MTRP: A High-Performance Cost-Efficient Buffering Scheme for Multi-Tenant Cloud Services

Zekai Zhu, Peiquan Jin*, Xiaoliang Wang, Yigui Yuan
School of Computer Science and Technology
University of Science and Technology of China, Hefei, China
jpq@ustc.edu.cn

***Abstract*—Nowadays, more and more enterprises are moving their critical business towards cloud data service, i.e., database-as-a-service (DaaS). Multi-tenancy is a crucial criterion for cloud service providers since it enables them to share resources among tenants, thus reducing costs. As all tenants share the same buffer space in the cloud server, the overall performance of a tenant will be affected by other tenants. Thus, developing an efficient buffering scheme to ensure tenant isolation becomes an urgent need. Aiming to solve this problem, this paper proposes a novel buffering scheme for managing the shared buffer in a multi-tenant cloud database. We first present an SLA (service level agreement)-based model to quantify the buffering performance of each tenant. Then, we propose MTRP (Multi-Tenant Replacement Policy) for multi-tenant scenarios. The novelty of MTRP lies in three aspects: (1) it partitions the buffer into logical zones to implement tenant isolation. Each zone manages the pages used by one tenant, but the zone space can be dynamically adjusted by buffer replacements. (2) it proposes a two-step replacement algorithm, including global replacement and local replacement, to select victim pages from the buffer, which can improve the space efficiency and reduce the SLA cost; (3) it adopts a reinforcement learning model to select the most appropriate zone to perform a page replacement. We conduct extensive experiments on various multi-tenant workloads to prove the effectiveness of our method. The comparative results with LRU, LFU, LRU-2, and LeCaR show that the proposed MTRP achieves a 1.33× higher hit ratio and reduces 70.1% SLA costs than its competitors on average.**

***Index Terms*—Multi-tenant database, Cloud service, Buffer management, Replacement policy**

I. Introduction

Database systems play the role of storing and querying data for enterprises. Recently, the global public cloud database services market has been increasingly proliferating. Major public cloud database services include Alibaba Cloud, Amazon Web Services (AWS), Google Cloud Platform, IBM Cloud, Microsoft Azure and Oracle Cloud [1].

For a cloud database, its service is necessarily multi-tenant, which means workloads from different tenants share the resources in the physical center [2]. When some tenant's workload executes, it may exert an influence on other tenants adversely. At present, the dominant consumption model for DaaS is provisioned. In this scenario, customers pay for resources they claim to need in advance, regardless of their actual use of them. Meanwhile, there has been an increasing interest in serverless DaaS like Amazon Aurora Serverless and Azure SQL DB Serverless [3]. Serverless databases are allowed to utilize resources up to a pre-specified maximum, which is the same as provisioned databases [4]. However, Serverless databases apply a pay-by-use consumption model. That is to say, customers only pay for resources they actually use instead of the pre-specified maximum of the resources. In scenarios where it is difficult to determine the amount of resources to provision in advance, serverless databases are more suitable.

DaaS poses a significant challenge to cloud service providers in reducing costs. If the service provider reserves the maximum of resources for each tenant, their desire for performance can all be satisfied. However, this approach is not cost-effective [5]. As a result, DaaS providers would like to oversubscribe resources, i.e., promise more resources to tenants than what is actually available on the physical machine [6]. In the multi-tenant resource-sharing scenario, each tenant may actually use fewer resources than promised at any time. For service providers, increasing the degree of oversubscription can significantly reduce costs. However, it may also affect tenant performance because of resource shortages [7]. Even so, oversubscribing is still feasible. If DaaS providers can dynamically allocate more resources to tenants with higher resource requirements, it can reduce the negative impact of resource shortage to a large extent, thus minimizing performance degradation.

In this paper, we study the problem of how to share a buffer pool in a multi-tenant database. The buffer pool stores some of the database pages in memory, thereby reducing the number of direct accesses to the disk. Hence, when the buffer pool is full, choosing which page to evict will exert a significant influence on the performance of the system. In the resource-sharing environment, we need to define the accountability of the service provider. As mentioned above, the performance of one tenant will be affected by other active tenants in this setting. The same workload of one tenant will definitely achieve different hit ratios if the buffer pool is shared among different tenants. If one tenant's hit ratio decreases due to multi-tenancy, we compare it with that in the environment of static resource reservation. Specifically, we use the difference in hit ratio as the quantitative criterion for performance degradation.

Currently, classic buffer replacement policies are focused on the goal of improving the hit ratio without considering

each tenant's SLA (Service Level Agreement) [8], [9]. In the resource-sharing environment, they lack the ability to allocate resources dynamically, resulting in a poor experience for tenants. Also, they are global replacement policies and cannot isolate the essential elements pertaining to requirements from multi-tenancy. In recent years, some learning replacement policies have been proposed. Combined with machine learning techniques, they demonstrate better self-adaptability. However, after consolidating the workloads of more tenants, they are unable to learn useful information effectively.

This paper proposes a novel idea to improve the efficiency of buffer management in the resource-sharing cloud environment. Different from previous traditional approaches, we present an SLA-aware page replacement policy called MTRP (Multi-Tenant Replacement Policy). Briefly, we make the following contributions in this paper.

- MTRP divides the shared buffer into sub-buffers called *zones*, and each zone is responsible for managing the data pages from one tenant. The size of each zone is adjusted by buffer replacements, and each zone is associated with a weight. We demonstrate that such a buffer organizing mechanism can ensure tenant isolation.
- MTRP proposes a two-step buffer replacement policy, which consists of a global replacement and a local replacement. The global replacement aims to select a zone for replacement according to the weights of zones, while the local replacement is to evict a page out of the zone selected by the global replacement. With this approach, we can improve space efficiency and reduce the SLA cost.
- MTRP adopts a reinforcement learning model for the global replacement to select the most appropriate zone. The weights of zones are dynamically updated during each replacement, which can minimize the SLA cost and improve zone isolation according to the different access patterns of each tenant. We will design a zone-reset operation to re-distribute zone weights to avoid serious performance degradation of a zone and guarantee the robustness of MTRP.
- We conduct extensive experiments on various multi-tenant workloads to prove the effectiveness of our method. The comparative results with LRU, LFU, LRU-2, and LeCaR show that the proposed MTRP achieves an average 1.33× and up to 4.25× higher hit ratio than its competitors. Meanwhile, MTRP outperforms all other algorithms by reducing 70.1% on average and up to 96% SLA costs.

