Towards Multitenancy for IO-bound OLAP Workloads

Hatem A. Mahmoud*
UC Santa Barbara
hatem@cs.ucsb.edu

Hyun Jin Moon†
Google
hyunm@google.com

Yun Chi
NEC Laboratories
ychi@sv.nec-labs.com

Hakan Hacıgümüş
NEC Laboratories
hakan@sv.nec-labs.com

Divyakant Agrawal
UC Santa Barbara
agrawal@cs.ucsb.edu

Amr El-Abbadi
UC Santa Barbara
amr@cs.ucsb.edu

ABSTRACT

Consolidation of multiple databases on the same server allows service providers to save significant resources because many production database servers are often under-utilized. We consider the problem of minimizing the number of servers needed to host a set of tenants, while satisfying the service level agreement (SLA) on the throughput of each tenant. Recent research investigates this problem under the assumption that the working sets of tenants are kept in main memory (e.g., OLTP workloads, or in-memory OLAP workloads), thus the buffer size of each tenant is dictated by the working set size of that tenant. In this paper we instead investigate the problem when the throughput SLAs of tenants are low enough for queries to be answered from disk. We study the trade-off between buffer size and query execution time for IO-bound workloads and propose an algorithm, called Greedy Memory Reduction (GMR). GMR approximates a globally-optimum memory allocation function (i.e., a function that assigns a buffer size to each tenant) for tenants running in private DBMS instances, such that the number of servers needed to host all tenants is minimized. GMR further inspires the design of another online heuristic that, though lacks an approximation guarantee, performs very well in practice. We then present a heuristic algorithm, called Greedy Tenant Consolidation (GTC), for consolidating tenants into shared DBMS instances whenever their throughput SLAs allow. Finally, we conduct extensive experimental evaluations of our algorithms to demonstrate their effectiveness, scalability, and correctness.

1. INTRODUCTION

Workload consolidation is an effective method to achieve cost efficiency in cloud computing. The consolidation of multiple small workloads – also known as multitenancy – can avoid significant resource waste given that many production servers are often over-provisioned for the peak workload [3]. Database services in the cloud can benefit from the same principle through database multitenancy by co-locating multiple databases on the same server. In this paper, we consider the problem of mutlitenant database placement given per-tenant service-level agreements (SLAs) on query throughput. By *throughput SLA* we mean that the service provider agrees to handle queries from a particular tenant up to a given arrival rate, whereas query latency is not the main concern. We aim to minimize the number of servers required to host a given set of tenants, while meeting the throughput SLA of each tenant. Previous work has considered consolidation of OLTP workloads whose working sets can fit in main memory [6], and consolidation of OLAP workloads that are small enough to entirely fit in main memory [16]. While these are important steps towards database multitenancy, previous studies build their solutions based on a key assumption that the total working set size of all tenants hosted by a server is no more than the available buffer memory on that server. This is typical for low-latency workloads, where the required low latency makes it infeasible to answer queries from disk. In this paper, in comparison, we consider tenants whose queries are allowed to be answered from disk. An example of such workloads are OLAP queries over large data sets for periodic report generation. We believe that tenants with such workloads may very well benefit from multitenancy if their throughput SLAs are low enough. To the best of our knowledge, previous research has not investigated such cases.

The problem of multitenancy for IO-bound OLAP workloads turns out to be closely related to the well-known "5 minute rule" that was proposed by Gray et al. [13, 12] and recently brought up-to-date by Graefe [11]. According to the 5-minutes rule, under the given costs for memory vs. IO bandwidth, a data item can be served either in-memory (memory-resident) or through IO access (disk-resident), resulting in different costs; furthermore, there exists a *break-even* frequency of access that separates the regions where one choice is better than the other. In our work, we extend this simple but powerful principle in the following two directions. First, we demonstrate that in handling an OLAP workload (instead of a single data item [13] or certain sequential SQL operations [12, 11]), there exists a continuous spectrum of configurations which we can exploit, in addition to the choices of 100% memory-resident and 100% disk-resident. For example, by increasing the size of the buffer pool dedicated to a workload, we can trade off a portion of the IO bandwidth required by the workload. Second, instead of optimizing for a single workload, we study how to place and configure a set of tenants in a manner that minimizes the number of servers required to host tenants. Here, the challenge for service providers is that all tenants have to be considered together in order to achieve globally-optimum solutions.

In this paper, we study the trade-off between memory and disk

*Work partially done while the author was at NEC.

†Work done while the author was at NEC.

.

bandwidth when co-locating IO-bound OLAP tenants. We design and implement a set of algorithms and heuristics that balance these two critical resources, namely memory and IO bandwidth. The goal is to minimize the total number of servers required to host a set of tenants, while meeting the throughput SLA requirements of each tenant. Toward this goal, we make the following contributions.

- We propose an approximation algorithm, called Greedy Memory Reduction (GMR), that approximates globally optimum buffer sizes of tenants running on private DBMS instances with a worst case approximation ratio of 3.
- We present an online heuristic, called Balanced Memory Reduction (BMR), that performs well in practice although it does not have a theoretical approximation guarantee.
- We present a heuristic, called Greedy Tenant Consolidation (GTC), that consolidates tenants into shared DMBS instances whenever throughput SLAs allow, based on a profiling approach that takes into account *cache warmness* (where cache warmness indicates what percentage of the pages cached in the DBMS buffer pool are relevant to a given tenant that runs on that buffer pool).
- We conduct an experimental evaluation to demonstrate that our algorithms and heuristics effectively reduce the number of servers needed to host a given set of tenants compared to other baseline algorithms and heuristics, and we also demonstrate that the tenant placement plans generated by our algorithms meet the throughput SLAs of tenants.

The rest of the paper is organized as follows. In Section2 we present related work. In Section 3 we formulate our optimization problem, state our assumptions and restrictions, and show the impracticality of searching for an exact solution to the optimization solution. In Sections 4 and 5 we present our approximate solution to the problem on two phases; first we solve the problem using private DBMS instances in Section 4, then we present a heuristic for consolidating tenants into shared DBMS instances in Section 5. In Section 6 we evaluate the effectiveness, scalability, and correctness of our solution. Finally, we conclude in Section 7.

2. RELATED WORK

The emergence of cloud computing has induced many new research challenges on database multitenancy [8]. Most recent research investigates the problem of database multitenancy when the workloads of tenants are kept in main memory. For example, OLTP workloads are studied in [6] and in-memory OLAP workloads are studied in [16]. The assumption that all tenants on a given server have to answer their queries from main memory leaves little room for optimization since the buffer size of each tenant is dictated by its working set size, and the CPU time is dictated by the high throughput requirements of the tenant. Furthermore, none of these solutions is optimized for OLAP workloads whose throughput SLAs allow for queries to be answered from disk. Other related research [19, 21, 18] investigates database multitenancy at a virtual machine level, where the allocation of CPU and memory is the main focus. However, these VM-based methods are not directly applicable to IO-bound multitenancy, at least before IO virtualization technologies become mature enough to achieve satisfactory IO isolation with negligible overhead [20].

Another direction of research on database multitenancy focuses on consolidation of large numbers of almost-inactive tenants by sharing the same schema among tenants [4, 14]. The main challenge in this type of systems is scalability, due to the limit on the number of tables a DBMS can handle for a given schema. Our work, in comparison, targets workloads with given throughput requirements. Other previous research, for example [15], attempts to model the performance of disk IO for different types of workloads analytically. We take a more empirical approach by profiling tenants for different buffer sizes and different cache warmness levels. However, it is still possible to make use of the model in [15] in our solution to avoid the profiling of workloads, if profiling is overly expensive.

3. PROBLEM FORMULATION

We consider the problem of tenant placement for IO-bound OLAP workloads. The objective is to configure and place OLAP tenants in a manner that minimizes the total number of servers needed to host tenants, while satisfying the throughput SLA requirements of each tenant. Figure 1 shows a block diagram of our multitenancy architecture, where each server hosts one or more DBMS instances, and each DBMS instance hosts one or more tenants. Note that the database server allocates and controls one buffer pool per database instance, which is typical for most widely used database products in large cloud data centers that are created with cost efficiency. The main questions that we investigate are: (1) which DBMS instances should be hosted together on the same server?, (2) how much memory should we assign to each database instance?, and (3) which tenants should be placed together within the same DBMS instance?

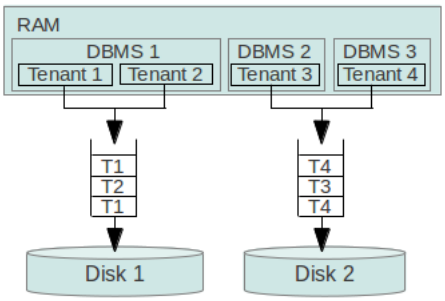

Figure 1: Block diagram of 3 DBMS instances hosting 4 tenants on a single server with 2 disks. All three DBMS instances share the same server. Tenants 1 and 2 share the same DBMS instance.

In our framework we allow two levels of multitenancy:

Multiple DBMS instances share the same DB server: Although a single DBMS instance per sever allows a larger buffer pool shared among hosted tenants, one tenant may greedily evict the cached pages of other tenants that share the same buffer pool, which may result in uncontrollable and inferior performance. Thus, we use multiple DBMS instances on the same server in order to impose controlled boundaries between the buffer pools of different (groups of) tenants.

Multiple databases (tenants) share the same DBMS instance: We do however consolidate some tenants into the same DBMS instances whenever their throughput SLAs allow. Note that some tenants need non-trivial buffer sizes to satisfy their throughput SLAs but do not need their buffers to remain warm from one query to another; that is, the buffer pool is merely used as a working space (e.g., in a nested-loop join, a table is scanned multiple times during the running time of a single query).

