PMAX: Tenant Placement in Multitenant Databases for Profit Maximization

Ziyang Liu, Hakan Hacıgümüş, Hyun Jin Moon*, Yun Chi, Wang-Pin Hsiung
NEC Laboratories America
{ziyang,hakan,hjmoon,ychi,whsiung}@nec-labs.com

ABSTRACT

There has been a great interest in exploiting the cloud as a platform for database as a service. As with other cloud-based services, database services may enjoy cost efficiency through consolidation: hosting multiple databases within a single physical server. Aggressive consolidation, however, may hurt the service quality, leading to *SLA violation penalty*, which in turn reduces the total business profit, called *SLA profit*. In this paper, we consider the problem of tenant placement in the cloud for SLA profit maximization, which, as will be shown in the paper, is strongly NP-hard. We propose SLA profit-aware solutions for database tenant placement based on our model for expected penalty computation for multitenant servers. Specifically, we present two approximation algorithms, which have constant approximation ratios, and we further discuss improving the quality of tenant placement using a dynamic programming algorithm. Extensive experiments based on TPC-W workload verified the performance of the proposed approaches.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Algorithms, Performance

Keywords

Multitenancy, Database, Cloud, SLA, Profit Optimization

1. INTRODUCTION

With the increasing popularity of cloud computing, the benefits of hosting applications on a cloud service provider (including IaaS, PaaS and SaaS) becomes more and more apparent and widely accepted. It not only avoids high capital expenditure, but also minimizes the risk of under-provisioning and over-provisioning using an elastic, pay-as-you-go type approach. From the cloud service provider's perspective, to achieve economies of scale, an important practice is to use each physical server to host multiple tenants (a.k.a. multitenancy). One approach is to use virtual machines with fixed resources such as Amazon EC2 instances. It provides good isolations among different tenants, but the downside is the compromise of resource sharing. An alternative is that all tenants use the same resource pool, so that occasional workload bursts from a few tenants can be gracefully handled. Such a model is often used in Software as a Service platforms where tenants run Web-based services that are accessed by Web users [21, 32, 12, 11, 15, 6, 16, 13]. Usually the revenue of a cloud service provider is generated by delivering the contracted services. The provider negotiates the service level agreements (SLAs) with each tenant, in terms of various criteria such as response time, availability, throughput, etc. [8, 9]. For example, a service level agreement may be that if the response time of the query is less than 5 seconds, the tenant pays the provider for the service, otherwise the provider pays the tenant a penalty. Since maximizing the profit is the ultimate goal of a cloud service provider, the profit should be the main optimization criterion of decision-making in multitenant databases.

Cloud service providers have conflicting goals when distributing the tenants on compute servers. On the one hand, purchasing, renting or using each server incurs an expense, and thus the provider is motivated to use as few servers as possible. On the other hand, if a server hosts too many tenants and is too crowded, the situation may lead to a high chance of violating the service level agreements, which incurs penalties. Thus, the placement of tenants should be carefully planned in order to minimize the total cost.

One may naturally think of the connection between tenant placement and bin packing, a classic NP-hard problem. It is worthwhile to point out that tenant placement has several additional complications compared with bin packing. First, in bin packing, each item has a fixed size, whereas each cloud tenant's size may vary with time (e.g., a tenant may have more users on weekdays than on weekends). Second, in bin packing, a hard constraint is that the total size of all items in a bin cannot exceed the bin's capacity. In tenant placement there is no such constraint, however, the more tenants a server hosts, the more SLA violations may occur. Lastly, the optimization goal of bin packing is minimizing the total number of bins, whereas in tenant placement the goal is to maximize the profit by minimizing the total cost, which includes the cost of the servers and the penalty of SLA violations.

There are several existing tenant placement strategies for cloud service providers [10, 32, 21, 13, 22]. However, none of them is profit-driven or aims at minimizing the SLA violation penalty. Discussion of these works is presented in Section 2.

In this paper we propose a new tenant placement strategy with the following features that previous works do not exhibit: (1) The

*The current affiliation is Google Inc.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

resources consumed by a tenant may change with time and is described using probabilistic distributions; (2) We do not assume that historical data is available; (3) Under the described system models, the total cost (the cost of purchasing/renting servers and the SLA violation penalties) has a provable constant bound.

Specifically, we will provide two approximation algorithms, one for the case of uniform query processing time and the SLA penalty (i.e., all tenants' queries have the same processing time, and all SLA penalties are the same), and another algorithm for the general case. The first algorithm is proved to have an approximation ratio of 3, and the second is proved to have an approximation ratio of 4. Besides, we also propose to couple the second approximation algorithm with a dynamic programming procedure, which further improves the quality and reduces the cost. In the end, we will further discuss how to handle online placement as well as the trade-offs of the algorithms.

The contributions of the paper are summarized as:

- We propose a solution to the tenant placement problem to maximize the provider's profit by minimizing the total cost, which includes the cost of the servers and the penalty of SLA violations. Our solution allows the tenants' resource usage to vary with time and be described using probabilistic distributions, and does not rely on historical data.
- We provide two approximation algorithms, one for a restricted case (uniform query processing time and SLA penalty across tenants) whose approximation ratio is 3, and one for the general case whose approximation ratio is 4.
- A dynamic programming algorithm is also introduced, which can be coupled with the approximation algorithm for better quality and lower cost.
- Experiments verify the effectiveness of the proposed approach.

The remainder of this paper is organized as follows. Related works are discussed in Section 2. Section 3 explains the system model used in this paper. The formal problem definition and hardness proof is presented in Section 4. Section 5 illustrates the algorithms for tenant placement, including a greedy algorithm adapted from Best Fit for bin packing, approximation algorithms for the uniform processing time/SLA penalty case and the general case, respectively, as well as a dynamic programming algorithm that can be coupled with the approximation algorithm. Experimental evaluations are reported in Section 6. Section 7 briefly discusses the complexities of the algorithms as well as online placement, and Section 8 concludes the paper.

2. RELATED WORK

Multitenancy Models. There are several possible multitenancy schemes, such as private virtual machine, private DB instance, private database, private table, shared table, etc. [20] provides an excellent overview of these tradeoffs.

Private virtual machine Using virtual machine technology, we may host multiple virtual machines on a single physical machine. Each virtual machine, however, hosts only a single database tenant. This option gives good isolation across tenants, but there is virtual machine performance overhead, and the number of VMs that can be hosted within a physical machine is quite limited (i.e. a dozen or two) using today's technology.

Private DB instance Within a physical machine, we may start multiple database instances and let each tenant use one database instance. Running a DB instance involves a high memory overhead, so this option also has a limited scalability.

Private database DBMS allows that multiple databases can be created within a single instance. Hence we may allow each tenant to have its own database while sharing the DB instance. In many DBMS offerings, each database usually comes with its own buffer pool, so memory isolation is well implemented with this scheme.

Private table In this scheme, multiple tenants share a database, while each tenant uses its own set of tables. This scheme is attractive for its relatively low memory overhead of each table compared to that of a database mentioned above, i.e. 4KB per table in DB2 [3], and 9KB per table in MySQL [19].

Shared table This is the most scalable scheme, since many DBMSs are specially designed for big tables. Packing many tenants into a single table to create a big table can nicely exploit the DBMS' inherent capability. The main challenge, however, is the heterogeneity of schemas across tenants, which requires a solution for putting them together into a single table. Several solutions have been proposed and evaluated in [3, 4, 19].

In our implementation of this paper we adopt the private table scheme. However, we believe that multitenancy models are orthogonal to the tenant placement problem, and cloud service providers should be able to freely choose the best sharing scheme for their tenants, considering scalability and isolation requirements.

Tenant Placement. There are several existing tenant placement strategies for cloud service providers [10, 32, 21, 13, 22]. However, none of them is profit-driven or aims at minimizing the SLA violation penalties. The optimization goal of [10] is to minimize the number of servers, subject to the constraint that even when the tenants' load reaches maximum (based on historical monitoring), none of the server's resources (CPU, memory, disk) should be saturated. There are two problems with this approach. First, no SLA is considered. As we show in this paper, SLA plays a critical role in our tenant placement strategy; placing the tenant without considering their SLAs leads to undesirable performance. Second, even if some servers are occasionally saturated, as long as the SLA penalty is smaller than the cost using more servers, there's no need to use more. In other words, minimizing the total cost (SLA penalty + server cost) should be the ultimate goal. [32] assumes that the resource consumed by each tenant is fixed and does not vary with time and the goal is to place tenants on servers such that the resource usage on each server is less than 100%, subject to certain constraints such as the maximum number of application/DB instances installed on each server. [21] uses a very similar approach except that it adopts a more complicated resource usage model that needs to be learned from historical data. In both [32] and [21], a simple best-fit algorithm is used for tenant placement, which finds the server with the least remaining resource that can accommodate each tenant, and in case such a server cannot be found, either relaxes the constraints gradually or use a new server. [13] starts with a random placement, then improves it using simulated annealing, with the aim of optimizing the "fitness" of the system, where "fitness" can be flexibly defined. [22] uses tps (transaction per second) as the SLO (service level objectives). It assumes a fixed number of tenant types and, similar as [32, 21], assumes that each tenant's tps value doesn't change. On each server, it assigns tenants of the same type to the same SQL server instance with a fixed amount of main memory allocated. The amount of memory allocated to each SQL server depends on the SLO of the tenant type, as well as the number of tenants of that type. The tenant placement problem is formulated as an integer programming problem and solved using a brute-force solver. In comparison, our approach to be introduced in this paper does not assume a fixed workload for each tenant, does not assume a fixed number of tenant types and uses the dollar cost of the service provider as the ultimate optimization goal.

