Fault Tolerant Multitenant Database Server Consolidation

by

Joseph Mate

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation, these include:

- Dr. Khuzaima Daudjee
- Dr. Shahin Kamali
- Fiodar Kazhamiaka

Abstract

Server consolidation is important in situations where a sequence of database tenants need to be allocated (hosted) dynamically on a minimum number of cloud server machines. Given a tenant's load defined by the amount of resources that the tenant requires and a service-level-agreement (SLA) between the tenant customer and the cloud service provider, resource cost savings can be achieved by consolidating multiple database tenants on server machines. Additionally, in realistic settings, server machines might fail causing their tenants to become unavailable. To address this, service providers place multiple replicas of each tenant on different servers and reserve extra capacity to ensure that tenant failover will not result in overload on any remaining server. The focus of this thesis is on providing effective strategies for placing tenants on server machines so that the SLA requirements are met in the presence of failure of one or more servers. We propose the Cube-Fit (CUBEFIT) algorithm for multitenant database server consolidation that saves resource costs by utilizing fewer servers than existing approaches for analytical workloads. Additionally, unlike existing consolidation algorithms, CUBEFIT can tolerate multiple server failures while ensuring that no server becomes overloaded. We provide extensive theoretical analysis and experimental evaluation of CUBEFIT. We show that compared to existing algorithms, the average case and worst case behavior of CUBEFIT is superior and that CUBEFIT produces near-optimal tenant allocation when the number of tenants is large. Through evaluation and deployment on a cluster of up to 73 machines as well as through simulation studies, we experimentally demonstrate the efficacy of CUBEFIT in practical settings.

Acknowledgements

I would like to thank Professor Wojciech Golab and Professor Tamer Özsu for improving the content and presentation of my thesis. Secondly, I would like to thank Dr. Shahin Kamali for lending his expertise on the theoretical aspect of the problem. Third, thank you Fiodar Kazhamiaka for your help with CPLEX. Most of all, I must thank Professor Khuzaima Daudjee for guiding me through all the obstacles in creating this thesis and for the extra effort he took to take my writing to a whole new level.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing has transformed the information technology sector by providing software-as-a-service (SaaS) and infrastructure-as-a-service (IaaS) on demand. Cloud service providers, such as Amazon Web Services [1], host client applications and their data on their cloud servers. This relieves customers from technical tasks such as system operation, maintenance and provisioning of hardware resources. In a typical SaaS system, performance service level agreements (SLAs), agreed between clients and the service provider, define the minimum performance requirement for software. The objective of a service provider is to meet the SLA requirement and, at the same time, minimize the operational cost involved in providing such service. There is generally a trade-off between performance as perceived by customers, and the operational costs associated with using resources.

Cloud providers commonly consolidate client applications, called *tenants*, on shared computing resources to improve utilization and, as a result, reduce operating and maintenance costs. A service provider should have effective strategies for assigning or allocating tenants to reduce the number of servers (machines) that host tenants. This is critical for avoiding *server sprawl* in which there are numerous under-utilized active servers which consume more resources than required by tenants. Preventing server sprawl is particularly important for green computing and saving on the energy-related costs which account for 70 to 80 percent of a data center's ongoing operational costs [9].

To meet SLA requirements, server consolidation should be performed in a way such that servers are not *overloaded*. Data centers usually have a large number of machines with homogeneous server resources, providing significant resource capacity [24]. Similarly, each tenant has a *load* defined as the minimum amount of server compute resources required by the tenant to meet its SLA. If a server is overloaded, i.e., the total tenant load that it hosts exceed its capacity then

the SLA requirements will not be satisfied.

In an ideal scenario, a cloud service provider has access to all tenants before assigning any of them to servers. This can provide efficient tenant placement while meeting the SLA requirements. However, in practice, tenants appear dynamically, i.e. in an online manner, and each tenant needs to be assigned to a server without any knowledge about forthcoming tenants. Another challenge is when one or more server machines hosting tenants fail. As a result, tenants can suffer from performance degradation or loss of availability. To address this issue, tenants should be replicated on more than one server so that when a server fails, the load of a replica hosted on the failed server can be distributed among other servers that host replica(s) of the same tenant (until a new server takes over for the failed server or the failed server is recovered). The SLA requirements should be met in case of a server's failure, i.e, the extra load redirected to other servers (as a result of the server's failure) should not result in overloaded servers. To meet this requirement, service providers need to reserve extra capacity on each machine in anticipation of server failure.

Recently, cloud hosting of analytical workloads experienced explosive growth. Redshift, Amazon's data warehousing solution is their largest growing web service [16]. SAP HANA, a platform focusing on in-memory analytics has become SAP's primary focus [29]. Many leading cloud service providers like Micrsoft and Oracle, are starting to offer platforms for analytics [3, 2]. Application of these platforms range from protecting customers by detecting anomalous fraudulent transactions to determining when a part of a tractor, jet engine, or production line machinery might fail [20, 29].

In this thesis, we consider the problem of server consolidation for multitenant analytical workloads for cloud service providers, described by the following requirements:

- Each tenant has a load, defined by its compute resource needs. The load of each tenant is defined with respect to the SLA requirements, e.g., a database with higher query load has higher server load.
- The cloud service provider should assign tenants to servers such that the SLA requirements will be met. This implies that the total load of tenants assigned to each server should not exceed the load capacity of servers.
- Tenants appear sequentially in an online manner. This implies that upon arrival of a tenant, the service provider needs to assign it to servers without knowledge of forthcoming tenants. Assignment of tenants is permanent, i.e., the service provider cannot change its previous decision upon arrival of a new tenant. This is consistent with most real settings in which moving tenants across servers dynamically (on-the-fly) is often impractical.

- To provide a fault-tolerant solution, each tenant is replicated on two or more servers. In case of a server's failure, the load of each tenant hosted on the server is redirected to other servers which host the tenant. The SLA requirements should be met in case of a server's failure, i.e., the extra load redirected to a server should not cause it to be overloaded. In anticipation of this, the service provider should reserve a part of the capacity of each server for potential redirected load from a failed server.

- The objective of a service provider is to minimize cost by reducing the number of servers which host tenants. This way, the service provider reduces the cost involved in purchasing new machines and also avoids server sprawl which is essential for green computing and operational costs savings in terms of, for example, electricity.

The rest of this thesis is organized as follows. The rest of this chapter formalizes the server consolidation problem and reviews algorithms and concepts related to bin packing. Chapter surveys existing work related to database server consolidation. Chapter 3 presents the CUBEFIT algorithm and proves its correctness as well as provides a theoretical analysis of the algorithm. Chapter 4 describes the model of the system that CUBEFIT is implemented and evaluated under. Chapter 5 presents the performance evaluation of CUBEFIT before Chapter 6 concludes the thesis and outlines future work.

1.1 Background and Contributions

Most existing research [27, 28, 14] consider server consolidation in the offline setting where all tenants are available before the consolidation starts. Moreover, these approaches do not provide fault-tolerant solutions. A practical model for server consolidation was introduced by Schaffner et al. [24]. They introduced online algorithms, in particular the Robust First Fit Interleaving (RFI) algorithm, which provides a fault-tolerant solution with the objective of reducing the number of active servers while meeting the SLA requirements. Unfortunately, these algorithms are well-defined only when there are two replicas per tenant, i.e., the resulting packings are not tolerant against failure of more than one server. Even in the case of two replicas per tenant, there is a large room for improvement with respect to the number of hosting servers. A theoretical analysis of the same model under the framework of *competitive ratio* was performed in [11]. The competitive ratio of an online algorithm is the maximum ratio between the cost of an online algorithm and that of an optimal offline algorithm OPT [26]. In [11], it is proved that the heuristics from [24] do not have good competitive ratios. The same paper introduced another algorithm, Horizontal Harmonic (HH), which has an improved competitive ratio. Unfortunately,

competitive ratio is a worst-case measure which does not capture the average-case performance of algorithms. It is well-known that packing algorithms that optimize the worst-case performance, may not perform well on average [6, 7], e.g., all Harmonic-based bin packing algorithms have better competitive ratio than Best Fit, however, they perform much worse on average (see Section 1.3 for a review).

