DRL-Clusters: Buffer Management with Clustering based Deep Reinforcement Learning

Kai Li
Syracuse University
kli111@syr.edu

Qi Zhang
Facebook Research
qizhang@fb.com

Lei Yu
IBM Research
lei.yu1@ibm.com

Hong Min
IBM Research
hongmin@us.ibm.com

Abstract

Buffer cache has been widely implemented in database systems to reduce disk I/Os. Existing database systems typically use heuristic-based algorithms for buffer replacement, which cannot dynamically adapt to changing workload patterns. This paper proposes a deep reinforcement learning-based approach, DRL-Clusters, to manage the buffer pool when handling changing workloads. DRL-Clusters can dynamically adapt to different workload patterns without incurring high inference overhead and miss ratio with page re-clustering and continuous interactions with the cache environment. Our evaluation results demonstrate that DRL-Clusters can achieve a lower or comparable miss ratio than the heuristic policies while reducing 13.3% - 26.8% page access overhead under changing workloads.

1 Introduction

The buffer pool is one of the critical optimizations that databases use to reduce disk I/Os to improve reads/writes performance. By holding the most recently accessed disk pages in the buffer pool, subsequent transactions requesting the same page can directly retrieve it from memory rather than from disk, thus improving the page access performance. In practice, the buffer pool has a limited size and cannot accommodate all disk pages, so when a new page is accessed while the buffer pool has no space, a victim page in the buffer pool must be evicted and replaced with the new page.

Buffer replacement policies have been heavily studied over many years. Existing database systems typically use heuristics algorithms that exploit commonly observed patterns, such as Least Recently Used (LRU) (1), Most Recently Used (MRU), CLOCK algorithms (2) and their variants (3). The drawback of these heuristic-based policies is that they are customized for a limited class of known page access patterns and cannot adapt to changing workloads that preserve dynamic patterns.

On the other hand, deep learning (DL) has achieved dramatic breakthroughs in computer vision, speech recognition, and natural language processing by capturing complex and useful patterns from large data corpora. Recently we have seen a rapid growth of database research in optimizing database configuration, indexing, and query execution with DL techniques (4; 5; 6). Therefore, it is natural to explore if DL models could improve buffer performance for databases and further advance the field. There is little existing work that applies DL to the buffer replacement management for optimizing database systems' performance. The most relevant work is iBTune (5), which uses a deep neural network to tune the buffer pool size to achieve a guaranteed level of service level agreement. Another work (7) proposed to apply DL to buffer replacement problems in the hardware and predict whether the accessed data should be cached or not. In this paper, we consider replacing the heuristic algorithms with a DL component to manage the buffer pool and predict which page should be evicted.

Despite the potential improvement, it is challenging to apply a DL model to manage buffer replacement in database systems in an effective way. Because traditional DL models learn the patterns from the collected data set and focus on recurring workloads, but the workloads in a database system usually present various access patterns. Therefore, to handle a dynamic workload, the model should be able to adapt rapidly to new patterns, which poses challenges. In addition, the decision space for the model to decide which page to evict is as large as the number of pages in the buffer, which introduces significant overhead to the training and inference process. Compared with traditional heuristic algorithms, a practical DL model for managing the buffer pool in production database systems should not introduce non-negligible overhead when making a page eviction decision. Otherwise, it will increase page access overhead and undermine the database performance, which is unacceptable in production environments.

To address these challenges, this paper proposes a practical deep reinforcement learning-based system, named DRL-Clusters, to manage the buffer pool in database systems. To address the adaptivity problem under a dynamic workload, DRL-Clusters uses the reward functions in reinforcement learning to evaluate the buffer performance of the database measured by hit ratio and learn to decide which page to evict through a trial-and-error process. Besides, to address the overhead issue, DRL-Clusters reduces the decision space by clustering the pages into clusters so that the decision space is reduced to the selections of a cluster. For each page, DRL-Clusters maintains short-term, mid-term, and long-term access counts as the features. The pages with similar features are grouped into the same cluster.

