Buffer pool management using Reinforcement learning

Deeksha Dixit
Aadesh Bagmar*
deeksha@umd.edu
aadesh@umd.edu
University of Maryland
College Park, Maryland

ABSTRACT

Buffer pool management is a popular area of research in Databases. For databases larger than memory, optimal buffer pool management can help in speeding up the processing of user queries. An optimal cache manager will avoid unnecessary operations, maximise the cache hit rate which results in fewer round trips to a slower backend storage system and minimize the size of storage needed to achieve a high hit rate. Buffer management is thus, critical for the efficiency of the page accesses to decide which pages are frequently used or will be used in near future for faster processing of upcoming queries. Various heuristic-based policies aiming at selecting the right page to evict have been proposed, however, existing database systems typically implement generalized heuristics for all workloads, without taking the workload structure into consideration.

In this work, we explore the potential of reinforcement learning to learn a policy for buffer management. In particular, we propose a Q-Learning based framework focusing on page replacement task for a predictive workloads. During execution of a workload and when the buffer is full, our system proposes a page in the buffer for eviction. Our system learns from past execution and creates a generalized, easy to implement solution that databases can adopt. Our simulation results demonstrate that our approach outperforms existing algorithms in certain cases suggesting the benefit of bringing modern learning based techniques into traditional database system design.

*All authors contributed equally to this research.

MD '20, May 18, 2018, College Park, MD

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/n

KEYWORDS

Q Learning, Reinforcement learning, cache eviction, buffer pool optimization

ACM Reference Format:
Deeksha Dixit and Aadesh Bagmar. 2020. Buffer pool management using Reinforcement learning. In *Proceedings of VLDB (MD '20).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/n

1 INTRODUCTION

Buffer replacement is an important problem in DBMS and as been studied for a long time. Typically, a buffer is divided into frames of same size where each frame can hold a page. When a page is requested which is not part of the buffer, it is first added to the buffer for future accesses. If there is no space available in the buffer, an existing page in the buffer needs to be evicted to make space for this new page. Choosing the right page to evict is important to reduce latency of accesses.

Existing database systems typically implement a heuristic based page replacement policy, such as Least Recently Used (LRU), Most Recently Used (MRU), etc. DBMIN algorithm suggests a strong approach where it uses different types of page replacement algorithms for different types of queries. DBMIN however suggests strategies per file (or per query). It misses out on optimizations that can be done if a workload is predictive. The algorithm does not perform any look ahead or predictive optimizations.

Buffer pool optimization has rewards (hits) and penalties (misses) associated with it. We believe that the problem has an optimal substructure for Reinforcement learning. In our evaluation, we compare our solution against several generic heuristic based algorithms including LRU, MRU, and Random. We ran simulations on workloads containing queries of different types. The results are promising and demonstrate that our learning-based policy achieves a higher hit-rate than baseline policies. It matches the optimal eviction policies at times.

The remained of the paper is organized as follows. We first discuss related work in Section 2. We describe the design of our framework in Section 3 and evaluation setup and results

in 4. In Section 5, we discuss future directions and conclude the paper in Section 6.

2 RELATED WORK

Liu et al. [5] use a Reinforcement Learning (RL) and ϵ greedy strategy to manage hybrid workload in a time-critical distributed computing environment. They are able to reduce the number of missed deadlines and minimize the job delay for all jobs. This corroborates the ability of RL to learn effective management strategies for similar applications. Tan et al. introduced iBTune [6] which optimizes the configuration of individual buffer pool sizes for database instances in cloud databases using a neural network-based approach. They use the information from similar workloads to train a pairwise neural network which exploits the relationship between miss ratios and allocated memory size.

There are well-known classical algorithms proposed for this problem including Working Set Algorithm [4], Hot Set algorithm[7] and DBMIN [3]. All of these algorithms focus on identifying the Query locality set and optimizing the buffer replacement policy and buffer allocation for the running query. However, these policies do not "look-ahead" or make use of any information if the workload is predictive.