The remainder of the paper is structured as follows. Section II reviews related work. Section III details our approach and algorithms. Section IV reports experimental results, and finally, Section V concludes the whole paper and discusses future work.

II. Related Work

Extensive work has been done on buffer replacement policies over the last two decades. These strategies are focused on maximizing the database system's hit ratio. Our work can be viewed as an extension of these algorithms in the multi-tenant environment.

LRU (Least Recently Used) is the best-known replacement policy in buffer management [10]. It always evicts the page according to their last reference time. It has constant complexity per request, and it is simple to implement. It can achieve very excellent performance on workloads with high temporal locality. However, while LRU captures the recency of the reference, it does not consider the frequency. That is to say, it cannot distinguish pages with different reference times in the buffer. Also, LRU is not scan-resistant.

LFU (Least Frequently Used) is a frequency-based policy and removes the least frequently used page whenever the buffer is overflowed [11]. The simple way to implement LFU is to assign a counter to each page in the buffer. Whenever a page is accessed, the counter is increased by one. Each time a page miss occurs, and a page has to be evicted, LFU will choose the page with the minimum counter value. LFU requires logarithmic time complexity in the buffer size and pays almost no attention to recent history. As a result, it may accumulate stale pages with high counters that are no longer useful.

LRU-k is an improvement of LRU, as it captures both recency and approximate frequency of references [12]. It can be viewed as a significant practical step forward. To be specific, LRU-k takes the last k references to each page when determining the victim for replacement. The authors recommended setting k to 2. Though LRU-2 tends to work better than LRU, its runtime complexity is higher than that of LRU. Since it needs to maintain a priority queue, it requires logarithmic time complexity in the buffer size. Also, it contains two crucial tunable parameters, which may exert an influence on its performance to a large extent.

ARC (Adaptive Replacement Cache) is an adaptive buffering algorithm which can recognize both recency and frequency of the reference [13]. Specifically, ARC divides the whole buffer into two lists, namely T1 and T2. They are both managed by LRU. T1 maintains items that are accessed only once, while T2 keeps items that are accessed more than once since admission. It also contains a parameter p, which represents the desired size of T1 and can be updated in an online manner. Since ARC uses an LRU list for T2, it is unable to capture the full frequency distribution of the workload and performs well for LFU-friendly workloads [14]. As a result, it is unable to distinguish between items of equal importance in T2 when facing churn workloads, which leads to continuous buffer replacement.

LeCaR (Learning Cache Replacement) is a novel replacement policy which is based on machine learning [15]. It uses reinforcement learning and regret minimization to choose from two policies, LRU and LFU. To manage regret, it

maintains a FIFO history queue of metadata on the most recent evictions from the cache. A decision is considered as poor if a request causes a page miss, but the requested page is found in the history queue. When a poor decision is made by a specific policy, LeCaR will increase the regret associated with it, thus decreasing the possibility of choosing it to apply the next time a miss occurs. LeCaR is able to learn the optimal probability distribution for every state of the system in an online manner. Experiments have shown that LeCaR is a competitive strategy and especially outperforms its competitors, like ARC, when cache sizes are low.

Recently, machine learning-based buffer replacement policies have become a hot topic [16]–[18]. In the literature [16], the authors presented two learned buffering schemes called LBR-c and LBR-r, which utilized classification models and regression functions to predict the future accessing probability of each page. Similarly, LRB [19] performed a training operation on historical accesses and predicted the page that has the lowest accessed probability in the future. However, these techniques all suffered from the time-consuming training process, causing high latencies during page replacements. To address this issue, GL-Cache [17] proposed a grouped caching policy to transform page-based training into group-based training, thereby fastening the training process. HALP [18] presented another new idea to reduce the training cost by only considering the four pages at the LRU list as the input of the learning model. It can be easily integrated with the LRU policy and support online learning-based page replacement. However, all of the above works were not proposed for the shared buffer in multi-tenant cloud services and could not be applied directly to solve the multi-tenant buffering problem. For instance, LRB, GL-Cache, and HALP were all towards key-value caches, while multi-tenant databases need a page-based buffering scheme.

III. Multi-Tenant Replacement Policy

In this section, we detail the design of MTRP. We first describe the SLA metric in Section III-A. Then, in Section III-B, we present the cost model. Section III-C discusses the architecture of MTRP, and finally, Section III-D introduces the operations of MTRP.

A. Measure of SLA

SLA refers to a tenant's performance low-bound, which is guaranteed by the cloud service provider. Regarding shared buffer management, SLA can be measured by the hit ratio because it dominates the buffer manager's performance.

Database management system caches a portion of database pages in memory through the buffer pool, which can greatly reduce the number of direct accesses to the disk. For a given buffer size and sequence of page accesses, the performance of the system can be measured by hit ratio. When the accessed page is in the buffer, we refer to it as a hit; otherwise, the page must be loaded from the disk, and we refer to it as a miss. The hit ratio is defined as the ratio of the number of hits to the total number of page accesses.

The SLA measure is related to a baseline, i.e., the hit ratio for a tenant's requests when the data pages of the tenant are determined and the promised buffer size is allocated to the tenant by the service provider. In particular, the service provider promises to the tenant that if the default LRU policy is applied and the buffer pool is reversed statically, the hit ratio will no longer be lowered than a specific value. Let HR_b be the hit ratio corresponding to the baseline. When the service provider is unable to fulfil his promise completely, a compensation function will be included in the cost according to the degradation on hit ratio, as shown in Eq. 1.

$$HRD = max(HR_b - HR_a, 0) \tag{1}$$

Here, HR_a is the actual hit ratio. Note that HR_a may be higher than HR_b in some extreme cases. If this occurs, we set HRD to zero.