In this thesis, we study a general model for server consolidation under the SaaS and IaaS paradigms for analytical workloads. We introduce an online algorithm that achieves solutions which optimize the number of servers used and, at the same time, can tolerate failure of any given number of servers. We define *robustness* of a tenant placement solution as the size of the smallest set of servers whose simultaneous failure results in an interruption of service[1]. There is generally a trade-off between the quality of solutions, in terms of the number of used servers, and the robustness in terms of maximum tolerance for server failures. We present the CUBEFIT algorithm, which creates $\gamma \geq 2$ replicas per tenant and has robustness γ, i.e., is tolerant against failure of any set of $\gamma - 1$ servers. The algorithm is based on novel ideas which combine classifying tenants by their sizes, placing tenants of the same class into γ-dimensional "cubes", and consolidating smaller tenants into cubes formed by larger tenants which still 'fit' them. Unlike prior related work, we show that our algorithm is well-defined for any value of γ. The value of gamma is defined by the cloud service provider so in practice, this value is usually a small integer not more than 3 [24, 11], thus we target our evaluation for $\gamma = 2$ and $\gamma = 3$.

We study CUBEFIT under both theoretical and practical settings. Using competitive ratio, we prove that in the worst-case CUBEFIT is as good as the best existing algorithm. For typical settings, we use the *average-case* ratio to prove that the algorithm has a significant advantage over existing algorithms. The average-case performance ratio of an online algorithm is the expected ratio between the cost of the algorithm to that of OPT when item sizes follow a probability distribution. We provide upper bounds for the average-case ratio of CUBEFIT under uniform and Zipfian distributions. Our results indicate that the average-case ratio of CUBEFIT is much better than its competitive ratio. This implies a significant advantage for CUBEFIT when compared to counterparts introduced in [24, 11].

We implement our CUBEFIT algorithm and evaluate its effectiveness for multitenant server consolidation on a cluster of up to 73 server machines. Unlike related work (Section 2) that test on only a handful of machines or report results of only simulation studies, we deploy, run and test CUBEFIT on a large cluster of machines as well as conduct extensive simulation studies to analyse the behaviour of the algorithm. We provide results of extensive experiments run on a fleet of up to 73 machines running the CUBEFIT algorithm and compare with the RFI algorithm of Schaffner et. al. [24]. Moreover, we compare the CUBEFIT algorithm with an optimal

[1]Where appropriate, we use robustness and k fault tolerance synonymously.

algorithm implemented as a mixed integer program (IP). Our results indicate that the number of servers used by CUBEFIT is comparable to the optimal solution, while we observe significant performance and practical issues that make the IP algorithm impractical in online settings.

1.2 Problem

In this section, we formally define the robust tenant placement problem. Our formulation is inspired by studies of a restricted version of the same problem [24, 11].

We consider an online (dynamic) setting in which tenants appear one by one. Tenants have many characteristics, but the one that is important for server consolidation is the load of a tenant. Thus, we distinguish each tenant by its load, which we normalize to be in the range (0,1] and each server has a capacity of 1. Upon arrival of a tenant of load x, γ replicas of the tenant are created, where γ is a parameter of the problem and typically $\gamma \in \{2, 3\}$. The load of a tenant is distributed evenly among the replicas, i.e., each replica has a load x/γ. We call replicas associated with the same tenant *partner* replicas. A consolidation algorithm needs to *place* these replicas on γ different servers. Each replica might be placed on an existing server or the algorithm might *open* (allocate) a new server for it. We assume that servers are homogeneous (uniform) and have unit capacity. To meet SLA requirements, the total load of replicas on each server should not be more than 1, the unit capacity of the server.

When a server fails, the load associated with each replica hosted by the server is evenly distributed among servers that host its partner replicas. The resulting extra load should not exceed the unit capacity of these servers. For example, consider $\gamma = 3$, and assume a tenant X has three replicas x_1, x_2, and x_3 which are respectively hosted by servers S_1, S_2, and S_3. In case of S_1's failure, the load of x_1 is equally distributed between S_2 and S_3. In case of the simultaneous failure of S_1 and S_2, the load of x_1 and x_2 is redirected to S_3. The system needs to be tolerant against failure of at most $\gamma - 1$ servers. This implies that S_3 should have a reserved capacity at least equal to the total load of x_1 and x_2. To be more precise, without loss of generality, assume $|S_i|$ indicates the total load of replicas on server S_i. Moreover, assume $|S_i \cap S_j|$ denotes the total load of replicas hosted on server S_i which have a partner replica on S_j. To have a fault tolerant solution, for any server S_i, and for any set S^* formed by at most $\gamma - 1$ servers other than S_i, we should have $|S_i| + \sum_{S_j \in S^*} |S_i \cap S_j| \leq 1$, i.e., the load directed to S_i as a result of simultaneous failure of servers in S^* should not be more than its reserved capacity. In summary, we define the problem as follows. Figure 1.1 provides an illustration.

Definition 1 *In the* online server consolidation problem *with replication factor γ, the input is an online sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of tenants (also referred to as items) where $a_t \in (0, 1]$ indicates*

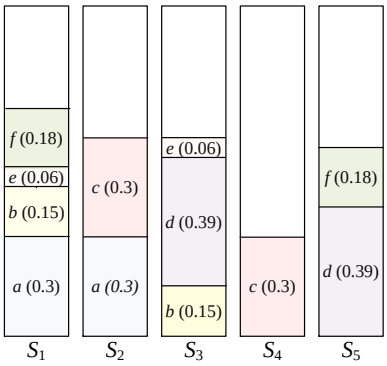
the load of the tth tenant $(1 \leq t \leq n)$. Upon arrival of the tth tenant, γ replicas of equal load a_t/γ should be placed onto γ different servers (also called bins). Servers have unit capacity, and the total load of replicas on each server should not exceed 1. In a valid packing of a sequence on n tenants, for each server S_i and for each set S^* of at most $\gamma - 1$ servers where $S_i \notin S^*$, we have $|S_i| + \sum_{S_j \in S^*} |S_i \cap S_j| \leq 1$. The objective is to form a valid packing of n tenants in which the number of hosting servers is minimized.

1.3 Online Bin Packing

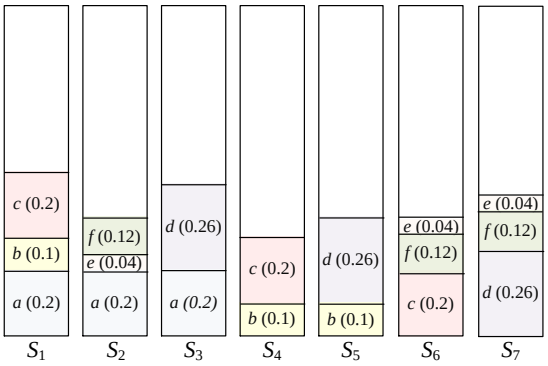
The server consolidation problem, as defined above, is closely related to the online bin packing problem. In this section, we review algorithms and concepts related to bin packing that are used later in the thesis. In the bin packing problem, the goal is to place a set of *items* with different *sizes* into a minimum number of *bins* of unit capacity. In the online setting, items appear one by one, and an algorithm has to place each item without knowledge of forthcoming items. In the context of server consolidation, each bin represents a server and each item represents a tenant. Note that bins have unit capacity which translates to the unit uniform capacity of servers. On the other hand, items have various sizes which translates to different loads for tenants. Minimizing the number of used bins is analogous to reducing the number of servers hosting tenants. In fact, server consolidation in the IaaS model is the same as online bin packing, except for the requirement of having fault-tolerant solutions which is not present in online bin packing.

A simple online bin packing algorithm is Next Fit, which maintains one open bin and places each item into the open bin; if there is not enough space the bin is closed and a new bin is opened. The Best Fit algorithm orders all used bins in non-increasing order of their *level*, defined as the total sum of items in each bin, and places an incoming item into the first bin that has enough space. Harmonic algorithm classifies items based on their sizes and treats items in each class separately using the Next Fit strategy.