Machine learning based approaches have not been greatly explored for buffer management problem. [2] discusses a deep learning based approach to solve a similar problem. Researchers have achieved success in end to end automatic Database knob tuning using Deep reinforcement learning. Ottertune [8], and CDBTune [9] are few works. [11] in its prior work created an agent modeled on maximizing long-term cache hit rates. Their agent achieved complementary results where it received stable long term cache hit rate in comparison to existing policies and improved short-term cache hit rate.

[1] discusses training a reinforcement learning based agent for maintaining CDN caches. They say that RL has been known to perform sub-optimally when compared to simple heuristics. They indicate that the main problem is that rewards (cache hits) include large delays preventing timely feedback to the agent. They solved this problem by modelling their agent to learn the OPT algorithm and their agent outperforms state-of-the-art CDN caches. Our approach is similar where we model our agent on learning existing techniques (MRU/LRU) and our rewards are not affected by time taken to manifest a hit/miss.

Zhang et al.[10] used model-free acceleration reinforcement learning to find an optimal policy for proactive caching. They used a linear predictive model to predict the popularity of content based on historical information. This work is highly related to our task. They differ from our application as their priority is to pre-cache the popular content on the other hand we want to optimize the eviction policy. However, similar to our work, their reward function consists of hits over time and and a replacement cost. Their success in training the RL agent for the task suggests that hits normalized by query runtime is a good measure of performance in this scenario. Our construction of the reward function is highly motivated by their approach. We use hits and misses normalized by query runtime as part of our reward function to solve the task at hand.

3 ENVIRONMENT AND SYSTEM DESIGN

In this section, we explain the task we are trying to solve and a few details of our simulation environment.

Task

Given a workload of queries of different types but consistent in order using a buffer pool of fixed size, find the optimal buffer pool eviction strategy.

We assume that the order of queries is consistent over an entire run but the parameters might change such as the size of table in a join query or the loop size in sequential queries. Also, we assume that the final strategy proposed is a combination of four traditional eviction schemes MRU and LRU.

System Design

We simulated the environment for this task. Our environment consisted of the following objects which users can tweak.

Cache. The Cache object is a map from keys to a Cache element. A cache object has a limit on maximum number of elements it can hold. If a user tries to add an element to an already filled cache, the user needs to specify an eviction strategy where it can choose which element to remove to accommodate the new cache element. Cache along with the elements also maintains information about when the element was added, how many hits does it have and when was it last accessed.

Query. The Query object requires the type of query (select, join, sequential) and the current time stamp to be defined. The queries have different parameters. The Query Type is indicated by the variable QT. e.g. **start** and **end** indexes for **select** queries, **start**, **end** and **loop_size** for **sequential** and **start**, **end** for both the tables in the **join** query.

Each Query object is assigned a cache. The cache may or may not be shared among different queries. Each query object has a **.step(action)** method where the query proceeds to the next time step altering the elements of the assigned cache. Every time an action is taken, the Query fast forwards to the next time step where the next action needs to be taken

while collecting the "hits" that it received for reaching the new state after taking the action.

e.g. Say the cache of size 3 has elements 1,2,3 in it. All three of these elements were last used in the order 1,2 and 3. The next elements which would be queried are 4,3,2,5. If the action taken is LRU, 1 would be evicted and the next cache would contain 2,3 and 4. Now, the next accesses are for 3 and 2 which are already present in the cache. Thus, the query fast forwards to element 5 while modifying the "hits" achieved to 2 (for 2 and 3).

While a query executes, every action increments **time** which we denote by t. At the end of every action, the environment returns the number of **Hits** and **Misses** noticed by the Query till time step t. We call it a cache hit if the required element is found in the cache while we call it a miss if the required element is missing from the cache. We define number of hits and misses at the end of time t as H_t and M_t. We define the total reward as H_t - M_t.

Action. For now our agent can choose to use either LRU (Least Recently Used) or MRU (Most Recently Used) as it's eviction policy. It needs to choose at every time step. We are adding FIFO (First In First Out) and LFU (Least Frequently Used) as other potential candidates.

Time. The "time" object is a key component in our simulation. Each query starts at time step zero and the time step increases by one after every action. The time steps are also used to label the cache elements which helps us identify the least recently used / most recently used elements.

Problem Formulation

In order to apply Reinforcement Learning to the aforementioned task we modeled it as a Markov Decision Process.

