Sharing Buffer Pool Memory in Multi-Tenant Relational Database-as-a-Service

Vivek Narasayya Ishai Menache Mohit Singh Feng Li Manoj Syamala Surajit Chaudhuri

Microsoft Research, Redmond, WA
{viveknar, ishai, mohits, fenl, manojsy, surajitc}@microsoft.com

ABSTRACT

Relational database-as-a-service (DaaS) providers need to rely on multi-tenancy and resource sharing among tenants, since statically reserving resources for a tenant is not cost effective. A major consequence of resource sharing is that the performance of one tenant can be adversely affected by resource demands of other co-located tenants. One such resource that is essential for good performance of a tenant's workload is *buffer pool memory*. In this paper, we study the problem of how to effectively share buffer pool memory in multi-tenant relational DaaS. We first develop an SLA framework that defines and enforces accountability of the service provider to the tenant even when buffer pool memory is not statically reserved on behalf of the tenant. Next, we present a novel buffer pool page replacement algorithm (MT-LRU) that builds upon theoretical concepts from weighted online caching, and is designed for multi-tenant scenarios involving SLAs and overbooking. MT-LRU generalizes the LRU-K algorithm which is commonly used in relational database systems. We have prototyped our techniques inside a commercial DaaS engine and extensive experiments demonstrate the effectiveness of our solution.

1. INTRODUCTION

Enterprises today rely on relational database systems for efficiently storing, indexing and querying data required by their applications. More recently, several commercial cloud relational database services such as Database.Com [8], Google Cloud SQL [11], Microsoft Azure SQL Database [16] (formerly known as Microsoft SQL Azure), and Oracle Cloud Database [21] have emerged. These relational database-as-a-service (DaaS) offerings are data platforms where the service provider assumes responsibility for provisioning machines, patching, ensuring high availability, geo-replication etc. The attraction of DaaS is the familiar relational database paradigm combined with promise of reduced total cost of ownership.

These cloud database services are by necessity *multi-tenant*. Multi-tenancy is crucial for the service provider to increase consolidation and reduce cost, since statically reserving resources up-front for each tenant can be too expensive. One architecture for such cloud database services (e.g., used in Microsoft Azure SQL Database) consists of multiple tenant databases hosted inside a single database server. In such a multi-tenant database system, the resources of the database server must be shared by the SQL workloads of all active tenants assigned to that server. Therefore, when one tenant's workload executes, it can affect the performance of other tenants' queries on the same server adversely.

There is a fundamental tension between the need for the service provider to reduce cost by increasing multi-tenancy, and the desire of tenants for assured resources for their workload. A DaaS provider would like to *overbook* resources, i.e., promise more resources to tenants in aggregate than is physically available on the machine, yet without affecting tenant performance. This is based on the observation that at any point in time some tenants on the machine are likely to use much fewer resources than they are promised. The intuition here is conceptually similar to airlines that overbook seats on a flight.

In the SQLVM project [19] we have developed resource governance techniques that enable the service provider to offer the assurance of *reserved resources* to a tenant, thereby isolating one tenant's performance from resource demands of another tenant without requiring static allocation of resources. We presented our approaches for addressing this challenge for CPU and I/O resources in [7] and [19] respectively. Recently, Microsoft Azure SQL Database has released offerings of Service Tiers and Performance Levels, which incorporate performance isolation for different resource dimensions [23]. These offerings use the techniques developed in [7] and [19] as the underlying control mechanisms for performance isolation.

In this paper, we study the problem of how to effectively share *buffer pool memory* in a multi-tenant setting. The buffer pool serves as a cache of database pages and is crucial for good performance of the tenant's workload. The first challenge is to define accountability of the service provider when buffer pool memory is shared among multiple tenants. We observe that the impact of a cache on performance of a tenant's workload depends not only on the size of the cache but also on the access pattern of the workload. Consider the case where a tenant is promised $1GB$ of buffer pool memory by the service provider. If the service provider statically reserves $1GB$ of buffer pool memory for the tenant, then the tenant's workload would achieve a certain *hit ratio*, i.e., a certain fraction of the pages accessed by the tenant's workload would be found in the buffer pool. Now suppose instead that the service provider does not statically reserve $1GB$ of buffer pool memory but allows memory to be dynamically shared among all tenants. Then the same workload of the tenant would potentially achieve a *different* hit ratio, depending on the memory demands of other tenants. If the tenant's hit ratio reduces due to multi-tenancy, we use this degradation in hit ratio of the tenant's workload compared to a "baseline" where memory

Proceedings of the VLDB Endowment, Vol. 8, No. 7

is statically reserved, as a quantitative measure of the impact of multi-tenancy on performance. Formally, we define the *Hit Ratio Degradation (HRD)* as the difference in total hits, between the baseline and any scheme that assigns memory dynamically, normalized by the total memory accesses. This metric captures the reduced number of page hits that the tenant incurs due to the fact that it is deprived of memory that it was promised.

Given the HRD definition, the next challenge is to efficiently measure HRD via *metering*. In particular, we need to compute the HRD for a tenant due to the effects of multi-tenancy. While measuring the hit ratio of the tenant's workload in the actual system is straightforward, computing the hit ratio for the baseline case requires *what-if* analysis. We show how to perform this computation with low memory and CPU overheads. Depending on the HRD value, the service provider may incur a certain *penalty* (specified via a penalty function) for not being able to deliver on its promise. We note that different *SLAs* could be defined between the tenant and service provider that determine the exact nature of this penalty (see Section 2).

The final and most crucial challenge pertains to the page replacement algorithm in a multi-tenant DaaS. The family of *LRU-K* [20] algorithms have been shown to be effective in managing buffer pool memory in relational database servers, e.g., the Microsoft Azure SQL Database service also uses a variant of LRU-K. The basic idea of LRU-K is to keep track of the timestamps of last K references to a page and use this information to estimate the probability of reference of the page. The page that is evicted is the one with the smallest estimated probability of reference. In a multi-tenant setting however, the page replacement algorithm must incorporate two additional requirements. First it needs to handle the *asymmetry* of tenants, who might differ on amount of promised memory and penalty functions used. Second, the page replacement algorithm must be flexible to optimize for different service provider objectives. For instance, if the penalty leads to money being refunded to tenants as compensation, the service provider may want to minimize the sum of total money refunded with a constraint on fairness across tenants. Alternatively, the service provide may wish to optimize a different aggregate measure, e.g., maximize the number of tenants whose HRD is below a certain threshold (say 10%). Due to these additional requirements, the penalty associated with evicting a page of one tenant may be much higher than evicting a page from another tenant, even though the two pages appear indistinguishable via LRU-K alone. As we show in this paper, using a standard page replacement policy globally across all tenants can lead to unnecessarily high penalties for the service provider. Furthermore, any solution must also take into consideration practical issues such as eviction of pages in batches and bookkeeping overheads. Thus, there is a need to rethink global page replacement in a multi-tenant setting while preserving the goodness of LRU-K for any individual tenant.

Leveraging ideas from prior work on weighted online caching [31], we first develop an "SLA-aware" page replacement algorithm for a simplified, non-batch version of the problem and provide worst-case guarantees on penalties incurred by the service provider. Intuitively, the algorithm logically associates with pages of each tenant, a weight equal to the marginal increase in penalty currently incurred by the service provider for that tenant. Pages with higher weight age more slowly in the buffer pool and hence are less likely to be evicted. Based on this intuition, we design *MT-LRU*, a multi-tenant page replacement algorithm that handles the practical considerations of a relational database engine. Besides leading to lower penalties for the service provider, MT-LRU has other desirable properties. First, it generalizes LRU-K to the case of multiple tenant with SLAs, i.e. (a) for any individual tenant the order of its pages evicted is the same as in LRU-K, and (b) if all tenants have identical SLAs, then the algorithm reduces to running LRU-K on the entire buffer pool. Second, MT-LRU can seamlessly incorporate non-linear penalty functions, as well as additional service provider constraints such as fairness across tenants. Third, it can be implemented with minimal overhead compared to LRU-K.

We have implemented a prototype of the above techniques: metering and the MT-LRU page replacement algorithm inside the database engine of Microsoft Azure SQL Database taking into account the above mentioned practical considerations. We have conducted an extensive set of experiments to measure accuracy and overheads of metering, and the effectiveness of the MT-LRU page replacement policy. The results of these experiments demonstrate the effectiveness of our techniques, and show that the new page replacement algorithm significantly reduces service provider penalties compared to using the standard LRU-K policy when the system is overbooked. In particular, in a range of experiments that span settings where a page replacement mechanism could affect total penalties, MT-LRU reduces penalties by 30 to 50 percentage points compared to LRU-K (cf. Section 5). We note that the techniques developed in this paper could in principle be applied to other large-scale cloud services where multiple tenants share a global cache (e.g., Amazon ElastiCache or Microsoft Azure Caching).