In a multitenancy environment, each tenant introduces a certain *load* to its hosting server. We use the term *load* to indicate the fraction of time that the hosting server spends on serving queries from the given tenant on average. The load of a tenant depends on two factors: (1) the average query arrival rate of the tenant, which is bounded by the throughput SLA of the tenant, and (2) the average query execution time of the tenant. The average query execution time of a tenant is affected by three main factors: (1) the workloads of other tenants running concurrently on the same DBMS instance, (2) the workloads of other tenants running on other DBMS instances but utilizing the same disk, and (3) the buffer pool size assigned to the tenant. Note that we focus on IO-bound workloads where the CPU is not highly utilized and therefore is not a critical resource (we leave the problem of co-locating IO-bound and CPU-bound workloads on the same servers as future work). Also, since we deal with tenants that have non-trivial throughputs, each server typically hosts a few tenants, thus disk space is not a bottleneck resource. The two main critical resources we focus on are IO bandwidth and memory space.

In the following subsection we investigate how IO bandwidth and memory affect the loads of tenants. Then we formally define our problem of placing and configuring tenants such that the total number of servers used to host tenants is minimized while satisfying per-tenant throughput SLAs.

3.1 Models for Two Critical Resources

We start by modeling the two critical resources, i.e., IO bandwidth and memory, and their effect on the loads of tenants.

3.1.1 IO bandwidth: concurrency

Previous studies (e.g., [17]) suggest that disk-bound OLTP workloads achieve maximum throughput when the ratio between the multi-programming level (MPL) (i.e., the number of transactions running concurrently) and the number of disks is small (e.g., around 2). We conduct our own experiments on disk-bound OLAP workloads to study the effect of concurrency on throughput. We begin by generating a set W that contains 7 random workloads. Each workload is comprised of 10 queries, randomly chosen from the query templates of the TPC-H benchmark, with randomly generated parameter values. Next, we construct a set P of 14 pairs of workloads randomly chosen from W. For each pair of workloads (w_i, w_j) in P, we run the two workloads w_i and w_j in parallel and serially, and compare the time required to execute the workloads in both cases. We consider two types of experiments: (1) the two workloads w_i and w_j run on the same DBMS instance, and (2) the two workloads run on two different DBMS instances. Each workload operates on a database whose size is 1 gigabyte. We run these experiments under two buffer configurations: (1) the total buffer pool size assigned to both workloads equals the total database sizes of both workloads (i.e., 2 gigabytes), and (2) the total buffer pool size assigned to both workloads equals half the total database sizes of both workloads (i.e., 1 gigabyte). We run our experiments on MySql 5.5 with InnoDB as a storage engine. We use Linux 2.6 machines each with a single disk and 2 Intel Quad-Core Xeon processors with hyper-threading.

Figure 2 shows the execution times of workload pairs plotted as a set of points whose x-coordinates represent the execution times when the workload pairs are executed concurrently, and whose y-coordinates represent the execution times when the workload pairs are executed sequentially. Our experiments show that, for disk-bound OLAP workloads running on MySql, maximum throughput is achieved in most cases by running 1 workload per disk at a time; that is, no two tenants run concurrently on the same disk. This observation confirms the claim in [17]. Since the SLA requirements of tenants are expressed in terms of maximum throughput, we make the design decision of restricting the number of tenants running on each disk to 1 at a time. We store the data of tenants that are hosted by the same DBMS instance on the same disk, thus tenants hosted by the same DBMS instance do not run their queries concurrently. Eliminating concurrency between workloads of tenants hosted by the same DBMS instance serves to avoid unpredictable interactions between workloads. Previous studies [2, 1] already show that such interactions result in unpredictable and counter-intuitive performance that is too complex to model. Therefore, in Figure 1, queries that are directed to the same DBMS instance are executed sequentially. We note that there are more sophisticated scan mechanisms, such as shared scans, implemented by some high-end database products to save IO when multiple queries scan the same set of records on the disk. However, as we mention before, in this work we mainly focus on cloud data center settings, where hardware and software components are chosen from commodity and lower-cost resources to increase cost efficiency, which is a common practice in the industry.

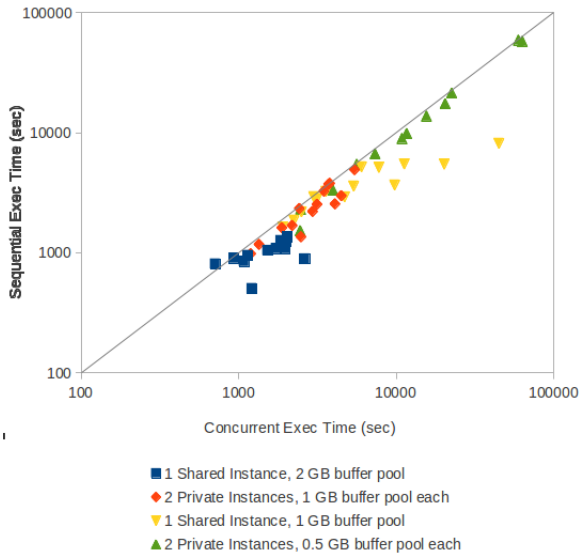

Figure 2: The execution time of random pairs of TPC-H workloads, concurrently versus sequentially.

3.1.2 Memory: buffer size and warmness

Although we restrict interaction between tenants by avoiding query contention on the same disks and the same DBMS instances, tenants still affect the loads of each others. For example, assigning less buffer size to a low-throughput tenant t_i slows down t_i, thus takes advantage of t_i's low throughput and leaves more memory for other higher-throughput tenants on the same server. But as t_i takes longer time to finish, it introduces more load on the server, and therefore leaves less time for other tenants that use the same disk and/or run on the same DBMS instance to finish their queries. Thus there is a trade-off between memory and time resources. Another way tenants affect each others is when tenants hosted by the same DBMS instance evict the buffered pages of each others. For example, if two tenants t_i and t_j share the same DBMS instance and the working set size of each of them is greater than or equal to the buffer size of the database instance, then each tenant of them will have to retrieve all data from disk each time they answer a query. In other words, the cache will be always *cold* for both tenants. Note that even in the case when the cache is always cold at the beginning

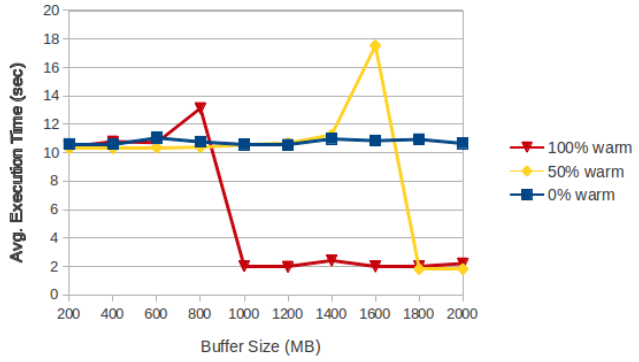

Figure 3: The average execution time when scanning the lineitem table in the TPC-H benchmark, for different buffer sizes, and different warmness levels.

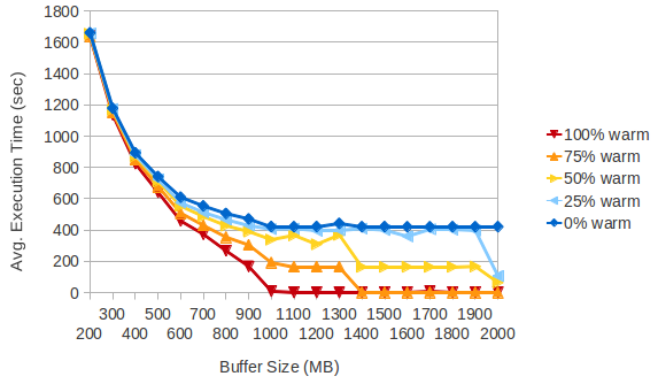

Figure 4: The average query execution time of Q8 in TPC-H, for different buffer sizes, and different warmness levels.

of each query, zero buffer may be not a good idea because a single query may re-use the data it retrieves multiple times (e.g., when performing a nested-loop join). Also, if the working set size of one of the two tenants, say t_i, is less than the buffer size, then only a fraction of t_j's cache will be evicted.

Figures 3 and 4 show the average query execution times of two types of TPC-H queries measured on MySql 5.5 for different buffer sizes and different *cache warmness levels*. For a given query (resp. tenant), cache warmness indicates what percentage of the buffer contains cached pages used by this query (resp. tenant). A tenant running in a private database instance always finds its cache 100% warm. Meanwhile, a tenant finds its cache 0% warm if it shares buffer with another tenant that has at least the same query arrival rate and has a working set size at least as large as the buffer size. If the working set size of one tenant is, say, 25% of the buffer size then the other tenant gets a cache that is 75% warm, and so on. To compute these profiles, we develop a profiler that measures the average query execution time of a given tenant under different buffer sizes and different warmness levels. The input of the profiler is a set of sample queries, say $Q(t)$, extracted from the workload of the tenant t, and the range of buffer sizes and warmness levels under which the tenant t needs to be profiled. The output of the profiler is a profile of the tenant t similar to Figures 3 and 4. To measure the average query execution time under different buffer sizes, the profiler restarts the database daemon multiple times, each time the database is assigned a different buffer size. To measure the average execution time under a certain warmness level, say $w\%$ warmness, the profiler scans a dummy table whose size equals $(100-w)\%$ of the buffer size before executing each query in $Q(t)$. This ensures that whenever a query $q \in Q(t)$ is executed, no more than $w\%$ of the pages cached in the buffer are relevant to q, but at the same time the entire buffer is available as a workspace for q.

Figures 3 and 4 show that the average execution time at any level of cache warmness drops almost monotonically as the buffer size increases, and higher levels of cache warmness drop faster. We observe this behavior in various TPC-H queries. However, some TPC-H queries perform differently due to the way InnoDB handles table scans. Figure 3 shows the average execution time when scanning a 1 gigabyte table on MySql 5.5. When most of the table is cached except for a few pages, MySql reads the remaining pages one by one using individual random reads, and in our example this results in an execution time greater than that when scanning the entire table from disk using a single sequential read. Some of the algorithms that we present later in this paper assume that the profiles of tenants are monotonically non-increasing. Because the profiles of tenants occasionally deviate from this monotonicity, we conservatively fix the profile of each tenant in the following way. We set the average query execution time at any given buffer size to be no less than the average query execution times at larger buffer sizes for the same tenant. Thus we force all profiles to be monotonically non-increasing at the cost of *conservatively overestimating* the average query execution time for some buffer sizes.