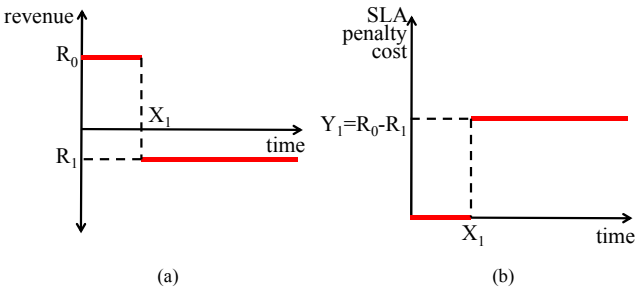


Figure 1: Service Level Agreement

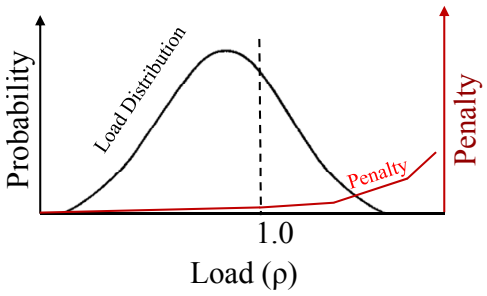


Figure 2: SLA Penalty wrt Server Load

Other Multitenancy Aspects. Various other aspects of multitenancy have been studied, including load balancing and database/VM migration and consolidation [12, 11, 5, 15, 6, 16], profit-driven query scheduling given users' Service Level Agreements [9, 8], profit driven query admission control [29], tenant migration with in-memory databases [27], schema mapping [3], etc. Since these problems are orthogonal to tenant placement, we omit the detailed discussion.

3. SYSTEM MODELING

Before formally defining the problem of tenant placement, we first introduce some important background regarding system modeling.

Service Level Agreement. SLAs in general may be defined in terms of various criteria, such as service latency, throughput, consistency, security, etc. In this paper, we focus on service latency, or response time.[1] Even with response time alone, there can be multiple specification methods: i) based on the average query response time [31], ii) based on the tail distribution of query response times (or quantile-based)[23], or iii) based on individual query response times [17, 28, 8]. We choose the last one, which has the finest granularity and most accurately captures the performance of the provider. Also, if preferred, there exist techniques (e.g., [14]) that directly map quantile-based SLAs to per-query SLAs.

In this paper we adopt one-step SLA revenue functions on query response time. An example is shown in Figure 1(a). The service provider gets a revenue if a query is processed before a certain deadline; otherwise the service provider pays a penalty. We adopt a similar setting as [8, 28, 29], where each tenant has one or more *query classes*; each query class is associated with an SLA and one or more queries associated. Similarly, an SLA cost function can be defined as the cost paid by the service provider with the increase of query response time, such as Figure 1(b).

Tenant and Server Load. The *load of a tenant* t is calculated by $load(t) = avgProc(t)/avgIntv(t)$, where $avgIntv(t)$ is the average interval length between two queries submitted by tenant t, which is the inverse of query arrival rate, and $avgProc(t)$ is the average query processing time of t.[2] The *load of a server* in a period of time is the number of queries received divided by the number of queries the server is able to process. Equivalently, the load of a server is the average query processing time of the server divided by the average interval length between two query arrivals. Thus server load>100% means that queries arrive at the server faster than the speed the server can process them, indicating an *overload* situation, and vice versa.

Note that the execution time of the same query may increase when more tenants are packed on a server. This is because a server's buffer pool size is limited, and with more tenants packed, the competition for buffer pool increases which leads to increased disk I/O. Thus if we use T to denote the set of tenants on a server s, then the load of server s can be expressed as

$$load(s) = f(T) \times \sum_{t \in T} load(t)$$

where $load(t)$ is the load of a tenant t if it is the only tenant on the server (we refer to this load as the *base load* of a tenant), and $f(T)$ is a factor that accounts for the increase of execution time when the set of tenants T is packed on the server.

For simplicity, in the remainder of the paper, we use "*tenant size*" to refer to the base load of a tenant (analogous to item size in the bin packing problem).

Tenant Workload. We model the query arrival of a tenant as a non-homogeneous Poisson process (i.e., the arrival rate may vary with time), which is widely used to model the arrival of items, e.g., customers, phone calls, and queries [7, 24, 28, 25]. Since an application may have different numbers of users at different times of day, days of week, etc., we use normal distribution to model the variation of arrival rate. It is shown in [2] that the amount of server usage (measured by the number of requests and amount of CPU usage) of Flickr servers exhibit normal distribution patterns. Since a tenant with a high query frequency usually observes a high variance, we assume that the variance of the normal distribution is proportional to its mean.

Server Penalty Estimation. A penalty estimation method is necessary to estimate how often a tenant's queries miss the SLA deadline, given the set of tenants on a server. This estimation is then used to guide the placement strategy. Nevertheless, the estimation method is usually orthogonal to the placement algorithms. Next we introduce the SLA penalty model we adopt, but it is not a main focus of the paper, and other SLA penalty models may be usable.

In this paper we adopt the following SLA penalty model: queries arrived during server overload (i.e., load>100%) will miss their SLA deadlines and the service provider needs to pay the penalty; and other queries will meet their SLA. This is because the load of a tenant will unlikely change very frequently (e.g., it may remain near a constant during morning rush hour, and near another constant during the night, etc.), and SLA penalties occur mainly because of prolonged system overload (e.g. more than 1 min or so), rather than a temporary burst in query arrival for a short period (e.g. a little bit dense arrival during 10 millisecond). If a system is overload for a long time (compared with the query execution time/SLA deadline), then the delay of processing the queries becomes longer

[1]We allow any level of throughput, as we aim to serve them using flexible cloud infrastructure.

[2]The average query processing time is computed over all queries, rather than a single query. If each query of tenant t takes 2 seconds and the server can process 2 queries in parallel, then the average query processing time of t is 1 second.

and longer, and the vast majority of the queries arrived during this time should miss the deadline. On the other hand, if the system is underload for a long time, then the vast majority of the queries should finish before the deadline (of course, the deadline should be reasonably longer than the query processing time). Therefore, the relationship between server penalty and server load should look like Figure 2. Note that SLA penalty starts to appear before load goes beyond 1.0. This is because the x-axis shows average load, while the actual load may fluctuate.

It is worth mentioning that, to make our approach widely applicable, we make no restrictions in the following aspects: (1) tenant sizes: different tenants may have arbitrarily different query frequencies; (2) Query processing times: queries of different tenants may have arbitrarily different average processing times; (3) SLA penalties: the SLA penalties for different query classes can be arbitrarily different; (4) Query scheduling algorithm: although we adopt the first come first served method due to simplicity of discussion and implementation, other scheduling algorithms such as cost based scheduling [8] can also be applied, as long as there is a way to calculate the expected SLA penalty given the information of tenants on a server.

4. PROBLEM DEFINITION

4.1 Uniform Query Processing Time and SLA Penalty

We start with discussing a relatively easier case in this section: all queries have the same execution time and the same SLA penalty. The only differences among the tenants are the frequencies of their queries (hence the loads). This scenario is not unrealistic if the tenants run Web applications, and all queries have similar execution times (such as a few milliseconds). Next we formally define the problem.

DEFINITION 4.1 (TP-UNIFORM). *We are given a set of tenants; each tenant t_i issues queries, such that the arrival of queries follows non-homogeneous Poisson process with rate λ_i, where λ_i follows normal distribution with mean μ_i and variance $\sigma_i^2 = k\mu_i$. Each query has an SLA penalty of L, and each server has a cost of D per time unit (e.g., hour). Also given in the input are the expected query execution time of each query when there is only one tenant on the server, and a function $f(T) \geq 1$ that describes how the execution time of a tenant increases if a set of tenants T is packed on a server. The goal is to place the tenants on the servers, such that the expected average load of each server is no more than 1, and the total cost is minimized. The total cost is defined as*

$$cost = mD + L\sum_{i=1}^{m} p(load_i > 1) \cdot arrival_i$$

where m is the number of servers used to host all tenants; $p(load_i > 1)$ is the probability that server i is overload, $arrival_i$ is the average query arrival rate (number of queries per time unit) of server i.