State. State is the combination of the query type and the current time step. Thus, our state S is a tuple of (QT, t).

Action. An agent can choose any one of the eviction strategies from a set of strategies at any time step. Thus, our action at time t is defined as:

$$A_t \subset [LRU, MRU, FIFO, LIFO] \tag{1}$$

Transition. After an action is taken, the time step is incremented and the agent moves to the next state. The transition also changes the state of the cache which is controlled by the environment.

Reward. The reward at time step t is defined as Equation [2]

$$(H_t - H_{t-1}) - (M_t - M_{t-1}) \tag{2}$$

We tune the reward in this way since this incentivises hits while imposing a penalty for misses.

Episode. Our episode consists of running a workload of queries where the query type and the query order is consistent while the size of tables / parameters may change. The reward at the end of an episode which took t time steps is defined as:

$$Total\ Reward = H_t - M_t \tag{3}$$

Reinforcement Learning Model. We use the Bellman equation to model our reward for the task. Our approach uses Dynamic Programming since we can generate millions of samples through our simulation. We learn the Q values using the following formula:

$$\begin{aligned} Q(S_t, A_t) = (1-\alpha)Q(S_t, A_t) + \\ \alpha(reward + \gamma \max(S_t + 1, all\ actions)) \end{aligned} \tag{4}$$

Pseudo-code for our agent

The pseudocode used to train our agent can be found in Algorithm 1.

4 RESULTS

We use the Q-Learning algorithm to train our RL agent. In Q learning the Q value associated with a state action pair is representative of quality of the action for that particular state. We store the Q values in Q Table which has the dimension $Number\ of\ states \times Number\ of\ actions$. A sub-part of the Q Table is shown in Table [4].

State	MRU	LRU
(Sequential, 1)	-2.157043	-2.157042
(Sequential, 2)	-2.148681	-2.148683
(Sequential, 3)	-2.128328	-2.128330
(Sequential, 4)	-2.083146	-2.083149

We ran a strategic set of experiments to evaluate and build our RL agent from the ground up. First, we ran experiments for training the RL agent to learn eviction policy for a workload with only single query types (select, join, sequential) but variable parameters randomly selected from a predefined range. These parameters vary the table size, overlap of the values between the tables being accessed and the loop size, empty or pre-filled cache. In those initial experiments we expected the agent to converge to the known optimal action for a single query such as MRU for a sequential query. The next section explains how we empirically found the required hyperparameters which gave us desirable results.

Individual Query Workload Experiments

We ran experiments by running queries of a similar type but with varying parameters. Here, we list some results that we got for them.

Join Query. We trained our agent on sequential queries Figure 1 shows the number of hits our policy managed to achieve

Algorithm 1: Algorithm for training the reinforcement learning agent

```
Result: Returns a q_table of shape (1000, 4) where each row indicates a timestamp multiplexed with query type and
        each column shows probability for choosing one of the four actions
reward_policy = [];
q_values_difference = [];
i = 0;
while i < number_of_runs do
    query_workload = get_workload();
    while not query_workload.is_done() do
        // Epsilon decay
        epsilon = 0.1 / pow(10, round(i / epsilon_div));

        if random.uniform(0, 1) < epsilon then
            action = random.choice(env.actions) // Explore action space
        else
            _action_ = np.argmax(q_table[state]) // Exploit learned policy
            action = ["mru", "lru", "lfu", "fifo"][_action_]
        end

        hits, misses = env.step(action);
        next_state = encode_queries(env.query_type, env.time.now());

        _action_ = ["mru", "lru", "lfu", "fifo"].index(action);

        old_value = q_table[state, _action_];
        next_max = max(q_table[next_state]);

        // Alpha Decay
        alpha = 0.01/pow(10, round(i/alpha_div));

        // Bellman Equation
        new_value = (1 - alpha) * old_value + alpha * (r + gamma * next_max);

        // Update q_table
        q_table[state, _action_] = new_value;

        state = next_state;

        // Update the convergence of Q values
        q_values_difference.append(sum(sum(abs(old_q - q_table))));

        // Update the received reward
        reward_policy.append(evaluate_policy());

        // Update Q table
        old_q_table = q_table;
    end
    i++;
    return reward_policy, q_table;
end
```