B. SLA-Based Cost Model

The SLA-based cost model is to evaluate the penalty caused by the hit-ratio degradation of tenants. It includes a compensation function that imposes a certain penalty on the service provider based on the degree of hit ratio degradation. If the actual hit ratio of the tenant differs significantly from the promised one, the SLA-based cost associated with the tenant will increase accordingly. From the perspective of tenants, the compensation function quantifies the compensation they deserve based on the degradation on hit ratio. From the perspective of service providers, the compensation function provides a basis for dynamically allocating resources among tenants.

The penalty mechanism mentioned above can be applied to different scenarios in a cloud environment. For instance, service providers may want to maximize the number of tenants whose performance degradation is within a certain threshold or minimize the compensation money related to performance degradation. In this paper, we define the SLA-based cost model by Eq. 2. Particularly, for one tenant i, its compensation function includes two elements: m_i and $f(HRD_i)$. m represents the SLA price that the tenant is willing to pay if the provider can meet its performance requirements completely. f is a normalized penalty function with both a scope and a value range of $[0, 1]$. The compensation function has an intuitive explanation in the sense that it is the part of the SLA price that the service provider refunds to the tenant due to its inability to meet the promised performance. The goal of our algorithm is to minimize the sum of the compensation function for all tenants.

$$Cost = \sum_{i=1}^{N} m_i * f(HRD_i) \tag{2}$$

In this paper, we normalize the penalty function for evaluation, which takes the form of piecewise linear [6]. Piecewise linear penalties have a plausible interpretation. For example, in Fig. 1, when HRD is below 10%, the provider will pay

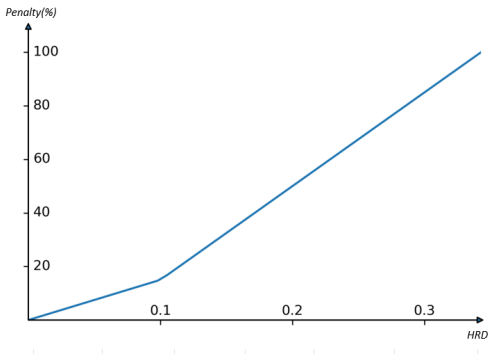

Fig. 1. One example of the penalty function.

back 1.5% of the SLA price for one percentage of HRD. From that point onwards, the per-unit penalty will increase to 3.5%, which reflects stronger tenant dissatisfaction.

C. Architecture of MTRP

Fig. 2 shows the architecture of MTRP. MTRP divides the shared buffer into logical zones. Each zone is responsible for managing the pages for each tenant, i.e., each tenant corresponds to a zone in the buffer. When deciding which page to evict, MTRP divides the entire process into two steps: *global replacement* and *local replacement*. The global replacement chooses a zone in the buffer as the targeted zone for page replacement. After choosing a specific zone, MTRP evicts a page from the zone, which is termed local replacement.

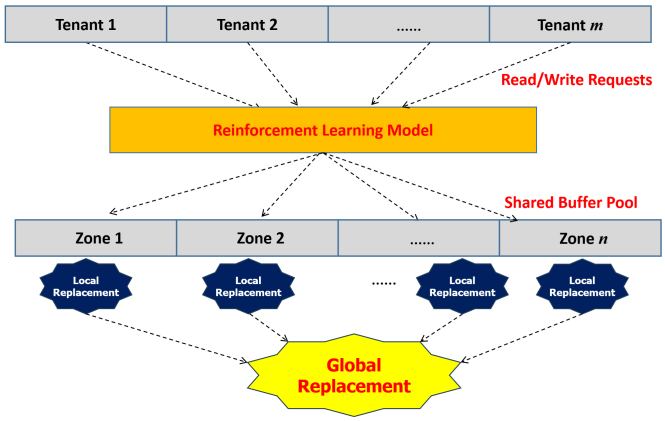

Fig. 2. Architecture of MTRP.

Unlike traditional learning buffer strategies like LeCaR, MTRP takes each tenant's SLA into consideration. When a page miss occurs, the intuition is that the shared resources should be taken from the tenant with low resource demands. If one evicted page is the optimal choice, its corresponding tenant must display relatively low demand for the buffer pool. In the process of global replacement, MTRP aims to obtain the zone corresponding to the tenant. MTRP can learn the optimal probability distribution for every state of the system. Differing from LeCaR, MTRP maintains a probability distribution of all zones instead of a probability distribution of two policies.

Another distinguishing feature of MTRP is that the weight of zones is not some direct function of SLA. Instead, MTRP utilizes the current associated regret to update the distribution. As a result, MTRP models the process of global replacement as an online reinforcement learning problem with regret minimization [6]. Upon a page miss, the zone with the smallest weight will be selected as the targeted zone of local replacement.

Note that the size of each zone in Fig. 2 is not static. On the contrary, it will be dynamically adjusted by the global and local replacement. The key idea is to allocate more buffer spaces to those tenants involving a larger volume of data and high I/O requests and, meanwhile, to reduce the buffer space of the tenants with low workloads. For example, if Tenant 1 (associated with Zone 1) triggers a page miss, but its weight is higher than others, it means that page replacements in Zone 1 will highly increase the SLA-based cost, which we need to avoid. Thus, we will select another zone (e.g., Zone 2) with the smallest weight because page replacements in Zone 2 have the least impact on the overall performance and the SLA-based cost a lot, meaning evicting a page from Zone 2 is the best choice according to the current access characteristics of all tenants. After replacing a page in Zone 2, we will put the requested page of Tenant 1 into Zone 2 and update the page metadata in the buffer. As a result, the size of Zone 1 has been expanded while the size of Zone 2 is decreased. Such adjustments will be triggered by each global and local replacement.

D. Operations

MTRP maintains a history list which contains the metadata of the most recently evicted pages in order to manage regret. It is managed by FIFO and its maximum length is equal to the size of the actual shared buffer size. Each entry in the history list is labeled by the zone that the evicted page belongs to. That is to say, each entry represents a decision to choose a zone in the previous process of global replacement. MTRP considers a decision poor if the accessed page causes a miss, but it is found in the history list. It indicates that if the page was not selected for eviction, it might cause a hit for the system. As a result, the corresponding tenant's compensation can be decreased. The behaviour of choosing the zone that the page belongs to can be rectified by a wiser decision, and thus, it brings about regret. When quantifying the regret, MTRP takes the SLA price and the current HRD into consideration at the same time. When a poor decision is made, the selected zone is penalized by increasing the regret with it. The detailed MTRP algorithm is as in Algorithm 1.