2. SLA FOR BUFFER POOL MEMORY

In this section, we describe the model for SLAs for buffer pool memory. An SLA between the provider and the tenant is defined by the following three components: (a) the amount of buffer pool memory. (b) the SLA metric - *hit ratio degradation (HRD)*. (c) The penalty function that describes how the service provider is held accountable for any value of HRD. Issues pertaining to how the SLA is metered are discussed in Section 3.

2.1 SLA Metric

The buffer pool in a DBMS is a cache of database pages and plays an important role in enabling good workload performance. When a query accesses a page, if it is found in the buffer pool, we refer to that access as a hit; otherwise it is a miss and the page must be fetched from disk. Thus, for a buffer pool of a given size and for a given workload (i.e., sequence of page accesses) the hit ratio is defined as:

$$\text{Hit Ratio}(HR) = \frac{h}{N} \quad (1)$$

where N is the total number page accesses and h is the number of page accesses found in the buffer pool. Note that $0 \leq HR \leq 1$. We observe that the hit ratio in the above equation HR has two implicit parameters: (1) The buffer pool size – the amount of memory in the buffer pool. (2) The workload – the sequence of pages accessed.

As described earlier, our model of SLA is relative to the "baseline", i.e., the buffer pool memory promised to the tenant. Say the tenant has been promised M GB of buffer pool memory. In effect, the service provider is promising the tenant that the actual hit ratio of the tenant's workload will be no worse than the hit ratio if M GB of buffer pool memory had been statically reserved for the tenant. We will refer to this as the *baseline*. If the service provider is unable to meet this promise, a penalty function (described in Section 2.2) is invoked depending on the degree to which the actual and baseline hit ratios differ:

$$\text{Hit Ratio Degradation}(HRD) = \max\{0, HR_B - HR_A\} \quad (2)$$

where HR_B is the hit ratio in the baseline case, and HR_A is the hit ratio in the actual (i.e. multi-tenant) case. Observe that the

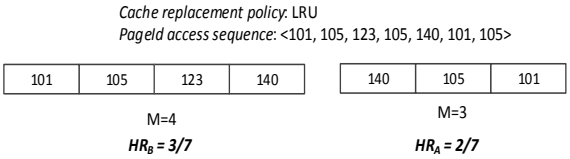


Figure 1: Example of hit ratio degradation.

Max operation is necessary since in general it is possible that HR_A is higher than HR_B. This might occur if the tenant is allocated *more* than M GB of buffer pool memory in the actual case - in such cases, we deem the hit ratio degradation to be 0. From the above definitions of HR and HRD, it follows that $0 \leq HRD \leq 1$. Combining Equations (1) and (2), we get:

$$HRD = \max\{0, \frac{h_B - h_A}{N}\} \quad (3)$$

Example 1. Consider a workload (sequence of pages accessed) shown in Figure 1. For the purposes of this example, suppose the page replacement policy used is LRU. Now, say the tenant is promised a buffer pool with M=4 pages in the SLA; but receives only M=3 pages in the multi-tenant system due to demands of other tenants. For the sequence of pages accessed by the tenant's workload, HR_B = 3/7 and HR_A = 2/7. The final contents of the buffer pool after all pages have been accessed (in each case) is shown in the figure. Thus $HRD = 3/7 - 2/7 = 1/7$.

2.2 Penalty Function

The penalty function defines how the service provider will be held accountable for any value of HRD that occurs in the actual case. The penalty function is an integral part of the SLA and is important from both the tenant and service provider's perspectives. For the tenant, a penalty function quantifies a potential compensation as a function of the performance degradation. For the service provider it provides the basis upon which to prioritize resource allocation across different tenants.

Note that the above notion of penalty can support a variety of scenarios ranging from internal memory provisioning within an enterprise to external DaaS offering. For example, a service provider may aim to maximize the number of tenants whose HRD is below a certain threshold. Alternatively, the service provider may want to minimize the monetary penalty which could be associated with the SLAs. For concreteness, we henceforth focus on a particular penalty model, which covers the above scenarios. In particular, we assume throughout that the penalty function consists of two elements: (i) an SLA "price" d; (ii) a normalized penalty function $g : [0, 1] \rightarrow [0, 1]$. The penalty for a given HRD is obtained by multiplying the two elements, namely $d \cdot g(HRD)$. Such cost structure has a simple interpretation in the sense that $g(\cdot)$ is the percentage price refunded to the tenant from the SLA price. In the rest of the paper, we often refer to g for simplicity as the *penalty function*, keeping in mind that its value ought to be multiplied by the SLA price d. We emphasize that the SLA prices are exogenous in our model, namely, how the provider does/should set prices is outside the scope of this work.

We perform our evaluation on (normalized) penalty functions whose functional form is either step-based or piecewise linear. See Figure 2 for examples of these two types of normalized penalty functions. Step-based functions are quite powerful, and can capture real world requirements, which are often non-linear. For example, availability SLAs in some cloud service providers today

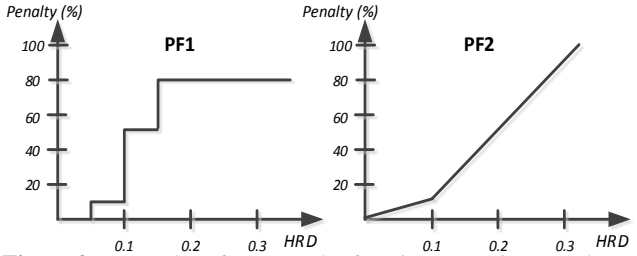


Figure 2: Examples of two penalty functions. PF1 is a step-based function, namely consists of multiple steps. PF2 is a piecewise-linear function.

takes the form of a step function such as: no penalty if availability 99.9% or higher, 10% penalty if availability between 99% and 99.9%, 50% penalty if availability between 90% and 99% etc. In our context, PF1 in Fig. 2 refunds at least half of the SLA cost if the HRD goes above 10%, and up to 80% refund for HRD greater than 15%. Piecewise linear penalties, also have a plausible interpretation: Each linear section corresponds to a fixed per-unit-HRD penalty. For example, under PF2 in Fig. 2, the provider will pay back 1.5% per one percent of HRD until reaching an HRD of 10%. From that point onwards, the per-unit refund increases to 3.5%, reflecting increased tenant dissatisfaction. While our experiments use the above two types of penalty functions, we note that the algorithm we develop in Section 4 does not make any assumptions about the particular form of the underlying penalty function; all it requires is a penalty function whose derivative is well defined or at least can be approximated.

2.3 Discussion

Today's DaaS offerings do not expose SLA metrics which are directly related to buffer pool memory performance. Instead, providers expose a fixed set of performance tiers, each associated with some level of "throughput units" (e.g., DTUs – Database Throughput Units – by Microsoft Azure SQL Database [23]). These units are based on a blended measure of CPU, memory and other resources. While buffer pool memory SLAs, such as described above, may be exposed to more performance-sensitive customers in the future, even under existing offerings, providers could still use our HRD-based SLAs internally (without exposing them to customers). For example, Premium, Standard and Basic contracts can be internally associated with penalties $d_i \cdot g(HRD)$ ($i = 1, 2, 3$ respectively), where $d_1 = 2d_2 = 4d_3$. Such mapping would facilitate dynamic division of buffer pool memory based on actual usage, rather than a static and potentially inefficient allocation.

3. EFFICIENT SLA METERING

Consider a tenant with an SLA for buffer pool memory as described in Section 2, whose workload is currently executing. Recall that the performance metric upon which the SLA is based is the hit ratio degradation (HRD) described by Equation (3) in Section 2.1. To enforce the SLA, the metric HRD needs to be computed. Depending on the HRD and the penalty function, the penalty incurred by the service provider is determined. The metering can be done at a suitably fine-grained interval (e.g., 30 secs or 1 minute).

To compute HRD (see Equation (3)), three quantities need to be measured for the tenant's workload: h_B,(the number of hits in the baseline), h_A (number of hits in the actual system) and N (the total number of pages accessed by the tenant's workload). Measuring h_A and N is relatively straightforward, and can be done at low

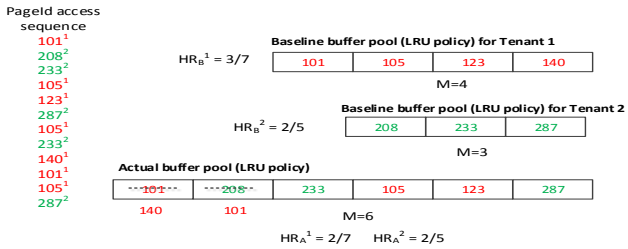


Figure 3: . Example of hit ratios in Baseline and Actual Buffer Pools for two tenants.

overhead - in fact today's DBMSs already keep counters that measure the number of hits and the total number of pages accessed in the buffer pool. The only extension required is to keep these counters *per tenant*. The main challenge however is measuring h_B of the tenant, since this is the hit ratio in the baseline setting. We first describe how the h_B can be measured with high accuracy. Next we analyze the memory and CPU overheads associated with such metering.

