DeepBM: A Deep Learning-based Dynamic Page Replacement Policy

Xinyun Chen
xinyun.chen@berkeley.edu

Abstract

In database systems, buffer management is critical for the efficiency of page accesses, which caches pages that are frequently used or will be used shortly for faster future queries. Various heuristic-based policies aiming at selecting the right page to evict have been proposed; however, existing database systems typically implement a generalized heuristics for all workloads, without taking the workload structure into consideration.

In this work, we are the first to explore the potential of leveraging deep neural networks to learn a policy for buffer management. In particular, we propose DeepBM, a deep learning-based dynamic buffer management policy, focusing on the page replacement task. During the execution of the database workload, when the buffer becomes full, DeepBM predicts a page in the buffer for eviction. Different from existing heuristic-based algorithms, which use a generic heuristic throughout the execution, DeepBM learns from the past execution and dynamically adapts to the workload. Our simulation results on two realistic workloads demonstrate that the learned workload-specific page eviction policy gradually outperforms existing general-purpose algorithms, suggesting the benefit of bringing modern learning-based techniques into traditional database system design.

1 Introduction

Buffer replacement is an important problem in database management that has been studied for a long time [18, 43, 21, 30]. In particular, because fetching data from the disk takes a much longer time than from RAM, modern database management systems (DBMSs) use an area in the main memory as a buffer to cache recently accessed pages, so that future queries on pages in the buffer could be faster. Typically, the buffer is divided into frames of the same size, where each frame can hold a page. When a database transaction requests a page that is currently not stored in the buffer, this page needs to be promoted to the buffer. If there is no more space for caching this page, a page in the buffer must be evicted to make space for the new accessed page. Such a choice of page replacement is essential for decreasing the latency of page access, while the performance could degrade to the case of mostly fetching the data from the disk when the eviction decisions are inappropriate.

To make the best use of the buffer pool, a long line of work have been devoted into the buffer management problem. Existing database systems typically implement a general-purpose heuristic page replacement policy, such as Least Recently Used (LRU), Most Recently Used (MRU), and CLOCK algorithm [18, 21]. Although these heuristic-based algorithms work well in practice, such algorithms do not take the workload structure into consideration, thus forego the potential benefit of workload-specific optimization. It is reasonable to design a generic policy to start with, since manually developing optimization heuristic for each single workload is infeasible in practice. However, we will demonstrate that there could be a considerable performance gap between a heuristic-based page replacement algorithm and the optimal workload-specific policy, which makes it desirable to develop a buffer replacement policy that can effectively utilize the characteristics of different workloads.

Inspired by the recent advancement of machine learning, especially deep learning techniques, in this work, we make the first step to study the effectiveness of leveraging deep neural networks to learn the buffer replacement policy. We present DeepBM [1], a general-purpose deep learning-based framework for learning workload-specific page replacement policies. During the execution of the database workload, when the buffer becomes full, DeepBM predicts a page in the buffer to be replaced. Different from existing heuristic-based algorithms, which use a generic heuristic throughout the execution, DeepBM learns the workload characteristics from the past execution, and dynamically adapts its eviction

[1] This abbreviation stands for Deep neural network for Buffer Management. We want to clarify that the current implementation focuses on page replacement, and we consider extending our approach to deal with other database buffer management problems as future work.

choices to the page access pattern of the workload.

In our evaluation, we compare DeepBM against several generic heuristic-based algorithms, including LRU, MRU, and the buffer replacement algorithm implemented in PostgreSQL [39, 1], which is based on the clock-sweep algorithm [16, 32]. We conduct experiments on TPC-C [2, 25] and Yahoo! Cloud System Benchmark (YCSB) [3, 12], which are two widely studied workloads with different characteristics. We execute both workloads in PostgreSQL to evaluate the clock algorithm, and run the workloads in a simulator for other baseline algorithms and our DeepBM policy. The results demonstrate that our learning-based policy gradually achieves a higher hit rate than baseline policies, and in particular, matches the optimal eviction policy significantly better. These preliminary results show the promise of bringing modern learning-based techniques into traditional database system design.

The remainder of the paper is organized as follows. We first discuss the related work in Section 2. We present the design of our framework in Section 3, and describe the evaluation setup and results in Section 4. We provide a discussion of future directions in Section 5, and conclude the paper in Section 6.

2 Related Work

A variety of heuristic-based buffer replacement algorithms have been proposed and implemented in database management systems. Classical general-purpose page replacement algorithms include Least Recently Used (LRU) and its variants [18, 33], CLOCK and GCLOCK algorithms [16, 32], 2Q algorithm [22], Least Reference Density algorithm [18, 16], and Frequency-based Replacement strategy [35]. Although these algorithms achieve a reasonable performance in practice, compared to the optimal eviction policy [5], which is an oracle algorithm assuming the knowledge of the subsequent page access requests, there is still large room for improvement, as we will confirm in Section 4.