Algorithm 2 shows the details of updating weight. The weight of each zone is equal when the system starts off. When a regrettable page miss occurs, MTRP decreases the weight for the corresponding zone. It indicates that the next time when a page miss occurs, the likelihood of the zone

Algorithm 1: MTRP

Input: shared buffer B, requested page p, number of active tenants N

```
zone_i = find_zone(p);
if p is in B then
    B.UPDATEDATASTRUCTURE(p);
else
    UPDATEWEIGHT(p, i);
    if p is in Hlist then
        Hlist.delete(p);
    end
    if B is full then
        if HRD_i == 0 then
            zone_j = zone_i
        else
            zone_j = (1, ..., N) w/prob(w_1, ..., w_N);
            if |zone_j| <= B/(N*N) then
                for k = 1 to N do
                    if |zone_k| <= B/(N*N) then
                        temp_w_k = 0;
                    else
                        temp_w_k = w_k;
                    end
                end
                zone_j = (1, ..., N)
                  temp_w/prob(temp_w_1, ..., temp_w_N);
                RESET();
            end
        end
        if Hlist is full then
            Hlist.delete(LRU(Hlist));
        end
        Hlist.add(LRU(zone_j));
        B.delete(LRU(zone_j));
    end
end
B.add(p);
```

Algorithm 2: UPDATEWEIGHT(p, i)

Input: shared buffer B, page p, zone id i, number of active tenants N, SLA price m

```
m_max = max(m_1, ..., m_N);
if p is in Hlist then
    w_i = w_i * e^(-m_i/m_max * HRD_i)
end
sum_w = sum(w_1, ..., w_N);
for i = 1 to N do
    w_i = w_i/sum_w;
end
```

Algorithm 3: Zone Reset

Input: shared buffer B, number of active tenants N

```
origin_sum_w = 0;
new_sum_w = 1;
for i = 1 to N do
    if |zone_i| <= B/(N*N) and w_i > 1/N then
        new_sum_w = new_sum_w - 1/N;
    else
        origin_sum_w = origin_sum_w + w_i;
    end
end
for i = 1 to N do
    if |zone_i| <= B/(N*N) and w_i > 1/N then
        w_i = 1/N;
    else
        w_i = w_i/origin_sum_w*new_sum_w;
    end
end
```

being chosen for eviction is decreased. The format of updating weight is inspired by regret minimization [20]. The amplitude of the updating weight is related to the tenant's current HRD and SLA price. In order to calculate the real-time HRD, MTRP maintains a ghost buffer for each tenant. The ghost buffer is of the tenant's promised size and is managed by LRU. Since the ghost buffer only maintains some metadata of page accesses, its overhead can be neglected. MTRP can obtain the HRD for a tenant in a real-time manner by managing the ghost buffer.

Note the size of a zone can be decreased to zero if its weight remains relatively large for some time. It might happen when some of the tenants are in high demand for resources currently. However, it is statistically unlikely that some tenants stay in this state during the whole execution process. Also, if the size of one zone becomes too small, the tenant's performance will be affected severely and it takes a long time to recover. Based on the above reasons, MTRP includes a reset mechanism. The reset operation is given as Algorithm 3. When global replacement chooses a zone of size no more than a specific value, the reset operation is invoked. When the reset operation is triggered, MTRP will not evict a page from the zone whose size reaches the lower bound. Instead, it will choose another zone whose size is above the lower bound according to weight by random. Since some zones have reached the lower bound size, MTRP definitely tends not to choose them for eviction for a period of time. Thus, MTRP resets their weight to the initial value according to their current value during the process. For those zones whose sizes are above the lower bound, the reset operation just redistributes their weight in a linear way. The principle is that the sum of all zones' weight remains 1. When the reset operation ends, MTRP just restarts the learning process. Reset ensures the minimal resources for each tenant, thus improving the robustness of the MTRP algorithm.

After the global replacement, we get the targeted zone for page replacement. Then, the local replacement is invoked to evict a specific page from the chosen zone. When deciding which page to evict in the zone, MTRP uses the LRU policy. Even though LRU is much reviled for its inability to resist scan, the problem can be handled in the resource-sharing environment. Since a zone is chosen according to weight at first, some of the zones can hardly be chosen. Thus, some tenant's scan workload will not pollute the buffer pool for those zones. Also, the eviction of pages follows the LRU order only within each zone instead of the whole buffer pool.

As a result, some of the pages for the scan workload will be evicted within the zone itself during its execution. Other replacement policies can also be used for local replacement, but it does not change the core idea of MTRP. In the future, we will consider some learning replacement schemes [16]–[18] for the local replacement to use different replacement algorithms for tenants.

IV. Performance Evaluation

In this section, we compare our proposed MTRP with several existing replacement policies, including LRU, LFU and LeCaR. We mainly focus on two metrics: hits ratios and SLA-based costs.

A. Experimental Setting

Workloads. We consider four access patterns: sequential, looping, temporally-clustered and probabilistic, which have also been used in previous work [21]. A sequential access pattern is a pattern where all pages are accessed sequentially and never be re-visited. A looping access pattern is one where a subset of pages are accessed repeatedly in a regular interval. A temporally-clustered access pattern is a pattern where recently accessed pages are more likely to be re-accessed in the future. A probabilistic access pattern is one where each page has a stationary access probability independently.

In our experiment setting, we use the aforementioned four patterns to simulate different tenants' access to the cloud database. Specifically, workload W1 contains 4 tenants, and the database contains 40,000 pages. Each tenant accesses 10,000 pages and has one of the four access patterns, respectively. Each access pattern contains 20,0000 accesses. Half of the tenants have a price of $4 for the SLA, and the other half have a price of $1. Each tenant is promised a buffer pool of 1,024 pages.