3.1 Measuring hit ratio in Baseline

To illustrate what it means to measure the number of hits in the baseline for a tenant's workload, consider the following example.

Example 2. Suppose we have two tenants: Tenant 1 and Tenant 2; and execution of their workloads results in a sequence of page accesses as shown in Figure 3. The superscript denotes which tenant accessed the page (note that tenants do not share data). Also, suppose that the actual buffer pool has 6 pages, and Tenant 1 and Tenant 2 were promised 4 pages and 3 pages respectively in their SLAs; thus the system is overbooked. Observe that in the actual case, the hit ratio achieved for Tenant 1 is $HR_A^1 = 2/7$. Figure 3 depicts which pages are in the buffer pool at the end of the sequence of page accesses, and also shows which pages were replaced (e.g. 101, 208, 233 are replaced assuming LRU policy). Similarly the hit ratio for Tenant 2 in the actual case is $HR_A^2 = 2/5$. However in the baseline case for Tenant 1, which has 4 pages, the first 4 page accesses are misses, but after the remaining 3 pages accesses are hits. Thus $HR_B^1 = 3/7$. Likewise, in the baseline case for Tenant 2, $HR_B^2 = 2/5$.

The basic observation which enables us to measure HR_B efficiently is that we only need to know the following two pieces of information to compute HR_B for a tenant: (a) The sequence of pages accessed by that tenant. (b) The number of pages promised to the tenant. Since we know the page replacement policy of the DBMS, we can then simulate the exact page replacement policy that the DBMS would have carried out for the given workload sequence and the promised memory size.

The key question is therefore the computation and memory overheads incurred by such a "what-if" simulation. The new data structure required is a baseline buffer pool for each tenant T. The baseline buffer pool is identical to a real buffer pool except it does not contain the actual data pages, rather it only tracks the *page-IDs* of pages accessed by tenant T.

The page replacement policy in Microsoft Azure SQL Database is a variant of LRU-K [20], which we need to simulate during metering. Thus in our implementation, we maintain: (a) A hash table of page-IDs per tenant, and associate the LRU-K counters with each page id. (b) A free list to store page-id reclaimed by the replacement scheme. Our simulation of the replacement policy of Microsoft SQL Server captures many important details. For example, the policy defines a notion of correlated references ([20]) to a page to avoid giving unnecessary importance to pages that experience a burst of references very close to one another (e.g. within a single transaction). Such correlated references are treated as a single reference to the page.

The above algorithm can, in principle, be adapted to other DBMSs that might use a different page replacement policy. The CPU and memory overheads would vary depending on the exact page replacement policy being simulated. For example, in the original LRU-K algorithm there is an additional data structure for tracking pages that have been discarded from the buffer pool for a certain period of time – referred to as the Retained Information Period [20]. Such logic could also be implemented, although it would add some memory overhead for tracking page-id discarded from the baseline buffer pool.

3.2 Analysis of Metering Overheads

CPU Overheads: During the "what-if" simulation, when the page id is found in the baseline buffer pool, the overheads only pertain to finding the page-id in a hash table and incrementing a counter. When the page-id is not found in the hash-table, then additional CPU overheads are incurred to simulate page replacement in the baseline buffer pool. Since this code is invoked only when the page-id is not found in the baseline buffer pool, the CPU time it incurs is proportional to the hit ratio of the tenant's workload in the baseline buffer pool. As we show in our experiments (Section 5), across a variety of different settings, this overhead is typically very small - negligible enough in practice that it falls within bounds of measurement error.

Memory Overheads: There are two components of memory overheads. First there is a fixed overhead per page frame in the actual buffer pool. This is required to track which tenant that page belongs to. Assuming that a tenant id can be stored in 4 bytes and each page is 8KB, this fixed overhead is 0.048% per page in the actual buffer pool. Second, there is a variable amount of memory overhead for the data structures of the baseline buffer pool of each tenant. This depends on the memory promised to the tenant. In the worst case, this is the memory for tracking page-IDs and associated page replacement counters (e.g., LRU-K counters) per page promised to the tenant. Observe that at any point in time, if the tenant's workload does not actually fill its promised size in the SLA, then the worst case may not be reached. We illustrate the variable memory overhead with an example. Consider an actual buffer pool of size 16GB and a page size of 8KB. Suppose that the tenants in aggregate are promised 32GB buffer pool memory, and each tenant accesses sufficient number of pages such that their respective baseline buffer pools would have been full. Suppose also that bookkeeping overhead of the LRU-K page replacement scheme is 40 bytes per page. In that case the total variable amount of memory would be 140MB (which is around 0.85% of 16GB). Finally, it is important to note that the variable memory overhead also depends on how much the service provider overcommits memory on the system (as described earlier) - the greater the degree of overcommitment, the higher the worst case memory overhead.

4. MULTI-TENANT PAGE REPLACEMENT ALGORITHM

In this section, we present our online page replacement algorithm MT-LRU that enables effective support of SLAs for buffer-pool memory in a multi-tenant setting. We start in Section 4.1 by framing the problem as an online caching problem and identifying the key differences from the "classic" online caching models. We outline the algorithmic challenges in Section 4.2. In Section 4.3, we

highlight the intuition behind MT-LRU, using a simplified version of the algorithm, which we term Sequential-MT-LRU; theoretical guarantees for the latter are also provided. A full description of MT-LRU, including a discussion of implementation issues appears in Section 4.4.

4.1 Formulation

As mentioned earlier, we treat our problem as an *online caching* (or *paging*) problem [24], where the buffer-pool plays the role of a cache (we use the terms "buffer pool memory" and "cache" interchangeably throughout this section). In the basic online caching problem, we have a cache of size B and a sequence of page requests $\sigma = (p_1, \ldots, p_T)$. At every time step t, we receive an access to page p_t and if the page is not in the cache, we incur a page miss and the page p_t is brought in the cache. If the cache is full and there is no space for page p_t, the algorithm has to make a decision about which page to evict from the cache. The cost of any algorithm is the number of page misses over the complete sequence.

In our context, we deal with a multi-tenant setting with asymmetric tenants: We have a fixed set of tenants that share a common buffer pool of size B. Each tenant has an SLA with the database provider. The individual SLA includes (i) a cache of size b_i, (ii) a (possibly non-linear) *penalty function* $f_i : [0, 1] \rightarrow \mathbb{R}^+$. Each tenant is promised a cache hit ratio for its workload commensurate to having a cache of size b_i and running LRU (or LRU-K) on this cache for the pages belonging to the tenant. Unlike traditional caching models, the algorithm is not charged by each miss individually, but rather pays for the total number of memory misses. That is, if the algorithm makes n_i misses for tenant i's pages while the baseline algorithm running on a cache of size b_i has m_i misses, then the hit ratio degradation (HRD) for tenant i is $HRD_i = \max\{0, \frac{n_i - m_i}{N_i}\}$, where N_i is the total number of page accesses of tenant i. The service provider's objective is to minimize the sum of penalties $\sum_i f_i(HRD_i)$.

As mentioned in Section 2, the above formulation can correspond to maximizing revenue, namely $f_i(HRD_i) = d_i g_i(HRD_i)$ (examples for g_i are given in Figure 2). The same formulation, however, can be applied for additional objectives; for example, the provider may wish to minimize the number of tenants whose HRD is greater than 10%. This objective could be easily expressed using penalty functions, by having a step-based function for each tenant, with a single step at $HRD = 10\%$.

4.2 Challenges

Variants of LRU, in particular LRU-K [20], perform well in practice for database systems – theoretical guarantees can be established under stochastic assumptions. Page replacement in LRU-K depends only on the *times* of the last K accesses to a page. Our focus is on designing an algorithm that would reduce to LRU-K for any individual tenant while being able to handle new requirements arising from multiple tenants with SLAs including: (i) Asymmetry of tenants. Tenants could differ in the amount of memory promised in the SLA and their penalty functions. Furthermore, different tenants could be active at different times. (ii) Unlike LRU-K which aims to maximize the likelihood of a page being found in the cache, we need to optimize for different objectives, e.g., minimizing total penalties. This is hard, particularly since penalty functions could be non-linear (see Section 2.2). Observe that the requirement of reducing to LRU-K for an individual tenant means that the sequence of evictions of pages of a tenant is the same as in LRU-K. Therefore, the challenge is to devise a method that can capture both the effects of *time* and *penalty* when making page replacement decisions. Finally, any viable page replacement algorithm must also accommodate practical considerations that arise while implementing in a relational database engine (see Section 4.4.2).

```
1  Let current time be T and let p_T be the accessed page
2  Let i be the tenant owning this page, and let n_i be its total
   number of misses
3  Set c(p_T) = f'(n_i)
4  if page p_T is not in the cache then
5      n_i = n_i + 1
6      Set c(p) = c(p) + f'(n_i) − f'(n_i − 1) for any other
       page of tenant i
7      if there is space in cache then
8          Place page p_T in the cache
9      else
10         Let p* be the page with the smallest c(p) value in
           the cache; evict page p* and replace with p_T.
11         Set c(p') = c(p') − c(p*) for each page p' in the
           cache
12     end
13 end
```

Algorithm 1: Pseudo-Code of Sequential-MT-LRU algorithm.

