different words, i.e., *page ID*. To measure the impact of the input size on the performance of LBR, we vary the input size and report the performance in Fig. 5. In this experiment, we use the multi workload. We can see that all input sizes from 64 to 256 have a similar effect on the performance of LBR-r. The figure also indicates that an embedding size of about 128 is sufficient for all workloads. Therefore, we set the input size to 64 and the embedding size to 61 in all experiments.

The network in LBR-r and LBR-c is structured as in Figure 1. The word2vec layer only embeds the *page ID*, and the embedded vector is concatenated to the other three features. Together they form the "word vector" that represents an access. We set the embedding size to be 61 so that the input size is 64. We have tried a larger embedding size, and it does not improve the performance of the predictor. The length of the access sequence that we put into the Transformer encoder is 1024. We use a 6-layer 8-headed Transformer encoder, and the hidden size of the feed-forward layer is 1024. At last, the output of the encoder is reshaped to size 1 by a linear network. In LBR-c, the output needs to go through an extra sigmoid function to be scaled to [0, 1].

5 CONCLUSIONS

This paper shows how we can use machine learning to help design a new buffer replacement policy for database systems. We name the new policy LBR (Learned Buffer Replacement). The key idea of LBR is to use machine learning models to periodically learn the access pattern from historical requests to make the buffer replacement adaptive to the workload change. Notably, we present two ways to learn the access pattern. One is a classifier, named LBR-c, which can distinguish hot pages from cold ones based on the training on historical requests; the other is a regressor called LBR-r, which can predict the future replacement behavior according to historical accesses. Finally, we implement the proposed LBR-c and LBR-r and compare them to several existing schemes. The results suggested our proposal's effectiveness in increasing the buffer manager's hit rate.

In the future, we will consider using other machine learning models in our framework. In addition, we will use different kinds of workloads to verify the performance of our proposal.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation of China (62072419). Peiquan Jin is the corresponding author.

REFERENCES

[1] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for Virtual-Storage Computer. IBM System Journal 5, 2 (1966), 78–101.
[2] Zhou Zhang, Peiquan Jin, Xiao-Liang Wang, Yan-Qi Lv, Shouhong Wan, Xike Xie. COLIN: A Cache-Conscious Dynamic Learned Index with High Read/Write Performance. Journal of Computer Science and Technology. 36, 4 (2021), 721-740.
[3] Wolfgang Effelsberg and Theo Härder.1984.Principles of Database Buffer Management. ACM Transactions on Database Systems 9, 4 (1984), 560–595.
[4] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In Proceedings of ISCA. 78–89.
[5] Song Jiang and Xiaodong Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In Proceedings of SIGMETRICS. 31–42.
[6] Peiquan Jin, Yi Ou, Theo Härder, and Zhi Li. 2012. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. Data & Knowledge Engineering 72 (2012), 83–102.
[7] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In Proceedings of VLDB. 439–450.
[8] Zhi Li, Peiquan Jin, Xuan Su, Kai Cui, Lihua Yue. CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory. IEEE Transactions on Consumer Electronics. 55, 3 (2009), 1351-1359.
[9] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In Proceedings of ICLR. OpenReview.net.
[10] Sameen Maruf, Fahimeh Saleh, and Gholamreza Haffari.2021.A Survey on Document-level Neural Machine Translation: Methods and Evaluation. Comput. Surveys 54, 2 (2021), 45:1-45:36.
[11] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In Proceedings of FAST.
[12] Liana V. Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In Proceedings of FAST. 341-354.
[13] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. 2015. To ARC or Not to ARC. In Proceedings of HotStorage.
[14] Subhash Sethumurugan, Jieming Yin, and John Sartori. 2021. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In Proceedings of HPCA. 291-303.
[15] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In Proceedings of MICRO. 413-425.
[16] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In Proceedings of HotStorage.
[17] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines. Proceedings of the VLDB Endowment 13, 11 (2020), 1976-1989.