There is little prior work on applying modern learning-based approaches for buffer management. Some existing work propose to use data mining approaches for buffer management, where they obtain rules that reflect the relationships among database objects and transactions by extracting the reference patterns from the page access streams [43, 18, 17]. Specifically, their approaches include two phases: (1) offline training, where they mine rules to predict when the future reference of a page would occur further down the stream; (2) applying their predicted rules to the online stream; note that once the future access patterns are predicted, a corresponding optimal eviction policy can be constructed as in [5], which always evicts the page that will be referenced the latest in the future, or never be used again. However, their mining approaches use hard-coded templates to infer the reference patterns. Although these work demonstrate their effectiveness on synthetic page reference traces generated by themselves, such a formulation could limit the flexibility to learn the patterns of more realistic database workloads. Meanwhile, their algorithms require the separation of offline training and online deployment, and can not perform real-time adaption to the workload. On the contrary, our DeepBM framework employs deep learning, which is demonstrated to have the impressive capability of extracting sophisticated patterns from the training data for different applications, without the need of incorporating additional domain knowledge [19, 46, 38, 15]. Furthermore, in addition to offline learning, our model could also be trained online during its deployment.

Besides buffer replacement, another closely related important topic of buffer management is buffer allocation, which tackles the problem of how many buffer frames should be allocated for each query. There have been some well-known classical algorithms proposed for this problem, including the Working Set algorithm [14], Hot Set algorithm [36, 37] and the DBMIN algorithm [11]. Meanwhile, some variants of these buffer management algorithms are proposed to deal with the situation where the database transactions have different priority levels, such as Priority-LRU and Priority-DBMIN [8]. We consider extending our framework to handle these more complicated scenarios as future work.

Although machine learning approaches have not been greatly explored for the buffer management problem, some recent work have studied the usage of these techniques for other computer system problems, such as job scheduling [26, 9, 27], index structure modeling [24], device placement [31], and code optimization [10, 13]. In this work, we make an initial step toward applying deep learning for buffer management, and discuss some properties and challenges of this application domain.

3 DeepBM Framework

In this section, we present the design of our DeepBM framework. We start the discussion by describing the formulation of page replacement problem in our setting. Next, we present the model architecture and how different components of the model are connected to jointly make the prediction. Lastly, we describe our model training approach.

3.1 Problem Formulation

In this work, we formulate the page replacement problem as a decision making problem. Assuming that we have a buffer with a maximal capacity of B pages. At each timestep t, a new page access is required, and the database system tries to find the page of request in the buffer. If the page is missed in the buffer and the buffer is full, the page eviction policy needs to predict an eviction action $a^t \in \{0, 1, ..., B-1\}$, indicating the index of the page in the buffer that needs to be

evicted. Our main goal in this problem is to maximize the *hit rate*, which is the ratio of accessed pages that are already in the buffer.

Note that this formulation is different from previous work on using data mining methods for buffer replacement, which try to predict the next timestep that requests the same page [43, 17]. Although it is useful to infer such information, aiming at precisely predicting the future access time could sometimes unnecessarily complicate the problem, because for page eviction, it is sufficient to know which page(s) would be accessed the latest in the future, and the concrete timestep does not matter. By training the neural network policy to directly predict the index of the page in the buffer with the latest next accessed time, the output space of our model is bounded by B, which is not dependent on the time horizon of the database workload.

3.2 Overview of Model Architecture

Figure 1 presents an overview of our DeepBM framework. We briefly summarize the key components in DeepBM model architecture as below.

Page access encoder. This component is a fully-connected network, which embeds the attributes of a new page access into a d_e-dimensional vector e^t.

Buffer page encoder. This component is another fully-connected network, which embeds the attributes of each page in the buffer into a d_b-dimensional vector b_i^t.

Page access history encoder. This component is a Long-Short Term Memory neural network (LSTM) [20], a popular and effective model architecture for modeling sequential data, which encodes the entire trace of page access records into a d_h-dimensional vector h^t.

3.3 Page Replacement Process

In the following, we elaborate how each component in the model framework gets involved in the page eviction process.

Page access information encoding. At each timestep t, the database system could receive a new page access request. We can extract important attributes of each page access as the input features to our model. Table 1 illustrates the attributes we use in our evaluation, and more attributes could be included if needed.

Inspired by existing page replacement algorithms such as LRU, besides the current page access information, previous references of the same data record also play an important role in buffer management. Therefore, in addition to the attributes of current page access, if the current page of request is already in the buffer, we also include its index in the buffer as an input feature of the model; on the other hand, if the current accessed page is missed, we set the index to be an overflow value, e.g., B. In our evaluation, we find that adding this feature to the input dramatically improves the model's ability to fit to the data, which is consistent with the heuristic algorithm design.

Let the number of features included in the model input to be $|F|$. Since different features have different possible values, for each feature F_i, let $|F_i|$ be the number of possible values that it can take, we first use one-hot encoding to represent each feature, which is denoted as x_i, then feed x_i into an embedding matrix E_i with the dimension of $|F_i| \times d_F$, so that each feature is transformed into a d_F-dimensional continuous vector $\tilde{e}_i^t$. Formally,

$$\tilde{e}_i^t = E_i^T \cdot x_i \tag{1}$$

This embedding approach was originally introduced in the domain of natural language processing [29], and it has been widely used for different applications.

After embedding each attribute, we concatenate the embedding vectors of all attributes to get a $|F| \cdot d_F$-dimensional vector, then feed this vector into a fully-connected layer FC_E with the output dimension of d_e. In this way, a page access record at timestep t is embedded as a d_e-dimensional vector e^t. Formally,

$$e^t = FC_E^T \cdot [\tilde{e}_0^t; \tilde{e}_1^t; ...; \tilde{e}_{|F|-1}^t] \tag{2}$$

Here, $[\tilde{e}_0^t; \tilde{e}_1^t; ...; \tilde{e}_{|F|-1}^t]$ denotes the concatenation of vectors $\tilde{e}_0^t, \tilde{e}_1^t, ..., \tilde{e}_{|F|-1}^t$. $\{E_i | i = 0, 1, ..., |F| - 1\}$ and FC_E are trainable weights of the *page access encoder*.