Note that due to the uniform query processing time and SLA penalty, each tenant can be assumed to have only a single query class.

LEMMA 4.1. TP-UNIFORM *is strongly NP-hard.*

PROOF. Consider a special case of TP-UNIFORM where the variance of query arrival rate is 0, i.e., queries of all query classes have a fixed average arrival rate. Then, each tenant's average base load is fixed.[3] Besides, assume $f(T) = 1$ for all T (i.e., the buffer pool has unlimited size). Then TP-UNIFORM becomes an equivalent problem as bin packing: each tenant corresponds to an item in the bin packing problem, and the goal is to use the fewest servers for the tenants, such that the load of each server is no more than 1. Therefore, bin packing is a special case of TP-UNIFORM, and thus TP-UNIFORM is strongly NP-hard. □

4.2 Nonuniform Query Processing Time and SLA Penalty

Similar as TP-UNIFORM, we define the TP-GENERAL problem as follows.

DEFINITION 4.2 (TP-GENERAL). *We are given a set of tenants; each tenant t_i issues multiple classes of queries, such that the arrival of queries in each class j follows non-homogeneous Poisson process with rate λ_{ij}, where λ_{ij} follows normal distribution with mean μ_{ij} and variance $\sigma_{ij}^2 = k\mu_{ij}$. Queries of the jth query class of tenant t_i have and an SLA penalty of L_{ij}, and each server has a cost of D per time unit (e.g., hour). Also given in the input are the expected execution time of each query class when there is only one tenant on the server, and a function $f(T) \geq 1$ that describes how the execution time of a tenant increases if a set of tenants T is packed on a server. The goal is to place the tenants on the servers, such that the expected average load of each server is no more than 1, and the cost is minimized. The cost is defined as*

$$cost = mD + \sum_{i=1}^{m}[p(load_i > 1) \cdot \sum_{t_k \in M_i} \sum_{j} perc(k,j) \cdot L_{kj}]$$

where m is the number of servers used to host all tenants; $p(load_i > 1)$ is the probability that server i is overload, $perc(k,j)$ is the percentage of queries on server M_i, which belong to the jth query class of tenant t_k.

TP-UNIFORM is a special case of TP-GENERAL, and thus TP-GENERAL is also strongly NP-hard.

5. TENANT PLACEMENT ALGORITHMS

5.1 Baseline Greedy Algorithm Based on Best Fit

Due to the similarity of tenant placement and bin packing, one would naturally consider an adaptation of the bin packing algorithm. For bin packing, a simple yet effective heuristics (Best Fit) is to place each item in the bin that can fit the item and has the smallest remaining space, or create a new bin if no existing bin can fit the item. For any bin packing instance, Best Fit uses at most twice the number of bins as any other solution.

We can easily use such a strategy for TP-UNIFORM and TP-GENERAL, with a small modification that we optimize for the cost, rather than trying to pack each server as full as possible. Specifically, since our goal is to minimize the cost, for each tenant, our greedy algorithm places it on one of the existing servers, or create a new server for this tenant, whichever choice minimizes the current expected cost.

Although the Best Fit algorithm for the bin packing problem has a constant approximation ratio, we will show that *the performance of such a strategy for either* TP-UNIFORM *or* TP-GENERAL *can be arbitrarily worse than the optimal solution*.

[3]The actual load may still vary due to the Poisson arrival, but based on the SLA penalty estimation method in Section 3, whether SLAs are met depends only on the average arrival rate.

EXAMPLE 5.1. *Suppose the sizes of all tenants are sufficiently small, such that even when the load of a server is close to 1, adding one more tenant is still cheaper than placing this tenant on a new server. Note that this is possible no matter how high the SLA penalty of each tenant is: we can always make the tenant sizes sufficiently small to get this effect. Thus in the output of the greedy algorithm, each server's expected load is close to 1. A problem with such a placement is that, although it costs more to move any single tenant to a new server, it may cost less if we move multiple tenants from a server to a new server, which may significantly reduce the expected SLA penalty of the original server.*

Specifically, we can construct a case where in the optimal solution, each server's average load is much smaller than 1, e.g., 0.5; while in the greedy solution, each server's average load is close to 1, e.g., 0.9. Although the optimal solution uses more servers, the number of servers cannot be arbitrarily large (in this case, it is no more than twice the number of servers in the greedy solution). On the other hand, if the SLA penalty is sufficiently high, then the SLA penalty of the greedy solution can be arbitrarily larger than that of the optimal solution (because the optimal solution will try to make the SLA violation rate, and hence the SLA penalty, close to 0, while the SLA penalty of the greedy solution is much larger). Therefore, the greedy solution can be arbitrarily worse than the optimal one.

As we can see, when the tenants are small, the greedy strategy tends to pack each server very full, which incurs a high SLA penalty due to a high probability that a server goes overload.

5.2 Approximation Algorithm for TP-Uniform

The main problem of the greedy algorithm introduced in Section 5.1 is that it tends to pack lots of tenants on a server when the tenants are small. And in multitenant databases, tenants are indeed small [26, 18]. Therefore, the idea of improvement is to proactively create new servers, even if it increases the current expected cost. To achieve this, we introduce the concept of *half full servers*. This is used to measure whether a server is too full and we should create new servers.

DEFINITION 5.1. *A* half full server *is a server whose total tenant size is S_H, such that: if we have another tenant t of size S_H, it is equally costly to place t on this server, compared with placing t on a new server.*

To compute S_H, we introduce the following lemma.

LEMMA 5.1. *Let $S(M)$ denote the average total size (i.e., total base load) of all tenants on server M. Given four servers M_a, M_b, M_c and M_d, if $S(M_a) + S(M_b) = S(M_c) + S(M_d)$ and $|S(M_a) - S(M_b)| < |S(M_c) - S(M_d)|$, then the expected cost of M_a and M_b is lower than that of M_c and M_d.*

We omit the detailed proof due to space limit. Basically it says that two servers with skewed loads cost more than two servers with balanced loads. The reason is the following. In the TP-UNIFORM case, since all queries have the same SLA and execution time, the SLA cost of a server depends on the probability of SLA violation, which in turn depends on the server overload probability. The overload probability of a server increases *superlinearly* with the load. Intuitively, when the average load of a server increases from 10% to 20%, the overload probability remains close to 0. But when the average load of a server increases from 80% to 90%, the overload probability should have a much bigger increase. And since a server with higher load has a higher query arrival rate, the SLA violation penalty increases even more from 80% to 90%, compared with from 10% to 20%. Therefore, two servers both with $x\%$ load cost less than two servers with $(x-\delta)\%$ and $(x+\delta)\%$ load, respectively.

Algorithm 1: Approximation Algorithm for TP-UNIFORM

```
Input : n tenants, the average query arrival rate of each
        tenant, query processing time r, SLA penalty L,
        server cost D
Find the value of S_H via binary search in (0, 1)
T = {t_1, ..., t_n} = sorted tenants in decreasing order of size
Create server M_1 for t_1
foreach t_i ∈ T do
    j = arg min_j additionalPenalty(t_i, M_j) where M_j is
    an existing server such that S(M_j) + S(t_i) ≤ 2S_H and
    additionalPenalty(t_i, M_j) ≤ Penalty(t_i) + D
    if j exists then
        Place t_i on M_j
    else
        Create a new server M_k
        Place t_i on M_k
    end
end
```

It follows from the conclusion of Lemma 5.1 that given the cost of a server and the SLA penalty of a query, the value of S_H can be computed via binary search in (0, 1).

The approximation algorithm for TP-UNIFORM is shown in Algorithm 1. $additionalCost(t_i, M_j)$ denotes the additional penalty of placing t_i on M_j, and $Penalty(t_i)$ denotes the penalty of placing t_i alone on a server. First, Algorithm 1 finds the value of S_H, and sort the tenants in decreasing order of size. Then, for each tenant t_i, it finds an existing server M_j such that (1) placing t_i on M_j will not make the total tenant size on M_j exceed $2S_H$; (2) placing t_i on M_j is cheaper compared with creating a new server for t_i; (3) placing t_i on M_j is the cheapest among all existing servers. If such a server M_j exists, we place t_i on M_j. Otherwise, we create a new server for t_i.[4]

Next we prove that Algorithm 1 has an approximation ratio of 3. For an arbitrary instance, let OPT denote the optimal solution and APP denote the solution of Algorithm 1. Besides, for proof purpose, we define another problem similar as TP-UNIFORM. In this problem, a tenant (except those defined below) is allowed to be split into arbitrary number of parts, and placed on different servers. For example, a tenant with size 10 can be split into three parts with sizes 2, 3 and 5, and placed on three servers. However, two types of tenants are not allowed to be split: (1) the tenants whose sizes are more than S_H (referred to as *large tenants*); (2) the tenants placed on a server that contains a large tenant. Let OPT' denote the optimal solution of this problem. Since any solution to the original problem is also a solution to this problem, we have $cost(OPT') \leq cost(OPT)$. We will prove that $cost(APP) \leq 3 \cdot cost(OPT')$, which implies $cost(APP) \leq 3 \cdot cost(OPT)$.