3.2 MINLP Formulation

Given a set of tenants that we need to host, we use our profiler to profile each tenant, then use the profiles of tenants as input to our algorithms to come up with a *placement plan* that minimizes the total number of servers needed to host all tenants while satisfying per-tenant throughput SLAs. The placement plan assigns each tenant to a DBMS instance (either a private DBMS instance, or a DBMS instance shared with other tenants), assigns each DBMS instance to a server, and determines the buffer size of each DBMS instance. Before presenting our algorithms in the following sections, here we present a mathematical formulation of the problem whose solution, if obtained exactly, is the optimum one (i.e., minimizes the number of servers required to host tenants). We assume that all servers have the same resources (e.g., the same disk speed, main memory size, and processing power), which is reasonable given the current trend of using commodity servers in data centers. Under such assumptions, the optimum solution (i.e., minimum number of servers) can be found by solving the following mixed-integer non-linear program (MINLP), which we refer to as MINLP-1.

$$min \sum_k z_k$$

$$s.t. \quad \forall i \quad \sum_j x_{ij} = 1 \tag{1}$$

$$\forall j \quad \sum_k y_{jk} = 1 \tag{2}$$

$$\forall j,k \quad y_{jk} \le z_k \tag{3}$$

$$\forall k \quad \sum_j y_{jk} m_j \le z_k \tag{4}$$

$$\forall k \quad \sum_j y_{jk} \sum_i x_{ij} r_i e_i(m_j, \sum_l x_{lj}) \le z_k \tag{5}$$

$$\forall i,j,k \quad x_{ij}, y_{jk}, z_k \in \{0,1\} \tag{6}$$

$$\forall j \quad 0 \le m_j \le 1 \tag{7}$$

Decision variables are explained as follows. $x_{ij} = 1$ if and only if the tenant i is hosted by the DBMS instance j, $y_{jk} = 1$ if and only if the DBMS instance j is hosted by the server k, and $z_k = 1$ if and only if the server k is used (i.e., hosts at least one database

instance). m_j is the buffer size of the DBMS instance j, measured as a fraction of server memory. r_i is an input constant indicating the average query arrival rate (i.e., throughput) of the tenant i, and e_i is the average query execution time of the tenant i. The average execution time is a function in buffer size and warmness level; such function is determined by the profile of the tenant, as in Figures 3 and 4. For now let us assume that there are only two warmness levels, either 100% warm or 0% warm. The tenant i is 100% warm if and only if no other tenants run on the same DBMS instance (i.e., if and only if $\sum_l x_{lj} = 1$, where j is the DBMS instance of the tenant i), otherwise the tenant i is 0% warm. The objective function to be minimized the number of servers used. The first two constraints state that each tenant is assigned to one and only one database instance, and each database instance is assigned to one and only one server. The third constraint counts a server as used if it hosts at least one database instance. The fourth constraint states that the total memory assigned to all database instances on a server is no more than the server memory size. The fifth constraint states that the total load of a server is no more than 1. For now let us assume that each server has only one disk, thus the load of each tenant is computed as the average query arrival rate times the average query execution time of the tenant.

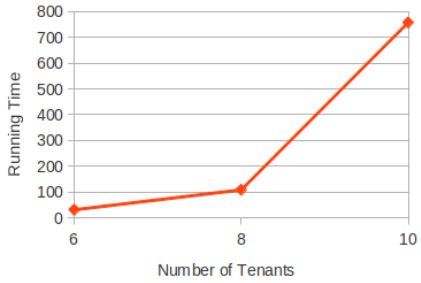

Figure 5: Running time of Couenne for different numbers of tenants.

We implement MINLP-1 in AMPL, and solve it using Couenne on NEOS[1]. We start by doing straightforward experiments where the number of tenants is very small, and the optimum solution consists of one tenant per database instance and one database instance per server. Figure 5 shows the time taken by NEOS to reach an optimum solution as the number of tenants increases. Our experiments show that, even for very simple input cases, the running time grows very rapidly as a function in input size. We believe that the complexity of the model stems from the non-convexity of the tenant profile function e_i. Thus it is impractical to use MINLP solvers to solve real-world cases that involve thousands of tenants. In the following sections, we develop algorithms and heuristics to approximate a solution to this optimization problem within a reasonable amount of time.

4. PRIVATE DATABASE INSTANCES

In this section, we study a special case where only the first level of multitenancy, namely multiple DBMS instances sharing the same DB server, is allowed. That is, we find a placement plan for a given set of tenants by assigning each tenant to a private DBMS instance and configuring the buffer size of each tenant. We want the total number of servers needed to host all tenants to be minimized while satisfying per-tenant throughput SLAs. The problem that we solve in this section is a sub-problem of that we describe in Section 3.2, since we do not allow for more than one tenant to be hosted by the same DBMS instance.

[1]http://www.neos-server.org/

4.1 Formulation as a 2-DVPP Problem

We begin by defining our sub-problem formally as follows. We are given a set of tenants $t_1, ..., t_n$. Each tenant t_i is associated with an SLA throughput r_i that has to be satisfied. Let m_i be the buffer size that we assign to the DBMS instance of tenant t_i, the average execution time e_i of tenant t_i is a function in m_i that is defined by the workload of t_i. We restrict concurrency on disks so that no two tenants run queries on the same disk at a time. Therefore, the average query execution time of each tenant depends only on the workload and the buffer size of that tenant. The load l_i imposed by the tenant t_i on its server equals $r_i \cdot e_i(m_i)$, where r_i is the throughput SLA of tenant t_i. We assume that e_i, and consequently l_i, are monotonically non-increasing functions in m_i. The total load of all tenants on a single server must not exceed the number of disks, and the total memory assigned to all tenants on a single server must not exceed the server memory size. Let $\mathcal{D}$ be the number of disks, and $\mathcal{M}$ be the memory size of each server. We assume that all servers have exactly the same resources. Therefore, without loss of generality, we set the unit of memory measurement to $\mathcal{M}$, so that the memory size of each server equals 1, and $m_i \leq 1$ for all t_i. We also scale down the average execution time function of each tenant by a factor of $1/\mathcal{D}$, so that the total load handled by each server equals 1, and $l_i \leq 1$ for all t_i.

The minimum number of servers needed to host tenants depends on how much memory we assign to each tenant (which in turn determines the load of each tenant), and how we pack tenants into servers. The problem of packing tenants into servers can be formulated as a 2-dimensional vector packing problem (2-DVPP) [9]. Each tenant t_i is represented by a 2-dimensional vector whose first dimension is the buffer size m_i, and whose second dimension is the load l_i. Each server has two capacities: a memory capacity of 1, and a load capacity of 1. As stated earlier, e_i and l_i depend on the workload and the throughput SLA of the tenant t_i, thus m_i is the only input of 2-DVPP that we can tune. In order to find the minimum number of servers needed, we need to find a memory assignment function $\mathfrak{m}^*$ that assigns a buffer size to each tenant such that the optimum output of the tenant placement 2-DVPP is minimized. That is, assume $Opt\_2DVP(\mathfrak{m})$ is an algorithm that computes the optimum packing of tenants into servers, where the buffer sizes of tenants are determined by the memory assignment function $\mathfrak{m}$, then

$$\mathfrak{m}^* = \arg\min_{\mathfrak{m}} \{Opt\_2DVP(\mathfrak{m})\} \quad (8)$$

4.2 Approximating Tenant Buffer Size

2-DVPP is an NP-hard problem in the strong sense (i.e., the worst case running time is exponential in the value of the input), therefore finding $\mathfrak{m}^*$ is NP-hard as well [should it be EXP??]. There are polynomial-time approximation algorithms for 2-DVPP with guarantees on the worst case ratio between the approximate solution value and the optimum solution value; such ratios are called approximation ratios. To the best of our knowledge, no approximation algorithm for 2-DVPP has a worst case approximation ratio less than 2. The approximation scheme in [7] approximates 2-DVPP with a worst case approximation ratio of 2+ϵ for any $\epsilon > 0$, but with a worst case running time that is exponential in $1/\epsilon^2$.

For approximation algorithms of NP-hard problems, worst case approximation ratios are typically guaranteed by proving a worst case ratio between the approximate solution value and a *lower bound* of the optimum solution value (rather than

the optimum solution value itself, which is hard to characterize). The lower bound that is used in this kind of guarantees is usually an infeasible solution value that can be computed efficiently for each instance of the hard problem, and is guaranteed to be less than the optimum solution value. Let $Aprx\_2DVP$ be an approximation algorithm for 2-DVPP with a worst case approximation ratio of ρ, and let Lwr_2DVP be the lower bound that is used to prove the approximation ratio of $Aprx\_2DVP$, thus $Aprx\_2DVP(\mathfrak{m})/Lwr\_2DVP(\mathfrak{m}) \leq \rho$, for any $\mathfrak{m}$. Since $Opt\_2DVP(\mathfrak{m}) \leq Aprx\_2DVP(\mathfrak{m})$, therefore $Opt\_2DVP(\mathfrak{m})/Lwr\_2DVP(\mathfrak{m}) \leq \rho$, for any $\mathfrak{m}$. Let $\hat{\mathfrak{m}}$ be a memory assignment that minimizes $Lwr\_2DVP()$; that is,

$$\hat{\mathfrak{m}} = \arg\min_{\mathfrak{m}}\{Lwr\_2DVP(\mathfrak{m})\} \tag{9}$$

Thus $Lwr\_2DVP(\hat{\mathfrak{m}}) \leq Lwr\_2DVP(\mathfrak{m}^*)$. Consequently,

$$\begin{aligned} Opt\_2DVP(\hat{\mathfrak{m}}) &\leq \rho \cdot Lwr\_2DVP(\hat{\mathfrak{m}}) \\ &\leq \rho \cdot Lwr\_2DVP(\mathfrak{m}^*) \\ &\leq \rho \cdot Opt\_2DVP(\mathfrak{m}^*) \end{aligned} \tag{10}$$

For our tenant placement problem, a possible lower bound can be computed by allowing tenants to span multiple servers. To demonstrate this, let us first formulate an integer linear program (ILP) for our tenant placement 2-DVPP as follows. We refer to the following ILP formulation as ILP-1.