Competitive analysis is used for studying worst-case behaviour of online bin packing algorithms. Similar to most related results, by competitive ratio, we mean *asymptotic* competitive ratio, in which only sequences are considered where the cost of OPT is sufficiently large. The competitive ratio of Next Fit and Best Fit are respectively 2 and 1.7 [15], while that of Harmonic converges to approximately 1.69 for large values of K [18]. Hence, in the worst-case Harmonic is slightly better than the other two algorithms. However, on *average*, Best Fit performs far better. When item sizes are generated from a uniform distribution, the average ratio of Best Fit is 1 while that of Next Fit and Harmonic are respectively 1.33 and 1.289 [6]. This shows that algorithms



(a) A packing with replication factor $\gamma = 2$



(b) A packing with replication factor $\gamma = 3$

Figure 1.1: Two solutions associated with a sequence of tenants $\sigma = \langle a = 0.6, b = 0.3, c = 0.6, d = 0.78, e = 0.12, f = 0.36 \rangle$. In the solution of (a), each tenant is replicated on two machines; hence, the load of each replica is half of the tenant's. In case of a single server's failure, the service continues without interruption. For example, if S_1 fails, the load of replica a redirects to S_2 ; this gives a total load of $0.6 + 0.3 < 1$ for S_2. Similarly, loads of e and f redirects to S_3 and load of f redirects to S_5. In the solution of (b), each tenant is replicated on three machines. In case of simultaneous failure of two servers, the system continues uninterrupted. For example, if S_1 and S_2 fail, the total load of replicas of a hosted on them redirects to S_3, resulting in a total load of $0.46 + 2 \times 0.2 < 1$.

that perform well with respect to competitive ratio do not necessarily have a better average-case performance. The same issue exists for server consolidation. Consider the Horizontal Harmonic (HH) algorithm of [11], which uses ideas similar to Harmonic bin packing algorithm. This al-

gorithm has competitive ratio of 1.59, which is the best among the existing algorithms for server consolidation. However, it does not perform well on average; for example, when replica sizes follow a uniform distribution , half of HH bins are expected to include only one replica which results in a big waste of resources. Clearly, it is desirable to achieve algorithms which have good performance in both worst-case and average-case scenarios.

Chapter 2

Background and Related Work

[24] and [11] are the only existing works that considered the problem of fault tolerant database server consolidation. However, neither of these proposals protect servers from multiple server failures. [24] proposes the RFI algorithm and [11] presents the HH algorithm but the latter does not evaluate the performance of HH nor does it experimentally compare with other algorithms. We compare against these two algorithms to show that CUBEFIT is superior both performance wise and importantly, CUBEFIT protects tenants against the failure of multiple servers. Moreover, the work from [24] reports only simulation results while we demonstrate the efficacy of CUBEFIT by implementing and evaluating it on a real system, through simulations, comparison with optimal and detailed theoretical analyses.

The remaining related work does not protect servers from becoming overloaded due to failure of other servers hosting tenant replicas. For example, [13] considers load sharing between servers but does not deal with fault tolerant overload management. Their proposal uses tenant classes that can deal with tenants that do not belong to a given class and can potentially consume different amounts of resources than expected. They study their proposals using only simulations and do not compare against other algorithms while we compare against the RFI algorithm proposed in [24] through extensive experiments on a large cluster.

Kairos places tenants by analyzing their usage of CPU, IO and various other system resources [8]. It places tenants on a minimal number of servers while not exceeding the capacities of the servers by using an optimization algorithm similar to CPLEX. Their experiments were run on just two servers and nine tenants. Schaffner et. al. [24] demonstrated that the algorithm from Kairos did not scale for a large number of tenants. Moreover, unlike our work, Kairos does not provide fault tolerant server consolidation. PMAX takes a different approach to the problem by considering the cost of SLO violations as well as the cost of servers. They use a modified version

of the best fit bin packing algorithm [19] to approximate a solution. This allows less costly solutions by potentially using less servers, but such solutions are also less effective in preventing server overload and can penalize tenants that are suffering from it. In contrast, CUBEFIT ensures there are no load violations, thereby avoiding performance degradation.

Lang et al. proposed a solution that has three tenant classes: 1 transaction per sec (tps), 10 tps, and 100 tps [17]. They experimentally determine the mixture of tenant classes that meet their (tps) SLO on various machine types. They compared a cheap disk machine to an expensive SSD machine. The algorithm searches for the mixture of classes on the machine type that gives the lowest cost but they do not provide for fault tolerance. In contrast, in addition to providing fault tolerance, our algorithm does not limit the tenants to classes and allows for more opportunity for packing as tenants in a particular class may not completely use their resources.

SWAT performs load balancing and load leveling by swapping the servers that maintain primary and secondary replicas [21] but does not consider the efficient online packing of tenants onto servers. Their algorithm can react to server failures by load balancing. However, unlike CUBEFIT, a server will be in overload until the algorithm can determine, and swap, the primaries in question.

Delphi-Pythia uses machine learning to determine placement of tenants on a server [12]. Building upon this machine learning algorithm, they use an AI search algorithm to move tenants away from bad placements but unlike CUBEFIT does not consider fault tolerant server consolidation.

2.1 Goals and Approaches

Multitenant database providers have various goals and methods for achieving their goals. We summarize these goals and their methods in Table 2.1. In CUBEFIT, a key goal is to strive to minimize the number of servers needed. This was also the goal in HH, RFI [24], and Kairos [8]. However, in other systems Lang et al. [17] and Floratou et al. [13] tried to minimize the dollar cost by having different server types. There are even models like PMAX [19], where they allow poor performance for their customers, but refund them for the inconvenience. There are also orthogonal objectives where the authors of SWAT [21] and Delphi-Pythia [12] tried to detect and remedy over packed situations.

The second goal to consider is server overload. CUBEFIT, HH, and RFI prevent overload situations, even in the event of failures by intelligently allocating tenants to servers and leaving sufficient reserved space. Kairos and Lang et al. prevent overload situations, ignoring the overload situation as a result of server failure. Floratou et al. was only able to mitigate overload situations in the event of failures by balancing the shared load of the servers. In PMAX, they allow overload situations, but refund the affected customers to save even more money at the cost of customer confidence. Finally, SWAT and Delphi-Pythia do not prevent overload situations, instead they try to detect and fix them.

These algorithms have a large array of methods for achieving their goals. Currently, the single load value method is best suited for this problem. Doing vector packing using multiple values can result in better packings than using only a single value. However, the current methods introduced by [8] are prohibitively expensive as shown in [24]. Additionally bucketing tenants into classes like in [17, 13, 12] is too coarse as it sacrifices better packings. Consider tenants bucketed into 1 and 10 transaction per second classes. A tenant that uses 2 transactions per second would have to be placed in the 10 TPS class, wasting a large chunk of capacity. As a result, with CUBEFIT we use a single load value for allocating tenants which was also used in [10, 24, 21, 19].

Paper	Objective	Tenant Characterization	Prevent Overload Due To Failure
CUBEFIT	Servers	Single load value	Yes
HH [10]	Servers	Single load value	Yes
RFI [24]	Servers	Single load value	Yes
Kairos [8]	Servers	Vector Packing CPU, Memory, IO	No
Lang et. al. [17]	Dollars	TPS classes	No
Floratou et. al. [13]	Dollars	TPS classes	No but, mitigates by leveling num. of shared tenants
SWAT [21]	Remedy Overloaded Servers	Single Load Value	No but, fixes by swapping primary and secondary when detected
PMAX [19]	Cost of Servers + SLA Penalties	Single Load Value	No but, pays tenants money for overloads
Delphi-Pythia [12]	Remedy Overloaded Servers	machine learned classes	No but, moves tenants on detection

Table 2.1: Goals and Approaches of Multitenant Database Server Consolidation

Paper	Replicas (Including Primary)	Fault Tolerance Overload	Fail Overload Strat.
CUBEFIT	γ	$\gamma - 1$	Prevent through allocation
HH [10]	2	1	Prevent through allocation
RFI [24]	any	1	Prevent through allocation
Kairos [8]	γ (untested)	0	N/A
Lang et. al. [17]	1	0	N/A
Floratou et. al. [13]	2	0	balances shared tenant load
SWAT [21]	2 or 3	0	on detection balances load
PMAX [19]	1	0	refund tenants on overloaded server
Delphi-Pythia [12]	2	0	on detection balances load

Table 2.2: Failure Models of Multitenant Databases

2.2 Failure Models

The various failure models of different multitenant database systems are summarized in Table 2.2. CUBEFIT can support any number of replicas, but it must be set before packing tenants. RFI can support any number of replicas at anytime; however, it is only able to protect against overload due to a single failure. If CUBEFIT has γ replicas, it can protect against overload for up to $\gamma - 1$ failures. Other than CUBEFIT, HH, and RFI, no other algorithm protects against overloads due to server failures.