LEMMA 5.2. *APP uses at most 1.5 times the number of servers as OPT'.*

PROOF. First, suppose there is no tenant whose size is larger than $2S_H$.

In the APP solution, all servers, except at most one, are at least half full, i.e., their tenant sizes are at least S_H. Otherwise, if there are 2 servers that are not half full, the APP algorithm would have combined the tenants on the two servers.

[4]If the size of t_i is greater than $2S_H$, we will always create a new server for t_i.

Suppose OPT' uses x servers, and APP uses y servers. Note that there are at most x tenants whose sizes are larger than S_H, because no two such tenants can be placed on the same server in either OPT' or APP. Since the APP algorithm places tenants in decreasing order of size, in servers $\#x+1$ to $\#y$ in the APP solution, each tenant's size is at most S_H, and thus each of these servers contains at least two tenants. We refer to these tenants as "extra tenants".

If $y > 1.5x$, then $y - x \geq \lfloor 0.5(x+1) \rfloor$, which means that there are at least x extra tenants. None of these x extra tenants can fit into any of the first x servers in the APP solution. This means that the total tenant size must be bigger than $2xS_H$. Then, since in the OPT' solution, no server's tenant size can be bigger than $2S_H$ (otherwise this server should be split), OPT' uses at least $x+1$ servers, which is a contradiction.

Now, suppose that there exist tenants whose sizes are larger than $2S_H$. Suppose there are k such tenants. Since no two of them can be placed together, both OPT' and APP uses k servers to host them. In APP, each of these k servers contains a single tenant. In OPT', there are 2 cases:

-Case 1: some of these k servers also contain other tenants, say server M_i contains another tenant t. This means that OPT' only uses these k servers. Otherwise, if there is another server M_{k+1}, its tenant size must be smaller than $2S_H$, and thus the extra tenant on M_i should be moved to M_{k+1} to balance the load of these two servers. And since the size of the additional tenants on each server cannot exceed S_H (otherwise OPT' would have placed it on a new server), the total size of all extra tenants is at most kS_H. APP will use at most $k/2$ servers for these tenants, because these additional tenants cannot be sequentially split, and the additional tenants on any two OPT' servers can be packed together on the same server by APP. Thus the number of APP servers will not exceed $1.5k$.

-Case 2: none of these k servers contains any other tenant. Suppose OPT' uses another k' servers for all other tenants. From the previous analysis, APP will use at most $1.5k'$ servers for these tenants. Thus APP uses at most $k + 1.5k'$ servers, which is smaller than $1.5(k+k')$.

Therefore, APP uses at most 1.5 times the number of the servers used by OPT'. □

THEOREM 1. *For any arbitrary instance of* TP-UNIFORM*, APP cannot be more than 3 times worse than OPT, i.e.,* $cost(APP) \leq 3 \cdot cost(OPT)$.

PROOF. We use H to denote the expected penalty/cost of a half full server. In OPT', all servers have the same SLA penalty, which is at least H (otherwise, two OPT' servers can be combined).

Consider the following two cases:

-Case 1: no tenant has a size of $2S_H$ or larger. Recall that the APP algorithm no longer places a tenant in a server if the tenant size of the server will exceed $2S_H$. This means that for any APP server, even if we evenly split its tenant size, the expected cost will still increase. Note that if we evenly split the tenant size, we will get two servers; the tenant size in each server is smaller than S_H. Therefore, for each APP server, the expected cost is at most $2H + D$. Suppose OPT' uses x servers, then we have

$$cost(OPT') \geq xH + xD$$

$$cost(APP) \leq 1.5x(2H+D) + 1.5xD \leq 3 \cdot cost(OPT')$$

-Case 2: there exist tenants with size at least $2S_H$. In this case, for both OPT' and APP, each server contains at most one such tenant. Again, we have two cases:

-Case 2.1: some of these k servers also contain other tenants. From the proof of Lemma 5.2, we know that these k servers are all the servers used by OPT', i.e., $k = x$. In this case APP uses at most $1.5x$ servers, and every APP server's cost is no larger than every OPT server's cost. Therefore, we have $cost(APP) \leq 1.5 \cdot cost(OPT')$.

-Case 2.2: none of these k servers contains other tenant. Then for these k servers, the cost of OPT' and APP are the same. For the remaining servers, since none of them has a tenant of size larger than $2S_H$, from Case 1, we know that the cost of APP is at most 3 times the cost of OPT', thus we have $cost(APP) \leq 3 \cdot cost(OPT')$.

Therefore, the cost of APP is at most 3 times the cost of OPT', and thus at most 3 times the cost of OPT. □

5.3 Approximation Algorithm for TP-General

For TP-GENERAL, we have the following observation: *if we do not consider the difference of SLA penalty, the performance can be arbitrarily worse than the optimal solution.*

EXAMPLE 5.2. *Consider two types of tenants. Type 1: small size (e.g., s) and large SLA penalty (e.g., xP); Type 2: large size (e.g., ys) and small SLA penalty (e.g., P). It is possible that one Type 1 tenant can only be placed together with one Type 2 tenant, and one Type 2 tenant can only be placed together with one Type 1 tenant (otherwise the cost is so high that we should split them into multiple servers). Suppose there are n Type 1 tenants and n Type 2 tenants. If we do so, we need n servers. If in the input, each Type 1 tenant is followed by a Type 2 tenant and each Type 2 tenant is followed by a Type 1 tenant, then the greedy approach introduced in Section 5.1 will use n servers. For Algorithm 1, even though it sorts the tenants by size, the problem still remains: suppose we split each Type 2 tenant into multiple small-size tenants; each one has the same size as the Type 1 tenant. Then Algorithm 1 will also use n servers.*

However, it is possible that all Type 1 tenants can be placed in a single server and all Type 2 tenants can be placed in a single server. As long as the size of Type 1 tenants are sufficiently small (specifically, the total size of all Type 1 tenant is much smaller than the size of a single Type 2 tenant), and the SLA penalty of Type 2 tenants are sufficiently small (specifically, the SLA penalty of placing all Type 2 tenants together is much smaller than the SLA penalty of placing one Type 1 tenant and one Type 2 tenant together), the total SLA penalty of the second placement using 2 servers can be arbitrarily smaller than the total SLA penalty of the first placement using n servers. Besides, since n can be arbitrarily large, the server cost of the second placement can also be arbitrarily smaller than the server cost of the first placement. Therefore, processing the tenants either in arbitrary order or in the order of their sizes can lead to arbitrarily bad performance.

Based on this observation, we must process the tenants *in an order that considers their SLAs*. We sort the tenants in the order of their *normalized SLA penalty* as defined below.

DEFINITION 5.2. *The* normalized SLA penalty *of a tenant t is the total expected SLA penalty of t's queries per time unit, if all these queries miss the SLA deadline. The normalized SLA penalty of t can be computed as*

$$NSP(t) = \sum_j (\mu_j \cdot L_j)$$

where μ_j and L_j are the average arrival rate and the per-query SLA penalty of the jth query class of tenant t.

To solve the problem in Example 5.2, we introduce the concept of *sequential split* as defined below.

Algorithm 2: Approximation Algorithm for TP-GENERAL

Input : n tenants, the average query arrival rate, processing time and SLA penalty of each tenant, server cost D
$T = \{t_1, \cdots, t_n\}$ = sorted tenants in the order of $L_i \cdot \lambda_i$ (either increasing or decreasing)
Create server M_1 for t_1
foreach $t_i \in T$ **do**
  M_j = the last created server
  if $additionalPenalty(t_i, M_j) \leq Penalty(t_i) + D$ *AND* $seqSplit(t_i, M_j)$ = *false* **then**
    Place t_i on M_j
  else
    Create a new server M_k
    Place t_i on M_k
  end
end

DEFINITION 5.3. *Given a server M that contains a list of tenants and an algorithm $\mathcal{A}$ used to place these tenants on M, a* sequential split *of M separates the tenants on M into two servers M_1 and M_2, such that all tenants on M_1 were placed by $\mathcal{A}$ on M before all tenants on M_2, and moreover, we are allowed to split one tenant into two parts and place them on different servers.*

Due to a similar conclusion as Lemma 5.1 (which we omit here due to space limit), whether tenants on a server can be sequentially split to make it more profitable can be determined by a binary search.