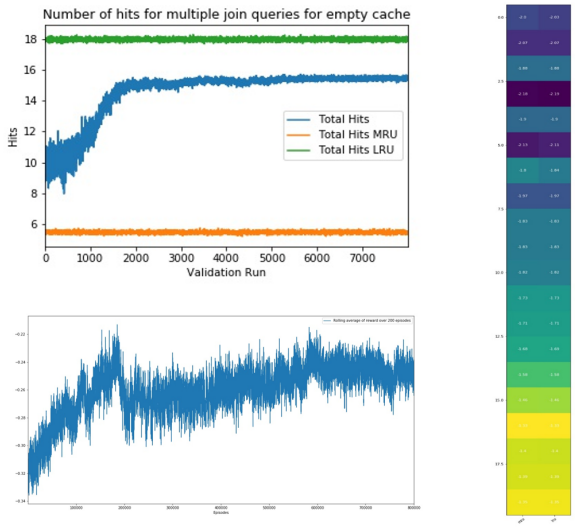

Figure 1: Hits, Reward and Action heatmap for Join Queries. Total Hits are hits received by our current policy. Our policy converges slightly before the optimal LRU policy in this case.

on join queries. Our policy nears performance of LRU as visible from the figure. Also the action Heatmap shows that LRU is preferred for most timestamps showing that our agent is getting primed towards the right policy. Our policy converges slightly before the optimal LRU policy in this case.

Sequential Query. In case of sequential queries where cache is already full, our policy performs better than other policies. Our agent learned that performing LRU initially followed by MRU is the optimal policy. This mixed policy is better than just MRU or just LRU. Figure 4 shows convergence of the reward and comparison of our policy with other policies.

Select Query. In this case, LRU performs better when multiple select queries sharing the same cache run together. Our policy follows closely. As expected MRU has 0 hits in this case. The reward function however converges really well. Figure 4 shows this.

Hyperparameter search

We did a hyperparameter search to ensure that we have the best possible strategy for a single query. We used the same hyper parameters from that point on. Once, we were certain that the agent was behaving as expected, we started running experiments with a workload consisting of multiple queries to see if the learnt policy can learn the pattern within the workload. We can also see that the reward seems to be converging as number of episodes increase. For α decay, we started with a learning rate of 0.01 and decayed it by a power of 10 every $n/5$ iterations where n is the total number of iterations. Similarly, we start ϵ at 0.01 and decay it every $n/4$ iterations.

Alpha Decay + Epsilon Decay. In this case, the Q values converged really quickly as evident from the graph. Similarly, the reward also seemed to have converged close to the high point. However, number of hits have converged at a slightly lower point than the high point it achieved initially. In this case alpha varied from $10^{-6} \leq \alpha \leq 10^{-2}$ and epsilon varied from $10^{-5} \leq \epsilon \leq 10^{-2}$.

Alpha Decay only. The Q values converged pretty quickly, however the reward seemed to converge at a lower value than the maximum value observed across the episodes. In this case alpha varied from $10^{-6} \leq \alpha \leq 10^{-2}$ and epsilon was fixed at 0.01.

Epsilon Decay only. In this strategy the q values decreased pretty smoothly and the reward converged at a high point.

We compute reward over validation runs because reward is tracking the behaviour of the RL agent over all the episodes. Hence, it provides a better idea about the running performance in comparison to validation runs which we perform intermittently after every 100 iterations.

Only Epsilon Decay enabled model seemed to perform the best for our use case. So, we conducted rest of the experiments with the same setting. We hypothesize a model with small learning rate from the beginning is better because of the randomness in parameters of the queries. While the ability to retain and rely on the information collected in past rather than exploring in the later episodes gave us the best overall reward. This is important as we want to do online learning eventually in our system, rather than learning the policies offline and then applying them to a system. Since most real world scenarios have a dynamic workload, we wished to cater to a similar setting.

Mixed Query Workload Experiments

We then started training our model on mixed workloads. Here are the combinations that we tried. Note that the cache is stored and carried on across the two queries. Thus, the agent needs to optimize evicting or keeping the elements thinking that the future query might use it. We tried running these experiments for multiple cases, empty cache vs full cache, overlapping vs non-overlapping scenarios, etc. Below, we show some of the interesting results.