We implement DRL-Clusters and compare its performance with other heuristic policies and the baseline. Under synthetic dynamic workloads, our evaluation shows that DRL-Clusters can save 13.3% - 26.8% page access time compared to existing heuristic policies while preserving a comparable miss ratio. Comparing to the non-clustered baseline, DRL-Clusters reduces the page access overhead by 2.25X.

2 Preliminaries

In this section, we describe the baseline design which manages the database buffer pool with a deep reinforcement learning model, without the clustering optimization. We denoted the baseline as DRL-BL. Then we analyze its limitations when managing a large buffer pool.

2.1 Baseline

In the baseline design DRL-BL, the system that using reinforcement learning model to manage database buffer pool consists of the following five components:

DRL Agent generates a buffer pool page eviction action based on the current state of the environment. The next state, as well as the reward, will be calculated based on this action. In this paper, we select Deep Q-Network (DQN) as the learning method. DQN was designed by combining reinforcement learning and deep neural networks at scale.

Environment represents the buffer in a database buffer pool.

State represents the current state of the buffer, which includes the features of all the pages in the buffer pool.

Action represents which page needs to be evicted. The agent generates the corresponding action $\mathrm{a_t}$ based on the current state of the buffer pool.

Reward reflects how the system performance changes before and after an action is taken. In this paper, the performance refers to the buffer pool miss ratio.

In the baseline design, the state space consists of the features from all buffer pages, and the action space comprises all of the buffer pages. Specifically, three different features are used for each page: short-term, mid-term, and long-term access frequencies. The state space is encoded as $\{\mathrm{F_0}, \mathrm{F_1} ..., \mathrm{F_i}\}$ where $\mathrm{F_i}$ denotes the features of the i-th page. Let $\mathrm{F_i} = \{\mathrm{f}_i^s, \mathrm{f}_i^m, \mathrm{f}_i^l\}$ denotes the short-term, mid-term, and long-term access times to the i-th page. Specifically, the baseline maintains a sliding window that captures most recent page requests and the window size is equal to the value of the defined long-term. For example, if the long-term is 1000, then the sliding window will store the most recent 1000 pages requests, Whenever a new page request is generated, either it is page hit or page miss, the sliding window will delete the earliest request and add the new request. In the case that the new page request results in a page miss, the baseline will count each distinct page's short-term, mid-term, and long-term access times from the 1000 requests in the sliding window.

2.2 Limitations of Baseline

The baseline works well when managing a small buffer, however, it becomes impractical when handling a large buffer pool, which is the typical use case in real-world database systems. We analyze the limitation of the baseline approach as follows. We formally formulate the average access time to database records as follows.

$$T_{avg} = T_{hit} \cdot (1 - miss\_ratio) + T_{miss} \cdot (miss\_ratio) \tag{1}$$

where T_{avg} denotes the average access time, T_{hit} denotes the overhead of a page hit, and T_{miss} denotes the overhead of a page miss. T_{miss} can be further broken down to:

$$T_{miss} = T_{IO} + T_{infer} \tag{2}$$

where T_{IO} refers to the system IO delay and T_{infer} refers to the buffer management inference time, namely the time used to decide which buffer page to evict. Substituting Eq. 2 to Eq. 1, we have

$$T_{avg} = T_{hit} \cdot (1 - miss\_ratio) + (T_{IO} + T_{infer}) \cdot (miss\_ratio) \tag{3}$$

In the above equation, T_{IO} and T_{hit} depend on the specific hardware but are fixed values, it can be seen that the average access time is not only determined by $miss\_ratio$ but also by T_{infer}, which varies in different buffer management policies.

For heuristic buffer replacement policies (e.g., LRU, LFU, MRU), T_{infer} is negligible as the eviction decision made by these policies is of O(1) complexity. However, in the baseline, T_{infer} becomes non-negligible when the decision space is large, as the overhead of making a decision[1] is of O(N) complexity, where N is the feature size used by the model. Consider a real-world use case where a database server allocates 32GB memory for the buffer pool, which can hold 8 million pages[2]. Therefore, the baseline would maintain up to 24 million features and feed them into the DQN agent under each reference time. Such a large feature size could significantly prolong the inference time and render the model inefficient for a large buffer pool.