SWAT and Delphi-Pythia are able to remedy an overload situation due to server failure only after detecting it. Their allocations do not reserve space to prevent overload. In Floratou et. al., they mitigate only overload situations by balancing the shared tenant load between servers, which is a weaker constraint. Finally, PMAX will refund tenants that are on overloaded servers.

2.3 Implementation Details

There are numerous considerations for multitenant databases systems to consider in addition to the packing and overload problems. The system must consider the Service Level Agreement (SLA) provided to tenants. An example SLA is 99% of queries must be answered within five seconds. The system must also isolate the effects of one tenant from another tenant sharing the same machine. Additionally, the solution must be placed on a cluster to evaluate the solution's feasibility. Finally, the system will have some targeted workload.

The most common SLA is to target a percentile latency like in CUBEFIT, RFI, SWAT and Delphi-Pythia. Kairos compared only the latencies of the tenants each running on a single machine with the latencies of that tenant running on a shared machine. This provides a good measure of the effects of consolidation on tenants, but does not provide the tenants with any performance guarantees. Lang et al. and Floratou et al. used transaction per second (TPS) SLAs instead of latencies. CUBEFIT is also able to support TPS and requires only computation of the relationship between TPS and load.

Tenant isolation prevents one tenant from reading or writing to another tenant's database and isolation ensures that the load a tenant places on a server does not affect other tenants. In this thesis we used the shared database management system (DBMS) to achieve tenant isolation. Shared DBMS is when all tenants on the same machine share a single DBMS. [8] has shown that shared DBMS is more efficient than having a virtual machine for each tenant because there is less overhead involved with having a single DBMS. The shared DBMS model was adopted by most existing work. Details of the tenant model used in [24] are unavailable as they used a proprietary system provided by SAP.

CUBEFIT is the first multitenant database to be tested on a fleet of machines at the scale of 70 machines. The largest number of machines used before this work was Delphi-Pythia running on 16 machines. Moreover, many works [24, 17, 13] only observed the interactions of shared tenants on a single machine and simulations were used to evaluate the effectiveness of the packing. Running simulations for the allocation only provides insight into the number of servers saved. With simulations you cannot report the latencies of tenants or discover system issues. One such system issue we discovered was that the load was not only affected by the total number of concurrent clients, but the number of tenants as well (which we describe in Chapter 4).

Targeting for a particular workload allows you to make some assumptions that simplify the problem. CUBEFIT, HH, and RFI look at OLAP workloads which predominantly consist of read queries. This allows us to assume that the load of a tenant is evenly divided between its replicas and that if a replica fails, its load is evenly distributed to the remaining replicas. All other work consider the OLTP model. CUBEFIT can support OLTP workloads as long as the there is a high

Paper	SLA	Tenant Isolation	Num. of Machines Deployed	Targeted Workload
CUBEFIT	p99 Latency	Shared DBMS	69	OLAP
HH [10]	N/A	N/A	N/A	N/A
RFI [24]	p99 Latency	Proprietary	Simulation	OLAP
Kairos [8]	compare isolated vs. consolidated latencies	Shared DBMS	2	OLTP
Lang et. al. [17]	TPS	N/A	Simulation	OLTP
Floratou et. al. [13]	TPS	shared DBMS	Simulation	OLTP
SWAT [21]	% of Queries Exceeding Threshold Latency	separate DBMS	6	OLTP
PMAX [19]	server load	shared DB	10	OLTP
Delphi-Pythia [12]	p95 and p99 Latency	shared DB	16	both

Table 2.3: Implementation Details of Multitenant Database Server Consolidation

proportion of reads over writes.

Chapter 3

Cube-Fit Algorithm

In this section, we introduce the CUBEFIT algorithm. CUBEFIT places replicas of almost equal sizes in the same bins. It defines K *classes* for replicas based on their sizes, where K is a small integer. For large data centers with thousands of servers, we suggest $K = 20$, while for smaller settings, it would be smaller, e.g., $K = 5$. Recall that γ denotes the number of replicas per tenant. The replicas with sizes in the range $(\frac{1}{\tau+\gamma}, \frac{1}{\tau+\gamma-1}]$ belong to class τ, where $1 \leq \tau < K$. Note that the size of each replica is at most $1/\gamma$. The replicas which have size in the range $(0, \frac{1}{K+\gamma-1}]$ belong to class K. Each bin also has a class which is defined as the class of the first replica placed in the bin. A bin of class i $(1 \leq i \leq K-1)$ is expected to receive i replicas of the same class. More precisely, it has $i + \gamma - 1$ *slots*, each of size $1/(i + \gamma - 1)$, out of which i slots are expected to be occupied by replicas of type i and $\gamma - 1$ slots are reserved to be empty in anticipation of servers' failure. If i slots of a bin of type i become occupied, we say the bin is a *mature bin*. There might be empty space in a mature bin which the algorithm uses to place smaller replicas, i.e., replicas belonging to classes larger than i.

Let $(x_1, x_2, \ldots, x_\gamma)$ denote the γ replicas of a tenant x. We say a mature bin B *mature-fits (m-fits)* a replica x_j if B has enough space for x_j and, after placing x_j in B, the empty space of B is no less than the total size of replicas shared between B and any set of $\gamma - 1$ bins. To place x, CUBEFIT first checks if, for all replicas of x, there are mature bins that m-fit them. If there are, the algorithm places replicas in them using the Best Fit strategy. More precisely, the replicas are placed one by one, each in the bin with the largest level (used space) that m-fits them. We call this the *first stage* of the algorithm for placing replicas of each tenant. Figure 3.1 provides an illustration of placing replicas in mature bins.

Assume that not all replicas of a tenant m-fit in the mature bins. In this case, the *second stage* of the algorithm is executed. The main idea is to place replicas in the same class into the same

bins and leave enough space in the bins in case of other bins' failure. As mentioned earlier, each bin of type τ ($1 \leq \tau \leq K$) is partitioned into $\tau + \gamma - 1$ *slots* of size $1/(\tau + \gamma - 1)$, and out of these slots, $\gamma - 1$ slot are left empty. The other τ slots in the bin are each filled with one replica of type τ. CUBEFIT performs the placement in a way that any two bins share replicas of at most one tenant. This ensures that the space available in the $\gamma - 1$ empty slots is sufficient to avoid overflow in case of the simultaneous failure of any $\gamma - 1$ servers. In what follows, we describe how the algorithm achieves such packing.