Select-Join. In this experiment we trained our model on a workload which consist of a select query followed by a join query. The model seemed to converge at a lower reward value. It performed better that vanilla LRU eviction policy but was sub par to vanilla MRU. However, the model did try to learn a good policy as the number of hits increased

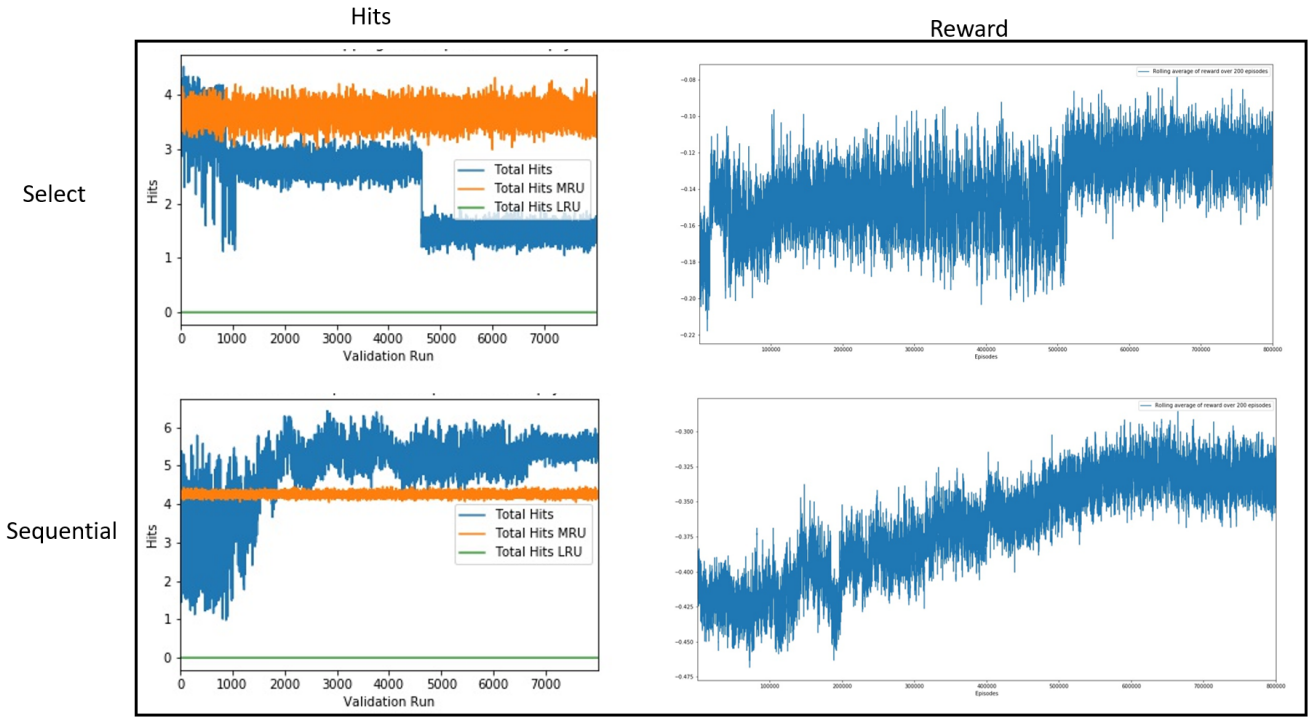

Figure 2: Hits and reward for select and sequential queries.