Workload W2 increases the number of access patterns that each tenant has to four compared with W1. Furthermore, workload W3 is based on W2, where each tenant's page accesses are skewed. To be more specific, in the first half of total page accesses, half of the tenants take up 75%, and the other half take up 25%. For the second half of page accesses, the opposite setting is made.

Workloads W4, W5, and W6 all contain 8 tenants, and the database contains 120,000 pages. Half of the tenants each accesses 10,000 pages, and the other half each accesses 20,000. Other settings are similar to W1, W2, and W3, respectively.

Competitors. We mainly compare MTRP with the following algorithms:

- *MTRP-r*: this is the MTRP policy without the zone reset function, which is used to reveal the effectiveness of the zone reset design.
- *LRU, LFU, LRU-2*, and *LeCaR*: these four policies use a fixed-size zone for each tenant, which is used to measure the effectiveness of zone-size adjustment in MTRP.
- *LRU (shared), LFU (shared), LRU-2 (shared)*, and *LeCaR (shared)*: these policies use a shared buffer for all tenants and apply the same replacement scheme for all tenants.

B. Results

We first MTRP with several existing replacement policies on W1, W2 and W3. MTRP-r represents the MTRP algorithm without the zone-reset mechanism. LRU (shared) represents the case where the LRU policy is applied in the shared buffer pool, while LRU represents the case where the actual buffer pool is partitioned to each tenant on average in an isolated way.

Fig. 3 shows the comparison of the hit ratios of MTRP and other algorithms on W1. We can see that MTRP achieves the highest hit ratio in all cases of different actual buffer sizes. Since the requirement of each tenant for the resource is not the same, MTRP can distribute more space of the shared buffer pool to the tenants who display a relatively high demand in an online manner. For example, when one tenant is under the access pattern of the scan, he will not have a significant performance improvement even if he can utilize more space in the buffer pool. By comparison, when one tenant is under the temporally-clustered access pattern, he is able to achieve much better performance with more resources of the buffer pool. Therefore, MTRP will distribute more space of the shared buffer pool to the tenant compared with others. In this way, MTRP is able to allocate resources among different tenants flexibly so that the overall hit ratio of the whole system can be improved. For MTRP-r, since it lacks the ability to re-distribute weight, it loses the chance to restart learning. As a result, when some zone's size is close to the lower bound, it does not have the opportunity to initialize weight, so it cannot regain the chance of performance improvement. Although MTRP-r displays some limitations, it still achieves a higher hit ratio compared with other existing replacement strategies.

Fig. 4 shows the comparison of the (normalized) costs of all compared replacement policies on W1. It can be noted that MTRP can reduce costs for the cloud service provider to a large extent with all three different actual buffer sizes. Since MTRP considers each tenant's SLA during the replacement process, it can update each tenant's weight in a timely manner. If some tenant's recent actual hit ratio is far lower than the baseline, his weight will be more likely to decrease during the learning process. Also, for those who share similar HRDs, MTRP will give priority to one who pays more for the SLA. If the tenant with a higher SLA price obtains more resources, its compensation function will decrease with a greater amplitude for the same HRD. As a result, the process of updating weight can ensure that each zone's possibility of being selected for the global replacement is directly related to its current demand for resources. Through this online updating mechanism, MTRP achieves the goal of saving costs for the service provider. As for MTRP-r, since it eliminates the step

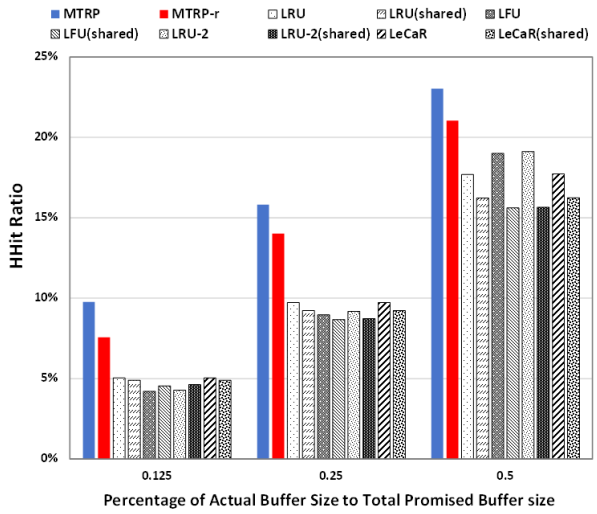

Fig. 3. Comparison of hit ratios (higher is better) on **W1**.

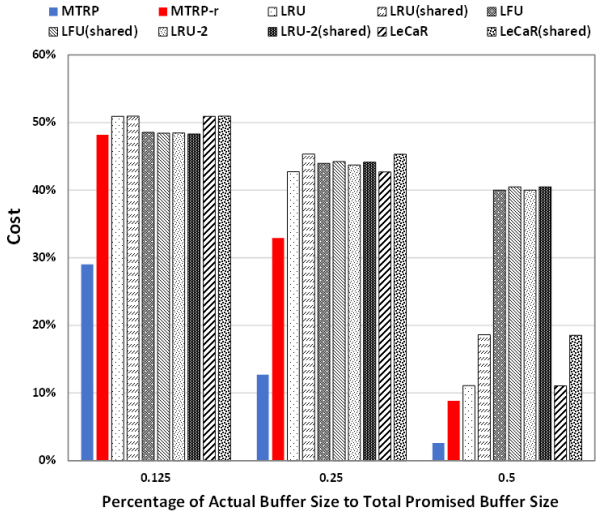

Fig. 4. Comparison of the SLA-based costs (lower is better) on **W1**.

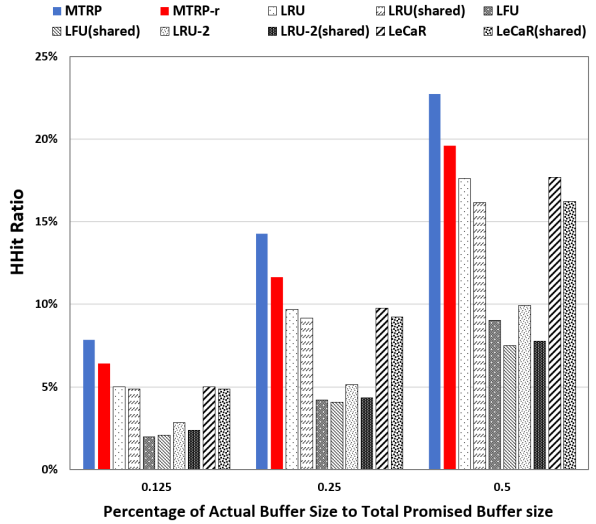

Fig. 5. Comparison of hit ratios (higher is better) on **W2**.