Buffer page encoding. The approach of encoding pages in the buffer is similar to the encoding of new page accesses, except that the embedding matrices included in the *buffer page encoder* use different weights. Using the buffer page encoder, we first embed pages in the buffer into d_e-dimensional vectors $\tilde{b}_0^t$, $\tilde{b}_1^t$, ..., $\tilde{b}_{B-1}^t$.

Afterwards, for each embedding vector $\tilde{b}_i^t$, we concatenate it to another d_e-dimensional vector ts_i^t, which is the positional encoding of i (i.e., the index in the buffer). Note that pages with larger indices are accessed more recently. We use an embedding matrix TS of dimension $B \times d_F$ to compute the positional encoding, i.e.,

$$ts_i^t = TS^T \cdot \mathbf{i} \tag{3}$$

Here, we use $\mathbf{i}$ to denote the one-hot encoding of index i.

Finally, we use another fully-connected layer FC_B with an output dimension of d_b to compute b_i^t, which takes the concatenation $[\tilde{b}_i^t; ts_i^t]$ as its input. Formally,

$$b_i^t = FC_B^T \cdot [\tilde{b}_i^t; ts_i^t] \tag{4}$$

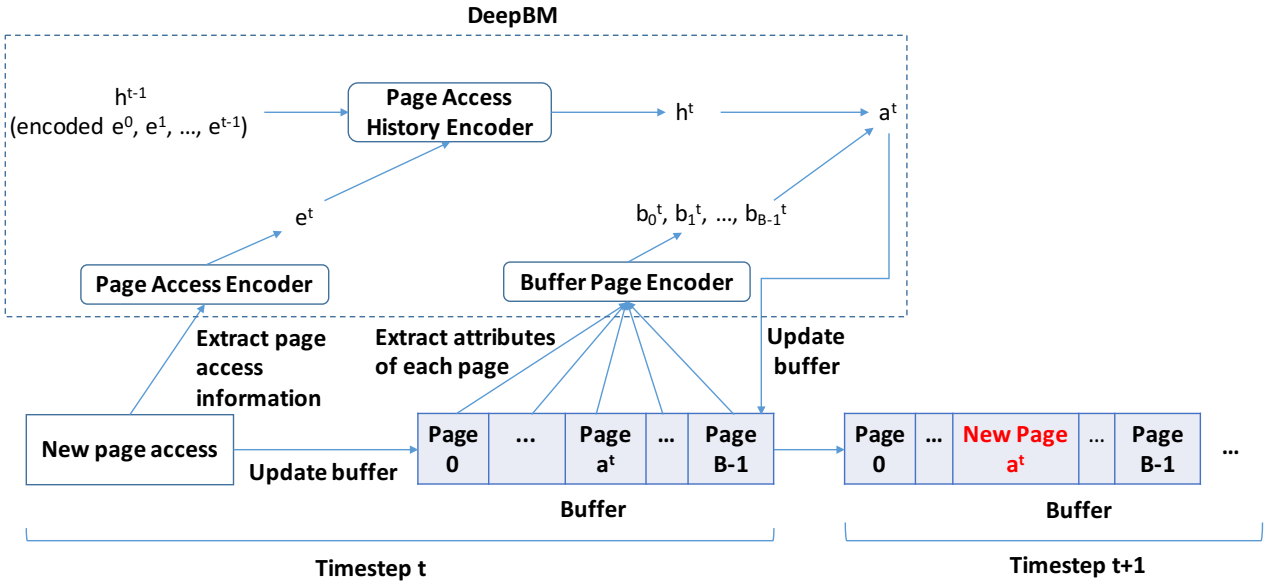


Figure 1: The DeepBM framework. At each timestep t, when a new page access request is received, DeepBM computes an embedding vector e^t using the *page access encoder*, which is then fed into the *page access history encoder*, a Long-Short Term Memory neural network (LSTM) [20] that is capable of modeling long-term sequential data, to update its internal state. When a page fault occurs and the buffer pool is full, DeepBM uses the *buffer page encoder* to compute embedding vectors $\{b_i^t\}, i = 0, 1, ..., B-1$ for all pages in the buffer, then predicts a^t as the index of page to be evicted.

Page access history encoding. To capture the page access pattern from the workload execution history, we employ a Long-Short Term Memory neural network (LSTM) [20] as the *page access history encoder*. LSTMs are a variant of recurrent neural networks (RNNs), which are popular and effective architectures of modeling sequential data, especially for NLP applications [4, 28]. The main advantage of LSTM-based model is that it is more capable of learning long-term dependency and encoding the intricate characteristics in the long run. Specifically, at each timestep t, the LSTM maintains a hidden state h^t, and a cell state c^t. Given a new page access record encoded as e^t, the hidden state and cell state are updated using Equation 5:

$$\begin{aligned} f^t &= \sigma(W_f \cdot [h^{t-1}; e^t] + b_f) \\ i^t &= \sigma(W_i \cdot [h^{t-1}; e^t] + b_i) \\ \tilde{c}^t &= \tanh(W_c \cdot [h^{t-1}; e^t] + b_c) \\ c^t &= f^t \cdot c^{t-1} + i^t \cdot \tilde{c}^t \\ o^t &= \sigma(W_o \cdot [h^{t-1}; e^t] + b_o) \\ h^t &= o^t \cdot \tanh(c^t) \end{aligned} \tag{5}$$

Here, σ is the Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. $W_f, b_f, W_i, b_i, W_c, b_c, W_o, b_o$ are trainable parameters in this component.