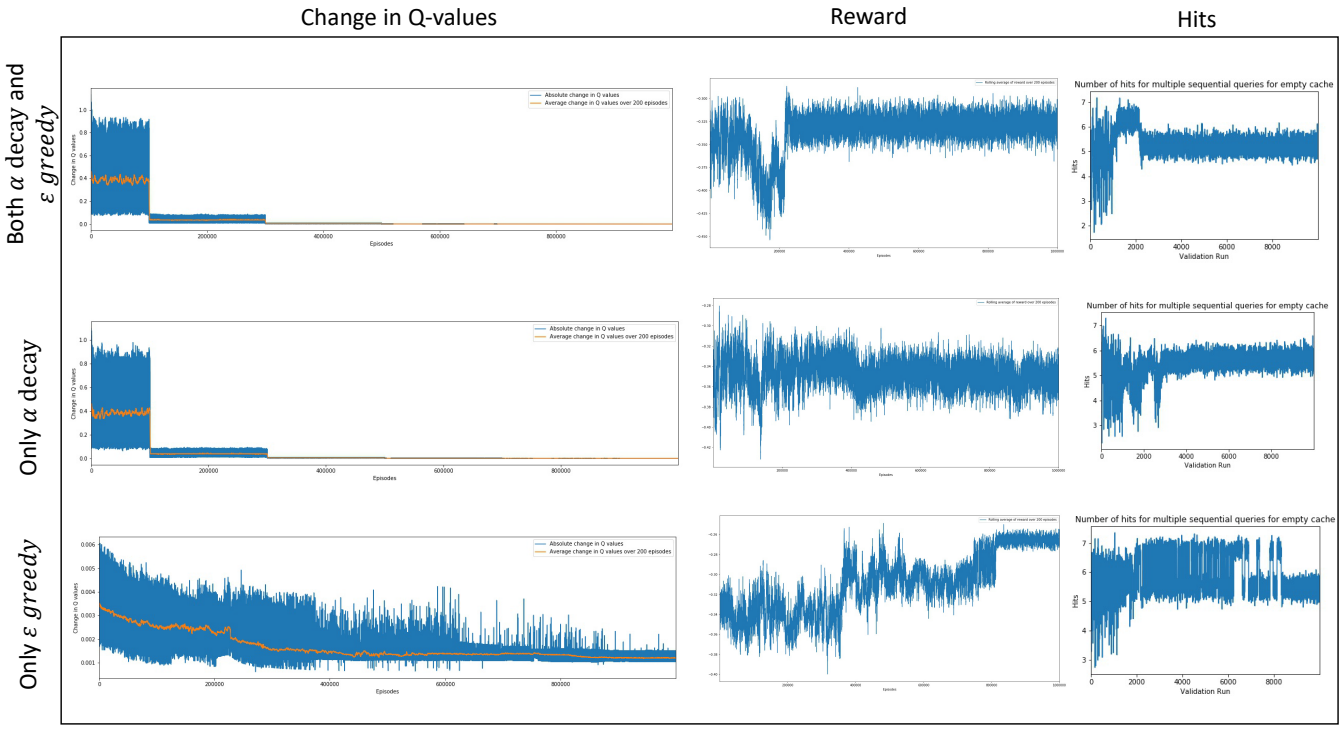

Figure 3: Hits and reward for select and sequential queries.