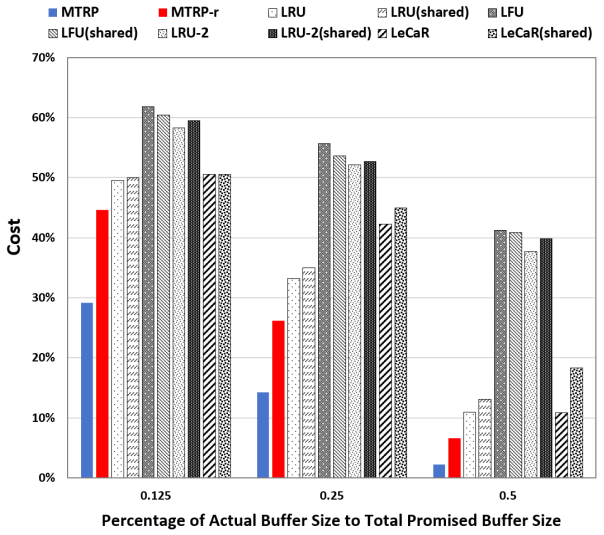

Fig. 6. Comparison of the SLA-based costs (lower is better) on **W2**.

of reset, it will bring about some inappropriate choices of the zone in the global replacement. Thus, MTRP-r cannot achieve the cost-saving effect as MTRP. We can see that when the percentage of actual buffer size to total promised buffer size is 0.125, the cost of MTRP-r is only 0.12% lower than that of LRU-2 (shared).

Fig. 5 and Fig. 6 show the experimental results on W2. MTRP still achieves the highest hit ratio and the lowest cost among all replacement policies for all settings. Since each tenant contains four different access patterns in this workload, one tenant's demand for the buffer pool resource varies as the access pattern varies. It can be concluded that our proposed MTRP is able to maintain good performance when facing changing access patterns instead of a fixed one.

Fig. 7 and Fig. 8 show the comparison of hit ratios and SLA-based costs on W3. Compared with the experimental results on W2, we can see that non-sharing strategies and sharing strategies both display relatively different performances. Compared with non-sharing strategies, MTRP achieves more improvement in hit ratio. To be specific, MTRP achieves 2.62×, 2.21×, and 1.69× hit-ratio improvements on average compared with non-sharing strategies in three different actual-buffer-pool-size settings, respectively. This is due to the fact that the accesses of each tenant in this workload are skewed. Since non-sharing strategies reserve a fixed-size buffer pool for each tenant, they cannot make adaptive changes in the face of imbalanced accesses of different tenants. By comparison, MTRP is a sharing strategy and is able to allocate more resources to the tenants who display relatively high demand currently. It can be noted that the update-weight step only happens when a page miss occurs for one tenant. If some

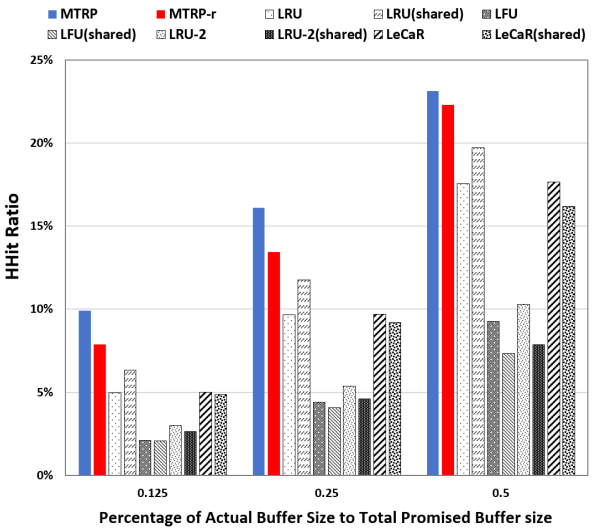

Fig. 7. Comparison of hit ratios (higher is better) on **W3**.

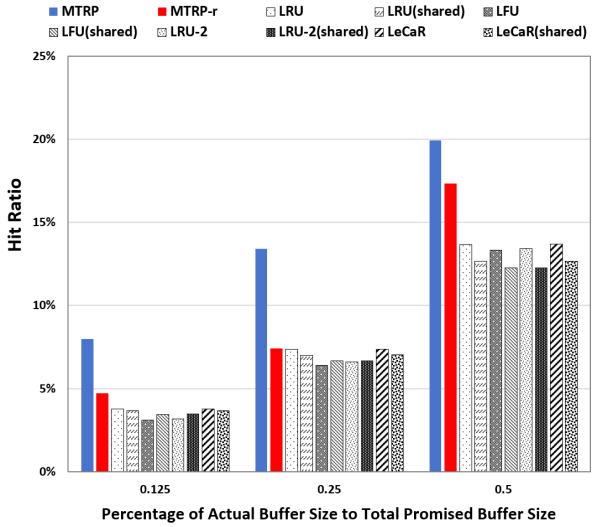

Fig. 9. Comparison of hit ratios (higher is better) on **W4**.

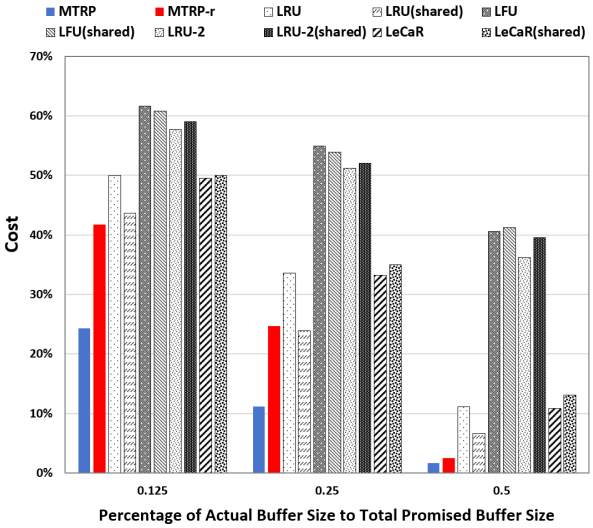

Fig. 8. Comparison of the SLA-based costs (lower is better) on **W3**.