Page eviction decision making. Finally, we discuss how to utilize the embedding vectors computed above to decide the page to evict. An intuitive design is to train another fully-connected network with the output dimension of B, then feed h^t and $\tilde{b}_0^t$, $\tilde{b}_1^t$, ..., $\tilde{b}_{B-1}^t$ as its input. In this way, the number of weights in this component is around $B^2 d_b$, which could be huge when a large buffer is used.

Note that for our prediction task, the choices of possible eviction indices are coherent with the number of pages in the buffer. Therefore, we incorporate a pointer network [44] to locate the index of the page for replacement in the buffer. The design of pointer network is based on the attention mechanism used in deep neural networks, which computes an attention map to highlight the important part in the input sequence [4]. In particular, we compute the probability distribution of evicting different pages p^t as follows:

$$\begin{aligned} u_i^t &= v_p^T \cdot \tanh(W_{p1} b_i^t + W_{p2} h^t) \\ p^t &= \text{softmax}(u^t) \end{aligned} \tag{6}$$

Here, v_p is a d_h-dimensional trainable vector, W_{p1} and W_{p2}

are two trainable matrices of dimensions $d_h \times d_b$ and $d_h \times d_h$ respectively. Such a design decreases the size of this component, which improves both the training and inference speed of the model.

With the predicted probability distribution, we can select the index of page for replacement a^t as

$$a^t = \arg\max_i p_i^t \tag{7}$$

3.4 Training Approach

The primary goal of the buffer replacement problem is to maximize the hit rate. Therefore, one option is to apply reinforcement learning algorithms to directly optimize for the metric we care about, i.e., the hit rate in our case [45, 40]. Such a training approach has been used in other system problems, e.g., job scheduling [26, 9, 27] and neural architecture search [41, 47]. However, reinforcement learning algorithms generally suffer from instability, especially when the model is trained online and the time horizon of the workload becomes long.

Compared to the above mentioned problems, where the optimal solution is prohibitively expensive to compute, a desirable property of page replacement problem is that the optimal eviction decision at each timestep can be obtained with a negligible computation overhead, which makes it preferable to train the neural network policy with supervised learning algorithms. In particular, as discussed in [5], for each workload, after seeing subsequent page access requests, the optimal policy can be computed in a greedy manner that evicts the page that will be accessed the latest in the future, or will never be accessed anymore. Therefore, we can compute the optimal eviction choice at each timestep in this way, and use it as the training supervision of the model.

Let $opt^t \in \{0, 1, ..., B-1\}$ be the optimal eviction decision at timestep t, we compute the objective function as follows:

$$J = -\frac{1}{T}\sum_{t=1}^{T} \log p^t_{opt^t} \tag{8}$$

Here we use the cross-entropy loss for optimization. Various optimization algorithms can be used to minimize such kind of loss functions, including stochastic gradient descent (SGD) [7], Adam [23], and RMSProp [42].

4 Evaluation

To demonstrate the effectiveness of our approach, we evaluate DeepBM against several general-purpose heuristic baseline algorithms on realistic database workloads that have been widely studied before. We first present our experimental setup, discuss the configuration details of baseline algorithms and DeepBM, then demonstrate the results.

4.1 Benchmarks

We evaluate on two popular database workloads, which are TPC-C [2, 25] and Yahoo! Cloud System Benchmark (YCSB) [3, 12]. We describe the characteristics of these benchmarks below.

TPC-C. The TPC-C benchmark is a representative workload of an OLTP (On-line Transaction Processing) environment. It models the order processing operations of a wholesale supplier with some number of warehouses, where each warehouse has 10 sales districts, and each sales district has 3,000 customers. In general, the accessed keys are generated under a uniform distribution; however, some transactions may exhibit a certain level of locality. For example, once a New-Order transaction is performed in a warehouse, with a high probability, the warehouse could supply all items of the order by its own stocks, thus further transactions would be performed in the same warehouse. Such a locality could be leveraged to make appropriate eviction decision, and eventually, increase the speed of order service.

Yahoo! Cloud System Benchmark (YCSB). YCSB is a benchmark for cloud data service systems. This benchmark models the setting where there is a table of records, and each operation could insert, update, read or scan records. This benchmark mostly considers the case of skewed record popularity, i.e., a small number of records are extremely popular and visited much more often than most other records. Therefore, by inferring the head and tail of the record access distribution, the performance of page replacement policy could be dramatically improved.

As we will demonstrate in Section 4.6, although existing general-purpose heuristic-based algorithms could achieve a reasonable performance on both workloads, there is still a noticeable performance gap compared to the optimal policy. Thus, we evaluate DeepBM on these two workloads, and see whether our learned page eviction policy could achieve a better performance.

For each of the two workloads, we run it for 18 minutes using PostgreSQL [39, 1], and record the page access information throughout the execution. In this way, we obtain more than 10 million page access records for each workload. Table 1 presents the attributes we extract for each page access. In this schema, rel_id, fork_num and blk_num jointly determines a unique page identifier.