The algorithm for TP-GENERAL is shown in Algorithm 2. It first sorts the tenants in the order of their normalized SLA penalty (Definition 5.2). We create a new server for the first tenant. For each subsequent tenant t, if the last created server M satisfies: (1) placing t on M is cheaper compared with creating a new server for t; (2) after placing t on M, the tenants on M cannot be sequentially split to make it more profitable, then t is placed on M. Otherwise, it creates a new server for t.

To prove the approximation ratio of Algorithm 2, similar as Section 5.2, we introduce the following problem similar as TP-GENERAL: suppose each tenant can be arbitrarily split, and placed on different servers, except those tenants whose sizes are large enough, such that this tenant itself can be sequentially split to reduce the cost. Let OPT' denote the optimal solution for this problem, and OPT denote the optimal solution of the original problem. Apparently $cost(OPT') \leq cost(OPT)$.

The following lemma is critical in proving the approximation ratio.

LEMMA 5.3. *In the OPT' solution, no two servers M_i and M_j can have the following tenants: there is a tenant on M_i whose normalized SLA penalty is higher than a tenant on M_j, but lower than another tenant on M_j.*

PROOF. Suppose there's a tenant a on M_i, and tenants b and c on M_j, such that $c.NSP > a.NSP > b.NSP$. Let $S(\cdot)$ denote the total tenant size of a server. There are two cases:

-Case 1: $S(M_i) \leq S(M_j)$. Then if $a.size \geq c.size$, we can move c to M_i, and move the same size of a to M_j. At this point, the expected penalty of c decreases or remains the same; the expected penalty of the part of a moved to M_j increases or remains the same. Since $c.NP > a.NP$, the total expected penalty in these two servers decreases or remains the same. Now, since the average expected penalty in M_i increases and the average expected penalty in M_j decreases, it will reduce the cost if we move a small size of tenant from M_i to M_j. Therefore, the original placement is not optimal; it can be improved, which is contradictory. On the other hand, if $a.size < c.size$, then we can move a to M_j, and move the same size of c to M_i. For the same reason, the cost can be reduced, which contradicts with that we started with the optimal solution.

-Case 2: $S(M_i) > S(M_j)$. The proof is similar as Case 1. If $b.size \geq a.size$, we can move a to M_j, and move the same size of b to M_i, and the total expected penalty will decrease. If $b.size < a.size$, then we can move b to M_i, and move the same size of a to M_j, and reduce the cost. □

Lemma 5.3 indicates that OPT' must place the tenant in the order (either increasing or decreasing) of normalized SLA penalty.

THEOREM 2. *For any arbitrary instance of* TP-GENERAL*, APP cannot be more than 4 times worse than OPT, i.e., $cost(APP) \leq 4 \cdot cost(OPT)$.*

PROOF. We first prove that APP uses at most twice the number of servers used by OPT'.

First, we assume that no tenant itself can be sequentially split.

Note that both APP and OPT' places tenants in the order of their normalized SLA penalty. Without loss of generosity, suppose they both use non-decreasing order. Also note that in the APP solution, no two adjacent servers can be merged to reduce the cost; otherwise the APP algorithm would have done so. Therefore, the first OPT' server must contain less tenant size than the first two APP servers combined. For similar reasons, the second OPT' server must contain less tenant size than the third and fourth APP servers combined. In this way, it is easy to see that APP uses at most twice the number of servers used by OPT'.

When there exist tenants that can be sequentially split (suppose there are k such tenants, and thus OPT' uses k servers for them), we have two cases:

-Case 1: some of these k servers also contain other tenants, say M_i contains another tenant t. This means that OPT' only uses these k servers. Otherwise, if there is another server M_{k+1}, its tenants cannot be sequentially split, and thus the extra tenants on M_i should be moved to M_{k+1} to balance the load of these two servers. Besides, if we combine the additional tenants on any two of these k servers, the combined tenants cannot be sequentially split, because otherwise the original extra tenants should have been sequentially split from those servers by OPT'. Thus if we place all these extra tenants using OPT', it must use at most $k/2$ servers, and by the conclusion above, APP will use at most k servers. Therefore, in this case APP uses at most $k + k = 2k$ servers.

-Case 2: none of these k servers contains any other tenant. Suppose OPT' uses another k' servers for all other tenants. From the conclusion above, APP will use at most $2k'$ servers for these tenants. Thus APP uses at most $k + 2k'$ servers, which is smaller than $2(k + k')$.

Now we prove the approximation ratio. To do so, we consider another solution OPT" to the adapted version of the problem. OPT" uses more servers than OPT', but each server contains fewer tenants. Each OPT" server (except the last one) can "just" be combined with the server next to it, i.e., combining any two adjacent OPT" servers (except the last one) has the same cost as not combining them. Obviously, the tenant penalty of OPT" is smaller than that of OPT'. We will prove that the tenant penalty of APP is at most twice that of OPT".

Again, we first assume that there's no tenant whose size is large enough to be sequentially split.

A key observation is that for each APP server, its tenants may come from at most three adjacent OPT" servers. So, for each APP server M, we discuss the following three cases:

-*Case 1:* M's tenants come from a single OPT" server M'. Then the penalty of M must be smaller than the penalty of this OPT" server.

-*Case 2:* M's tenants come from two OPT" servers M_1 and M_2. Then M's penalty is no larger than combining all tenants on M_1 and M_2. We refer to servers that can be "just sequentially split" as "just full" servers. For two "just full" servers, if one has a higher SLA penalty and smaller tenant size, then it also has a lower cost (the detailed deduction is omitted due to space limit). Thus the cost of M will be no larger than twice the SLA penalty of M_2 plus D.

-*Case 3:* M's tenants come from three OPT" servers M_1, M_2 and M_3. Since M has a higher average SLA penalty than combining M_1 and M_2 (suppose we use increasing order of SLA penalty), its tenant size must be smaller than combining M_1 and M_2. Thus the cost of M will be smaller than combining M_1 and M_2, i.e., smaller than twice the SLA penalty of M_2.

As we can see, for each APP server M, there is a distinct OPT" server M', such that $cost(M) \leq 2cost(M') + D$. Thus it is easy to see that if APP uses y servers, then

$$penalty(APP) \leq 2penalty(OPT'') + yD \leq 2penalty(OPT') + yD$$

Suppose OPT' uses x servers and APP uses at most $2x$ servers, thus

$$cost(OPT') \geq penalty(OPT') + xD$$

$$cost(APP) \leq 2penalty(OPT') + 2xD + 2xD \leq 4 \cdot cost(OPT')$$

Now we assume that there exists tenants whose sizes are large enough to be sequentially split. Suppose there are k such tenants, and OPT' uses k servers to host them. We have two cases:

-*Case 1:* some of these k servers also contain other tenants. Then these k servers are all the servers used by OPT', i.e., $k = x$. In this case APP uses at most $2x$ servers, and every APP server's cost is no larger than a distinct OPT' server's cost. Therefore, we have $cost(APP) \leq 2 \cdot cost(OPT')$.

-*Case 2:* none of these k servers contains other tenant. Then for these k servers, the cost of OPT' and APP are the same. For the remaining servers, since none of them has a large tenant, from Case 1, we know that the cost of APP is at most 4 times the cost of OPT', thus we have $cost(APP) \leq 4 \cdot cost(OPT')$.

Therefore, we have proved that $cost(APP) \leq 4 \cdot cost(OPT') \leq 4 \cdot cost(OPT)$. □

5.4 Quality Improvement Using Dynamic Programming

Note that in the approximation algorithm, tenants placed on the servers have a fixed order, i.e., if two tenants t_i and t_j are placed on servers M_x and M_y respectively, and $i < j$, then $x \leq y$. Since the order is fixed, we can use a dynamic programming approach to find the optimal solution wrt the fixed order and the number of servers.

The dynamic programming procedure is shown in Algorithm 3. Its input is the list of tenants sorted by their normalized SLA penalty and the number of servers used by the approximation algorithm (Algorithm 2), denoted by m.

Let $C(i, j)$ be the expected SLA penalty when co-locating tenants t_i through t_j on a single server. We construct and fill in an $n \cdot m$ table, where n is the number of tenants and m is the number of servers. In each cell, we store two information, $MP(i, j)$ and $PrevCut(i, j)$. $MP(i, j)$ is the minimum total expected penalty when placing the first i tenants on j servers. $PrevCut(i, j)$ tells us where the previous cut is, for the setup that realizes the minimum total expected penalty. For example, $PrevCut(100, 10)$=85 means that given one hundred ordered tenants and ten servers, we should cut after the 85th tenant (i.e. put tenants 86 through 100 on the last server).