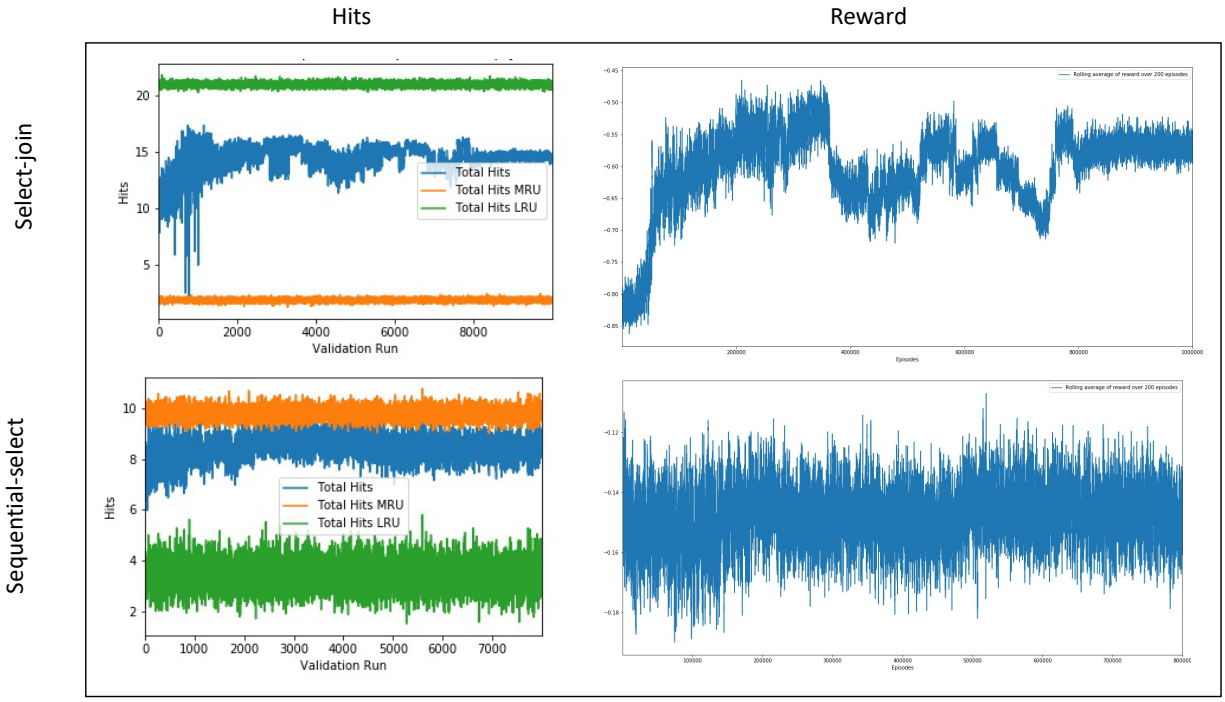

Figure 4: Hits and reward for select-sequential and select-join queries.

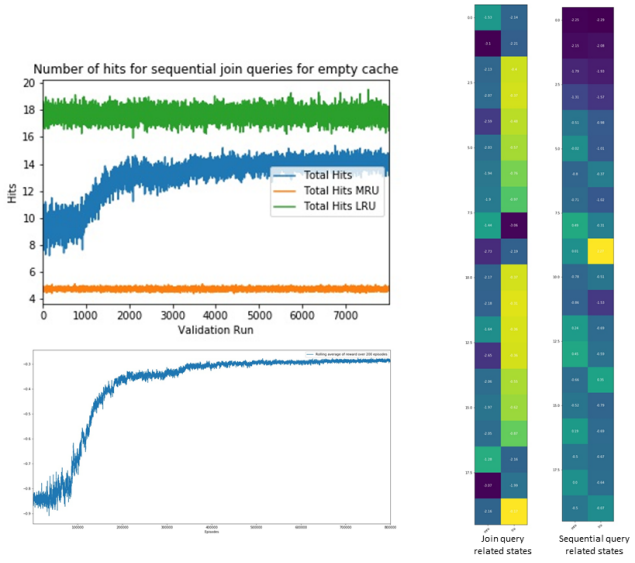

Figure 5: Hits, reward and heatmap for sequential-join queries.

overtime. This performance suggested that a policy which is only a combination of MRU and LRU is not complex enough for a workload like this. So, we decided to add more actions to our environment in order to give the agent more choices in form of LFU and FIFO.

Sequential-Select. In this, we ran a workload consisting of a sequential query followed by a select query. In this scenario the policy learnt by the RL agent was pretty close to vanilla MRU which is promising. As, it suggests that with even a simple action space the agent was able to learn a good policy. The reward also seemed to increase and converge at a high point.

Join-Sequential. Join-sequential workload consists of a join query followed by a sequential query. The reward increased smoothly over time. It can be seen from the heatmap in Figure 5 that action preferred in join query related states is LRU and the action preferred in sequential query related states is MRU in most cases but not all which was earlier observed in Individual query experiments. We hypothesize that the policy performedsub-optimally overall as it performed sub-optimally for sequential query part of the workload. We expect it to perform better on more training.