The above challenges rule out several obvious approaches to tackle the problem. We briefly discuss below the most prominent ones. One straightforward approach is to allocate a dedicated fixed part of cache to each tenant, in some proportion based on its SLA and possibly other competing tenants' SLAs. Unfortunately, such an algorithm is not able to utilize the fact that peak activity of tenants can be temporal and the cache size dedicated to a tenant should vary over time accordingly. An orthogonal approach is to run LRU-K on the whole cache. However, this algorithm does not distinguish between tenants and could suffer badly when tenants have asymmetric SLAs, for example when tenants have been promised different amounts of memory or the penalty functions differ significantly over tenants (see Section 5 for experimental results). Another approach is to directly use algorithms for weighted caching [2, 31] which generalize LRU to settings when pages have different weights. Such an approach fails in our setting since penalty functions can be non-linear, i.e. each additional miss does not necessarily cost the same as is assumed in weighted caching.

4.3 Key ideas behind MT-LRU

To understand the intuition behind MT-LRU, we first present a simplified version of the MT-LRU algorithm, which we refer to as Sequential-MT-LRU. We make two simplifying assumptions. First, we assume that cache eviction decisions can be made after each page access, whereas in practice, database systems need to perform evictions in batches to control overheads. Second, we assume that the provider is penalized according to the total number of misses of each tenant, rather than based on HRDs; with some abuse of notion, we use $f_i(n_i)$ for the penalty function of tenant i, where n_i is its total number of misses.

To better isolate the essential elements pertaining to requirements from multi-tenancy, we develop Sequential-MT-LRU as an "SLA-aware" version of LRU (rather than LRU-K, $K > 1$). We describe below the main aspects of the algorithm (see Algorithm 1 for the pseudo-code).

Handling penalties: Observe that when a page p is accessed, the Sequential-MT-LRU algorithm assigns it a *shadow price* $c(p)$ which is set equal to the *derivative* of the penalty function for the tenant owning the page, evaluated at the current time (Line 3). The intu-

ition is that this assignment is equivalent to the *additional* penalty that will be incurred if the tenant owning this page has a miss at the current time T. When the algorithm needs to make space in the cache to accommodate a newly referenced page, it evicts the page with the smallest $c(p)$ value, and reduces shadow prices of all other pages in the cache by that amount. The assignment of shadow prices to pages deals with the issue of asymmetry among tenants as well as non-linearity of penalty functions[1]. In particular, the update of the shadow price (Line 11) incorporates both *time* and *penalty* considerations: a page that remains in the cache sees its shadow price value reduced until reaching zero and being evicted; this achieves the effect of aging. The principle of accounting for the page's age appears also in traditional LRU-K. However, unlike LRU-K, each elapsed time unit is weighted differently. Intuitively, existing pages in the cache should be "punished" more for an eviction of a page with higher $c(p^*)$. This factor, along with the initial assignment of $c(p)$ which captures penalty considerations, distinguishes Sequential-MT-LRU from traditional LRU-K. Note that pages with high shadow price are expected to remain longer in the cache, but not necessarily in full correlation with their age as in LRU-K. For example, a newer page of a tenant with low penalty derivative could be evicted before an older page of a different tenant with high penalty derivative.

Reducing to LRU for an individual tenant: Observe that in Line 6, the $c(p)$ values of all pages of the tenant are either increased or decreased by the same amount equal to $f'(n_i) - f'(n_i - 1)$. Since at any point in time the $c(p)$ of an older page of the tenant has been reduced more times than a newer page (Line 11), the ordering of the tenant's pages by $c(p)$ values is consistent with the ordering that would have been achieved by LRU. Ties in $c(p)$ if any, could be broken using the standard LRU timestamp.

Guarantee on penalties: We next provide a worst-case guarantee on penalties for the sequential-MT-LRU algorithm. Towards that end, we assume that each $f_i(\cdot)$ is a convex increasing function. Under this setting, we give a guarantee on the total penalty incurred by the algorithm relative to the best *offline* algorithm for the problem, which knows the sequence of arriving pages in advance. We note that the technical assumption of convexity of $f_i(\cdot)$ is necessary for the proof; and it is natural in the sense that each additional miss has an increasingly greater impact on tenant performance.

THEOREM 1. *The algorithm Sequential-MT-LRU returns a solution such that* $\sum_i f_i(n_i) \leq \sum_i f_i(\gamma \cdot B \cdot o_i + B)$*, where* B *is the buffer-pool size,* o_i *is the number of misses for tenant* i *in the best offline algorithm and* $\gamma = \max_{i,x} \frac{x f_i'(x)}{f_i(x)}$.

Proof is omitted for brevity. We note that in order to demonstrate the robustness of our algorithm, we include in our experiments penalty functions that are not necessarily convex (see Section 5.4). Finally, it is worth noting that if $f_i(x) = x$ for each tenant i, then the objective is to minimize the total number of misses. Theorem 1 suggests that algorithm Sequential-MT-LRU is B-competitive which is the best deterministic guarantee of any algorithm [24].

Similar theoretical guarantees cannot be obtained for the more complicated setting of MT-LRU. Nevertheless, the main ideas behind Sequential-MT-LRU form the basis for MT-LRU's design: (i) The use of shadow price as the basis for page replacement (ii) The use of the penalty function's derivative as the initial shadow price of a page.

4.4 The MT-LRU algorithm

[1]In cases where the penalty functions are not differentiable or even discontinuous, we first take a smooth approximation of the function and then take its derivative.

We first highlight the practical considerations that must be taken into account in any viable implementation of a page replacement algorithm such as batch evictions and low additional bookkeeping overhead. We illustrate these requirements by summarizing relevant aspects of the implementation of LRU-K in Microsoft Azure SQL Database. Next we describe MT-LRU which carries over the main ideas behind the simpler sequential algorithm, while incorporating these practical requirements and ensuring that page replacement *within* pages of a tenant still reduces to the LRU-K algorithm.

4.4.1 Page replacement in Microsoft Azure SQL Database

The page replacement policy in Microsoft Azure SQL Database is a variant of LRU-K. It maintains a free list of pages from which a page is allocated when a new page is accessed. When the number of pages in the free list becomes small, it reclaims a certain fraction (α) of pages currently in the buffer pool and places them in the free list. The pages reclaimed are those having the largest estimated inter-arrival time as required by LRU-K. Observe that determining these pages exactly could be expensive when the buffer pool memory size is large. Therefore, for efficiency, this step is done by sampling l pages and determining a cutoff threshold based on those sampled pages for reclaiming the desired fraction α of pages. Then a "clock hand" (similar to the CLOCK algorithm [5]) sweeps across the pages reclaiming all pages whose LRU-K timestamp is below the previously determined cutoff value, until the desired number of pages has been reclaimed. The key points to note are that page evictions are done in batches, the use of sampling to determine cutoff threshold based on LRU-K timestamp, and that typically α is reasonably large (e.g., 0.25) to keep overheads low.

4.4.2 Practical considerations

Sampling: Since a relatively large fraction of the cache is evicted in each batch, we need to find the right thresholds for batch eviction using a sampling mechanism.

Smoothing: Despite evicting pages in batches, we want pages belonging to the same tenant with similar LRU-K time stamps to have similar shadow prices; this helps reduce unnecessary fluctuations in buffer pool memory allocated per-tenant.

Efficient bookkeeping: In principle, for each eviction, we need to account for the fact that multiple pages are evicted and reduce the shadow prices of existing pages in the cache accordingly. To keep bookkeeping overheads low we must avoid updating every page's shadow price (as in Line 11 of the Sequential-MT-LRU algorithm).

Time thresholding: While we use the shadow prices on the sampled pages to determine which pages to evict, the eviction criteria should be translated to using simple time thresholds (as in LRU-K) for two reasons. First, for an individual tenant, we want the sequence of pages evicted to be the same as in LRU-K. Second, we would like to avoid any shadow price calculations during the eviction loop itself. With time-thresholding, we simply maintain one time threshold for each tenant, and remove every tenant's pages in the buffer pool whose relevant LRU-K timestamp is earlier than the threshold.

4.4.3 Eviction in MT-LRU

The pseudo-code of the main loop of MT-LRU is given as Algorithm 2. For every tenant i, we are given a penalty function $f_i(HRD_i)$. For each tenant, we maintain the number of misses n_i as well as the number of misses m_i for the baseline case. Note that this logic must be implemented for SLA metering as discussed in Section 3, so no additional overheads are incurred due to MT-LRU. The shadow price of each page p is denoted by $c(p)$. We also maintain a *Virtual Cost* vector, denoted $VC(t)$ (indexed by time t). The vector $VC(t)$ stores an (approximate) α-percentile shadow

price among the pages in the cache (cf. Line 19). As we elaborate below, this vector is used for calculating the shadow prices. The vector is highly sparse, as it is updated only at times when the evict operation is called.