Algorithm 3: Dynamic Programming for TP-GENERAL

```
Input : n tenants, the average query arrival rate λ_i,
        processing time r_i and SLA penalty L_i of each tenant
        t_i, the number of servers m used by Algorithm 2
Find the value of S_H using binary search
T = {t_1, ··· , t_n} = sorted tenants in the order of L_i · λ_i
(either increasing or decreasing)
for i = 1 to n do
    MP(i, 1) = C(1, i)
end
for j = 2 to m do
    for i = j to n do
        l = arg min_{1≤k≤i−1}(MP(k, j − 1) + C(k + 1, i))
        PrevCut(i, j) = l
        MP(i, j) = MP(l, j − 1) + C(l + 1, i)
    end
end
Cuts(m − 1) = PrevCut(n, m)
for j = m − 2 to 1 do
    Cuts(j) = PrevCut(Cuts(j + 1), j + 1)
end
Output the optimal placement based on MP and Cuts
```

Algorithm 3 is based on the following recurrence relation:

$$MP(i, j) = \begin{cases} C(1, i), & j = 1 \\ \min_k(MP(k-1, j-1) + C(k, i)), & j > 1 \end{cases}$$

Algorithm 3 iteratively fills up the cells in the table. In the end, we compute $MP(n, m)$, and more importantly, we can find all cuts by following $PrevCut(i, j)$ in the backward direction, starting from $PrevCut(n, m)$.

As we can see, Algorithm 2 and Algorithm 3 can be coupled. We first use Algorithm 2 to find the number of servers used by APP, then use Algorithm 3 to find the placement. In this way, we guarantee to output a solution that is no worse than using Algorithm 2 alone.

6. EVALUATION

We compare three tenant placement algorithms presented in this paper in the experiments. The main comparison metric is the *Total Cost*, which is comprised of *Server Cost* plus *SLA Penalty Cost*. The SLA penalty costs are calculated based on the example model we showed in Section 3. However, as discussed, other SLA penalty models could be used for different systems. Therefore the important aspect of the comparison is the relative performance of the algorithms. We also present the results for varying capacity constraints and the scalability of the algorithms.

6.1 Experimental Setup

6.1.1 Comparison Systems

APP. APP is the approximation algorithm (TP-General) presented in Section 5.3 (Algorithm 2). Due to space limit, we do not report the result of uniform query processing time and SLA penalty (Algorithm 1).

Greedy. This is an adaptation of the classic bin packing algorithm (Baseline) as introduced in Section 5.1. It processes the tenants one by one. For each tenant, it either places it on one of the

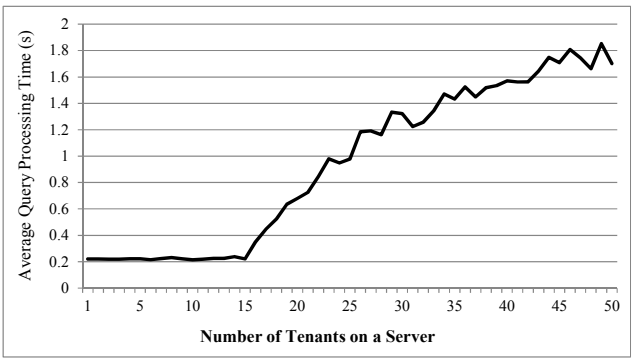


Figure 3: Relationship between Average TPC-W Query Processing Time and Number of Tenants on a Server

existing servers, or create a new server for this tenant, whichever choice minimizes the current expected cost.

DP. This is the dynamic programming approach (Algorithm 3). As we mentioned before, it guarantees to output the optimal solution if the order to place the tenants and the number of servers are fixed, thus it should outperform APP in terms of cost (server cost + SLA penalty). However, due to the higher time complexity of DP, it is expected to be slower.

6.1.2 Environment

The experiments were performed on a cluster of servers, each with eight Intel Xeon E5260 2.4GHz cores, 16GB memory, running CentOS 5.6 and MySQL 5.5.

We generated up to 150 tenants. Each tenant runs the TPC-W workload according to the ordering mix pattern. Regarding the sizes of the tenants, we considered the multitenant database settings presented in Microsoft SQL Azure [1] and [30], where a multitenant server usually hosts a large number of tenants. Accordingly, we set the data size of each tenant in the experiments to be 580MB, and each tenant has a base load of up to 3%. (We also show the effect of tenants that generate larger loads.) We adopt the shared-database multitenancy model, i.e., on each physical server, the tenants use separate tables of the same database. We chose to use MySQL as the DBMS. All tenants on the same server share a connection pool with multiple connections to the database, and we use an FCFS queue to process their queries. The buffer pool size is set as 10GB. A query that finishes before its deadline (set as 10 times its expected query processing time, or 2 seconds, whichever is longer) incurs no penalty, otherwise the corresponding SLA penalty has to be paid.

6.1.3 Query Processing Time Profiling

As explained in Section 3, due to limited buffer pool size, the load of a server is a superlinear combination of tenants' base loads, i.e., it is the sum of the base loads of its tenants, multiplied by a factor $f(T)$. According to [10], the number of I/O operations only depends on the working set size and the update rate. Therefore, we first conduct a profiling stage to compute this factor, where we measure how the processing times of the TPC-W queries increase with the number of tenants on a server.

The result of the profiling stage is shown in Figure 3. When a server contains no more than 15 tenants, the average query processing time of each tenant is stable. This is consistent with the fact that the buffer pool size is 10GB and the working set size of each tenant is roughly 0.6GB. When the number of tenants on a server further increases, I/O starts to increase and the query processing time of each tenant quickly goes up. This information is passed to the tenant placement algorithms in order to accurately estimate the actual load of a server in packing the tenants.

If different tenants have very different update rates, then a more complex profiling stage needs to be performed which measures query execution time as a function of both working set size and update rate.

6.1.4 Workload Generation

As mentioned in Section 3, although the tenants' traffics may exhibit fluctuations, the overall load characteristics of a tenant usually change gradually, rather than abruptly. Thus we divide the timeline into consecutive intervals, which is used by all tenants. The queries from each tenant are randomly assigned a mean arrival rate, variance of arrival rate and SLA penalty. In each interval, each tenant is assigned a query arrival rate, which is randomly generated following normal distribution based on its mean and variance values. Then, queries of each tenant are generated that follows Poisson process. In the experiment we set the length of each interval as 10 minutes.

6.2 Results and Analysis

The default parameter values are as follows: the cost of a server is 100 units per minute[5]; the SLA penalty of the queries of a tenant is randomly generated between 2 and 20 units; the base load of a tenant is randomly generated from 1% to 3%.

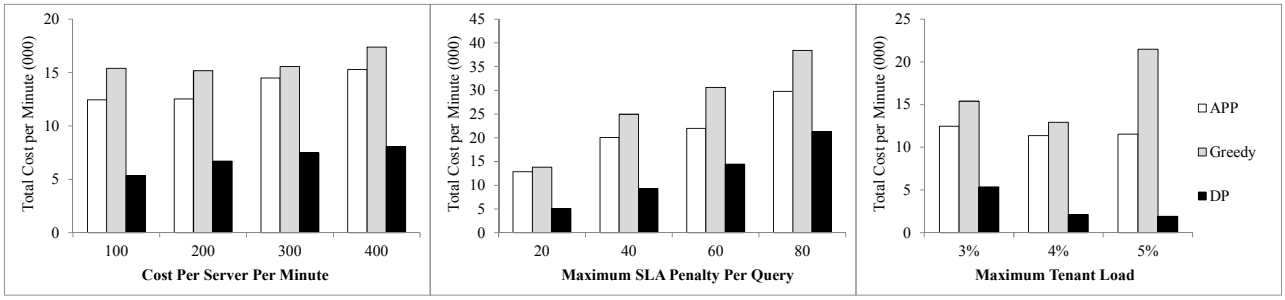
6.2.1 Performance by Server Cost

We vary the server cost from 100 units per minute to 400 units per minute and generate Figure 4(a). All approaches chose to use 9 servers, however, how they place the tenants on these 9 servers are very different. Although Greedy is more "flexible" than APP in that Greedy does not require a fixed order when placing the tenants, APP consistently outperforms Greedy and Greedy incurs up to 30% more cost than APP for two reasons: (1) it does not sort the tenants in the order of normalized SLA penalty, and thus it may suffer from the bad performance as exemplified in Example 5.2; (2) it does not consider sequentially splittable servers. As a result, if a tenant's size is sufficiently small, then even if a server is already very full, it may still be cheaper to place the tenant on this server compared with creating a new server. This will lead to very full servers with high probability of going overload and thus high penalties. On the other hand, as long as placing a tenant on a server makes this server sequentially splittable, APP will place the tenant on a new server, which is a key step in achieving the constant approximation ratio, and also gives a better performance.

We can also observe that although DP uses the same number of servers and the same order to place the tenants as APP, it has a significantly lower cost than APP, which suggests that a better placement makes a big difference for the service provider. DP saves up to 57% cost compared with APP and up to 67% cost compared with APP.