5 FUTURE WORK

The results of our reinforcement learning system seem to be promising. In the future, we hope to be able to train our model on more complex workloads so that it learns to optimize

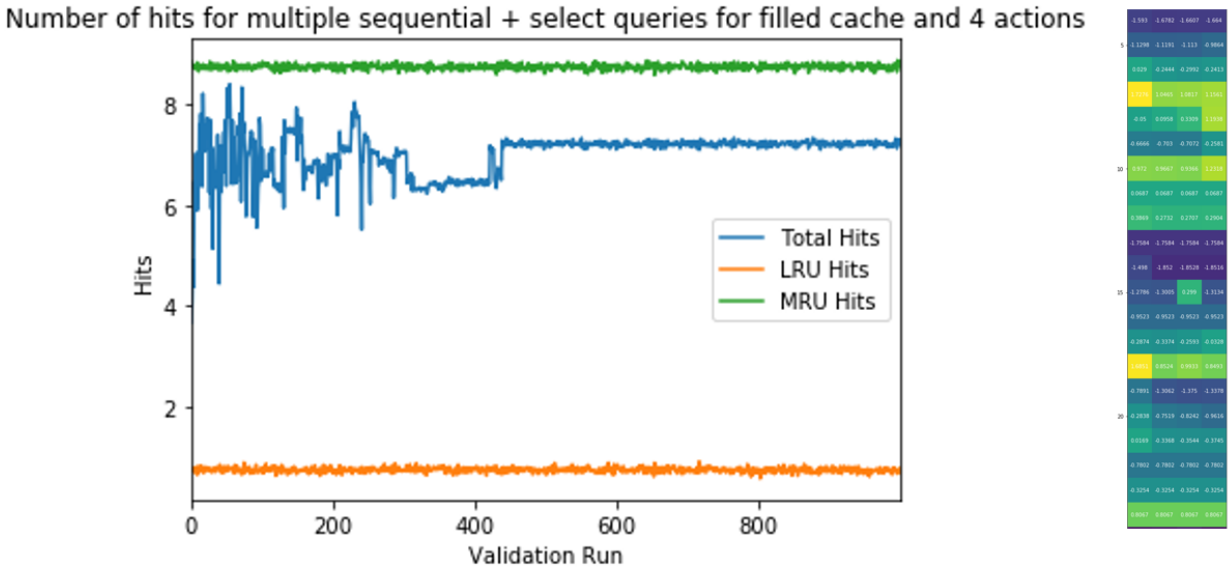

Figure 6: A look into the future: Performance on 4 actions for training. The agent is converging towards MRU

better. Figure 6 shows initial runs where an agent can choose from MRU,LRU,FIFO and LFU. We also hope to allow the agent to evict any element as opposed to just from MRU, LRU actions in case the agent comes up with something even better. Finally, our experiments need to benchmarked on real world systems to get a better understanding of it's performance and usability in real-world scenarios.

6 CONCLUSION

In this work, we propose a reinforcement 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 different varied workloads with queries of single type and mixed query workload shows that the learned workload-specific page eviction policy can outperform existing algorithms and come up with a generic approach which will be easy to implement. We hope that our preliminary results could shed light on the effective usage of modern learning-based techniques for traditional database system design.

7 RESOURCES

Code: https://github.com/cardwizard/DatabaseDeepQL

REFERENCES

[1] Daniel S Berger. 2018. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 134–140.

[2] Xinyn Chen. 2018. DeepBM: A Deep Learning-based Dynamic Page Replacement Policy. *Berkeley Course Project* (2018). https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F18/projects/reports/project16_report.pdf

[3] H-T Chou and DeWitt. 1986. An evaluation of buffer management strategies for relational database systems. *Algorithmixa 1* (1986), 311–336.

[4] Peter J Denning. 1968. The working set model for program behavior. *Commun. ACM* 11, 5 (1968), 323–333.

[5] Y. He, F. R. Yu, N. Zhao, V. C. M. Leung, and H. Yin. 2017. Software-Defined Networks with Mobile Edge Computing and Caching for Smart Cities: A Big Data Deep Reinforcement Learning Approach. *IEEE Communications Magazine* 55, 12 (2017), 31–37.

[6] Feifei Li. 2019. Cloud-native database systems at Alibaba: opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[7] Giovanni Maria Sacco and Mario Schkolnick. 1981. *A mechanism for managing the buffer pool in a relational database system using the hot set model.* IBM Thomas J. Watson Research Division.

[8] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024.

[9] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An

end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[10] Naifu Zhang, Kaibin Zheng, and Meixia Tao. 2018. Using grouped linear prediction and accelerated reinforcement learning for online content caching. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 1–6.

[11] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. 2018. A deep reinforcement learning-based framework for content caching. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 1–6.