At any given time, the algorithm has γ groups of bins for each type $\tau \leq K - 1$. Each group is formed by $\tau^{\gamma-1}$ bins of type τ. The τ slots in these $\tau^{\gamma-1}$ bins can be arranged to form a cube of size τ in the γ-dimensional space. Replicas are assigned to the slots in the cubes in the following manner. For each type $\tau \leq K - 1$, the algorithm has a counter cnt_τ which is initially 0. After placing replicas associated with a tenant of type τ in the second stage of the algorithm, the counter cnt_τ is updated to $(cnt_\tau + 1) \mod \tau^\gamma$. Note that the value of cnt_τ is always in the range $[0, \tau^\gamma - 1]$, i.e., it can be encoded as a number of γ digits in base τ. Let I_τ indicate this number before placing replicas of tenant x of type τ. The algorithm places replicas of x in the slots indicated by the γ cyclic shift value of I_τ. In other words, the γ digits of I_τ are used to address the slot at which the replica is placed at. For example, if $\tau = 3$, $\gamma = 2$, and $I = (21)_3$, the first replica of x is placed at slot $(2, 1)$ of the first 2-dimensional cube, and the second replica at slot $(1, 2)$ of the second cube. After placing these replicas, I is updated to $(22)_3$. As another example, if $\tau = 3$, $\gamma = 3$, and $I = (010)_3$, the first replica of x is placed at slot $(0, 1, 0)$ of the first 3-dimensional cube, the second replica at slot $(1, 0, 0)$ of the second cube, and the third replica at $(0, 0, 1)$ of the third cube. After placing these replicas, I is updated to $(011)_3$. Figure

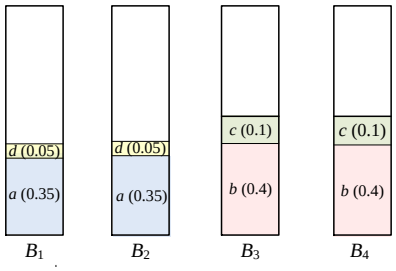


Figure 3.1: An illustration of the first stage of the algorithm. There are two replicas per tenant ($\gamma = 2$). Consider sequence $\langle a, b, c, d \rangle$ of tenants. There will be four bins of class 1, opened by replicas of a and b. After placing these replicas, the four bins become mature. When tenant c arrives, all these bins m-fit the two replicas of c. Bins B_3 and B_4 are selected since they have higher level (used space) when c arrives. Later, when d arrives, only mature bins B_1 and B_2 m-fit the replicas of d.

3.2 provides an illustration and pseudocode for CUBEFIT is shown in Algorithm 1.

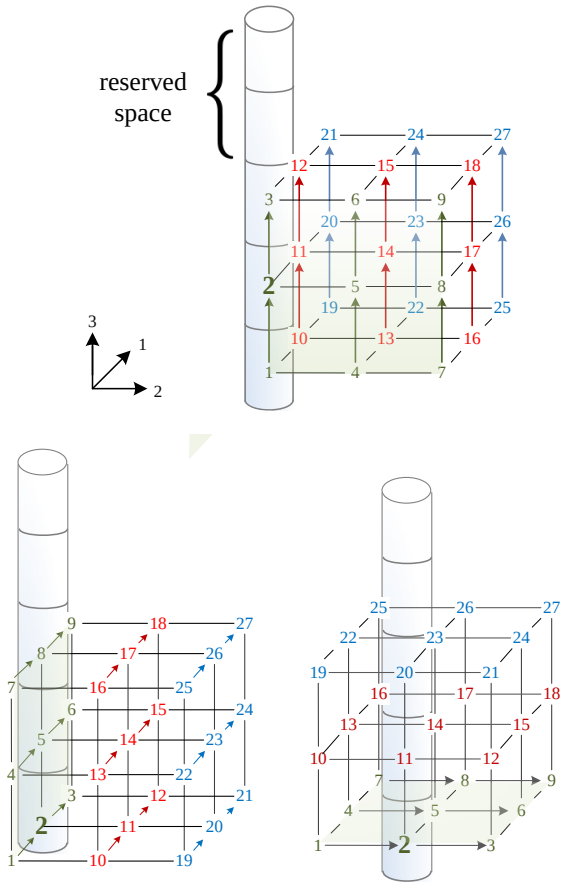


Figure 3.2: The idea behind CUBEFIT for placing replicas of the same type. In this example, we have $\gamma = 3$, i.e., there are three replicas per tenant, each placed in one of the three cubes. Also, replicas have type $\tau = 3$, i.e., the load of each is in the range $(1/6, 1/5]$. Tenants are labelled from 1 to 27. Each of the depicted cubes include one replica from each tenant (replicas with the same label). Replicas in the same group, where only the last digit differs, are placed on the same server. Note that no two servers share replicas of more than one tenant, e.g., tenant $x = 2$ is placed at slot $(0, 0, 1)$ of the first cube, slot $(0, 1, 0)$ of the second cube, and $(1, 0, 0)$ of the third cube. Any pair of the servers associated with these slots share only tenant $x = 2$.

Since each replica of a given tenant is placed in a different dimension in each of γ cubes (groups), we get the following lemma.

Lemma 1 *Consider tenants placed in the second stage of the* CUBEFIT *algorithm. No two bins of type $\tau \leq K-1$ share replicas of more than one tenant.*

Proof of Lemma 1. By definition, two bins in the same group (cube) include the jth replica of each tenant ($1 \leq j \leq \gamma$). Hence, they cannot share replicas of any tenant. Consider two bins B_1 and B_2 in two different groups. For the sake of contradiction, assume replicas of two tenants x and y of type τ are placed in both B_1 and B_2. Since replicas of x and y are placed in B_1, the value of I_τ for x, I_{x,B_1} and y, I_{y,B_1} differ in only the least significant digit because they are on the same server. Similarly, x and y are both placed on B_2, so I_{τ,x,B_2} and I_{τ,y,B_2} also differ in only the least significant digit. Also, I_{τ,x,B_1} and I_{τ,x,B_2} are cycle shifts of one another as well as I_{τ,y,B_1} and I_{τ,y,B_2}. This implies that I_{τ,x,B_2} and I_{τ,y,B_2} differ in only some digit other than the least significant. This contradicts the previous fact that I_{τ,x,B_2} and I_{τ,y,B_2} differ in the only least significant digit. □

To place the replicas in class K in the second stage of the algorithm, we consider the largest integer α_K so that $\alpha_K^2 + \alpha_K < K$, i.e., $\alpha_k = \lfloor \frac{\sqrt{4K+1}-1}{2} \rfloor$. This ensures that $\frac{1}{\alpha_K} - \frac{1}{\alpha_K+1} > \frac{1}{K}$; consequently, the algorithm can group set of replicas of class K into *multi-replicas* with total size in the range $(\frac{1}{\alpha_K+1}, \frac{1}{\alpha_K}]$. The algorithm treats these multi-replicas similar to the way that it treats replicas of class $\alpha_K - \gamma + 1$. There would be γ *active* multi-replicas at each stage of the algorithm, each associated with one of the γ cubes (initially, they are empty sets of replicas). For placing the i'th replica class K in the second stage ($1 \leq i \leq \gamma$), the algorithm checks whether adding the replica to the ith active multi-replica makes the multi-replica larger than $1/\alpha_K$. If it does not, the replica is added to the multi-replica. Otherwise, a new multi-replica which includes only the discussed replica is created and declared as the active multi-replica. Multi-replicas are placed in the same manner as replicas of type $\alpha_K - 1$, i.e., each occupy a slot in bins of type $\alpha_K - 1$. This way, the active multi-replicas in different groups include exactly the same replicas. So, we can treat multi-replicas as replicas of class $\alpha_K - \gamma + 1$. In the discussion that follows, when there is no risk of confusion, we ignore replicas of class $\tau = K$ and assume all replicas belong to classes $\tau < K$.

In summary, CUBEFIT has two stages for placing each tenant x. First, it checks if all replicas of x m-fit in the mature slots of bins of smaller type. If they do, then replicas of x are placed in these mature bins according to the Best Fit strategy. Otherwise, γ replicas of x are placed in γ different cubes as described above. Algorithm 1 illustrates the details of the CUBEFIT algorithm.