We use an experiment to demonstrate the relationship between the model inference time and the feature size in a DL model. We first train a DL model that maintains X features, then we test the model by asking the model to make a decision and measure the inference overhead. We test the model 10 times and report the average inference time. We vary X from 300 to 12000 and report the inference time in Figure 1a. The result shows that the inference overhead grows linearly with the feature size, from 0.14ms when the feature size is 300 to 0.38ms when the size is 12000.

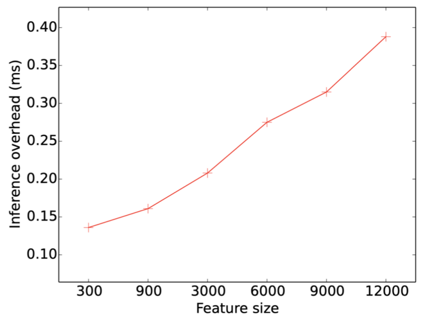
The experiment result conveys the following information: even though DL models can reduce the *miss_ratio*, it incurs extra overhead in the model inference process and increase the *T_infer*, which can offset the benefit of reduced *miss_ratio*. The larger the feature size is, the higher the inference overhead will be. The inference overhead makes DL models unable to scale up to support enterprise-class databases that are usually configured with a large buffer pool.

3 DRL-Clusters Approach

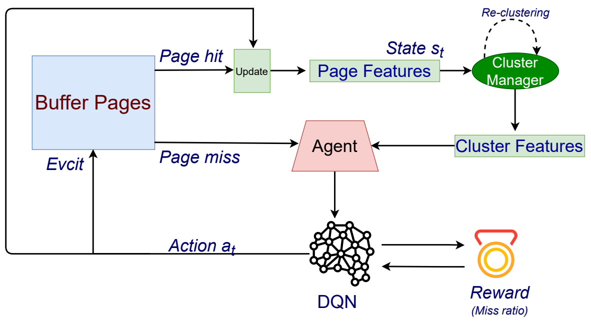
In this paper, we aim to design an efficient (with low T_{infer}) yet effective (with low $miss\_ratio$) DL-based buffer replacement policy. Inspired by the formal analysis in the previous section, there is a trade-off between the feature size and model inference overhead. The insight we obtained is that reducing the feature size can reduce the model inference overhead. So we propose a cluster-based

[1]also known as inference

[2]assume each page is of 4K size



(a) Relationship between inference overhead and feature size



(b) System of DRL-Clusters

Figure 1

DRL model to manage the buffer pool, denoted as DRL-Clusters. By grouping buffer pages with similar page features into clusters, we can derive cluster features of much smaller size and then feed the cluster features to the DQN agent.

In DRL-Clusters, the decision space is reduced from buffer pages to the clusters. At the time a cluster is chosen by the model, choosing which page from the cluster to evict is also a non-trivial problem. In our approach, we randomly choose a page from the cluster to evict, we will talk about different selection strategies in Sec. 5. Therefore, compared with the baseline DRL-BL, the DQN agent of DRL-Clusters uses a much smaller feature size and thus incurs a much lower inference overhead.

Figure 1b presents the workflow of our DRL-Clusters. In the case page access hits in the buffer pool, we update the page features based on short-term/mid-term/long-term access times, and the cluster manager will update the corresponding cluster features[3]. In the case of a page miss, the DRL-Cluster agent will read the current features of all clusters and feed them into the DQN. The DQN then generates an action of selecting a cluster and from which a page will be evicted[4]. The page eviction will result in a new state (e.g., new page features). The reward will be generated and used to update the DQN periodically.

Compared with the baseline design, the state space of DRL-Cluster only consists of clusters features, and the action space only comprises the clusters as well. Therefore, the size of features fed to the DQN agent can be significantly reduced, and so as the inference overhead, as learned by Figure 1a.