For the first test case (server cost = 100 units per minute), we also monitored the CPU and I/O usage of four servers, which contains 9, 16, 17 and 18 tenants respectively, using `collectl` utility to collect the server statistics every second. The mean and standard deviation of the CPU and I/O statistics are shown in Table 1. As we can see, when there are 9 tenants on a server, there are very few reads. When there are 16, 17 and 18 tenants, the number of

[5]We use the same unit for server cost and SLA penalty. Since the actual unit is irrelevant, it is omitted. The actual server cost in practice usually ranges from less than a dollar to several dollars/hour, e.g., Amazon EC2 standard instances costs $0.08-$0.64/hour.



(a) Max SLA = 20, Max Load = 3% (b) Cost per Server = 100, Max Load = 3% (c) Cost per Server = 100, Max SLA = 20

Figure 4: Effectiveness of Tenant Placement

	CPU		Reads		KBread		Writes		KBwrite	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
9 tenants	2.6%	4.4%	0.9	16.3	13.2	201.5	2.8	6.4	14.1	59.6
16 tenants	8.7%	6.9%	210.0	380.1	13812.3	27843.1	5.7	9.9	74.2	236.2
17 tenants	9.4%	6.6%	379.3	398.7	32535.2	37955.8	4.8	8.1	100.1	257.5
18 tenants	9.1%	6.3%	449.9	381	38435.3	35005.3	4.6	7.5	124.7	246.5

Table 1: Server Statistics

reads explodes, since the buffer pool is not large enough to store the working sets of that many tenants, and a lot of queries need to read data from the disk. On the other hand, the number of writes increases less dramatically, since writes occur when tenants update their tables. For 16, 17 and 18 tenants, the CPU usage does not vary much, indicating that I/O is the bottleneck.

6.2.2 *Performance by SLA Penalty*

We now vary the maximum SLA penalty per query and generate Figure 4(b). When the maximum SLA penalty is 20, the SLA penalty of each tenant is randomly generated from 2 to 20. When the maximum SLA penalty is 40, 60 and 80, we double, triple and quadruple the SLA penalty of each tenant, such that their SLA penalties are in the ranges of [4, 40], [6, 60] and [8, 80], respectively. The relative performances of the three algorithms are similar as Figure 4(a). Note that when the SLA penalties are higher, the cost difference between APP and Greedy tends to be bigger, because Greedy is more reluctant to add new servers, which leads to some tightly packed machines with relatively high chance of violating SLAs and hence high penalties. On the other hand, the APP algorithm proactively places tenants on new machines (using the sequential split method), thus does not have this problem and achieves a lower cost.

6.2.3 *Performance by Tenant Load*

Figure 4(c) is obtained by varying maximum tenant load. When the maximum tenant load is 3%, the base load of each tenant is randomly generated from 1% to 3%. When the maximum tenant load is 4% and 5%, the base load of each tenant is increased by 1/3 and 2/3, so that the maximum load of a tenant is 4% and 5%, respectively. The relative performances of the three algorithms are similar as Figures 4(a) and (b), indicating the consistent good performance of the proposed algorithms, especially DP. Note that when the maximum tenant load is 4%, the total costs of the algorithms are less than that when maximum tenant load is 3%. This is because when the maximum load is 4%, all three algorithms use one more server (10 instead of 9), which lowers the SLA penalty on each server, and thus the total cost is smaller even though they need to pay for an additional server. The cost of Greedy increases when maximum tenant load is 5%, because it still uses 10 servers and thus some servers suffer high SLA penalties. APP and DP maintain roughly the same cost due to better placement strategies.

6.2.4 *Performance with Fixed Capacity*

Sometimes the service provider may not have the number of servers recommended by our tenant placement algorithms. In this case, the service provider has to use fewer servers. To see how this affects the algorithms, we perform this test in which we only give each algorithm a certain number of servers to use, which is less than their recommendations. When all servers have been used, the algorithms are not allowed to add new servers, but can only place the remaining tenants on the existing servers.

In our default test case, all algorithms recommend 9 servers. Thus we test the effect if we allow the algorithms to use a maximum of 9, 8 and 7 servers, which is shown in Figure 5(a). Note that APP is not included in this test, since it is not able to use less servers that it recommends, without violating its placement policies (i.e., no server can be sequentially split). The cost of DP and Greedy increases when the number of servers is reduced, and DP consistently outperforms Greedy by a large margin.

Interestingly, when the number of servers is insufficient, DP and Greedy uses different strategies. Figure 5(b) and (c) shows the number of tenants placed on each server, and the SLA violation penalties of each server when the number of servers is 7. As we can see, DP saturates only one server (server #7) with 45 low value tenants (since it first sorts the tenants based on their normalized SLA penalties), while keeping the loads of the other servers low. This ensures that the other six servers operate normally and incur little SLA penalty. Besides, even if server #7 has a relatively high load, the normalized SLA penalties of the tenants on server #7 are so low that it does not incur too much SLA penalty. This turns out to be a good strategy. On the other hand, Greedy tends to evenly distribute the tenants across servers, and all servers end up having high SLA penalties, and the total penalty is much higher than that of DP.

We also report the average load, per-query SLA penalty and value (i.e., normalized SLA penalty) of tenants on each server for

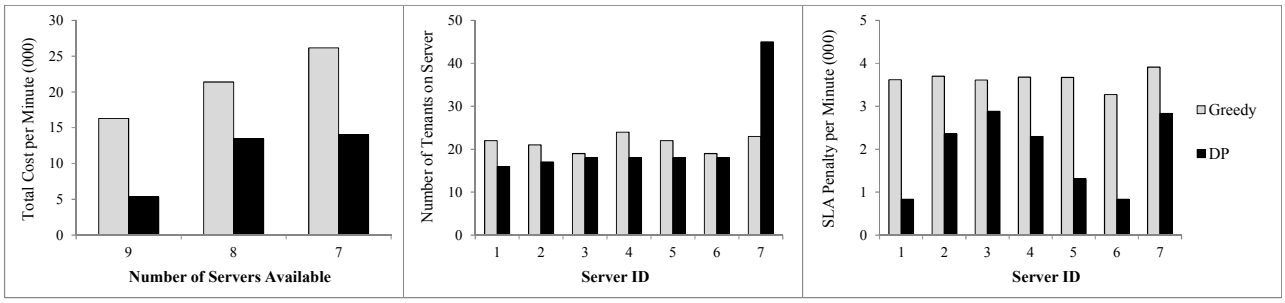


Figure 5: Effectiveness of Tenant Placement with Restricted Number of Servers

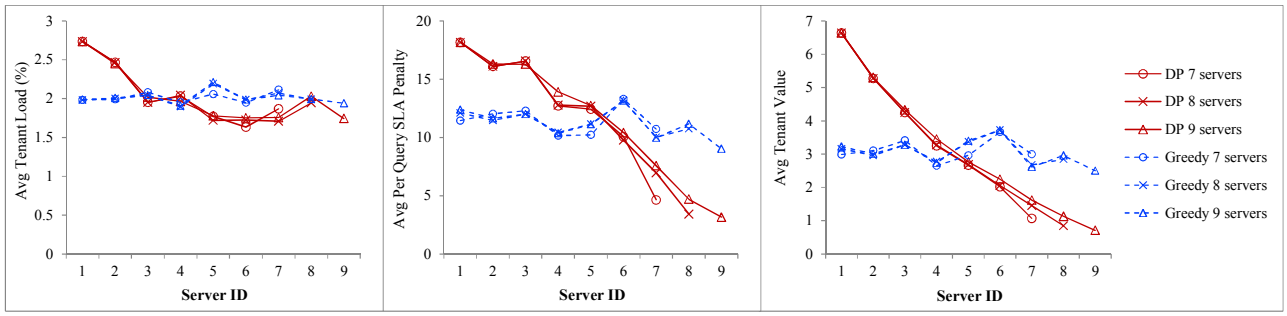


Figure 6: Statistics of Servers with Restricted Number of Servers

all 3 test cases (i.e., number of servers = 9, 8, 7) in Figure 6. Since DP sorts the tenants based on their values, the average value decreases with server ID. Since the value is proportional to the product of load and per-query SLA penalty, the average load and per-query SLA penalty also tend to decrease with server ID, although not necessarily. On the other hand, the average load, per-query SLA penalty and values of tenants are roughly stable for Greedy.

6.2.5 *Scalability*

To test the scalability of the algorithms we increase the number of tenants from 100 to 800. Figure 7 shows the time each algorithm takes to place certain number of tenants. Each point is the average time of 10 random test cases. The running times of APP and Greedy are insignificant, and are below 0.2 second even for 800 tenants. On the other hand, DP takes much longer time as its time complexity is $O(n^2m)$ (as to be shown in Section 7), where n is the number of tenants and m is the number of servers it uses, which is essentially $O(n^3)$. But since tenant placement algorithms are not interactive algorithms and their running times are not critical, DP's efficiency should generally be acceptable.