Theorem 1 *The schemes resulting from* CUBEFIT *algorithm are valid, i.e., no bin is overloaded in case of failure of at most $\gamma - 1$ servers.*

PROOF. Consider an arbitrary bin B^* of type τ in the packing of CUBEFIT. Also, consider an arbitrary set $S = \{B_1, B_2, \ldots, B_{\gamma-1}\}$ of bins so that $B^* \notin S$. We show that in case of

simultaneous failure of all servers in S, the extra load redirected to B^* does not cause overload. By Lemma 1, B^* and $B_i \in S$ share at most one replica of type τ. So, the extra load redirected to B^* from tenants placed in the second stage of the algorithm is at most $\frac{\gamma-1}{\tau+\gamma-1}$. Summing to this the total load of original replicas in the bin, which is at most $\frac{\tau}{\tau+\gamma-1}$, we get a total load of at most 1, i.e., no overflow for B^* from these replicas. Replicas that are placed in B^* in the second stage of the algorithm (i.e., placed after B^* becomes mature) are ensured to m-fit in B^*. This implies that the extra load resulting from these replicas do not cause an overflow in case of failure of any set of $\sigma - 1$ bins. $\square$

3.1 Average-case Complexity

In this section, we study the average-case complexity of CUBEFIT. For simplicity, we focus on the replication factor $\gamma = 2$. However, similar techniques to the ones introduced here can be applied to study the algorithm for larger values of γ. We assume tenants have random sizes in the range $(0, 1]$, i.e. replica sizes are independently and randomly from the range $(0, 1/2]$. This gives an upperbound for the average case performance as tenant replicas between (1/3,1/2] are the pathological case. They provide less opportunity for packing as shown in Appendix C. We study the *asymptomatic* average-case ratio [6] of CUBEFIT, which is the expected ratio between the cost of CUBEFIT and the cost of an optimal algorithm OPT for serving a sufficiently long random sequence. We consider two distributions for tenant sizes, namely uniform and Zipfian distributions. Our results for uniform distribution directly applies to other symmetric distributions where the chance of a tenant having size x is equal to $1 - x$. We show that the average-case ratio of CUBEFIT is at most 9/7 for uniform distribution, while it is even less for most settings of the Zipfian distribution.

3.1.1 Uniform Distribution

We make use of the techniques introduced for the *up-right matching* problem. An instance of this problem includes n points generated uniformly and independently at random in a unit-square in the plane. Each point is randomly assigned a $\oplus$ or a $\ominus$ label. The goal is to find a maximum matching of $\oplus$ points with $\ominus$ points so that in each pair of matched points the $\oplus$ point appears above and to the right of the $\ominus$ point. Figure 3.3 provides an illustration.

We relate the upright matching problem with the fault-tolerant server consolidation in the following sense. Consider an input sequence of server consolidation defined by a sequence σ of n tenants. Since $\gamma = 2$, each tenant has two replicas, which we refer to as blue and red replicas.

Algorithm 1: CubeFit Algorithm

input : An online sequence $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ of tenants
Positive integers K (number of classes) and
γ (number of replicas per tenant)

output: A packing of tenants in σ which is tolerant against simultaneous failure of any $\gamma - 1$ servers.

mature-bins $\leftarrow$ empty set of bins
for $\tau \leftarrow 1$ *to* K **do**
    $Cnt_\tau \leftarrow 0$ // a counter used in the second stage
    $Group_\tau^\gamma \leftarrow$ arrays of $\tau^{\gamma-1}$ empty bins
    // A γ-dimensional cube of τ^γ slots of type τ.
end
// Continued on next page ...

We create two instances of upright matching, one associated with blue replicas and one with red replicas. Each replica x is plotted as a point in the upright matching instance in the following manner. The vertical coordinate of the point corresponds to the index of x in σ (scaled to fit in the square) . If x is smaller than $1/3$, the point is labelled as $\ominus$ and its horizontal coordinate will be $3x$; otherwise, the point will be $\oplus$ and its horizontal coordinate will be $3 - 6x$. This way, the resulting point will be bounded in the unit square. Note that if in a solution for the up-right matching instance a $\oplus$ point associated with replica L is matched with a $\ominus$ point of replica S, then we have $3S \leq 3 - 6L$, i.e., $L + S \leq 1 - L$. In other words, the empty space in the bin is equal to the larger replica in the bin.

Consider the following up-right matching algorithm. In the instance formed by the blue replicas, we process $\ominus$ replicas in a top-down fashion so that each point S is matched with the leftmost $\oplus$ point among points that are above and to the right of x. In the resulting matching, all points except $n/3 + \Theta(\sqrt{n}\log^{3/4} n)$ of them are expected to be matched [25]. We apply the same algorithm for the matching instance created for the red replicas, except that when creating an edge between $\oplus$ point L and $\ominus$ points S, the following condition should hold: In case blue replicas of L and S are paired together (in the matching instance for blue replicas), we should have $2L + 2S \leq 1$. If this is not the case, instead of matching S with L, which is the leftmost point among the ones on top and right of S, we take the second leftmost point L'. In other words, in the upright matching instance for red replicas, each $\ominus$ point is matched with the leftmost or second left most $\oplus$ point located on its top and right. Similar proof to that of [25] shows that the number of unmatched points in the second matching instance is also $n/3 + \Theta(\sqrt{n}\log^{3/4} n)$ (the additive term is expected to be twice more than that of the other matching).

Algorithm 2: CubeFit Algorithm (continued)

```
// Placing tenants one by one
for i ← 1 to n do
    x ← a_i      // x is the current tenant
    x_1, x_2, ..., x_γ ← x/γ    // γ replicas of x
    first-stage ← True    // whether x is placed in the first stage
      // First stage:
    for j ← 1 to γ do
        if there is at least one mature bin that m-fits x_j
        then
            Place x_j in the mature bin with highest level.
        else
            Remove x_1, x_2, ..., x_{j-1} from their bins.
            first-stage ← False
            break
        end
    end
      // Second stage:
    if first-stage == False then
        I_τ = (I_1, I_2, ..., I_γ)_τ ← Interpretation of Cnt_τ as a number on γ digits in base τ.
        τ ← ⌊γ/x⌋ − 1 ;    //   type of the replicas
        for j ← 1 to γ do
            P ← The I_γ'th slot of the bin B, where B is the bin at index (I_1, I_2, ..., I_{γ-1})_τ
            of group Group_τ^j Place x_j in slot P.
            if B includes τ replicas of type τ then
                mature-bins ← mature-bins ∪ {B}.
            I_τ ← cyclic shift-right I_τ
        end
        Cnt_τ ← Cnt_τ + 1
        if Cnt_τ == τ^γ then
            Group_τ^γ ← arrays of τ^{γ-1} empty bins
            Cnt_τ = 0
        end
    end
end
```

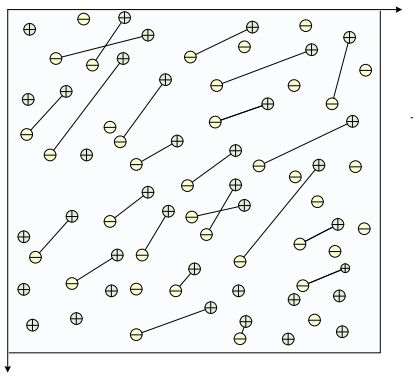


Figure 3.3: A solution for an instance of upright matching

The above algorithms for upright matching instances can be translated to a server consolidation algorithm, let us call it MATCHFIT, which works as follows. MATCHFIT places blue and red replicas separately and opens a new bin for any replica larger than 1/3. The algorithm uses the Best Fit strategy to place any replica x smaller than or equal to 1/3 into a bin B which has a replica larger than 1/3 so that B m-fits x (i.e., the empty space in each of the two bins including replicas of x is sufficient in case of the other bin's failure). If no such bin exists, a new bin is opened for x and no other replica is placed there. This is consistent with choosing the leftmost or second leftmost $\oplus$ bin which appears above and to the right of the $\ominus$ point associated with the replica. Note that when an $\ominus$ points S is matched with the leftmost (or second leftmost) $\oplus$ point L on its top and right, then we know the replica of L appears earlier than S (because L is on top of S), and among the $\oplus$ points that appear on the right of S (i.e., those replicas larger than 1/3 which fit with S in the same bin), it is the leftmost (its replica is the largest). We conclude that, for both blue and red replicas, MATCHFIT is expected to open $n/3$ bins for the replicas larger than 1/3, and among the other $2n/3$ replicas (those smaller than 1/3), it places $n/3 - o(n)$ replicas in the bins opened for replicas larger than 1/3.