As shown by the DRL-Clusters workflow diagram, both page hit and page miss will update the page features. To guarantee that the pages belonging to the same cluster still preserve similar page features after certain rounds, the cluster manager will conduct a re-clustering procedure to update the clusters. Since re-clustering is an expensive operation, which takes more than 2 seconds for clustering 10000 pages into 10 clusters as observed in our experiment, we run this procedure in a background thread and periodically execute it.

4 Implementation & Evaluation

4.1 Implementation

The baseline is implemented by (8). It builds a simulation environment of a database buffer and integrates classic buffer replacement policies, such as LRU, LFU, and MRU. To implement DRL-Clusters, we extend the baseline by implementing a cluster manager and integrating it with the baseline, and we also alter the feature vectors from page features to cluster features. We use KMeans (9) as the clustering algorithm. When the agent generates an action that the X_{th} cluster is selected, we randomly pick one page in the X_{th} cluster to evict. We set the delay to 0 ms for a page hit and 1 ms for a page miss as suggested by (10).

[3]each cluster's features are derived by averaging the page features in the cluster

[4]For the sake of simplicity, we randomly select a page to evict.

4.2 Evaluation

4.2.1 Workload Generation

We generate two types of workloads with different access patterns, namely random workload, and sequential workload. By mixing them, we can synthesize a dynamic workload. We ensure the number of distinct contents is much larger than the buffer size so as to trigger enough misses.

For the random workload, we refer to a popular data simulation model Zipf to generate both training and testing datasets. Zipf is widely used in the simulation of network content requests. It provides a simple but general way to simulate content requests that follow the Zipfian distribution (11). When generating the dataset, similar to existing works (8), we set the parameter to 1.3. We use Zipf to generate two samples, one for the training dataset and the other for the testing dataset. We generate $40K$ requests consisting of $15K$ distinct contents in each sample.

For the sequential workload, we use Zipf to generate a pair of random numbers, with the first one denotes the content starting index and the second denotes the scanning length. For example, with a pair of (1001, 10), 1001 will be the starting index, and 10 will be the length. It represents a sequential request which accesses the content from id 1001 to 1011. Similarly, we generate two samples that contain $47K$ requests within the same range of $15K$ distinct contents. We then mix them with the random workloads. In the end, we obtain two mixed samples, with each containing $87K$ requests.

4.2.2 Experiments

We use the mixed samples generated as described in Sec. 4.2.1 to evaluate the performance of the DRL-Clusters. For classic buffer replacement policies, we drive the testing sample sequentially to the target implementations. For the DRL-based implementations, we first train the model with the training samples for 100 epochs and then test it with the testing samples.

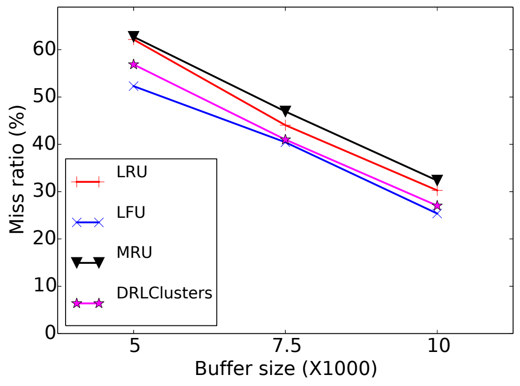
Comparison to classic policies: In this experiment, we configure the buffer size to $7.5K$ for all implementations and configure 3 clusters in DRL-Clusters. After driving the testing workloads, we report the metrics of miss ratio and average page access time in table 1. For the average page access time, we sum up each page access's time and divide it by the number of page accesses. The result shows that LFU achieves the lowest miss ratio among all buffer replacement policies, and our DRL-Clusters achieves a lower miss ratio than the other two classic policies (LRU, MRU). The result can be explained by the fact that the mixed workloads generated in Sec. 4.2.1 features the Zipfian distribution, which favors the LFU policy, and our DRL-Clusters also use frequency as the feature. However, LFU causes the highest average page access time. We investigate the cause of such a high access time and find it is due to the expensive metadata maintenance in LFU (which tracks all pages' accessed times). Compared to all classic policies, our DRL-Clusters achieves the lowest page access time and can save 13.3% - 26.8% access time.