$$min \textstyle\sum_k z_k \tag{11}$$

$$s.t. \quad \forall i \quad \sum_k x_{ik} = 1 \tag{12}$$

$$\forall i,k \quad x_{ik} \leq z_k \tag{13}$$

$$\forall k \quad \sum_i x_{ik} m_i \leq z_k \tag{14}$$

$$\forall k \quad \sum_i x_{ik} l_i \leq z_k \tag{15}$$

$$\forall i,k \quad x_{ik} \in \{0,1\} \tag{16}$$

$$\forall k \quad z_k \in \{0,1\} \tag{17}$$

$x_{ik} = 1$ if and only if tenant t_i is hosted by server s_k. $z_k = 1$ if and only if server s_k hosts at least one tenant. Since the number of servers needed to host all tenants is no more than the number of tenants, therefore $1 \leq i,k \leq n$, where n is the number of tenants. Solving ILP-1 gives an optimum solution to our tenant placement 2-DVPP, however it is NP-hard. Relaxing ILP-1 by replacing the integer constraints (i.e., the last two constraints) with non-negativity constraints (i.e., $x_{ik}, z_k \geq 0,\ \forall i,k$) turns ILP-1 into a linear program (LP) that can be solved in polynomial time. We refer to the linear program that results from this relaxation as LP-1. The solution to LP-1 allows a tenant to span multiple servers, thus it is infeasible for our tenant placement problem. However, the solution value of LP-1 is a lower bound of the solution value of ILP-1. From [5], the maximum ratio between the solution value of ILP-1 and the solution value of LP-1 is 3. We define $Opt\_2DVP(\mathfrak{m})$ as the solution value of ILP-1, and $Lwr\_2DVP(\mathfrak{m})$ as the solution value of LP-1, thus

$$Opt\_2DVP(\mathfrak{m}) \leq 3 \cdot Lwr\_2DVP(\mathfrak{m}) \tag{18}$$

By substituting in Equation (10) we get

$$Opt\_2DVP(\hat{\mathfrak{m}}) \leq 3 \cdot Opt\_2DVP(\mathfrak{m}^*) \tag{19}$$

A tighter bound can be obtained through the LP relaxation of another ILP formulation of 2-DVPP [5] that is based on the Gilmore-Gomory cutting stock model [10]. However, the number of decision variables in the Gilmore-Gomory ILP formulation is exponential in the number of tenants, therefore it is impractical when dealing with thousands of tenants.

We focus on finding a memory assignment $\hat{\mathfrak{m}}$ that minimizes $Lwr\_2DVP()$. From [5], the solution value of LP-1 equals $\max\{\sum_i l_i, \sum_i m_i\}$. Therefore

$$\hat{\mathfrak{m}} = \arg\min_{\mathfrak{m}}\{\max\{\sum_i l_i(\mathfrak{m}(t_i)), \sum_i \mathfrak{m}(t_i)\}\} \tag{20}$$

Finding $\hat{\mathfrak{m}}$ provides us with an input to 2-DVPP such that when 2-DVPP is solved optimally we get an output number of servers that is no more than three times the minimum number of servers required to host the tenants. The next section presents a greedy algorithm that approximates $\hat{\mathfrak{m}}$ with an absolute error of no more than 1 in pseudo-polynomial running time.

4.3 Greedy Memory Reduction (GMR)

In this section we present a greedy algorithm that approximates $\hat{\mathfrak{m}}$ with an absolute error of no more than 1; that is, our algorithm finds a memory assignment $\dot{\mathfrak{m}}$ such that $Lwr\_2DVP(\dot{\mathfrak{m}}) \leq Lwr\_2DVP(\hat{\mathfrak{m}}) + 1$. The high-level idea of our solution is as follows.

> Starting by assigning maximum buffer size to all tenants, we keep reducing the buffer sizes of some tenants (therefore trading memory for IO bandwidth) in a greedy fashion, until the number of servers needed to provide the total memory size of all tenants and that needed to handle the total load of all tenants reach a *balanced* state.

Our solution consists of two phases. First we profile each tenant under different buffer sizes, then we use these profiles to approximate $\hat{\mathfrak{m}}$. We begin by defining our notation. Let μ denote the minimum unit of memory allocation. Thus, server memory size $\mathcal{M}$ and all buffer sizes of all tenants are multiples of μ. A reasonable order of magnitude for μ is 100MB. Let m_i^{min} denote the smallest buffer size for which the load of the tenant t_i can be handled by a single server. That is, $m_i^{min} = \min\{m : l_i(m) \leq 1,\ m/\mu \in \mathbb{Z}\}$. Also, let m_i^{max} denote the data size of t_i if the data size of t_i fits in the memory of a single server, or $\mathcal{M}$ otherwise. That is, $m_i^{max} = \min\{\mathcal{M},\ \min\{m : m \geq m_i^{ds},\ m/\mu \in \mathbb{Z}\}\}$, where m_i^{ds} is the data size of t_i. In the first phase, we profile each tenant t_i by measuring its average query execution time $e_i(m)$, for every m ranging from m_i^{min} through m_i^{max} with μ step size. Since m_i^{min} is not known until $e_i(m_i^{min})$ is computed, we begin our profiling from m_i^{max} and decrement m until either $m = 0$, or $r_i \cdot e_i(m) > 1$. In the second phase, we use our Greedy Memory Reduction (GMR) algorithm to approximate $\hat{\mathfrak{m}}$ in pseudo-polynomial running time, with an absolute error of no more than 1, based on the profiles that we measure in the first phase.

Algorithm 1 lists our Greedy Memory Reduction algorithm. Initially, each tenant t_i is assigned a buffer size of m_i^{max}. Since the initial total memory is maximum, and since we assume that the load is monotonically non-increasing function in the buffer size, then the initial total load is minimum. After the initialization phase, the algorithm proceeds in iterations such that each iteration decreases the buffer size of a single tenant. More specifically, in each iteration k, we pick a tenant $t_{i(k)}$ whose current buffer size $m_{i(k)}(k)$ can be decreased by some amount of memory δ while incurring a minimum average load increase per unit memory decreased; that is, the cheapest available chunk of memory to remove from a single tenant. We terminate the loop either (1) when the total memory needed by all tenants becomes no more than the total load, or (2)

Algorithm 1 Greedy Memory Reduction (GMR)

input: $\{l_i(m): \quad \forall t_i,\ \forall m,\ m_i^{min} \leq m \leq m_i^{max},\ \frac{m}{\mu} \in \mathbb{Z}\}$

Set $k \leftarrow 0$
Set $m_i(k) \leftarrow m_i^{max},\ \forall t_i$
while $\sum_i m_i(k) > \sum_i l_i(m_i(k))$ and $\exists i : m_i(k) > m_i^{min}$ **do**
    Let $\bar{l}_i(k,\delta) = \dfrac{l_i(m_i(k)-\delta) - l_i(m_i(k))}{\delta}$
    Let $\delta_i(k) = \arg\min_{\delta}\{\bar{l}_i(k,\delta) : m_i(k) - \delta \geq m_i^{min},\ \dfrac{\delta}{\mu} \in \mathbb{Z}^+\}$, if tied pick the largest δ.
    Let $i(k) = \arg\min_{i}\{\bar{l}_i(k,\delta_i(k))\}$, break ties arbitrarily.
    Set $m_{i(k)}(k+1) \leftarrow m_{i(k)}(k) - \delta_{i(k)}(k)$
    Set $m_i(k+1) \leftarrow m_i(k),\ \forall i \neq i(k)$
    Set $k \leftarrow k+1$
end while
if $\max\{\sum_i m_i(k), \sum_i l_i(m_i(k))\} > \max\{\sum_i m_i(k-1), \sum_i l_i(m_i(k-1))\}$ **then**
    Set $m_{i(k)}(k) \leftarrow m_{i(k)}(k-1)$
end if
Set $\dot{\mathfrak{m}}(t_i) = m_i(k), \forall i$
return $\dot{\mathfrak{m}}$

when each tenant t_i is assigned its minimum feasible buffer size m_i^{min}. After the loop, we make one last check to see if it is better to rollback the last iteration of the loop. We denote the memory assignment returned by GMR as $\dot{\mathfrak{m}}$.

To analyze the running time of GMR, let us first assume that a priority queue is used to pick $i(k)$ at each iteration. The running time of GMR is affected by the value of $\frac{\mathcal{M}}{\mu}$; that is, the granularity at which memory is assigned to tenants. If $\frac{\mathcal{M}}{\mu}$ is a constant then the absolute approximation error equals 1, and the running time is $O(n\ \log(n))$, where n is the number of tenants. However, μ can be used as a parameter to control the running time as well as the approximation error of GMR. If μ is a parameter the running time of GMR is $O(n\frac{\mathcal{M}}{\mu}(lg(n)+\frac{\mathcal{M}}{\mu}))$, and the worst case absolute error equals $1 + n\ \frac{\mu}{\mathcal{M}}$. The extra error stems from the fact that each tenant may be assigned $\mu - \epsilon$ memory units more than its optimum memory assignment, where ϵ is a negligible amount of memory.

The following theorem states the approximation guarantee of GMR when $\frac{\mathcal{M}}{\mu}$ is constant.

THEOREM 4.1. $Lwr\_2DVP(\dot{\mathfrak{m}}) \leq Lwr\_2DVP(\hat{\mathfrak{m}}) + 1$

The proof of Theorem 4.1 is in Appendix A. From Equation (18) and Theorem 4.1, we get

$$\begin{aligned} Opt\_2DVP(\dot{\mathfrak{m}}) &\leq 3 \cdot Lwr\_2DVP(\dot{\mathfrak{m}}) \\ &\leq 3 \cdot Lwr\_2DVP(\hat{\mathfrak{m}}) + 3 \\ &\leq 3 \cdot Lwr\_2DVP(\mathfrak{m}^*) + 3 \\ &\leq 3 \cdot Opt\_2DVP(\mathfrak{m}^*) + 3 \end{aligned} \quad (21)$$

Let us define $V(\mathfrak{m}) = \{(\mathfrak{m}(t_i), l_i(\mathfrak{m}(t_i))) : \forall t_i\}$, for any $\mathfrak{m}$. We pass the set $V(\dot{\mathfrak{m}})$ as input to 2-DVPP to get a tenant placement plan that assigns tenants to servers.