4.2 Simulation Environment

After extracting the workload execution information, we implement a simulator in C++ to interact with our buffer replacement algorithm. Specifically, the simulator maintains the timestamp and a buffer with at most B pages. At each

No	Type	Name	Description
0	uint64	ts	time elapsed since the first buffer access (in ms)
1	uint32	rel_id	Relation ID
2	bool	is_local_temp	If this is a temporary relation in the current session
3	uint32	fork_num	(uint32)-1 - INVALID; 0 - MAIN; 1 - FSM; 2 - VISIBILITYMAP; 3- INIT
4	uint32	blk_num	disk block (page) number in a file
5	uint32	mode	0 - NORMAL (normal read) 1 - ZERO_AND_LOCK (dont read from disk; only lock the page) 2 - ZERO_AND_CLEANUP_LOCK (similar to 1, but lock the page in cleanup mode) 3 - ZERO_ON_ERROR (read page from disk, but return zeroed page on error) 4 - NORMAL_NO_LOG (the same as NORMAL here)
6	int32	strategy	-1 - DEFAULT 0 - NORMAL (random access) 1 - BULKREAD (large read-only scan) 2 - BULKWRITE (large multi-block write) 3 - VACUUM (vacuum)
7	uint32	rel_am	index access method; 0 - heap; 403 - btree; 405 - hash; 783 - gist; 2742 - gin; 4000 - spgist; 3580 - brin
8	uint32	rel_file_node	identifier of physical storage file
9	bool	has_toast	if it has toast table (for large values $\geq$ 8KB)
10	bool	has_index	if this table has or has had any index
11	char	rel_persistence	“p” - regular table; “u” - unlogged permanent table; “t” - temporary table
12	char	rel_kind	“r” - ordinary table “i” - secondary index “S” - sequence object “t” - for out-of-line values (toast table) “v” - view “m” - materialized view “c” - composite type “f” - foreign table “p” - partitioned table “I” - partitioned index
13	int16	rel_natts	number of user attributes
14	uint32	rel_frozen_xid	all xids that is smaller than this number are frozen
15	uint32	rel_min_mxid	all multixacts are greater than or equal to this number
16	bool	hit	whether the current page access is hit in the buffer or not. This attribute is obtained by running the page replacement policy used by PostgreSQL.

Table 1: Extracted attributes of each page access in our evaluation. These attributes are obtained by executing the workloads in PostgreSQL [39, 1].

timestep, it emits page access queries with the extracted attributes according to the timestamp, and when a page replacement is needed, the simulator evicts the page in the buffer based on the page replacement policy connected with the simulator. The implementation of core simulation functionalities includes over 1,000 lines of code.

Throughout our evaluation, we set the buffer size to be 1 MB, the page size to be 8192 B, thus the maximal number of pages in the buffer is $B = 128$.

4.3 Metrics

Since our experiments are performed in a simulated environment, we are not able to measure those metrics that require the integration of DeepBM into the database system, such as data access latency, throughput, and other workload-specific runtime statistics. Instead, we measure the following metrics, which are also important metrics to indicate the efficiency of buffer replacement algorithms, while could be accurately computed in our simulation.

Hit rate. This is the primary metric to demonstrate the effectiveness of page replacement policies, which is the proportion of accessed pages that are already in the buffer.

Match rate. This metric computes the proportion of eviction choices that are exactly the same as the optimal policy. This metric is mainly used to illustrate to what extent a page replacement algorithm aligns with the optimal policy.

4.4 Baseline Policies

We compare our approach with several existing page eviction algorithms, as described below.

Least Recently Used (LRU). This algorithm always evicts the page that has not been used for the longest time.

Most Recently Used (MRU). This algorithm always evicts the page that has been used most recently.

Page eviction policy used in PostgreSQL (PG). The page eviction algorithm implemented in PostgreSQL is based on the clock-sweep algorithm [1, 16]. Specifically, each page maintains a usage counter, which is incremented up to a small number limit whenever it is pinned. To select the victim page for eviction, a *clock hand* moves circularly through all pages. When a page fault occurs, the clock hand inspects the page it is pointing to. If the page has a usage count of 0, then this page is evicted; otherwise, the usage count of this page is decreased by 1, and the clock hand moves on to the next page. This inspection process continues until a page is found to be valid for replacement. In our evaluation, we run the fifth-chance version of this algorithm, i.e., the limit of usage counter is 5.

Hit rate	DeepBM (Ours)	PG	LRU	MRU	Optim
TPC-C	**67.11%**	66.02%	65.33%	4.83%	76.11%
YCSB	**48.70%**	47.17%	47.33%	3.12%	59.08%

Table 2: Hit rate of different page eviction policies on TPC-C and YCSB workloads.

Match rate	DeepBM (Ours)	LRU	MRU	Optim
TPC-C	**41.36%**	1.04%	0	100%
YCSB	**49.58%**	0.88%	0	100%

Table 3: Match rate of different page eviction policies on TPC-C and YCSB workloads.

Optim. As discussed in [5], this is an oracle algorithm that computes the optimal decision assuming the knowledge of the subsequent page access information. This policy can be used as a reference to illustrate the optimality of different page replacement algorithms.

4.5 DeepBM Configuration

From the extracted attributes in Table 1, we discard the 0-th attribute "ts", whose value is unimportant to determine the page eviction choices, and utilize the remaining attributes as the input to DeepBM.