Sampling: In each evict operation, we first obtain a sample set of l pages drawn uniformly at random from the cache.

Smoothing: Each *sampled* page is assigned a shadow price (Line 9). Note that for each tenant i, the page with the latest timestamp is assigned a shadow price equal to the value of its current penalty derivative. Older pages are assigned a shadow price through a linear interpolation using the vector $VC(t)$ for times between the earliest LRU-K timestamp and current time; the interpolation achieves the required smoothing of shadow price values. Observe further that this update preserves the original LRU-K property that for each tenant, pages with more recent timestamps cannot be evicted before pages with older timestamps.

Efficient bookkeeping: The vector VC allows for efficient bookkeeping – instead of decreasing all shadow prices in each eviction operation, we simply hold the required price updates within that vector and do the updates in retrospect. $VC(T)$ is assigned the shadow price of p_{cric} (Line 19), where p_{cric} is the page whose shadow price is at the α-fraction of shadow prices among the l sampled pages (Line 11). Thus if we reduce the shadow price of all pages in the cache by $VC(T)$, then approximately α-fraction of pages will have their shadow price go below zero and therefore, can be removed. VC is an approximate measure because it is calculated based on the sampled pages. This means that in practice we will be evicting slightly more or slightly less pages than required, but this is not a significant issue given the scale involved.

Time thresholding: For each tenant i, we find the page p_i that is closest to p_{cric} in the sorted order. We then set τ_i (the time threshold of i) as the timestamp of page p_i. The actual evictions are done in the same manner as in the original Microsoft Azure SQL Database page replacement implementation: a clock hand loops over the buffer pool (Line 15) and removes all pages whose timestamp is earlier than the corresponding τ_i.

Computational Overhead: The computational overhead of MT-LRU over the standard Microsoft Azure SQL Database eviction mechanism is minimal. Since the algorithm outputs a time threshold for each active tenant, we conservatively sample a larger number of pages compared to the original implementation (2X); however, the increased sample size has negligible impact on running time. Further, the computational overhead of sorting shadow prices is similar to the overhead of sorting timestamps in LRU-K. Similarly, memory requirements of the algorithm are comparable to LRU-K. In particular, we emphasize that the memory overhead of the vector VC is independent of the number of tenants, and only needs to be updated during batch evictions, which are relatively infrequent. In addition, the system can ignore very old virtual costs for all practical purposes (i.e., treat them as zeros). Together these imply that in practice the memory requirements for the VC vector are at most a few thousands of entries, where each entry consists of a timestamp (of the eviction) and the corresponding VC value.

4.4.4 Additional implementation details

Handling step-based functions: To use MT-LRU when penalty functions are step-based (e.g., PF1 in Fig. 2), we require a preprocessing step of *smoothening* the penalty function. Intuitively, the reason is that the derivative of such functions does not capture useful information about the "marginal penalty" of the tenant (note that the derivative is either 0, or undefined at the break points). To address this issue, we smoothen the penalty function by interpolating the break points of the function, resulting in a piecewise linear function, whose derivative is well defined (with the exception of the break point themselves, for which we arbitrarily pick either the right or left derivatives).

Fairness: While our objective is to minimize the sum of penalties, we may also wish to enforce a certain *fairness* criterion to ensure that tenants with less stringent penalty functions are not completely starved. MT-LRU is flexible to allow for such refinements. To that end, in practice we modify the penalty function by increasing the derivative of the penalty function for high HRDs. This ensures, for example, that the HRD of low paying tenants will not be extremely high as compared to higher paying tenants. Using similar techniques, it is also possible to ensure that each active tenant receives a small minimum number of pages regardless of the penalty function.

```
1  Let current time be T.
2  Sample l pages uniformly at random from the cache
3  foreach tenant i do
4      Let t_i^0 denote the earliest LRU-K time stamp of any page
       of tenant i in the sample.
5      Set AVC_i = sum_{t=t_i^0}^{T} VC(t)
       // Observe that AVC_i is dependent on i via
          the starting point of the sum.
6      Let y_i be the total number of pages of i in the sample.
7      For each tenant, sort its pages by the LRU-K timestamp
       in increasing order;
8      Denote by pos_i(p) the position of page p of tenant i in
       the sorted order, pos_i(p) ∈ [0, y_i − 1]
       // Shadow price assignment:
9      Set c(p) = (f_i'(HRD_i) − AVC_i) + AVC_i · pos_i(p)/(y_i − 1)
10 end
11 Sort the l pages in increasing order of c(p)'s, let p_cric denote
   the page at α fraction.
12 foreach tenant i do
13     Let p_i denote a page of tenant i whose shadow price is
       closest to c(p_cric). Let τ_i denote the timestamp of p_i
       (set τ_i to zero if p_i doesn't exist)
14 end
15 foreach page p do
16     Let i be the tenant owning page p
17     Remove p from memory if its time-stamp is less than τ_i.
18 end
19 Set VC(T) = c(p_cric).
```

Algorithm 2: Pseudo-Code of MT-LRU's $\mathbf{EVICT}(\alpha)$ operation; α is the fraction of pages that need to be evicted.

5. EXPERIMENTS

We have implemented the techniques described in this paper for SLA metering (Section 3) and multi-tenant buffer pool page replacement policy (Section 4) in a commercial database engine: Microsoft Azure SQL Database. The goals of our experiments are to evaluate: (a) The accuracy and overheads of SLA metering. (b) The effectiveness of the new buffer pool page replacement policy.

5.1 Experimental Setup

Machine and Database Server: We use a machine with a 2 processor, Quad-Core, 2.1GHz CPU, and 32GB RAM. We run an instance of the Microsoft Azure SQL Database server extended as described in this paper for supporting multi-tenancy. Although the machine has 32GB physical memory, for a given experiment we

can configure the database server so that it uses only a specified amount of memory for the buffer pool (e.g. 8GB). Database: Each tenant's data resides in a separate database. We use a copy of the TPC-H [23] 10GB database for each tenant.

Workload: Since our techniques pertain to the buffer pool, we generate workloads so that we can systematically vary the page access patterns, and hence control the buffer pool memory demands of each tenant. We therefore generate synthetic workloads, where each query performs a range access of q pages (q is a parameter with a default value of 10) on the clustered index of the lineitem table in TPC-H. The entire clustered index is around 8GB in size. We control locality in the workload by generating queries where the start of a range is selected at random using a Zipfian distribution (which follow a power law); where the Zipfian factor Z=0 is a uniform distribution that gets progressively skewed as Z increases. In addition to the above workload that accesses pages at random in the table, we also generate workloads that sequentially scan a given number of pages from the table. We refer to these two kinds of workloads as RAND and SCAN respectively. For each tenant we execute the workload on 4 connections simultaneously and on each connection we run 100,000 queries.

Number of Active Tenants: For any given experiment, we pick a certain number of tenants (between 1 and 32) to execute their workloads. In practice in DaaS, it is rare that more than a few tenants (typically around 10) are *active* on a server at a particular point in time. By controlling the physical memory configured for the buffer pool, and the memory promised to tenants, we can control the overall degree of overbooking by the service provider.

Buffer Pool Aware Page Replacement strategies: As a default for comparison we use the native policy of Microsoft Azure SQL Database, which uses an adaptation of LRU-K as described in Section 4. We refer to this policy as SQL. In addition to MT-LRU, we also include for comparison another SLA-aware page replacement policy that we refer to as CURRENT. In this policy, the importance of a page during eviction is computed as the *current* penalty for the tenant according to the penalty function divided the inter-arrival time as estimated by LRU-K. Thus, while CURRENT captures effects of both LRU-K and penalty, it does not incorporate the marginal changes in the penalty function, and is thus potentially susceptible to non-linear penalty functions.

5.2 Accuracy of SLA Metering

We consider the case where there is a single tenant who has been promised X GB of memory (X is varied between 1 and 8) and the actual buffer pool memory of the server is also set to X GB. In this case since all the memory in the actual buffer pool is available for this single tenant, we know that if SLA metering is perfectly accurate, then the number of hits in the actual (h_A) and baseline (h_B) should be identical. We record the hits measured in the actual buffer pool, and also the hits measured by the SLA metering code for the baseline case. We define accuracy as $1 - \left|\frac{h_A - h_B}{N}\right|$. Thus, if the actual and baseline hits are identical, accuracy is 100%. We report the average (and min) accuracy over all metering intervals. Note that by taking the absolute value of the difference between h_A and h_B we count both overestimation or underestimation errors.