4.4 Balanced Memory Reduction (BMR)

In this section we present another approach to obtaining a memory assignment function that minimizes the total number of servers. The high-level idea of this approach is as follows.

For each tenant, we keep reducing the memory allocated to it until, for this particular tenant, the buffer size and the load reach a *balanced* state.

For any memory assignment $\mathfrak{m}$ let

$$\mathfrak{D}(\mathfrak{m}) = \max_{t_i}\{\max\{\mathfrak{m}(t_i), l_i(\mathfrak{m}(t_i))\}\} \quad (22)$$

That is, $\mathfrak{D}(\mathfrak{m})$ is the maximum dimension of all vectors in $V(\mathfrak{m})$. From [5], if $V(\mathfrak{m})$ is used as input to ILP-1 and LP-1 (both defined in Section 4.2), the worst case ratio between the solution value of ILP-1 and the solution value of LP-1 is $1+2\mathfrak{D}(\mathfrak{m})$. Since the maximum value of $\mathfrak{D}(\mathfrak{m})$ is 1, the worst case ratio between the solution values of ILP-1 and LP-1 is no more than 3. Inequality (18) is based on the maximum of $\mathfrak{D}(\mathfrak{m})$, but a more general inequality that takes $\mathfrak{D}(\mathfrak{m})$ into account can be stated as follows.

$$Opt\_2DVP(\mathfrak{m}) \leq (1 + 2\mathfrak{D}(\mathfrak{m})) \cdot Lwr\_2DVP(\mathfrak{m}) \quad (23)$$

The GMR algorithm that we present in Section 4.3 focuses on minimizing $Lwr\_2DVP(\mathfrak{m})$. In this section we present another approach to obtain a tight bound on $Opt\_2DVP(\mathfrak{m})$; that is, minimizing $\mathfrak{D}(\mathfrak{m})$. We make use of the profiling phase explained in Section 4.3 to compute the function l_i for each tenant t_i. Then we assign to each tenant t_i, independently of other tenants, a buffer size m_i that minimizes $\max\{m_i, l_i(m_i)\}$. We refer to this memory assignment function as $\ddot{\mathfrak{m}}$. If load is a non-monotonic function in buffer size, we compute $\ddot{\mathfrak{m}}(t_i)$ by trying every buffer size for which t_i is profiled from m_i^{min} to m_i^{max}. However, assuming that load is a monotonically non-increasing function in buffer size, we compute $\ddot{\mathfrak{m}}$ as follows. For each tenant t_i we initialize $\ddot{\mathfrak{m}}(t_i)$ to m_i^{max}, then iteratively decrease $\ddot{\mathfrak{m}}(t_i)$ until $\ddot{\mathfrak{m}}(t_i) < l_i(\ddot{\mathfrak{m}}(t_i))$. Then we check whether rolling back the last iteration minimizes $\max\{m_i, l_i(m_i)\}$; if yes, we rollback the last iteration. We refer to this heuristic as Balanced Memory Reduction (BMR).

The main disadvantage of BMR is that it lacks an approximation guarantee. It is possible to come up with corner cases where BMR performs very badly. For example, consider the case when we have n tenants, all of them have the profile $\{l_i(1-\epsilon) = 0, l_i(0) = 1\}$. BMR sets the buffer size of each tenant to $1-\epsilon$ so as to minimize $\mathfrak{D}(\ddot{\mathfrak{m}})$. However, this memory configuration requires n servers to host the n tenants, which is the maximum number of servers any memory configuration algorithm would require. When the same input is given to GMR, GMR sets the buffer sizes of half the tenants to $1-\epsilon$, and the buffer sizes of the other half to 0, thus requiring $\lceil n/2 \rceil$ servers to host the given tenants. Nevertheless, BMR has several practical advantages over GMR. First, it is more efficient in terms of running time. If μ is not a parameter, the running time of BMR is linear. Otherwise, if μ is a parameter, the running time of BMR is $O(n\frac{\mathcal{M}}{\mu})$, where n is the number of tenants. Second, BMR assigns a buffer size to each tenant independently of other tenants, thus it can be used as an online algorithm. Third, as we show in our experiments (Section 6.3), BMR performs at least as good as GMR in practice, and in many cases BMR generates tenant placement plans that require less servers compared to GMR. Fourth, BMR is much simpler and easier to implement compared to GMR.

We also examine a hybrid approach that attempts to balance the buffer size and the load of each tenant, while taking advantage of the theoretical approximation guarantee of GMR. We refer to this approach as GMR+BMR. In GMR+BMR, we begin by running GMR till its end to obtain $\dot{\mathfrak{m}}$, then we iteratively reduce $\mathfrak{D}(\dot{\mathfrak{m}})$ as long as this reduction does not degrade the theoretical approximation guarantee of $\dot{\mathfrak{m}}$. In each iteration k we reduce $\mathfrak{D}(\dot{\mathfrak{m}})$ by μ by going through each tenant t_i and ensuring that $\max\{\dot{\mathfrak{m}}_k(t_i), l_i(\dot{\mathfrak{m}}_k(t_i))\} \leq \mathfrak{D}(\dot{\mathfrak{m}}_{k-1}) - \mu$; if this inequal-

ity does not hold, we either decrease $\dot{\mathfrak{m}}(t_i)$ by μ if $\dot{\mathfrak{m}}_{k-1}(t_i) > l_i(\dot{\mathfrak{m}}_{k-1}(t_i))$, otherwise if $\dot{\mathfrak{m}}_{k-1}(t_i) \leq l_i(\dot{\mathfrak{m}}_{k-1}(t_i))$ we increase $\dot{\mathfrak{m}}(t_i)$ until $l_i(\dot{\mathfrak{m}}(t_i))$ is decreased by μ. We terminate once it is no longer possible to reduce $\mathfrak{D}(\dot{\mathfrak{m}})$ by μ, or if decreasing $\mathfrak{D}(\dot{\mathfrak{m}})$ by μ violates the inequality $(1 + 2\,\mathfrak{D}(\dot{\mathfrak{m}}_k))\,Lwr\_2DVP(\dot{\mathfrak{m}}_k) \leq (1 + 2\,\mathfrak{D}(\dot{\mathfrak{m}}))\,Lwr\_2DVP(\dot{\mathfrak{m}})$. Our experiments show that, while GMR+BMR improves the approximation guarantee of GMR significantly, it rarely decreases the actual number of servers needed by GMR, since the actual number of servers needed by GMR is usually much less than the upper bound imposed by the approximation guarantee of GMR.

5. SHARED DATABASE INSTANCES

In this section, we extend our solution by additionally allowing the second level of multitenancy, namely multiple tenants sharing the same DB instance. The new solution to the shared instance case is based on the solution to the private instance case. The key idea of our solution to this more general multitenancy is as follows.

> First we find a memory assignment for tenants in private database instances as explained in Section 4. Then we consolidate database instances iteratively, one pair at a time, as long as the number of servers needed to host tenants after iterative consolidation is no more than the number of servers before the consolidation.

To guarantee that the number of servers never grows, when doing iterative consolidation, we maintain two invariants: (1) total memory never increases, and (2) total load never increases.

We compute the load of a tenant t in a shared database instance as follows. Let $\mathfrak{I}(t)$ be the database instance that hosts t. We make a conservative assumption that any other tenant t_l hosted by the same database instance $\mathfrak{I}(t)$ uses m_l^{max} memory units as cache for each query it executes; that is, t_l uses all its data set to answer each query. Let $\mathfrak{m}(\mathfrak{I}(t))$ be the buffer size of $\mathfrak{I}(t)$. If $\sum_l m_l^{max} > \mathfrak{m}(\mathfrak{I}(t))$, then t is 0% warm. Otherwise, the warmness level of t is defined by $\mathfrak{m}(\mathfrak{I}(t)) - \sum_l m_l^{max}$ rounded down to the closest warmness level at which t is profiled. For example, if $\mathfrak{m}(\mathfrak{I}(t)) - \sum_l m_l^{max} = 0.74\mathfrak{m}(\mathfrak{I}(t))$, and the set of warmness levels at which t is profiled is $\{0, 25, 50, 75, 100\}$, then t is considered 50% warm. Having decided the warmness level of t, the load of t is computed as a function in the buffer size and the warmness level (e.g., as in Figures 4 and 3).

We explain our Greedy Tenant Consolidation (GTC) heuristic as follows. The input to the heuristic is a set of tenant profiles that provide the load of each tenant as a function in buffer size and warmness level. The output is an instance assignment function $\mathfrak{I}$ that assigns each tenant to a database instance, and a memory assignment function $\mathfrak{m}$ that assigns a buffer size to each database instance. The cost function that we want to minimize is $\max\{\sum_i l(t_i), \sum_j \mathfrak{m}(I_j)\}$. Instead of searching for a global minimum for the cost function, our heuristic operates greedily by picking at each iteration a pair of database instances whose consolidation reduces the cost function as much as possible.

In order to compute how much reduction in the cost function can be achieved by consolidating two database instances, say I_1 and I_2 where $m(I_2) \geq m(I_1)$, first we need to choose a buffer size for the potential consolidated instance, say I_{new}, in the range from $\mathfrak{m}(I_2)$ through $\mathfrak{m}(I_1) + \mathfrak{m}(I_2)$. These upper and lower bounds guarantee that Invariants (1) and (2) are true, respectively. The upper bound is obvious. To justify the lower bound we make two notes. First, note that the load is a monotonically non-increasing function in buffer size and warmness level. Since we start from the output of private instance configuration (Section 4), decreasing the buffer size of any tenant must increase the load because private instance configuration terminates only after taking advantage of any chance of decreasing the buffer size of a tenant while maintaining the load of that tenant intact. Second, note that consolidation either decreases warmness (which increases load), or keeps warmness and load intact (if warmness is already 0%). Based on these two notes, if we choose a buffer size for I_{new} less than $\mathfrak{m}(I_2)$, the buffer sizes of all tenants in I_{new} decrease, and their warmness levels either decrease or remain the same, thus total load increases, which violates Invariant (2).