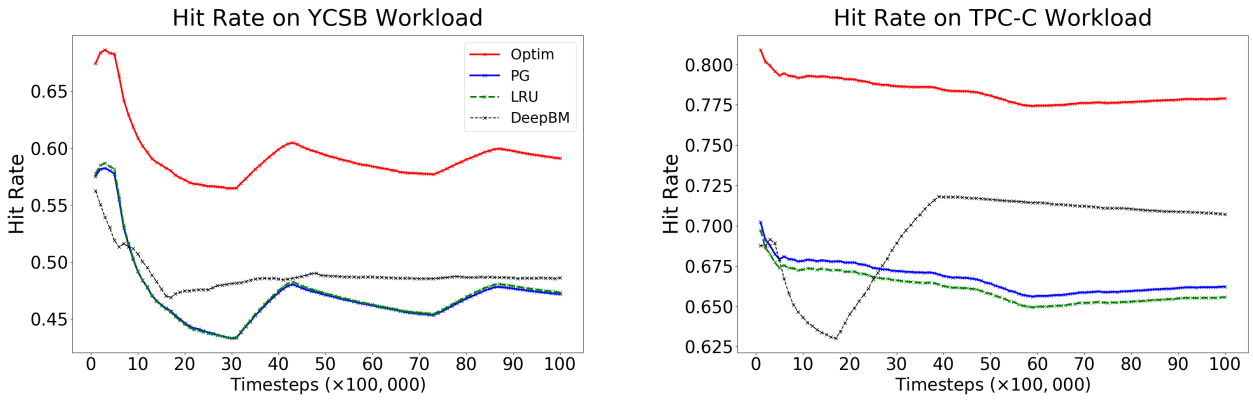
For hyper-parameters in the model architecture, we use a 1-layer LSTM as the page access history encoder. The dimensions of page embedding vectors and LSTM hidden sizes are $d_e = d_b = d_h = 512$, the dimension of the embedding vector for each attribute is $d_F = 32$.

For our training approach, we use the Adam optimizer [23] to update the model parameters. The learning rate is 5e-2, and is not decayed throughout the execution. We set the batch size to be 200 for training, and gradients with L_2 norm larger than 5.0 are scaled down to have the norm of 5.0. All weights are initialized uniformly randomly in $[-0.1, 0.1]$.

Our DeepBM model is implemented using PyTorch [34], which is a popular open-source Python machine learning library. The implementation of the core model architecture includes around 300 lines of code, excluding the code for logging, debugging, monitoring the workload execution and model prediction, etc.

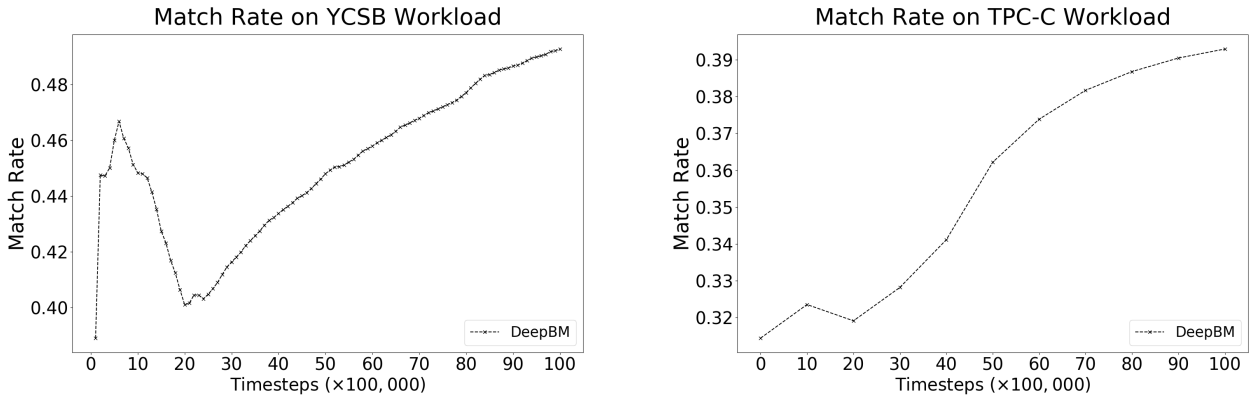
4.6 Results

We first present the overall hit rates and match rates of different buffer replacement algorithms in Table 2 and 3 respectively. From Table 2, we observe that for both benchmarks, MRU policy performs poorly, since in these workloads, a page could be accessed frequently in a short period,



(a) (b)

Figure 2: The hit rate of different page eviction policies throughout the workload execution. (a) YCSB workload. (b) TPC-C workload. Note that in (a), the curves of PG and LRU largely overlap with each other.



(a) (b)

Figure 3: The match rate of DeepBM throughout the workload execution. (a) YCSB workload. (b) TPC-C workload.

which makes this eviction algorithm highly ineffective. On the other hand, LRU algorithm demonstrates a competitive performance compared to PG policy, which suggests that this simple heuristic could achieve a reasonable performance in general. However, the hit rates of both LRU and PG algorithms are still over 10% lower than the optimal policy. In fact, from Table 3, the optimal eviction decision does not align with the LRU eviction decision in most cases, which shows that using a generic algorithm for different workloads may not be the best practice.

In particular, our DeepBM policy outperforms PG and LRU on both workloads, which consistently increases the best hit rate by around 1.5%. Although the results are still not comparable to the optimal policy, such an improvement is more significant than the performance difference between LRU and PG. Furthermore, the eviction choices made by our DeepBM algorithm align much better to the optimal policy, achieving a match rate of $40\% - 50\%$. Compared to LRU, where only around 1% pages are replaced in the optimal way, these results demonstrate the potential of training a deep neural network to characterize the workload-specific page access patterns, and fit to the buffer replacement policy with a higher efficacy.