We first report the accuracy of SLA metering as the amount of buffer pool memory (M) promised to the tenant is varied between 1GB and 8GB and the skew in the workload is varied between Z=0 and Z=1.25. We find that the average accuracy is almost always close to 100%. There are a few time intervals where there are small errors (the largest of these never exceeds 3%). The explanation for this is that our technique for estimating h_B that simulates the actual buffer pool does not model all nuances of the behavior of the actual buffer pool. For example, the actual buffer pool replacement policy withholds under certain situations a small amount of memory in reserve (around 10 MB per core), and does not always give that to the workload. Since we do not model this effect in our simulation, we occasionally observe small errors. In principle, even these small errors could be further reduced by modeling such additional effects in the simulation code. Finally, we also note that there is a small amount of non-determinism arising out of the fact that in the page replacement policy of Microsoft Azure SQL Database the cutoff threshold is determined via a random sample. Thus two successive runs of the same workload could exhibit a slightly different behavior with respect to pages in the buffer pool. These experiments demonstrate that it is feasible to achieve very accurate SLA metering of hit ratio for buffer pool memory of a commercial DBMS.

5.3 Overheads of SLA Metering

To measure the computational overheads of SLA Metering, we run each workload with SLA metering turned off and turned on (but keeping all other settings identical), and measure the difference in end-to-end time of the workloads.

Varying promised memory: First, we consider a single tenant case. We vary the amount of memory promised to the tenant while keeping the physical memory for the buffer pool constant (to 4GB) and measure the percentage difference in time with and without SLA metering for different values of promised memory. We notice that the overheads are indistinguishable from the inherent variations of the system and experimental error. Only for one data point are the overheads positive, and in that case it is 0.04%.

Varying number of tenants: To measure the sensitivity of overheads to number of tenants, we vary the number of active tenants (1, 2, 4, 8, 16, 32), promising each tenant a memory of 2GB and holding the overbooking ratio constant at 2x. Once again we observe that the CPU overheads of what-if simulation for SLA metering are negligible (below 0.25%).

Varying workload skew: In the final experiment, we measure how the workload characteristics affect overheads. This time we hold the memory promised to a tenant constant at 8GB and the physical memory in the actual buffer pool to 4GB; and vary the Zipfian factor for the workload ($Z = 0, 0.5, 1.0$). We observe again that the overheads are negligible (below 0.5%). Overall, this experiment indicates that SLA metering of hit ratio in the baseline case can be implemented with very low overheads. As described in Section 4.2, the memory overheads are also modest. The accuracy and overheads results above indicate that SLA metering can be done effectively for hit ratio of buffer pool memory.

5.4 Effectiveness of MT-LRU

We next study the benefits of our SLA-aware page replacement policy. As described in Section 4, MT-LRU is designed to minimize the sum of penalties incurred over all tenants. We report here conclusive experimental results which show the advantage of MT-LRU over other page replacement algorithms.

5.4.1 Sensitivity to SLAs

In our first experiment, we study how the different buffer pool memory promised to tenants affects penalties. We set the actual buffer pool memory to 4GB and have two active tenants, each being promised a certain amount of memory (varied from 1GB to 5GB). In effect this varies the factor by which the service provider overcommits memory from 0.5 (for 1GB case) to 2.5 (for 5GB). Each tenant runs an identical SCAN workload. Both tenants use penalty function PF1 (see Figure 2), however Tenant1 has a price of $100 for the SLA, Tenant2 has a price of $10. This is a simple scenario

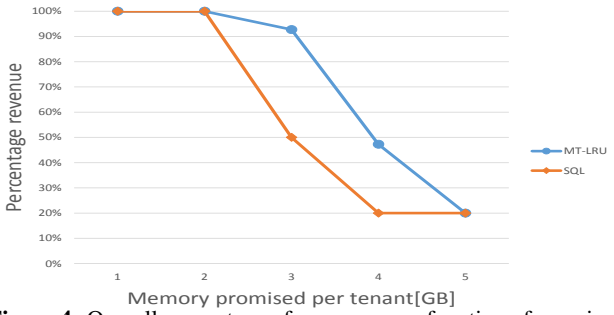


Figure 4: Overall percentage of revenues as a function of promised memory.

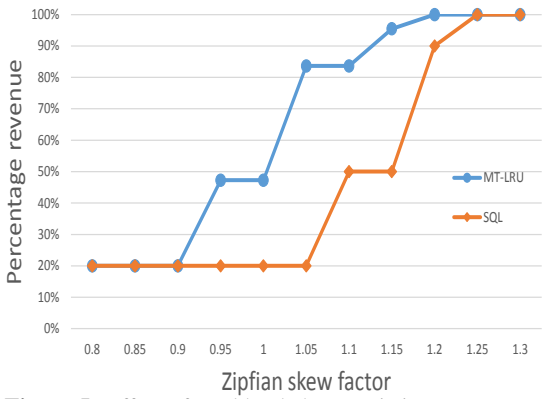


Figure 5: Effect of workload characteristics on revenue.

	Zipfian factor							
Revenues	0.9	0.95	1.0	1.05	1.1	1.15	1.2	1.25
T1 - SQL	20	20	20	20	50	50	90	100
T1 - MT-LRU	20	50	50	90	90	100	100	100
T2 - SQL	2	2	2	2	5	5	9	10
T2 - MT-LRU	2	2	2	2	2	5	10	10
Total - SQL	22	22	22	22	55	55	99	110
Total - MT-LRU	22	52	52	92	92	105	110	110

Table 1: Effect of workload characteristics- Revenue breakdown

where Tenant1 is much more performance sensitive than Tenant2 and is willing to pay more for it. In this setting, observe that minimizing penalty implies maximizing service provider *revenue* from the tenants.

Fig. 4 shows the results. The y-axis is the percentage revenue achieved by the policy, and the x-axis is the memory promised to each tenant . When the total memory promised to tenants is below the actual memory (4GB), both SQL and MT-LRU obtain the maximal revenue. This is expected, because each tenant gets at least its promised memory, thus the HRDs are 0 and no penalty is incurred. The other extreme occurs when each tenant is promised a large amount of memory (5GB or above), which substantially exceeds the total memory available. In this scenario, both tenants will exhibit maximal HRDs, hence the total revenue of 20%, which corresponds to a maximal penalty of 80% per tenant. The "interesting" scenario is when the promised memory per tenant lies within the interval $(2GB, 5GB)$ (excluding $2GB$ and $5GB$). For that memory range, the hit ratio degradation for at least one of the tenants is unavoidable since the memory available is less than what the tenant is promised and is demanded by its workload; on the other hand, for that range, the provider can avoid paying large penalties, by appropriately favoring one tenant over the other, based on penalty considerations. Accordingly, we can view the interval $(2GB, 5GB)$ as the *active range*, i.e., the range in which MT-LRU (or any other algorithm) can obtain penalty reductions. We shall use the notion of *active range* in similar contexts throughout this section.

As expected, the default SQL policy is unable to distinguish between tenants thereby incurring similar hit ratio degradations for each tenant. This results in a substantial penalty for Tenant1, the higher paying tenant. Notably, for a memory requirement of 4GB per tenant, the total penalty is maximal. On the other hand, MT-LRU allocates a disproportionate amount of buffer pool memory to Tenant 1 and is thus able to significantly reduce penalty over the active range.

5.4.2 *Sensitivity to Workload*

In this experiment, we vary the tenant workloads while holding other factors constant. Specifically, we set the actual buffer pool memory to 3GB and the number of tenants to 2, each being promised 4GB buffer pool memory. As before, both tenants use PF1. Tenant1 pays \$100 for the SLA, while Tenant2 pays \$10. We then run the RAND workload varying Z from zero to 1.3. The (normalized) total revenue is shown in Figure 5. We observe that MT-LRU consistently outperforms SQL in the active range, which is $Z \in (0.9, 1.25)$. To understand why, we also provide the revenue breakdown by tenant (Table 1). Compared to SQL, MT-LRU always obtains greater or equal revenues from the higher-paying tenant (Tenant1), which leads to higher total revenue. Interestingly, for $Z = 1.1$, MT-LRU "sacrifices" Tenant2 (obtains a lower revenue of \$2, compared to \$5 under SQL) to extract more revenue from Tenant1.

Generally, the active range in this set of experiments corresponds to scenarios where the access pattern over pages in the table is not too skewed, i.e., each tenant still accesses a large number distinct pages. In this case, there is significant pressure on the buffer pool memory and the page replacement policy plays an important role. As expected, MT-LRU is able to appropriately incorporate penalty considerations and thus do much better than the SQL policy. Outside the active range, both algorithms perform the same: For small Z's the effective working set of each tenant is too large, in the sense that the buffer-pool size is too small to accommodate even a single tenant (independent of the underlying algorithm). At the other extreme, for $Z > 1.25$ the workload has a fairly skewed access pattern (i.e., only a few distinct pages are accessed). Since the frequently accessed pages of both tenants combined likely fits within the buffer pool memory, the specific page replacement policy plays little role in this case, and both achieve revenues of 100% (i.e. no penalties).