Within the range $[\mathfrak{m}(I_2), \mathfrak{m}(I_1) + \mathfrak{m}(I_2)]$, we choose a buffer size that minimizes the cost function $\max\{\sum_i l(t_i), \sum_j \mathfrak{m}(I_j)\}$. If more than one buffer size minimize the cost function, we choose the one that also minimizes $\min\{\sum_i l(t_i), \sum_j \mathfrak{m}(I_j)\}$. After we choose a buffer size for the potential instance I_{new}, which corresponds to the pair of instances I_1 and I_2, we compare this pair of instances against other candidate pairs of instances, and choose the pair whose consolidation achieves as much reduction of the cost function $\max\{\sum_i l(t_i), \sum_j \mathfrak{m}(I_j)\}$ as possible. We terminate once no consolidation decreases the cost function.

6. EXPERIMENTS

We carry out a set of experiments to demonstrate the effectiveness, scalability, and correctness of our algorithms and heuristics. Our experiments are based on the TPC-H benchmark[2], which is representative of OLAP workloads. As mentioned before, we run our experiments on MySql 5.5 with InnoDB as a storage engine. We use Linux 2.6 machines each with a single disk and 2 Intel Quad-Core Xeon processors with hyper-threading. We start by explaining the tools that we use to conduct our experiments.

6.1 Tenant Synthesizer

We develop a tenant synthesizer to generate synthetic tenant profiles. For each synthetic tenant we generate: (1) a random mixture of TPC-H queries that constitute the workload of the tenant, (2) a random data size, and (3) a feasible random throughput value. We compute the profile of each tenant from its query mixture and data size, then use the computed profile to determine the range of feasible throughput values from which we pick the random throughput value of the tenant. We use two approaches to compute the profile of a tenant from its random query mixture and random data size. We begin by explaining our two approaches informally as follows. In the first approach, we profile individual TPC-H queries on a database with 1 gigabytes of data, generate a random TPC-H query mixture for each tenant, compute the profile of each tenant from the profiles of its query mixture, generate a random data size for each tenant, then *scale* the profile of the tenant according to its random data size. This approach allows us to pick arbitrary data sizes, without having to re-profile TPC-H queries for each random data size that we generate, but scaling the profiles of tenants generates profiles that do not correspond to real measurements. In the second approach we define a small set of data sizes, profile individual TPC-H queries on each of these data sizes, generate a random query mixture for each tenant, pick a random data size for each tenant from the pre-defined set of data sizes, then compute the profile of each tenant from the profiles of its query mixture that correspond to the data size of the tenant. This approach generates tenants with profiles that correspond to real measurements, but we are restricted to pick the data size of each tenant from a small set of data sizes. We explain the process of tenant synthesis in details as follows.

We define a set D of data sizes, and for each data size $d \in D$ we

[2]http://www.tpc.org/tpch/

create a TPC-H database with data size d. On each of these TPC-H databases, we profile TPC-H query templates as follows. Let H be the set of all query templates in the TPC-H benchmark. For each template $h \in H$, we generate a set of queries Q_h by assigning random values to the parameters of h, then we use Q_h as input to our profiler to generate a profile of h. The profile of the template h is a function $p_h(d, b, w)$ that returns the average query execution time of h when run on a database with a data size d, a buffer size b, and a cache warmness level w. In our experiments, the set D contains three data sizes: 1 gigabyte, 2 gigabytes, and 3 gigabytes. We profile TPC-H query templates for each data size $d \in D$, with buffer sizes that range from 100 megabytes to d, at 100 megabytes intervals, and under five warmness levels: 0%, 25%, 50%, 75%, and 100%.

After profiling all templates in H, we generate two sets of synthetic tenants, T_{real} and T_{scaled}. First, we explain the set T_{real}. For each tenant $t \in T_{real}$ we randomly pick a data size d_t from the set D. We also generate a random query mixture for t by tossing a fair coin for each template $h \in H$ to decide whether t uses h or not. Let H_t denote the set of TPC-H queries used by t. We compute the profile p_t of the tenant t as follows. For each buffer size b, and each warmness level w, we set $p_t(b, w)$ to the average of the profiles of all templates in H_t with a data size of d_t, a buffer size of b, and a warmness level of w; that is, $p_t(b, w) = \sum_{h \in H_t} p_h(d_t, b, w)/|H_t|$. Second, we explain the set T_{scaled}. For each tenant $t \in T_{scaled}$ we randomly pick a data size d_t following an exponential distribution whose mean λ is an input parameter to the tenant synthesizer. We also generate a random query mixture for t by tossing a fair coin for each template $h \in H$ to decide whether t uses h or not. Let H_t denote the set of TPC-H queries used by t. We compute the profile p_t of the tenant t as follows. For each buffer size b, and each warmness level w, we set $p_t(d_t \cdot b, w)$ to d_t times the average of the profiles of all templates in H_t with a data size of 1 gigabytes, a buffer size of b, and a warmness level of w; that is, $p_t(d_t \cdot b, w) = d_t \cdot \sum_{h \in H_t} p_h(1, b, w)/|H_t|$.

Finally, we generate for each tenant t a feasible SLA throughput value r_t as follows. Let $e_t^{min} = \min_{b,w}\{p_t(b, w)\}$; that is, the minimum query execution time in the profile of the tenant t. Since we assume that p_t is monotonically non-decreasing, e_t^{min} can be achieved when b and w are maximum. Similarly, let $e_t^{max} = \max_{b,w}\{p_t(b, w)\}$, which is achieved when b and w are minimum. We are interested in generating a throughput value r_t that the tenant t can satisfy, otherwise the placement problem is unsolvable, thus $1/e_t^{min}$ is the maximum value of r_t. However, $1/e_t^{max}$ does not represent a lower bound on the value of r_t. We generate a random average query execution time, e_t^{rand}, uniformly from the range $[e_t^{min}, \mathcal{C} \cdot e_t^{max}]$, where $\mathcal{C}$ is our *coldness factor*, then we set $r_t = 1/e_t^{rand}$. The coldness factor is a parameter to the tenant synthesizer, whose value is no less than e_t^{min}/e_t^{max}, and that we use to push the average throughput of tenants lower or higher.

6.2 Evaluation: Effectiveness

We show that our algorithm GMR is effective by comparing it against other baseline memory configuration heuristics. We use three heuristics for comparison: (1) Max: sets the buffer size of each tenant t_i to its data size m_i^{max}, (2) Min: sets the buffer size of each tenant t_i to the minimum amount of memory m_i^{min} that satisfies its SLA constraints, and (3) Const: sets the buffer sizes of all tenants to the same value, which is the minimum buffer size that satisfies the SLA constraints of all tenants.

We construct two sets of tenants, T_{real} and T_{scaled}, each contains 1000 synthetic tenants generated using our tenant synthesizer as explained in Section 6.1. We use these synthetic tenants as input to GMR, BMR, and the three baseline heuristics. The output of each of these five memory configuration heuristics is a set of 2-dimensional vectors that represent the buffer sizes and the loads of tenants after memory configuration. This set of 2-dimensional vectors is used as input to an approximation algorithm for 2-dimensional vector packing to generate a *tenant placement plan*. We use First Fit Decreasing (FFD) [9] as an approximation algorithm for 2-dimensional vector packing, which has a worst case approximation ratio of 7/3. Also, we generate one more tenant placement plan by using the output of GMR as input to GTC, to consolidate tenants into shared database instances, then we apply FFD to the output of GTC. Finally, we compare the total number of servers needed by each of these six tenant placement plans.

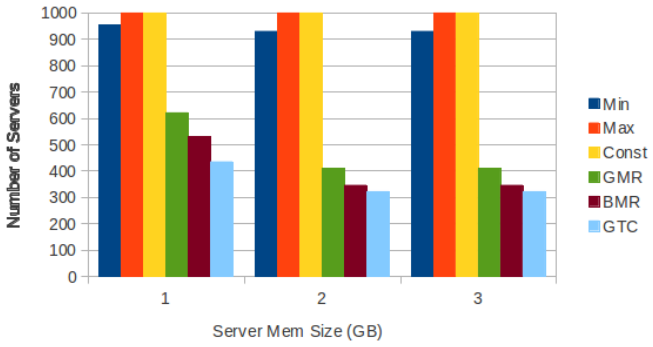

Figure 6: The number of servers needed by each placement plan for different server memory sizes.

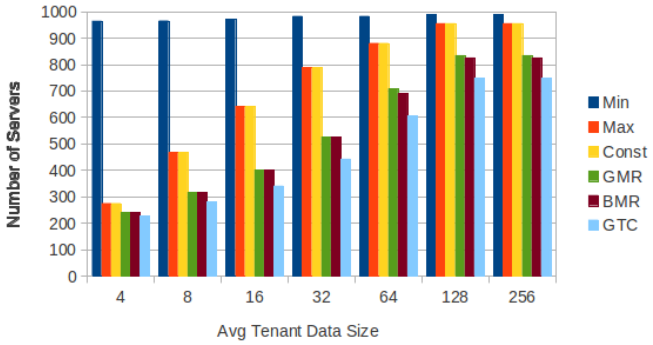

Figure 7: The number of servers needed by each placement plan as the average tenant data size increases. Coldness = 2.

