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

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 7
Copyright 2015 VLDB Endowment 2150-8097/15/03.

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

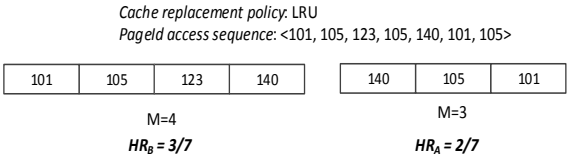

Figure 1: Example of hit ratio degradation.

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

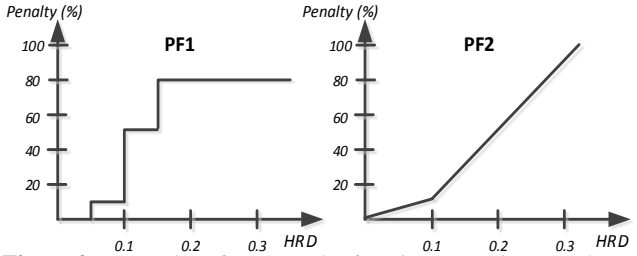

Figure 2: Examples of two penalty functions. PF1 is a step-based function, namely consists of multiple steps. PF2 is a piecewise-linear function.

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

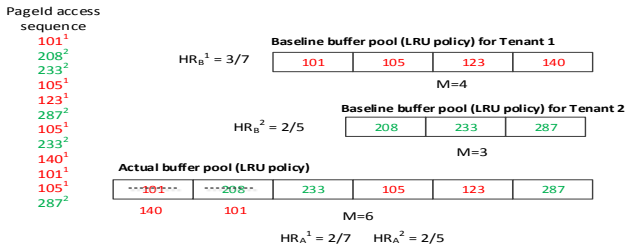

Figure 3: . Example of hit ratios in Baseline and Actual Buffer Pools for two tenants.

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