Both the above experiment sets demonstrate the same qualitative properties. When the memory pressure is either too low or two high, then algorithms are not distinguishable in penalties. On the other hand, in the active range, the algorithm choice is crucial, and MT-LRU has significant merits. The experiments that we have reported so far compare the algorithms in a controlled setting: i.e., we modify a single dimension of the input (either the promised memory or the skew coefficient). We have also run experiments that examine hundreds of different settings; all of which exhibit substantial penalty reductions for MT-LRU. The average performance of the different algorithms (MT-LRU, SQL and CURRENT) is described at the end of this section.

5.4.3 *Drill-down into behavior of MT-LRU*

In order to better understand the behavior of MT-LRU, we drill down into the point $Z = 1.1$ in Figure 5 - where each of the two tenants is promised 4GB (for a total of 8GB), and the actual buffer pool memory on the machine is 3GB. We measure: (a) how much

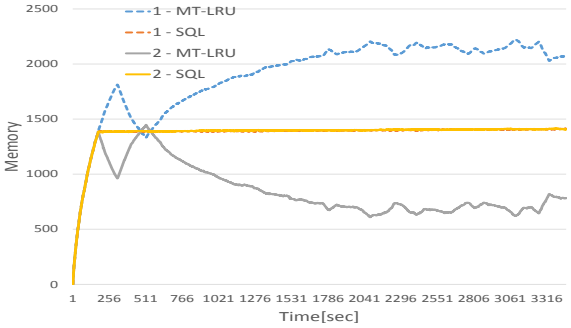
memory each tenant receives according to the policy, measured every second (see Figure 6(a)); (b) the hit ratio degradation of the tenants (Figure 6(b)). Recollect that the tenants execute identical workloads and that Tenant1 pays much more than Tenant2. For about the first minute or so, the tenants get almost equal amounts of memory (of around 1.5GB each) since the page replacement policy does not kick in until then. After that, there is no longer adequate memory in the buffer pool to keep all pages of both tenants. From that point on, MT-LRU policy tends to evict pages of Tenant2 much more, and thus more of Tenant1's pages are in memory. As time progresses, the system reaches a fairly steady memory allocation of around 2.15GB for the high-paying tenant, Tenant1, and 0.65GB for Tenant2. This leads to an HRD of less than 0.1 for Tenant1, which results in a relatively low penalty of just 10%. The HRD for Tenant2 reaches 0.3, which result in a maximal penalty of 80%. Overall, by aggressively reducing Tenant1's penalty, MT-LRU is able to achieve 84% of the maximal revenue. SQL on the other hand treats both tenants the same, resulting in almost equal memory shares and HRD. Because both HRD are above 10%, the provider pays 50% penalty for each tenant, and obtains overall revenue of just 50% of the maximum.

One point to notice in Figures 6(a) is the spike in memory allocation which occurs at early stages of the experiment (around 5th minute). Such fluctuation is expected due to initialization effects when HRDs become non-zero. The algorithm starts with an empty buffer-pool, hence the HRDs are zero for both tenants until the first time there is memory pressure. Subsequently, when the HRDs become non-zero and penalties are manifested, this leads to an aggressive attempt to recover from high penalties, which results in transient lumpy behavior. In practice, one could mitigate this initialization effect by adjusting the derivative of the penalty functions for this brief transient period.

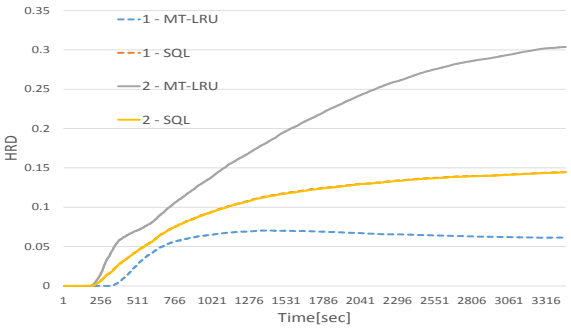
5.4.4 *Varying number of tenants and workloads*

We now consider an experiment in which we vary *both* the payment and workload characteristics across tenants. We have eight active tenants running workload RAND, each requiring a memory of 2GB; the total buffer-pool size is 6GB. In particular, we assign two tenants to each of the four combinations: ($100, $Z = 1.2$), ($100, $Z = 1.1$), ($10, $Z = 1.2$), ($10, $Z = 1.2$). We refer to these four groups of tenants as (high/low paying,higher/lower skew) tenants.

For this setting, MT-LRU is able to achieve 84% of the maximal revenue, by keeping the HRDs of high paying tenants between 6 and 9 percent; the steady state memory allocation for these tenants are around 1GB for the higher skew tenants, and 1.2GB for the lower skew tenants. Low paying tenants get memory allocations of around 0.2GB (higher skew) and 0.4GB (lower skew), and their resulting HRD is around 30%. SQL allocates around 0.6GB to higher skew tenants and 0.8GB to lower skew tenants, regardless of their payment. This leads to an almost identical HRD of around 14.5% across tenants, and an overall revenue of 50% of the maximum revenue. CURRENT actually does worse than SQL in this particular example. The high-paying tenants get around 0.8GB (higher skew) and 1.0GB (lower skew) of memory, but their HRD remains above 10% (around 11%), hence the provider still pays a penalty of 50%. Because the low-paying tenants now get less memory compared to SQL (around 0.4GB and 0.65GB for higher and lower skew tenants, respectively), their HRD becomes higher than 15% (almost 20%), and their penalties are at maximal level, higher than the SQL policy penalties. This example demonstrates once more the advantage of MT-LRU in being able to suitably weight the time and penalty dimensions.



(a) Buffer pool memory given to tenants vs. Time



(b) HRD of tenants vs. Time

Figure 6: Temporal behavior of MT-LRU and SQL. The buffer-pool size is 3GB, and each tenant is promised 4GB. Both tenants have RAND workflow ($Z = 1.1$). Both tenants use PF1, but Tenant1 (dotted lines) pays $100 for the SLA, while Tenant2 pays $10.

5.4.5 *Comparing different algorithms*

We next provide a more comprehensive comparison of average performance for the different algorithms. To that end, we run MT-LRU, SQL and CURRENT on 320 different settings. The settings correspond to different configurations within the active-range of the algorithms. To obtain different settings, we vary the number of tenants (either 2 or 4), the penalty functions (in each experiment we choose either step-based or piecewise linear penalties from a pool of several penalty functions), the promised memory, the type of workload and the skew (if relevant). The performance results are summarized in Figure 7. MT-LRU stands out as the best algorithm, retrieving almost 90% of the maximum revenue on average. CURRENT, which is also an SLA-aware policy, does better than SQL, but still lags behind MT-LRU. We wish to emphasize that the results are *consistent*, in the sense that in each individual run, MT-LRU would do better than the other two algorithms. Finally, we point out that additional heuristic policies that we tried performed worse than CURRENT, and hence are not included in the figure.

5.5 Experiments with Real Workloads

We conclude this section by reporting the results we obtain when running real customer workloads. We note that some of the real workloads that we experimented with turned out to be CPU-bound with very small buffer pool memory requirements ($100-300$ MB). For such workloads there are no benefits in using penalty-aware memory replacement algorithms because there is no substantial memory pressure even with tens of tenants. However, in a significant fraction of the real workloads there is substantial pressure on buffer pool memory. We report below two such workloads, which we term REAL1 and REAL2. REAL1 is a reporting and analysis

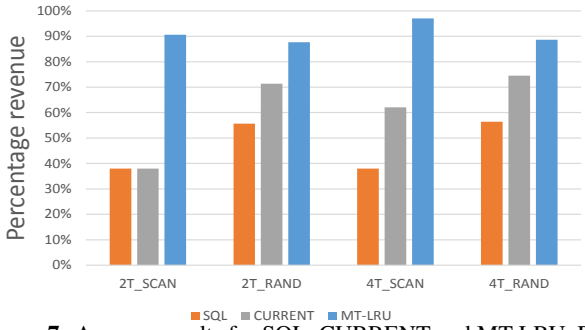


Figure 7: Average results for SQL, CURRENT and MT-LRU. Results are averaged over 320 different settings. The different settings vary by the number of tenants (either 2 or 4), the penalty functions, the amount of memory promised to tenants and workload.

	Memory promised (Gb)											
Revenues	2	3	4	5	6	7	8	9	10	11	12	13
T1 - SQL	100	90	50	20	20	20	20	20	20	20	20	20
T1 - MT-LRU	100	100	90	90	90	90	50	50	50	50	50	20
T2 - SQL	10	9	5	2	2	2	2	2	2	2	2	2
T2 - MT-LRU	10	9	5	2	2	2	2	2	2	2	2	2
Total - SQL	110	99	55	22	22	22	22	22	22	22	22	22
Total - MT-LRU	110	109	95	92	92	92	52	52	52	52	52	22

Table 2: REAL1 workload. Effect of promised memory per tenant - Revenue breakdown

database for a large cosmetics retailer and REAL2 is a data warehouse that tracks sales of different products sold by the company. Both workloads consist of joins over 6 to 12 tables, typically with grouping and aggregation.