One concern of a deep learning-based approach is that it could require a huge amount of training data to achieve a reasonable performance. To address this concern, for each workload, we demonstrate the curve of hit rates and match rates for its first 10 million page access records in Figure 2 and 3 respectively. We can observe that although the performance of DeepBM is worse than LRU and PG at the beginning, it gradually adapts its predictions, and consistently outperforms other baseline algorithms after processing no more than 3 million page access requests, while the overall hit rate of DeepBM also changes more smoothly. Compared to the length of the entire workload, this number is acceptable. Meanwhile, the match rate of DeepBM increases

quickly during the training process. Empirically, we find that less than 10,000 page access records is needed to achieve a match rate of over 30% for both workloads.

Meanwhile, we observe that although the overall hit rate of every buffer replacement policy is higher on TPC-C, it is harder for DeepBM to fit to the optimal policy on this workload. This may due to the fact that deep neural networks are better at capturing the overall statistics, i.e., the popular pages that are accessed frequently during a period of time, than the structural characteristics of the transactions.

5 Future Work

While our preliminary results demonstrate the promise of leveraging deep neural networks to learn buffer replacement policies, there is still plenty of room for improvement. In this section, we discuss challenges we have encountered when working on this project, and outline several future directions for exploration.

Integration of DeepBM into the database system. In this work, although we have evaluated on standard database workloads, so far our experiments are performed in a simulation environment, which limits the metrics that we can measure. One important future step is to implement DeepBM as part of the buffer replacement policy in a real database system. An immediate challenge is to minimize the computation overhead caused by the neural network policy; in particular, the gradient propagation steps during training could be time-consuming if the implementation is not carefully optimized. Thus, we may need to make a tradeoff between the size of the neural network architecture and the prediction overhead of the model. Meanwhile, we may consider extending our model to deal with other related buffer management problems when it is deployed in the database system, such as buffer allocation.

Combining with existing buffer replacement algorithms. In our experiments, we observe that although the overall performance of DeepBM is better than baseline algorithms, there are some cases where the prediction of DeepBM is highly inappropriate, especially at the beginning of the workload execution. Therefore, instead of solely deploying our deep learning-based policy, we can combine it with existing policies, and run the learning-based policy after it is well-trained and outperforms the base policies. Although it is a promising direction, we observe that it is challenging to come up with a proper way of doing so. The central difficulty comes from the distribution mismatch of buffer contents with different buffer replacement policies. In particular, if we train our model with the deployment of LRU, once we start to use our learned policy, both the hit rate and match rate would drop dramatically, then gradually go up again after the model is utilized for some time. Therefore, the overall performance and the length of the time range when the learning-based policy performs worse than the base policies would largely remain similar to simply employing DeepBM from the beginning. Such a phenomenon also exists in natural language modeling problems. To mitigate this issue, scheduled sampling approach is proposed in [6], where at each timestep in the training process, instead of always picking the decision from one approach, we can randomly select the policy to make the decision. We leave the extension of such algorithms to appropriately select the best eviction choice from different candidate policies as future work.

Learning a meta policy to switch among different base page eviction algorithms. We observe that for both workloads, the match rate of DeepBM would converge to a certain point no more than 50%, and could not further improve even if we provide more page access records to train the model. This indicates that it may be hard to learn a single model to fit to the entire workload. On the contrary, we could consider learning a meta policy to switch among different base page eviction algorithms, where each of the base policy can either be a manually-designed heuristic-based algorithm, or a learning-based policy. Using this approach, since the supervision of optimal decision at each timestep is not directly available, we need to use reinforcement learning techniques to train such a meta policy. Although it could be challenging to train the policy in this way, since the expressiveness of this meta policy is higher, it could potentially learn a better solution than a single neural network policy.

Exploration of different supervised training objectives. In our evaluation, we observe that while the match rate of our neural network policy is tremendously higher than the baselines, the hit rate does not show a similar level of improvement, which suggests that while aligning with the optimal policy is a good choice, it is not always necessary. Actually in practice, all pages that are not accessed in the near future could be good candidates for replacement. We have attempted to train the neural network policy with multiple optimal targets, so that it is sufficient for the model to predict one of the specified objectives. Surprisingly, in our preliminary exploration, we find that this training method is not effective. One possible explanation is that the model gets confused about which target is a better fit at each timestep, so that it is not able to learn a consistent and well-performed policy. We believe that further tuning of this alternative approach could make the training easier and more effective.

6 Conclusion

In this work, we propose DeepBM, the first deep learning-based buffer replacement policy. Different from existing generic heuristic-based buffer replacement algorithms, our approach learns to dynamically adapt to the page access patterns of different workloads during the execution. Our simulation on two realistic database benchmarks demonstrates that the learned workload-specific page eviction policy gradually outperforms existing general-purpose algorithms, and in particular, matches the optimal eviction policy significantly better. We hope that our preliminary results could shed light on the effective usage of modern learning-based techniques for traditional database system design.

7 Acknowledgments

We would like to thank Prof. John Kubiatowicz for his guidance and suggestions on this project. We also thank Min Du and Zhuoyue Zhao (University of Utah), who help us get a more in-depth understanding of different database workloads and existing buffer management policies.

References

[1] Postgresql. https://www.postgresql.org/.

[2] Tpc-c. http://www.tpc.org/tpcc/.