Figure 6 shows the number of servers needed by each of the six placement plans to host T_{real} as the server memory data size $\mathcal{M}$ increases from 1 gigabytes to 3 gigabytes, while the coldness factor $\mathcal{C}$ remain constant at 2. Figure 6 shows that our algorithms and heuristics give superior results in all cases. For server memory size of more than 1 gigabyte, the gap between our algorithms and the baseline algorithms is more than 500 servers. Our experiments show that performance remains the same for $\mathcal{C} = 5$. Figure 7 shows the number of servers needed by each of the six placement plans to host T_{scaled} as the average tenant data size λ increases from 4 gigabytes to 256 gigabytes. The server memory size $\mathcal{M}$ remains constant at 16 gigabytes, and the coldness factor $\mathcal{C}$ equals 2. Figure 7 shows that our algorithms and heuristics give superior results in all cases. Min performs worst, requiring almost as many servers as the number of tenants. Both Max and Const give exactly the same results. The gap between Max/Const and GMR is maximum when $\lambda = 32$; that is, 261 servers. GTC further reduces the

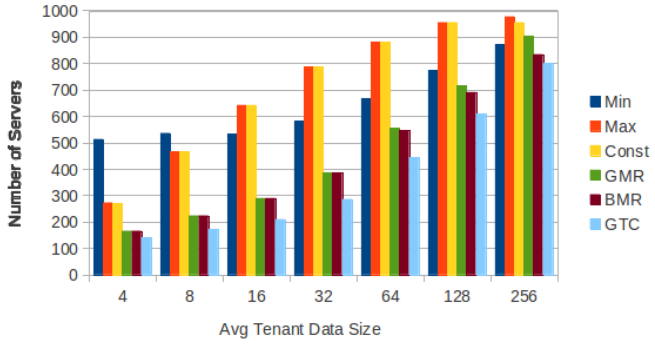

Figure 8: The number of servers needed by each placement plan as the average tenant data size increases. Coldness = 5.

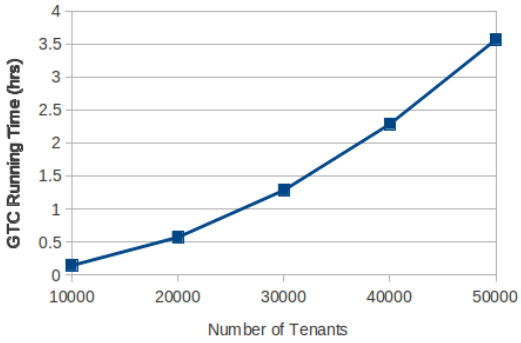

Figure 9: The running time of GTC as the number of tenants increases.

number of servers, and the maximum reduction is achieved when $\lambda = 64$; that is, 104 servers. When λ is too small or too large compared to $\mathcal{M}$, the gap decreases between Max/Const and GMR on the one hand, and between GMR and GTC on the other hand. As mentioned in Section 4.4, although BMR does not have a theoretical approximation guarantee like GMR, BMR performs as good as GMR in practice, and is even better when λ is much larger than $\mathcal{M}$. Figure 8 shows the same experiment when the coldness factor $\mathcal{C}$ equals 5. Max and Const maintain nearly the same results as in Figure 7, with Const doing slightly better in some cases. Min performs better compared to Figure 7 as the number of servers is initially smaller, and grows slower. GMR and BMR still give superior results compared to other baseline heuristics, and perform better than Figure 7 for most data sizes.

6.3 Evaluation: Scalability and Correctness

We also evaluate the scalability of our algorithms and heuristics by measuring the running time as the number of tenants increases. GMR and BMR process up to 50,000 tenants in less than a second. We focus on GTC since it has quadratic time complexity. Figure 9 shows the running time of GTC as the number of tenants increase from 10,000 to 50,000. The server memory size $\mathcal{M}$ is 16 gigabytes, and the coldness factor $\mathcal{C}$ is 2. Figure 9 shows that GTC can process tens of thousands of tenants within a few hours on single machine.

Finally we evaluate the correctness of our algorithms and heuristics by showing that the placement plans generated by GMR, BMR, and GTC, when implemented in practice, satisfy the throughput SLA requirements of tenants. Figure 10 compares the SLA throughputs of tenants against the actual throughputs that our generated placement plans can handle. First, we pick 30 random tenants from T_{real}, and run GMR, BMR, and GTC on them, followed

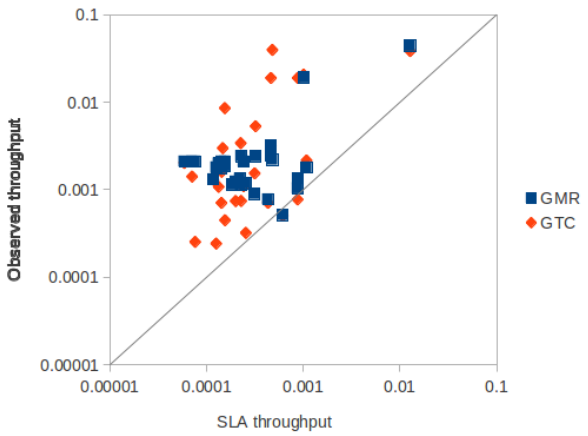

Figure 10: Observed throughput versus SLA throughput.

by FFD, to get placement plans. GMR and BMR give the same placement plan, requiring 10 servers, while GTC requires 8 servers. We deploy these tenants on servers according to the resulting placement plans, then we run the workloads. We schedule the workloads of the tenants hosted by each server in a round robin fashion to see the maximum throughput supported by the placement plans. Figure 10 shows that our placement plans satisfy the throughput SLAs of tenants in the real deployment, and for most tenants there is enough room to handle spikes of traffic.

7. CONCLUSION

We present a set of algorithms and heuristics that balance memory and time resources in order to minimize the total number of servers required to host a set of IO-bound tenants, while meeting the throughput SLA of each tenant. We first tackle the problem for private DBMS instances by approximating a globally-optimum memory assignment that minimizes the total number of servers needed, and present an online version that works well in practice, then we develop a heuristics for consolidating tenants into shared DBMS instances. Finally we evaluate our algorithms and heuristics experimentally, and show that they effectively reduce the number of servers needed to host a given set of tenants compared to other baseline heuristics. We also demonstrate that the tenant placement plans generated by our algorithms and heuristics meet the throughput SLA requirements of the tenants as expected.

8. REFERENCES

[1] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Interaction-aware scheduling of report-generation workloads. *The VLDB Journal*, 20:589–615, 2011.

[2] M. Ahmad and I. T. Bowman. Predicting system performance for multi-tenant database workloads. In *DBTest*, 2011.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical report, University of California at Berkeley, 2009.

[4] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, 2008.

[5] A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, 2001.

[6] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[7] W. F. de la Vega and G. S. Lueker. Bin packing can be solved within 1+epsilon in linear time. *Combinatorica*, 1(4):349–355, 1981.

[8] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Who's driving this cloud? towards efficient migration for elastic and autonomic multitenant databases. Technical report, University of California at Santa Barbara, 2010.

[9] M. R. Garey, R. L. Graham, and D. S. Johnson. Resource constrained scheduling as generalized bin packing. *J. Comb. Theory, Ser. A*, 21(3):257–298, 1976.

[10] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[11] G. Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM*, 52, 2009.

[12] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.*, 26, 1997.

[13] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. *SIGMOD Rec.*, 16, 1987.

[14] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting database applications as a service. In *ICDE*, 2009.

[15] O. Ozmen, K. Salem, M. Uysal, and M. H. S. Attar. Storage workload estimation for database management systems. In *SIGMOD*, 2007.

[16] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, 2011.

[17] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to determine a good multi-programming level for external scheduling. In *ICDE*, 2006.

[18] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *SOCC*, 2011.

[19] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35:7:1–7:47, February 2008.

[20] C. A. Waldspurger and M. Rosenblum. I/o virtualization. *Commun. ACM*, 55(1):66–73, 2012.

[21] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigumus. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE*, 2011.

APPENDIX

A. PROOF OF THE GUARANTEE OF GMR

To keep the notation simple, we use $\delta_i(k)$ to indicate both the amount of memory, and the range $[m_i(k) - \delta_i(k), m_i(k)]$. For any m in the range $\delta_i(k)$, such that m is a multiple of μ, we refer to the range $[m_i(k) - \delta_i(k), m]$ as a *tail* of $\delta_i(k)$, and the range $[m, m_i(k)]$ as the *remainder* of the tail. For any tail β, we use the symbol β to refer to the range of the tail as well as the size of the tail. Similarly, for any remainder α of the tail β, we use the symbol α for both the range and its size. For any m in the range $[m_i^{min}, m_i(k) - \delta_i(k))$, such that m is a multiple of μ, we refer to the range $[m, m_i(k) - \delta_i(k)]$ as a *prefix* of $\delta_i(k)$. For any prefix γ, we use the symbol γ to refer to both the prefix range and its size. We also use the function $l()$ to indicate the load drop across a range, and $\bar{l}()$ to indicate the average load drop across a range; for example, $l(\delta_i(k)) = l_i(m_i(k) - \delta_i(k)) - l_i(m_i(k))$ and $\bar{l}(\delta_i(k)) = l(\delta_i(k))/\delta_i(k)$. The proof of the approximation guarantee of GMR is based on the following two lemmas.

LEMMA A.1. *If β is a tail of $\delta_i(k)$ then $\bar{l}(\beta) \leq \bar{l}(\delta_i(k))$.*

Proof: Assume for contradiction that $\bar{l}(\beta) > \bar{l}(\delta_i(k))$. We consider the non-trivial case when $0 < \beta < \delta_i(k)$, since otherwise the lemma is obvious. Let α be the remainder of β. Since $l(\delta_i(k)) = l(\alpha) + l(\beta)$, therefore $\bar{l}(\delta_i(k)) = \frac{1}{\delta_i(k)} l(\alpha) + \frac{1}{\delta_i(k)} l(\beta) = \frac{\alpha}{\delta_i(k)} \bar{l}(\alpha) + \frac{\beta}{\delta_i(k)} \bar{l}(\beta)$. Sine $\delta_i(k) = \alpha + \beta$, therefore $\bar{l}(\delta_i(k))$ is a convex combination of $\bar{l}(\alpha)$ and $\bar{l}(\beta)$. Since we assume that $\bar{l}(\beta) > \bar{l}(\delta_i(k))$, and $0 < \beta < \delta_i(k)$, therefore $\bar{l}(\alpha) < \bar{l}(\delta_i(k))$. Note that one of the endpoints of the range α is at $m_i(k)$, thus α is among the possible δ's for tenant t_i at iteration k. But by definition, $\delta_i(k)$ is the δ with the smallest value of $\bar{l}()$ for the tenant t_i at iteration k. Therefore we reach a contradiction. ■