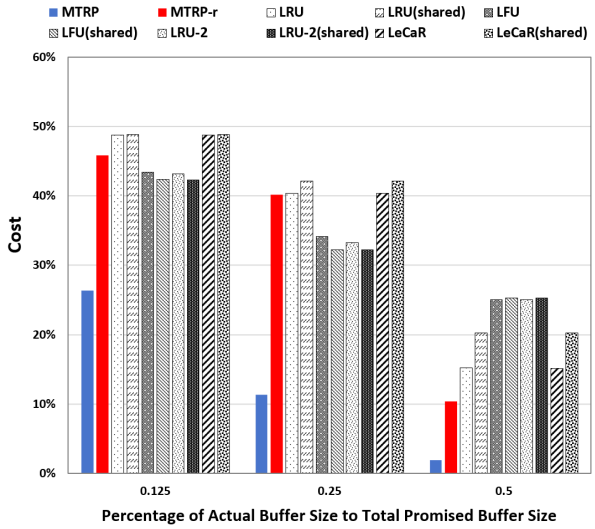

Fig. 10. Comparison of the SLA-based costs (lower is better) on **W4**.

tenant accesses pages in a heavier manner for some time, they will obtain more chances to update their weight. As s result, the zone for him will be less likely to be selected during global replacement. Also, for a page miss, the system will definitely load the page to the buffer pool. For those tenants whose workloads are more intensive at this moment, they will occupy more space in the buffer pool compared with others, thus improving the overall hit ratio of the whole system. Compared with other sharing strategies, MTRP displays more performance improvement in terms of cost reduction. In particular, the average cost for traditional sharing strategies is 2.19×, 3.67×, and 15.29× as much as the cost for MTRP in three different actual-buffer-pool-size settings, respectively. For traditional sharing strategies, when they give priority to the intensive workload, they won't consider the SLA for tenants. For those tenants whose workloads are non-intensive at the moment, their performance will be severely influenced. As a result, although traditional sharing strategies can experience an improvement in hit ratio in this scenario, their cost only decreases with a small amplitude. By comparison, when allocating relatively more resources to tenants with heavy workloads, MTRP still manages to satisfy the SLA for those tenants with lightweight workloads. When facing imbalanced workloads, MTRP is able to achieve a balance among different tenants according to the current situation. As a result, at the same time increasing the hit ratio, MTRP also reduces the cost to a large extent.

Figs. 9-14 show the comparison results on W4, W5, and W6. When there are more tenants, MTRP still achieves the highest hit ratio and the lowest cost among all replacement policies for all settings. By analyzing the results, we can draw similar conclusions to those on W1, W2, and W3.

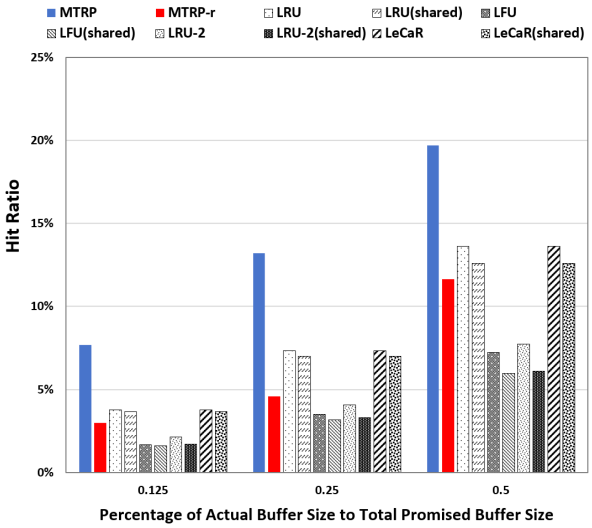

Fig. 11. Comparison of hit ratios (higher is better) on **W5**.

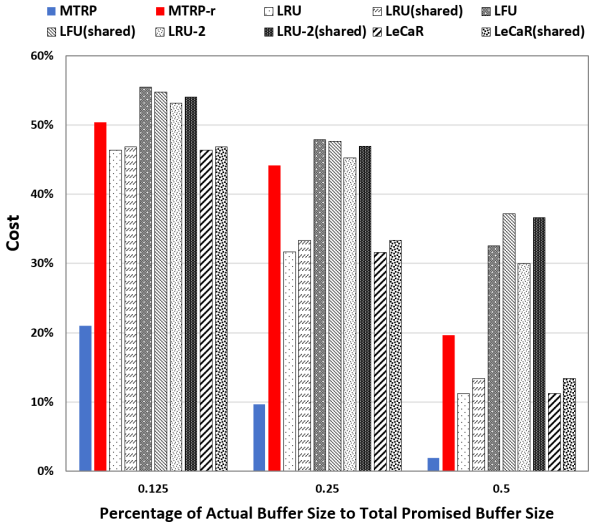

Fig. 12. Comparison of the SLA-based costs (lower is better) on **W5**.

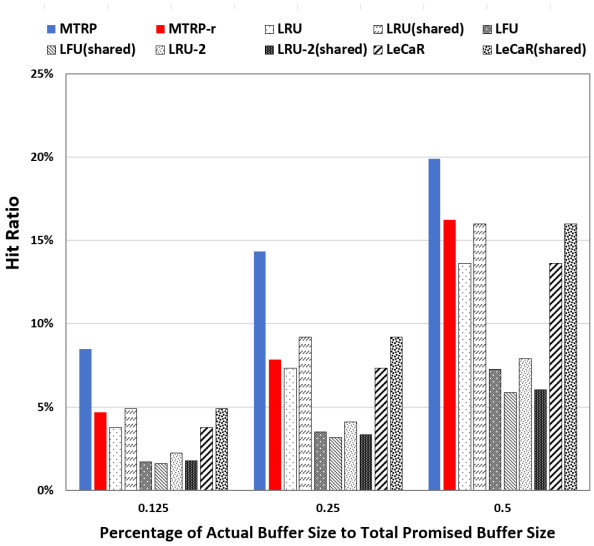

Fig. 13. Comparison of hit ratios (higher is better) on **W6**.