A close look at CUBEFIT and MATCHFIT shows similarities between the two algorithms. CUBEFIT opens a new bin for any replica larger than 1/3, i.e., a replica of type one. For other replicas, in its first stage, it checks whether the replicas fit into the mature bins (including bins of type one). If they do, they are placed in mature bins using the Best Fit strategy (similar to MATCHFIT). This indicates that CUBEFIT performs similarly to MATCHFIT in its firsts stage. The difference between the two algorithms comes from the second stage, where CUBEFIT creates a cube structure for placing replicas smaller than 1/3 while MATCHFIT places each of them into a new bin and *closes* the bin right after. By closing a bin, we mean the algorithm does not use the bin for placing future replicas (as a result each bin contains at most two replicas which implies a

matching).

Clearly, CUBEFIT does not open more bins than MATCHFIT because it efficiently uses the empty space in the bins opened by replicas smaller than 1/3 by forming the cube structures. On the other hand, as discussed above, for a sequence of n tenants, MATCHFIT is expected to match (place) $n/3 - o(n)$ of replicas smaller than 1/3 with (in the bins opened for) $n/3 - o(n)$ replicas larger than 1/3, in each of the instances formed by the blue and red replicas. This implies that $n/3 - o(n)$ of replicas are expected to be placed in the second stage of the CUBEFIT in each instance. This observation yields the following lemma.

Lemma 2 *For a sequence of n tenants, the expected number of bins opened by* CUBEFIT *is at most n.*

Proof of Lemma 2. Let m denote the number of replicas, i.e., $m = 2n$. The expected number of bins opened for replicas larger than 1/3 is $m/3$. Among the other $2m/3$ replicas, $m/3 - o(m)$ of them are expected to be placed in the bins opened for the $m/3$ replicas larger than 1/3. This is because, as indicated above, only $o(m)$ points are unmatched in the upright matching instance. The remaining $m/3$ replicas are placed in the second stage of the algorithm. Since each bin of type larger than 1 includes at least two replicas, there will be at most $\frac{m/3}{2}$ extra bins opened for these replicas. In total, the number of opened bins would be $m/3 + m/6 = m/2$, which is n. □

The following theorem implies that the packings resulted from CUBEFIT are away from an optimal *offline* packing with at most a small factor of 1.28 when item sizes follow uniform distribution.

Theorem 2 *The asymptotic average-case performance ratio of* CUBEFIT *is at most* $9/7 \approx 1.28$.

PROOF. Consider a sequence of n tenants, i.e., $m = 2n$ replicas. For each replica of type one (in the range $(1/3, 1/2]$), OPT opens a bin. This is because a bin that includes two of these replicas will be overloaded in case of some server's failure. Since a replica has type one with a chance of $1/3$, OPT is expected to open $m/3$ bins for these replicas. The expected size of the replicas in these bins is $5/12$, i.e., there is an available space of $1 - 10/12 = 1/6$ in each bin for other replicas. This gives an expected available space of size $m/3 \times 1/6 = m/18$ for all bins opened by replicas larger than 1/3. The expected number of replicas smaller than 1/3 is $2m/3$, each having an expected size of $1/6$, i.e., an expected total size of these replica is $m/9$. OPT might place at most $m/18$ of these replicas in the available space in bins of type one. Other replicas have size at least $m/9 - m/18 = m/18$, which should be placed in at least $m/18$ bins.

As a result, the number of bins opened by OPT is expected to be at least $m/3 + m/18 = 7m/18$. Lemma 2 indicates that CUBEFIT is expected to open at most $m/2$ bins. This would result an average-case ratio of at most for $9/7$ for CUBEFIT. □

3.1.2 Zipfian Distribution

In this section, we analyse CUBEFIT when tenant sizes follow the Zipfian distribution. This is consistent with practical scenarios [23] in which most tenants have small sizes while a relatively small number of tenants have larger sizes. We assume tenant take sizes, independently and randomly, from the set $\{1/M, 2/M, \ldots, 1\}$ in which M is a sufficiently large positive integer. Let $s > 1$ denote the value of the exponent characterizing the distribution. The chance of an tenant having rank i is $p(i) = \frac{1/i^s}{H(K,s)}$, where $H(K, s)$ is the generalized harmonic number defined as $\sum_{n=1}^{K}(1/n^s)$. The mean size a tenant is $(H(M, s-1)/H(M, s)) \times 1/M$, and consequently, the total size of tenants in a sequence of length n, denoted by T_n, is expected to be $(H(M, s-1)/H(M, s)) \times n/M$.

Next, we consider tenants smaller than $1/p$, i.e., tenants with sizes in the set $\{1/M, 2/M, \ldots, \lfloor M/p \rfloor/M\}$. The mean size of these tenants is $H(\lfloor M/p \rfloor, s-1)/H(M, s) \times 1/M$. Moreover, the chance that a tenant be smaller than $1/p$ is $H(\lfloor M/p \rfloor, s)/H(M, s)$. Hence, the total size of tenants smaller than $1/p$ in a sequence of length n is expected to be

$$(H(\lfloor M/p \rfloor, s) \times H(\lfloor M/p \rfloor, s-1))/H(M, s)^2 \times n/M$$

Comparing this with T_n, we conclude the following:

Proposition 1 *Total size of tenants that are smaller than $1/p$ is expected to constitute a fraction $f_p = \frac{H(\lfloor M/p \rfloor, s) \cdot H(\lfloor M/p \rfloor, s-1)}{H(M,s) \cdot H(M,s-1)}$ of the total size T_n of all tenants.*

Next, we consider the total utilization (filled space) in the bins of CUBEFIT when parameter K, number of classes, is sufficiently large. For bins opened by replicas of tenants smaller than $1/p$, at least a fraction $(2p-1)/(2p+1)$ of bin space is used. Let $p_1 = 10$, $p_2 = 5$, $p_3 = 2$, and $p_4 = 1$. From Proposition 1, we can find the fraction of the total size of tenants in the following intervals among the total size of tenants. The ranges are $(0, 1/p_1]$, $(1/p_1, 1/p_2]$, $(1/p_2, 1/p_3]$, and $(1/p_3, 1/p_4]$. For example, for M =1,000 and $s = 2$, these fractions can be calculated as 0.687, 0.094, 0.1239, and 0.093, respectively. The utilization of bins opened by tenants in these intervals is lower bounded by respectively $19/21$, $9/11$, $3/5$, and $1/3$. The number of bins opened

s \ M	500	1,000	5,000	10,000
2	1.3910	1.3597	1.3157	1.3006
3	1.1105	1.1079	1.1058	1.1056
4	1.1054	1.1053	1.1053	1.1053
5	1.1053	1.1053	1.1053	1.1053

Table 3.1: Upper bounds for average ratio of CUBEFIT (with sufficiently large parameter K) when tenant sizes follow Zipfian distribution with different parameters s and M.

for tenants in each interval is upper bounded by the expected total size of tenants in the interval divided by these utilizations. In the example above, the total number of bins opened by CUBEFIT is expected to be upper bounded by $0.687\ T_n \times 21/19 + 0.094\ T_n \times 11/9 + 0.1239\ T_n \times 5/3 + 0.093\ T_n \times 3 \approx 1.3597 T_n$. Note that T_n, the expected total size of tenants in the sequence, is a lower bound for the number of bins used by OPT. Hence, for M =1,000 and $s = 2$, the average-case ratio of CUBEFIT is at most 1.3627. Table 3.1 shows the upper bounds for average-case ratio of CUBEFIT calculated in a similar manner for some other values of M and s. Note that as the parameters s and M grow, the average ratio of CUBEFIT becomes better. Intuitively speaking, many tenants become smaller and the sequences become easier to pack. This is consistent with the same observation for classic bin packing (see, e.g., [6]). In particular, for sufficiently large values of s and M, the average ratio of CUBEFIT converges to at most 1.1053. This implies that the packings of CUBEFIT are only a factor 1.1053 away from optimal *offline* packings, i.e., CUBEFIT is close-to-optimal when tenant sizes follow Zipfian distribution with large exponents.