LEMMA A.2. *If γ is a prefix of $\delta_i(k)$ then $\bar{l}(\gamma) > \bar{l}(\delta_i(k))$.*

Proof: Assume for contradiction that $\bar{l}(\gamma) < \bar{l}(\delta_i(k))$. Let δ' be the concatenation of the ranges γ and $\delta_i(k)$. Then $\bar{l}(\delta')$ is a convex combination of $\bar{l}(\gamma)$ and $\bar{l}(\delta_i(k))$. Since we assume that $\bar{l}(\gamma) < \bar{l}(\delta_i(k))$, and by definition $\gamma > 0$ and $\delta_i(k) > 0$, therefore $\bar{l}(\gamma) < \bar{l}(\delta') < \bar{l}(\delta_i(k))$. Note that one of the endpoints of the range δ' is at $m_i(k)$, thus δ' is among the possible δ's for tenant t_i at iteration k. But by definition, $\delta_i(k)$ is the δ with the smallest value of $\bar{l}()$ for the tenant t_i at iteration k. Therefore we reach a contradiction. ■

Proof of Theorem 4.1

Proof: We begin by proving that there is no memory assignment $\mathfrak{m}$ that satisfies both of the following two inequalities: (1) $\sum_i \mathfrak{m}(t_i) < \sum_i \dot{\mathfrak{m}}(t_i)$ and (2) $\sum_i l_i(\mathfrak{m}(t_i)) < \sum_i l_i(\dot{\mathfrak{m}}(t_i))$. Then we show that if there exists a memory assignment $\mathfrak{m}$ that satisfies only one of the two inequalities, then the difference between $Lwr\_2DVP(\mathfrak{m})$ and $Lwr\_2DVP(\dot{\mathfrak{m}})$ is no more than 1.

Assume for contradiction that there exists a memory assignment $\mathfrak{m}$ that satisfies both Inequality 1 and Inequality 2. Let t_a be any tenant such that $\mathfrak{m}(t_a) < \dot{\mathfrak{m}}(t_a)$. Also, let t_b be any tenant such that $l_b(\mathfrak{m}(t_b)) < l_b(\dot{\mathfrak{m}}(t_a))$. Since l_b is monotonically non-increasing, therefore $\mathfrak{m}(t_b) > \dot{\mathfrak{m}}(t_a)$. Let Δ_a denote the range $[\mathfrak{m}(t_a), \dot{\mathfrak{m}}(t_a)]$, and let k_a be the iteration at which GMR sets the buffer size of tenant t_a to $\dot{\mathfrak{m}}(t_a)$. Similarly, let Δ_b denote the range $[\dot{\mathfrak{m}}(t_b), \mathfrak{m}(t_b)]$. GMR removes the range Δ_b from the buffer of t_b in one or more δ's. Let these δ's be ordered chronologically as $\delta_b^1, \delta_b^2, ..., \delta_b^c$, where δ_b^1 is the first removed δ. Let k_b be the iteration at which δ_b^1 is picked by GMR; that is, $\delta(k_b) = \delta_b^1$. Note that δ_b^1 may be not completely contained within the range Δ_b. We refer to the intersection between the ranges Δ_b and δ_b^1 simply as $\Delta_b \cap \delta_b^1$. The range $\Delta_b \cap \delta_b^1$ is actually a tail of δ_b^1. Therefore, from Lemma A.1, $\bar{l}(\Delta_b \cap \delta_b^1) \leq \bar{l}(\delta_b^1)$. We prove that for any such t_a and t_b, $\bar{l}(\Delta_b) \leq \bar{l}(\Delta_a)$, then we use this inequality to reach a contradiction with Inequality 2. Consider the two possible cases, either (1) $k_a < k_b$, or (2) $k_b < k_a$. For Case 1, at iteration k_b, the buffer size of tenant t_a is still at $\dot{\mathfrak{m}}(t_a)$, unchanged since k_a. Therefore Δ_a is one of the δ's that are considered for comparison at iteration k_b. Since GMR picks δ_b^1 at iteration k_b, therefore $\bar{l}(\Delta_a) \geq \bar{l}(\delta_b^1) \geq \bar{l}(\Delta_b \cap \delta_b^1)$. Similarly, since $\delta_b^2, ..., \delta_b^c$ are all

picked by GMR at iterations subsequent to iteration k_b, therefore $\bar{l}(\Delta_a) \geq \bar{l}(\delta_b^j)$, for all j. Note that $\bar{l}(\Delta_b)$ is a convex combination of $\bar{l}(\Delta_b \cap \delta_b^1), \bar{l}(\delta_b^2), ..., \bar{l}(\delta_b^c)$. Therefore $\bar{l}(\Delta_b) \leq \bar{l}(\Delta_a)$. For Case 2, at iteration k_b, the buffer size of tenant t_a is at $m_a(k_b) > \dot{\mathfrak{m}}(t_a)$. Let Δ'_a denote the range $[\dot{\mathfrak{m}}(t_a), m_a(k_b)]$. GMR removes Δ'_a from the buffer of t_a in one or more δ's. Let these δ's be ordered chronologically as $\delta_a^1, \delta_a^2, ..., \delta_a^d$, where δ_a^1 is the first removed δ. Note that these δ's are prefixes of one another, therefore $\bar{l}(\delta_a^1) < \bar{l}(\delta_a^2) < ... < \bar{l}(\delta_a^d)$. Note also that Δ_a is a prefix of δ_a^d, therefore $\bar{l}(\delta_a^d) < \bar{l}(\Delta_a)$. Since δ_b^1 is picked by GMR at iteration k_b, rather than δ_a^1, therefore $\bar{l}(\Delta_b \cap \delta_b^1) \leq \bar{l}(\delta_b^1) \leq \bar{l}(\delta_a^1) < \bar{l}(\Delta_a)$. Similarly, for all δ_b^j, if δ_b^j is picked by GMR at an iteration before k_a, then $\bar{l}(\delta_b^j) < \bar{l}(\Delta_a)$. Otherwise, if δ_b^j is picked by GMR after k_a, then we apply the logic of Case 1 to show that $\bar{l}(\delta_b^j) \leq \bar{l}(\Delta_a)$. Therefore, for all j, $\bar{l}(\delta_b^j) \leq \bar{l}(\Delta_a)$. As with Case 1, since $\bar{l}(\Delta_b)$ is a convex combination of $\bar{l}(\Delta_b \cap \delta_b^1), \bar{l}(\delta_b^2), ..., \bar{l}(\delta_b^c)$, therefore $\bar{l}(\Delta_b) \leq \bar{l}(\Delta_a)$. From Inequality 1, since $\sum_i \mathfrak{m}(t_i) < \sum_i \dot{\mathfrak{m}}(t_i)$, therefore $\sum_a \Delta_a > \sum_b \Delta_b$. And since $\bar{l}(\Delta_a) \geq \bar{l}(\Delta_b)$ for all a and b, therefore $\sum_a \Delta_a \bar{l}(\Delta_a) > \sum_b \Delta_b \bar{l}(\Delta_b)$, which can be re-written as $\sum_a l(\Delta_a) > \sum_b l(\Delta_b)$. Therefore, $\sum_i l_i(\mathfrak{m}(t_i)) > \sum_i l_i(\dot{\mathfrak{m}}(t_i))$, which contradicts with Inequality 2. Thus, there is no memory assignment $\mathfrak{m}$ that satisfies both inequalities, 1 and 2.

Consider the case when $\mathfrak{m}$ satisfies only one of the two inequalities, either 1 or 2. We prove that in such case $Lwr\_2DVP(\dot{\mathfrak{m}}) \leq Lwr\_2DVP(\mathfrak{m}) + 1$. To prove this, we consider two possible cases, either (1) $\sum_i \dot{\mathfrak{m}}(t_i) \leq \sum_i l_i(\mathfrak{m}(t_i)), \sum_i \mathfrak{m}(t_i) < \sum_i l_i(\dot{\mathfrak{m}}(t_i))$, or (2) $\sum_i l_i(\dot{\mathfrak{m}}(t_i)) \leq \sum_i l_i(\mathfrak{m}(t_i)), \sum_i \mathfrak{m}(t_i) < \sum_i \dot{\mathfrak{m}}(t_i)$. Other than these two cases, $Lwr\_2DVP(\mathfrak{m}) > Lwr\_2DVP(\dot{\mathfrak{m}})$, and the theorem follows directly. For Case 1, note that the loop of GMR terminates once total memory is no more than total load. Let k_l be the last iteration, and let $d = \sum_i m_i(k_l) - \sum_i l_i(m_i(k_l))$, and $d' = \sum_i m_i(k_l - 1) - \sum_i l_i(m_i(k_l - 1))$. Since each iteration of the loop of GMR decreases the buffer size of a single tenant, and since the buffer size and load of each tenant is at most 1, therefore each iteration decreases total memory by no more than 1 and increases total load by no more than 1. Thus the difference between total memory and total load decreases by no more than 2 after each iteration. In our case, $d \leq 0$, therefore $d' \leq 2$. If $d' \geq 1$, then $d \geq -1$, thus the gap between total memory and total load when GMR terminates is no more than 1. If $d' < 1$, then d may be less than -1, however in such case the last iteration degrades the solution rather than improving it. We check for this case after the loop terminates, and if it occurs we rollback the last iteration, thus the final gap size is d'. Therefore, in both cases, the gap between total memory and total load is no more than 1. Since $Lwr\_2DVP(\mathfrak{m})$ lies within this gap, therefore the difference between $Lwr\_2DVP(\mathfrak{m})$ and $Lwr\_2DVP(\dot{\mathfrak{m}})$ is no more than 1. For Case 2, $\sum_i \dot{\mathfrak{m}}(t_i) > \sum_i \mathfrak{m}(t_i)$. Then there must be a tenant t_a such that $\mathfrak{m}(t_a) < \dot{\mathfrak{m}}(t_i)$. But from the definition of GMR, the two termination conditions are either (i) $m_i(k) = m_i^{min}$ for all t_i, which is not true in this case since $m_a(t_a) \neq m_a^{min}$, or (ii) $\sum_i m_i < \sum_i l_i(m_i)$, which is also not true by the definition of Case 2. Therefore we reach a contradiction. ■