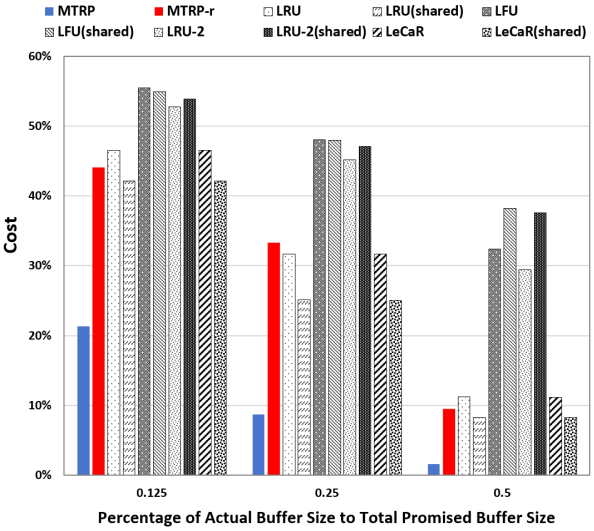

Fig. 14. Comparison of the SLA-based costs (lower is better) on **W6**.

V. Conclusions and Future Work

In this paper, we present a new idea to improve the efficiency of buffer management for multi-tenant cloud services. Differing from classic replacement policies, we propose an SLA-aware page replacement policy called MTRP (Multi-Tenant Replacement Policy), which is based on reinforcement learning and regret minimization. MTRP divides the whole buffer into zones and assigns a weight to each zone. By maintaining a history list, MTRP can update the weights of zones in an online manner. As a result, MTRP can dynamically allocate the resources of the shared buffer pool among tenants, ensuring zone isolation. Also, MTRP contains a zone-reset operation which improves the robustness of the algorithm. We conduct extensive experiments on various multi-tenant workloads to prove the effectiveness of our method. The comparative results with LRU, LFU, LRU-2, and LeCaR show that the proposed MTRP achieves a 1.33× higher hit ratio and reduces 70.1% SLA costs than its competitors on average.

In the future, we will investigate a more generalized model. Presently, we only take the resource of the buffer pool into consideration. We will include more resources, such as CPU cores, in our future model.

Acknowledgements

This work was supported by the National Science Foundation of China (No. 62072419) and CCF-Huawei Populus Grove Challenge Fund. Peiquan Jin is the corresponding author.

REFERENCES

[1] V. Narasayya and S. Chaudhuri, "Multi-tenant cloud data services: state-of-the-art, challenges and opportunities," in *SIGMOD*, 2022, pp. 2465–2473.

[2] P. Arora, S. Chaudhuri, S. Das, J. Dong, C. George, A. Kalhan, A. C. König, W. Lang, C. Li, F. Li *et al.*, "Flexible resource allocation for relational database-as-a-service," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, pp. 4202–4215, 2023.

[3] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, "Firestore: The nosql serverless database for the application developer," in *ICDE*, 2023, pp. 3376–3388.

[4] J. Schleier-Smith, "Serverless foundations for elastic database systems," in *CIDR*. www.cidrdb.org, 2019.

[5] V. Narasayya, S. Chaudhuri *et al.*, "Cloud data services: Workloads, architectures and multi-tenancy," *Foundations and Trends in Databases*, vol. 10, no. 1, pp. 1–107, 2021.

[6] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri, "Sharing buffer pool memory in multi-tenant relational database-as-a-service," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 726–737, 2015.

[7] B. Nougnanke, J. Loye, J. Baffier, S. Ferlin, M. Bruyere, and Y. Labit, "gperfisol: Gnn-based rate-limits allocation for performance isolation in multi-tenant cloud," in *ICIN*, 2024, pp. 194–201.

[8] F. Qazi, D. Kwak, F. G. Khan, F. Ali, and S. U. Khan, "Service level agreement in cloud computing: Taxonomy, prospects, and challenges," *Internet Things*, vol. 25, p. 101126, 2024.

[9] A. Badshah, A. Jalal, U. Farooq, G. U. Rehman, S. S. Band, and C. Iwendi, "Service level agreement monitoring as a service: An independent monitoring service for service level agreements in clouds," *Big Data*, vol. 11, no. 5, pp. 339–354, 2023.

[10] Y. Yuan, Z. Chu, P. Jin, and S. Wan, "Access-pattern-aware personalized buffer management for database systems." in *SEKE*, 2022, pp. 475–480.

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

[12] 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.

[13] N. Megiddo and D. S. Modha, "{ARC}: A {Self-Tuning}, low overhead replacement cache," in *FAST*, 2003.

[14] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with {CACHEUS}," in *FAST*, 2021, pp. 341–354.

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

[16] Y. Yuan and P. Jin, "Learned buffer replacement for database systems," in *DSDE*, 2022, pp. 18–25.

[17] J. Yang, Z. Mao, Y. Yue, and K. V. Rashmi, "Gl-cache: Group-level learning for efficient and high-performance caching," in *FAST*, 2023, pp. 115–134.

[18] Z. Song, K. Chen, N. Sarda, D. Altinbüken, E. Brevdo, J. Coleman, X. Ju, P. Jurczyk, R. Schooler, and R. Gummadi, "HALP: heuristic aided learned preference eviction policy for youtube content delivery network," in *NSDI*, 2023, pp. 1149–1163.

[19] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in *NSDI*, 2020, pp. 529–544.

[20] S. Bubeck, N. Cesa-Bianchi *et al.*, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Foundations and Trends in Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012.

[21] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "Towards application/file-level characterization of block references: a case for fine-grained buffer management," in *SIGMETRICS*, 2000, pp. 286–295.