[3] Ycsb. https://github.com/brianfrankcooper/YCSB.

[4] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[5] BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal 5*, 2 (1966), 78–101.

[6] BENGIO, S., VINYALS, O., JAITLY, N., AND SHAZEER, N. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems* (2015), pp. 1171–1179.

[7] BOTTOU, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[8] CAREY, M. J., AND JAUHARI, R. *Priority in DBMS resource scheduling*. 1989.

[9] CHEN, T., ZHENG, L., YAN, E., JIANG, Z., MOREAU, T., CEZE, L., GUESTRIN, C., AND KRISHNAMURTHY, A. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166* (2018).

[10] CHEN, X., AND TIAN, Y. Learning to progressively plan. *arXiv preprint arXiv:1810.00337* (2018).

[11] CHOU, H.-T., AND DEWITT, D. J. An evaluation of buffer management strategies for relational database systems. *Algorithmica 1*, 1-4 (1986), 311–336.

[12] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[13] CUMMINS, C., PETOUMENOS, P., WANG, Z., AND LEATHER, H. End-to-end deep learning of optimization heuristics. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on* (2017), IEEE, pp. 219–232.

[14] DENNING, P. J. The working set model for program behavior. In *Proceedings of the first ACM symposium on Operating System Principles* (1967), ACM, pp. 15–1.

[15] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[16] EFFELSBERG, W., AND HAERDER, T. Principles of database buffer management. *ACM Transactions on Database Systems (TODS) 9*, 4 (1984), 560–595.

[17] FENG, L., LU, H., TAY, Y., AND TUNG, K. Buffer management in distributed database systems: A data mining-based approach. In *International Conference on Extending Database Technology* (1998), Springer, pp. 246–260.

[18] FENG, L., LU, H., AND WONG, A. A study of database buffer management approaches: towards the development of a data mining based strategy. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on* (1998), vol. 3, IEEE, pp. 2715–2719.

[19] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385* (2015).

[20] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[21] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: an effective improvement of the clock replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (2005), USENIX Association, pp. 35–35.

[22] JOHNSON, T., AND SHASHA, D. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases* (1994), Morgan Kaufmann Publishers Inc., pp. 439–450.

[23] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[24] KRASKA, T., BEUTEL, A., CHI, E. H., DEAN, J., AND POLYZOTIS, N. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (2018), ACM, pp. 489–504.

[25] LEUTENEGGER, S. T., AND DIAS, D. *A modeling study of the TPC-C benchmark*, vol. 22. ACM, 1993.

[26] MAO, H., ALIZADEH, M., MENACHE, I., AND KANDULA, S. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), ACM, pp. 50–56.

[27] MAO, H., SCHWARZKOPF, M., VENKATAKRISHNAN, S. B., MENG, Z., AND ALIZADEH, M. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963* (2018).

[28] MIKOLOV, T., KARAFIÁT, M., BURGET, L., ČERNOCKỲ, J., AND KHUDANPUR, S. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association* (2010).

[29] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (2013), pp. 3111–3119.

[30] MIN, J.-K. Beast: A buffer replacement algorithm using spatial and temporal locality. In *International Conference on Computational Science and Its Applications* (2006), Springer, pp. 67–76.

[31] MIRHOSEINI, A., PHAM, H., LE, Q. V., STEINER, B., LARSEN, R., ZHOU, Y., KUMAR, N., NOROUZI, M., BENGIO, S., AND DEAN, J. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning* (2017), pp. 2430–2439.

[32] NICOLA, V. F., DAN, A., AND DIAS, D. M. *Analysis of the generalized clock buffer replacement scheme for database transaction processing*, vol. 20. ACM, 1992.

[33] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record 22*, 2 (1993), 297–306.

[34] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[35] ROBINSON, J. T., AND DEVARAKONDA, M. V. *Data cache management using frequency-based replacement*, vol. 18. ACM, 1990.

[36] SACCO, G. M., AND SCHKOLNICK, M. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the 8th International Conference on Very Large Data Bases* (1982), Morgan Kaufmann Publishers Inc., pp. 257–262.

[37] SACCO, G. M., AND SCHKOLNICK, M. Buffer management in relational database systems. *ACM Transactions on Database Systems (TODS) 11*, 4 (1986), 473–498.

[38] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., ET AL. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[39] STONEBRAKER, M., AND ROWE, L. A. *The design of Postgres*, vol. 15. ACM, 1986.

[40] SUTTON, R. S., BARTO, A. G., ET AL. *Reinforcement learning: An introduction*. 1998.

[41] TAN, M., CHEN, B., PANG, R., VASUDEVAN, V., AND LE, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626* (2018).

[42] TIELEMAN, T., AND HINTON, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning 4*, 2 (2012), 26–31.

[43] TUNG, A. K., TAY, Y., AND LU, H. Broom: Buffer replacement using online optimization by mining. In *Proceedings of the seventh international conference on Information and knowledge management* (1998), ACM, pp. 185–192.

[44] VINYALS, O., FORTUNATO, M., AND JAITLY, N. Pointer networks. In *Advances in Neural Information Processing Systems* (2015), pp. 2692–2700.

[45] WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning 8*, 3-4 (1992), 229–256.

[46] XIONG, W., DROPPO, J., HUANG, X., SEIDE, F., SELTZER, M., STOLCKE, A., YU, D., AND ZWEIG, G. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256* (2016).

[47] ZOPH, B., AND LE, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).