We use the same experimental setting as in Section 5.4.1. Namely, we have an even number of tenants. All tenants are promised the same memory, however half of them pay $100 (per tenant) for the SLA, while the other half pays only $10. The total size of the buffer-pool memory is set to 6GB. In each run, we vary the size of the memory promised per tenant, and examine the effect on revenues. We first run the above experiment for REAL1. Table 2 provides the revenue breakdown by tenant. As before, MT-LRU consistently achieves higher revenues than SQL. As observed from the revenue breakdown, this is a consequence of giving preference to the high paying tenant T1.

We repeat this experiment for REAL2, this time with six tenants. The aggregate revenues of both SQL and MT-LRU are shown in Figure 8. Besides the superior performance of MT-LRU, we can draw additional interesting observations. First, comparing to the experiment done with synthetic data (cf. Figure 4), the active range here is much larger. Since the active range is directly related to the overbooking ratio, this shows that the range of overbooking ratios for which the provider can get adequate gain is *workload dependent*, and needs to be set on a trial and error basis. Second, this graph demonstrates that even a considerably high overbooking ratio can be sustained with minimal penalties. For example, notice that for an overbooking ratio of 10 (corresponding to 10 GB of promised memory for each of the 6 tenants), the provider extracts 85% of the maximal revenue.

6. RELATED WORK

Multi-tenant database-as-a-service commercial offerings such as Database.com [8], Google Cloud SQL [11], Windows Azure SQL Database [16], Oracle Database Cloud Service [21] are available today. Our work is a first step in enabling such systems to go beyond static reservation of buffer pool memory on behalf of a tenant, and enable sharing of buffer pool memory across tenants in an accountable and principled manner. All commercial database engines

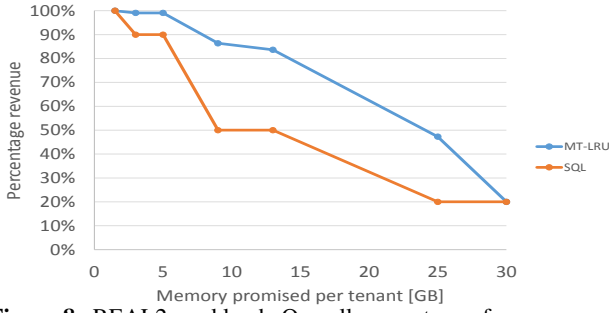


Figure 8: REAL2 workload. Overall percentage of revenues as a function of promised memory.

today have resource governance capabilities. For example, IBM DB2 Workload Manager [9] provides the ability to provide limits on resources such memory assigned to a particular application. However, our techniques goes well beyond this by clearly defining an SLA for minimum resources that a tenant is promised and can be objectively metered. Further, we provide new resource allocation policies (via our page replacement algorithm) that allows the service provider to trade-off performance with cost.

Our technique for SLA metering for the buffer pool has some similarity to the technique of simulated buffer pool in [26], which simulates the impact of making the buffer pool larger (say by 10%) on the number of hits for the workload. In the setting of [26], such simulation is used for determining if it would be beneficial to give the buffer pool more memory or not. In contrast, in our case, buffer pool simulation is required to meter the performance SLA promised to the tenant. Unlike simulated buffer pool which only extends the current pool by a certain percentage, we are required to simulate the buffer pool for the entire memory promised to a tenant.

There is extensive work on buffer pool page replacement policy dating back to the early '80s [25] where it was shown why LRU is not ideal for relational database environments. Several improvements have been made over simpler schemes such as LRU, e.g. DBMIN [4], LRU-K [20], 2Q [12], ARC [15], CAR [3]). All these techniques focus on the goal of maximizing performance (i.e. hit ratio) for a given workload. Our work can be viewed as extending state-of-the-art policies (e.g. LRU-K) to the multi-tenant case to make the page replacement algorithm SLA-aware, thereby enabling the system to make judicious cost-performance trade-offs.

This work was done in the context of the SQLVM project [19] at Microsoft Research. We have previously studied how to define and implement resource SLAs for I/O [19] and CPU [7] resources. Curino et al. [6] and Lang et al. [13] approach consolidation of multiple databases in a single server by analyzing the workloads, identifying how these workloads interact with one another, and recommending which databases should be co-located in order to meet performance goals (or SLO – Service-Level-Objectives). Xiong et al. [29], [30] constructs machine learning models to predict query performance as a function of resources allocated to it, and then use such models to perform admission control and allocate resources so that query latency SLO can be met. Similarly, Duggan et al. [10] builds a machine learning model to predict query performance in a DBMS; [18] develops statistical models for helping predict resource usage if a workload were to change (e.g. queries/sec were to double). SQLVM is complementary to these approaches since it provides resource-level isolation from other tenants, and makes no assumptions about the specific workloads of tenants. SQLVM can potentially be used as a building block to build such recommenders, since it can ensure that the tenants are actually allocated

the resources that the models assume. In [14] and [17] methods for tenant placement and load balancing across nodes in a cluster are presented so as to maximize profit (or reduce SLA violations). Such mechanisms are orthogonal and more coarse-grained compared to our techniques which provides isolation across tenants *within* a server by controlling resource allocation at a fine granularity. Both techniques can be used to together in principle – e.g., the SLA metering mechanisms we provide can assist in detection of when such load balancing in necessary.

Sharing memory of a machine across tenants also occurs in IaaS offerings. For example, Amazon RDS [1] provides a MySQL or Oracle database server running inside a virtual machine (VM) to a tenant. Service providers have the ability to overbook memory to increase consolidation and reduce cost [27]. The hypervisor can automatically change the physical memory available to a VM using techniques such as ballooning [28]. Further, in [22], techniques for application level ballooning are developed for applications which need to manage their own memory (e.g. databases). SLAs on memory become important in this setting since tenants can no longer be assured of statically reserved resources for their applications.

7. CONCLUSION

Our model of SLAs for buffer pool memory is relative to a baseline of statically reserved memory, and it retains meaning regardless of the tenant's data size, data distribution, workload characteristics, and the extent to which the service provider overbooks resources. We design and prototype an SLA-aware page replacement algorithm, which enables dynamically sharing memory in overbooked DaaS.

Acknowledgements

We thank Arnd Christian König, Sudipto Das, and the anonymous reviewers for their useful feedback.

8. REFERENCES

[1] Amazon Relational Database Service (RDS). http://aws.amazon.com/rds.

[2] N. Bansal, N. Buchbinder, and J. S. Naor. A primal-dual randomized algorithm for weighted paging. *Journal of the ACM (JACM)*, 59(4):19, 2012.

[3] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *FAST*, volume 4, pages 187–200, 2004.

[4] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica*, 1(1-4):311–336, 1986.

[5] F. J. Corbato. A paging experiment with the Multics system. Technical report, DTIC Document, 1968.

[6] C. Curino, E. P. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database service for the cloud. In *CIDR*.

[7] S. Das, V. R. Narasayya, F. Li, and M. Syamala. CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-Service. *Proceedings of the VLDB Endowment*, 7(1), 2013.

[8] Database.com. http://www.database.com.

[9] DB2 Workload Manager for Linux, Unix and Windows. http://www.redbooks.ibm.com/redpieces/abstracts/sg247524.html.

[10] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[11] Google Cloud SQL. http://code.google.com/apis/sql.

[12] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[13] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance SLOs. In *ICDE*, pages 702–713.

[14] Z. Liu, H. Hacıgümüş, H. J. Moon, Y. Chi, and W.-P. Hsiung. Pmax: tenant placement in multitenant databases for profit maximization. In *EDBT*, pages 442–453. ACM, 2013.

[15] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[16] Microsoft Azure SQL Database (formerly SQL Azure). http://www.windowsazure.com/en-us/services/sql-database/.

[17] H. J. Moon, H. Hacıgümüş, Y. Chi, and W.-P. Hsiung. SWAT: A Lightweight Load Balancing Method for Multitenant Databases. In *EDBT*, pages 65–76. ACM, 2013.

[18] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. In *SIGMOD*, pages 301–312, 2013.

[19] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR*, 2013.

[20] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Record*, volume 22, pages 297–306. ACM, 1993.

[21] Oracle database cloud service. http://cloud.oracle.com.

[22] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application Level Ballooning for Efficient Server Consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 337–350, 2013.

[23] Azure SQL Database Service Tiers and Performance Levels. http://msdn.microsoft.com/en-us/library/azure/dn741336.aspx.

[24] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[25] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7), 1981.

[26] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB*, pages 1081–1092, 2006.

[27] The role of memory in VMWare ESX server 3. http://www.vmware.com/pdf/esx3_memory.pdf.

[28] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[29] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. HacigümüŞ. Intelligent management of virtualized resources for database systems in cloud environment. In *IEEE ICDE*, pages 87–98, 2011.

[30] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüŞ. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *SOCC*, page 15, 2011.

[31] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.