We also vary the buffer size from $5K$ to $10K$ and report the same metrics in Figure 2. The result shows the miss ratio and average access time decrease linearly as the buffer size increases. Under all settings, DRL-Clusters always achieves the second-lowest miss ratio and lowest average access time.

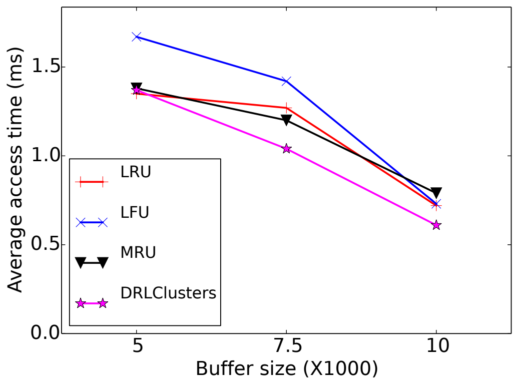
Policies	Miss ratio (%)	Average access time
LRU	50.16	1.27 ms
LFU	40.44	1.42 ms
MRU	46.94	1.20 ms
DRL-Clusters	41.0	1.04 ms

Table 1: Comparison between DRL-Clusters and classic policies

Vary cluster size: In this experiment, we configure the buffer size to $7.5K$ and vary the cluster size in DRL-Clusters. The cluster size varies from 5 to 30 and the results are reported in Figure 2. The result shows there is a trade-off between the performance and cluster size in DRL-Clusters. When the cluster size is smaller than 10, increasing cluster size can improve the performance (both miss ratio and access time). However, further increasing cluster size to beyond 10 can negatively affect performance.



(a) Miss ratio



(b) Average access time

Figure 2: Vary buffer size

Cluster size	Miss ratio (%)	Average access time
5	43.86	1.21 ms
10	40.38	1.15 ms
15	42.14	1.22 ms
30	42.44	1.31 ms

Table 2: Performance of varying cluster sizes in DRL-Clusters

Approaches	Miss ratio (%)	Average access time
DRL-BL	19.81	0.99 ms
DRL-Clusters	19.57	0.44 ms

Table 3: Performance comparison between DRL-Clusters and DRL-BL

Comparison to baseline: In this experiment, we compare the performance of DRL-Clusters with the baseline DRL-BL. We generate new mixed workloads in the same manner as Section 4.2.1. The random (sequential) workloads contain $40K$ ($180K$) requests that comprise $20K$ distinct contents. The buffer size is configured at $10K$. After training both models for 100 epochs, we drive the mixed testing workloads and report the miss ratio and average access time in Table 3. Compared to the baseline, the result shows that DRL-Cluster achieves a similar miss ratio but reduces the inference overhead from 0.99 ms to 0.44 ms, leading to a $2.25X$ improvement.

5 Discussion

In our DRL-Clusters, we simplify the victim page selection in a cluster by randomly selecting one. This is based on the assumption that all pages within the same clusters preserve similar page features, and choose any page in the cluster to evict would lead to the same effect or the same reward, in other words. However, this assumption may not always hold, as the DQN agent does not directly read the page features, so it is possible that randomly choose a page leads to a negative reward. One can explore alternative ways to select a victim page from a cluster, such as maintaining an LRU and MRU queue for each cluster and selecting the victim page accordingly.

Another shortcoming of DRL-Clusters is the feature set used in the model. We only consider the frequency-based page features in different periods, making the model achieve a comparable miss ratio with the LFU policy. One can further improve the model by including other page features into the feature set, such as recency information and table features (e.g., table space IDs).