6.2.6 *Summary*

DP is by far the most effective algorithm of the three. It generally saves more than 50% cost compared with APP and Greedy. APP is also more effective than Greedy and generally saves 10% to 20% of the cost. The fact that DP always uses the same number of servers as well as the same placement order as APP, yet outperforms APP by a large margin, shows that even a small alteration of tenant placement may affect the cost significantly. It is interesting to study whether the placement of DP can be further improved on a consistent basis, which is a future research direction.

APP and Greedy are highly efficient, whereas DP is relatively slower with $O(n^2m)$ complexity where n and m are the number of tenants and servers, respectively. However, generally the tenant

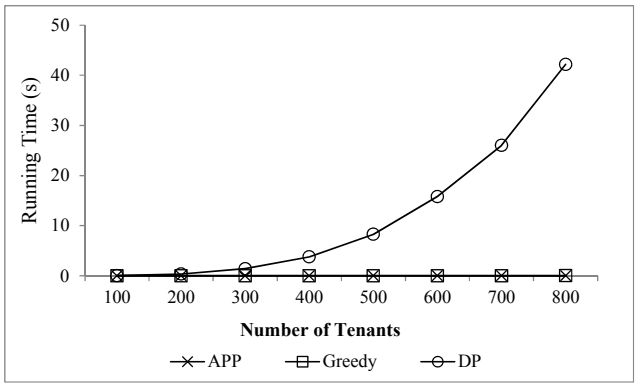


Figure 7: Scalability of Tenant Placement

placement is not a time-sensitive task, unless the scale is very large. Even with extremely large number of tenants, it is possible to cluster the tenants and execute the algorithm on the clusters. Hence, such a complexity should generally be acceptable.

7. DISCUSSION

7.1 Complexity Analysis

When all tenants have the same or similar SLA, and issue the same or similar queries, Algorithm 1 has the best performance, with a time complexity of $O(nm)$ where n is the number of tenants and m is the number of servers it uses. To see this, note that for each tenant, it takes $O(m)$ time to determine which server to place it. Since m is linear to n, the complexity is essentially $O(n^2)$.

Similarly, the placement phase of Algorithm 2 takes $O(nm)$, or $O(n^2)$ time. Since it needs to sort the tenants, the total time complexity is $O(n\log n + nm)$. It should be used when the tenants

have a large variance on query processing time and/or SLA penalty.

Algorithm 3 guarantees to output a placement no worse than that output by Algorithm 2, since it computes the optimal placement given a fixed order of tenants and a fixed number of servers. However, the time complexity of Algorithm 3 is $O(n^2m)$, since it needs to fill a table that has $O(nm)$ cells, and it takes $O(n)$ time to fill each cell. This complexity is essentially $O(n^3)$.

7.2 Online Placement

Tenants may arrive and depart over time. Both Algorithm 1 and Algorithm 2 can be adapted to handle incremental arrival of tenants. For algorithm 1, we have the following lemma.

LEMMA 7.1. *If we do not sort the tenant by size in Algorithm 1 and only place each tenant either on the last created server or a new server, the approximation ratio will be 4.*

The idea of the proof is that if the tenants are processed in random order, the number of servers used by APP will be at most twice, rather than 1.5 times, the number of servers used by OPT'. The proof is omitted due to space constraint.

Departure of tenants will involve tenant migration. To determine which tenant to migrate, a variety of factors should be considered, including servers' loads, tenants' data sizes, query rates, query frequencies, as well as SLA violations due to service interruption. This is a complicated problem and we do not intend to study it in this paper, as we focus on achieving a good initial placement. Nonetheless, it will be an interesting future work to explore.

8. CONCLUSIONS AND FUTURE WORK

Multitenancy supports a large number of tenants economically by accommodating multiple tenants within a single server. This paper studies how to find a good tenant placement from the perspective of service provider's profit, which can be an integral part of a service provider's strategies. We formalize the problem of tenant placement for cost minimization, which is strongly NP-hard. We then provide two approximation algorithms, one for a special case and one for the general case, with approximation ratios 3 and 4, respectively. We further provide a DP procedure that can be used to couple the approximation algorithm for better performance. DP is experimentally shown to consistently outperform the baseline and the approximation algorithms by a large margin, indicating its excellent effectiveness. While this paper focuses on the deployment of multiple tenants at once, an important future work is to study online placement with dynamic arrivals and departures of tenants, in more details.

9. REFERENCES

[1] Inside SQL Azure. http://social.technet.microsoft.com/wiki/contents/articles/1695.inside-sql-azure.aspx.

[2] J. Allspaw. *The Art of Capacity Planning: Scaling Web Resources*. O'Reilly Media, 2008.

[3] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *SIGMOD Conference*, pages 1195–1206, 2008.

[4] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. A Comparison of Flexible Schemas for Software as a Service. In *SIGMOD Conference*, pages 881–888, 2009.

[5] S. K. Barker, Y. Chi, H. J. Moon, H. Hacıgümüş, and P. J. Shenoy. "Cut Me Some Slack": Latency-Aware Live Migration for Databases. In *EDBT*, pages 432–443, 2012.

[6] N. Bobroff, A. Kochut, and K. A. Beaty. Dynamic Placement of Virtual Servers for Managing SLA Violations. In *Integrated Network Management*, pages 119–128, 2007.

[7] B. Cahoon, K. S. McKinley, and Z. Lu. Evaluating the Performance of Distributed Architectures for Information Retrieval Using a Variety of Workloads. *ACM Trans. Inf. Syst.*, 18(1):1–43, 2000.

[8] Y. Chi, H. J. Moon, and H. Hacıgümüş. iCBS: Incremental Cost-based Scheduling under Piecewise Linear SLAs. *PVLDB*, 4(9):563–574, 2011.

[9] Y. Chi, H. J. Moon, H. Hacıgümüş, and J. Tatemura. SLA-Tree: A Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing. In *EDBT*, pages 129–140, 2011.

[10] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD Conference*, pages 313–324, 2011.

[11] S. Das, S. Nishimura, D. Agrawal, and A. E. Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *PVLDB*, 4(8):494–505, 2011.

[12] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD Conference*, pages 301–312, 2011.

[13] C. Fehling, F. Leymann, and R. Mietzner. A Framework for Optimized Distribution of Tenants in Cloud Applications. In *IEEE CLOUD*, pages 252–259, 2010.

[14] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive Quality of Service Management for Enterprise Services. *TWEB*, 2(1), 2008.

[15] D. Gmach, J. Rolia, and L. Cherkasova. Satisfying Service Level Objectices in a Self-Managing Resource Pool. In *SASO*, pages 243–253, 2009.

[16] Z. Gong and X. Gu. PAC: Pattern-driven Application Consolidation for Efficient Cloud Computing. In *MASCOTS*, pages 24–33, 2010.

[17] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *IEEE Real-Time Systems Symposium*, pages 232–243, 1991.

[18] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[19] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting Database Applications as a Service. In *ICDE*, pages 832–843, 2009.

[20] D. Jacobs and S. Aulbach. Ruminations on Multi-Tenant Databases. In *BTW*, pages 514–521, 2007.

[21] T. Kwok and A. Mohindra. Resource Calculations with Constraints, and Placement of Tenants and Instances for Multi-tenant SaaS Applications. In *ICSOC*, pages 633–648, 2008.

[22] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards Multi-Tenant Performance SLOs. In *ICDE*, pages 702–713, 2012.

[23] Z. Liu, M. S. Squillante, and J. L. Wolf. On Maximizing Service-Level-Agreement Profits. In *ACM Conference on Electronic Commerce*, pages 213–223, 2001.

[24] Z. Lu and K. S. McKinley. Partial Collection Replication for Information Retrieval. *Inf. Retr.*, 6(2):159–198, 2003.

[25] Q. Mei, H. Fang, and C. Zhai. A Study of Poisson Query Generation Model for Information Retrieval. In *SIGIR*, pages 319–326, 2007.

[26] K. Ramamritham, S. H. Son, and L. C. DiPippo. Real-Time Databases and Data Services. *Real-Time Systems*, 28(2-3):179–215, 2004.

[27] 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*, pages 1264–1275, 2011.

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

[29] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacıgümüş. ActiveSLA: A Profit-Oriented Admission Control Framework for Database-as-a-Service Providers. In *SOCC*, pages 15:1–15:14, 2011.

[30] F. Yang, J. Shanmugasundaram, and R. Yerneni. A Scalable Data Platform for a Large Number of Small Applications. In *CIDR*, 2009.

[31] L. Zhang and D. Ardagna. SLA Based Profit Optimization in Autonomic Computing Systems. In *ICSOC*, pages 173–182, 2004.

[32] Y. Zhang, Z. H. Wang, B. Gao, C. Guo, W. Sun, and X. Li. An Effective Heuristic for On-line Tenant Placement Problem in SaaS. In *ICWS*, pages 425–432, 2010.