In addition, in our current design, the number of clusters requires manually tuning and the database operator needs to choose the best setting for their use case. Besides, the re-clustering procedure is quite expensive, and we periodically execute it in a background thread so that it won't affect the handling of page access requests. Instead of periodically executing it, there are other ways to execute the procedure. For instance, one can keep tracking the average distance of the pages to the cluster centroids and trigger the re-clustering when the distance is above a threshold. Another approach is to keep monitoring the miss ratio of the buffer pool and trigger the re-clustering if the miss ratio increases consecutively.

6 Related Work

Deep learning-based approaches have been researched and applied to improve the efficiency of many complex systems, such as databases and cache management.

To make a database perform better, Ji Zhang et al. (6) addressed the challenges of database auto-tuning by proposing CDBTune, which utilized deep reinforcement learning to find the optimal configurations in high-dimensional continuous parameter space. Adam et al. (12) introduced Self-Tuning Memory Manager, which provided adaptive self-running of database memory heaps and cumulative database memory allocation by using a combination of technologies such as control theory, runtime simulation modeling, cost-benefit analysis, and operating system resource analysis. To be able to maintain the machine learning models' performance when the test data diverges from the training data, Lin Ma et al. (4) proposed an active data collection platform that employed active learning to gather relevant data. They also developed an active learning approach that can combine multiple noisy signals for data gathering from database applications. To achieve more effective cache management, Zhan Shi et al. (7) analyzed an LSTM (13) based offline cache replacement model and proposed Glider, which is an online deep learning model that matches the offline model's accuracy but with lower cost. Sami Alabed et al. (14) also investigated the use of reinforcement learning to guide a general-purpose cache manager. In contrast to these works, DRL-Cluster addressed the challenge of finding a trade-off between model accuracy and performance by introducing a clustering-based reinforcement learning approach.

7 Conclusion

This paper proposes a deep reinforcement learning-based approach, DRL-Clusters, to manage the buffer pool when handling changing workloads. DRL-Clusters can dynamically adapt to different workload patterns without incurring high inference overhead and miss ratio with page re-clustering and continuous interactions with the cache environment. Our evaluation results demonstrate that DRL-Clusters can achieve a lower or comparable miss ratio than the heuristic policies while reducing 13.3% - 26.8% page access overhead under dynamic workloads.

References

[1] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[2] F. J. Corbato, "A paging experiment with the multics system," MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, Tech. Rep., 1968.

[3] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 323–336.

[4] L. Ma, B. Ding, S. Das, and A. Swaminathan, "Active learning for ml enhanced database systems," in *Proc. of SIGMOD*, 2020, pp. 175–191.

[5] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang, "ibtune: Individualized buffer tuning for large-scale cloud databases," *Proceedings of the VLDB Endowment*, vol. 12, no. 10, pp. 1221–1234, 2019.

[6] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proc. of SIGMOD '19*, 2019, p. 415–432.

[7] Z. Shi, X. Huang, A. Jain, and C. Lin, “Applying deep learning to the cache replacement problem,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 413–425.

[8] P. Wang, Y. Wang, and R. Wang, “A reinforcement learning-based replacement strategy for content caching,” 2020. [Online]. Available: https://peihaowang.github.io/archive/Wang_DRL_Cache_2020_report.pdf

[9] K-means clustering. [Online]. Available: https://en.wikipedia.org/wiki/K-means_clustering

[10] Hard disk drive performance characteristics. [Online]. Available: https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics

[11] Zipf’s law. [Online]. Available: https://en.wikipedia.org/wiki/Zipf%27s_law

[12] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, “Adaptive self-tuning memory in db2,” in *Proc. of VLDB*, 2006, pp. 1081–1092.

[13] M. Sundermeyer, R. Schlüter, and H. Ney, “Lstm neural networks for language modeling,” in *Thirteenth annual conference of the international speech communication association*, 2012.

[14] S. Alabed, “Rlcache: Automated cache management using reinforcement learning,” *arXiv preprint arXiv:1909.13839*, 2019.