Theorem 3 *The average-case ratio of* CUBEFIT*, when tenant sizes are take independently at random from a Zipfian distribution with parameter s and M converges to at most 1.1053 for large values of s and M.*

3.2 Worst-case Analysis

In this section, we provide upper bounds for the competitive ratio of the CUBEFIT algorithm, which reflect the worst-case behavior of the algorithm. Consider an input sequence $\sigma = \langle a_1, \ldots, a_n \rangle$ of length n. Recall that there are γ replicas for each tenant x that we denote with $x_1, x_2, \ldots, x_\gamma$, and each replica x_j ($1 \leq j \leq \gamma$) has a load x/γ. To provide an upper bound of r for competitive

ratio, we define a weight for each replica x_j, denoted by $w(x_j)$, and prove the following statements: (I) The total weight of replicas in each server, except a constant number of them, in the packing using CUBEFIT is at least 1. (II) The total weight of replicas in each server in an optimal packing scheme is at most r. The above statements imply a competitive ratio of r for CUBEFIT. This is because (I) implies that $\text{CUBEFIT}(\sigma) \leq W(\sigma)$ and (II) implies $\text{OPT}(\sigma) \geq W(\sigma)/r$ where $W(\sigma)$ denotes the total weight of all replicas of all tenants in σ.

Theorem 4 *For the fault-tolerant bin packing with replication factor $\gamma = 2$ and $\gamma = 3$, the competitive ratio of* CUBEFIT *with large values for K approaches to approximately 1.59 and 1.625, respectively.*

We define the weight of each replica x in the following manner. Note that all replicas have size at most $1/\gamma$. If $x \in (1/(i+1), 1/i]$ for some positive integer $i(\gamma \leq i \leq K+\gamma)$, then the weight of x will be $1/(i-\gamma+1)$. Recall that x_j belongs to class $\tau = i-\gamma+1$ in this case and $i-\gamma+1$ replicas of this type are placed in each bin of type τ (except potentially the last group of bins). The remaining replica are those of type K, i.e., those smaller than $1/(K+\gamma-1)$. These replicas form multi-replicas with total size in the range $(\frac{1}{\alpha_K+1}, \frac{1}{\alpha_K}]$. We define the weight of a replica of size x in class K to be $\frac{x(\alpha_K+1)}{\alpha_K-\gamma+1}$. This ensures that the resulting multi-replica has a total weight of at least $\frac{1}{\alpha_K-\gamma+1}$, which is the same as a replica of type $\alpha_K-\gamma+1$

We show that total weight of replicas in any bin, except a constant number of them, is at least 1. Let i denote the type of a given bin in the packing of CUBEFIT $(1 \leq i \leq K-1)$. If $i \neq \alpha_K-\gamma+1$, then the bin includes i replicas of type i. The only exception is the last γ groups of bins opened for replicas of type i which might include less than i replica. Assuming $K, \gamma \in O(1)$, there would be a constant number of such bins, which can be ignored in the asymptotic analysis. So, the total weight of replicas in bins of type i, except a constant number of them, is $(i-\gamma+1) \times \frac{1}{i-\gamma+1} = 1$. Bins of type $i = \alpha_K-\gamma+1$ might include multi-replicas. There will be i slots in these bins, each occupied with either a replica of type i or a multi-replica (except a constant number of bins in the last group). In both cases, the total weight of replicas in such slot would be $\frac{1}{i}$, which gives a total weight of 1 for all replicas in the bin.

Consider a bin B in the optimal packing. Assume B includes m_i replicas of type i $(1 \leq i \leq K-1)$. Since we look for an upper bound for total weight of replicas in B and all replicas of type i have equal weight, we might assume these replica have the smallest weight in their class, i.e., all replica of type i in B have size $\frac{1}{\gamma+1}+\epsilon$ for some small positive ϵ. Consider the largest $\gamma-1$ replicas in B. To have a valid packing, there should be an empty space of size at least equal to sum of the sizes of these $\gamma-1$ replicas. This condition is required to ensure that failure of $\gamma-1$ servers does not cause an overload in B. Let T denote the type of the smallest replica

among these $\gamma - 1$ replicas (excluding replicas of type K, i.e., $T \leq K - 1$), and M denote the number of replicas of type T among these $\gamma - 1$ replicas, i.e., $M = \gamma - 1 - \sum_{i=1}^{T-1} m_i$. Note that $0 < M \leq m_{_T}$. To maximize the total weight of replicas while satisfying the condition regarding to the empty space, we should solve the following integer program:

Maximize $regularWeight + tinyWeight$ where

$$regularWeight = \sum_{i=1}^{K-1} \left(m_i \times \frac{1}{i} \right)$$

$$tinyWeight = \frac{tinySize(\alpha_K + 1)}{\alpha_K - \gamma + 1}$$

Subject to:

$$regularSize + tinySize + emptySize \leq 1$$

$$regularSize = \sum_{i=1}^{K-1} m_i (\frac{1}{\gamma + i} + \epsilon)$$

$$reservedSpace = \sum_{i=1}^{T-1} m_i \frac{1}{\gamma + i} + M(\frac{1}{\gamma + T} + \epsilon)$$

In the above program $regularWeight$ and $regularSize$ respectively denote the total weight and size of replicas in B which belong to classes other than K. Similarly, $tinyWeight$ and $tinySize$ denote the total weight and size of tiny replicas in B, i.e., those in class K. Also, $reservedSpace$ denote the reserved space in B. The variables of the above programs are m_i's ($1 \leq i \leq K - 1$) which are non-negative integers, $tinySize$ which is in a real value in the range $(0, 1]$, and T which is a positive integer smaller than K. We can solve the above program for small values of γ and K.[1] In particular, we get the following result for $\gamma \in \{2, 3\}$.

[1] For larger values of γ, we can find lower bounds for the IP program, which give upper bounds for the competitive ratio of CUBEFIT.

Chapter 4

System Model

In this section, we describe our system model. Each server hosts multiple tenants and has a data store that is shared between tenants that it hosts, as shown in Figure 4.1. A tenant's load is generated by some number of concurrent clients, each having a workload consisting of a set of queries that are executed against the tenant's data store.

A server services all clients of all hosted tenants. There are multiple ways to implement such a multi-tenant model, e.g., virtual machines, shared table, shared database management systems (DBMS) and separate DBMS. We use shared DBMS as it has better performance than virtualization [8] and several multi-tenant environments have used it [8, 21]. In a shared DBMS environment, each tenant resides as a database instance on the single DBMS running on the server.

The load of a tenant is the ratio between the amount of the server's capacity used by the tenant and the server's total capacity (hence, a number in the range $(0, 1]$). Tenant loads form

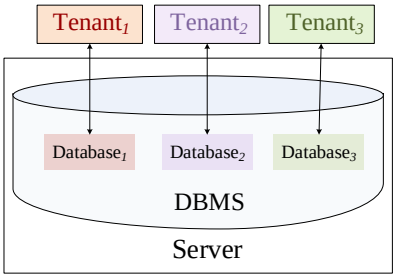


Figure 4.1: Shared DBMS model: Tenants 1, 2 and 3 share the data store on the server.

the input sequence for the tenant placement algorithm. The load of a tenant is shared equally between its γ replicas. This holds for the read heavy OLAP workloads we looked at. The load on the server is the sum of the loads of the tenant replicas on that server. The number of clients per tenant follow either a uniform or Zipfian distribution.

We use a simple but practical load model to demonstrate the placement algorithm's feasibility. As with the load model used in [24, 22], CUBEFIT uses as input a value that captures the in-memory load that the tenant places on a server, and load from multiple tenants on the same server is additive, as shown in [24]. It has also been shown that usages on memory bandwidth and CPU are additive [8], combined as a single load value representing the proportion of the total CPU and memory bandwidth used [24]. Moreover, [22] shows that a linear model accurately predicts latency.

There is a rich body of work that captures a tenant's resource usage on a server [22, 8, 17]. This tenant resource characterization problem is orthogonal to tenant placement that this thesis considers. As in [22], we model tenant utilization using a linear relationship between a tenant's properties. In our experiments the load of a tenant is defined by the function $\delta c + \beta$ where c is the number of clients, δ represents the amount of capacity each client takes up on the server, and β is the overhead each tenant places on the server. This function produces a value larger than 1.0 when the server's capacity is over-utilized. The value of δ and β are specific to a hardware configuration but can be generalized for multiple configurations. We determined the values for δ and β by running various numbers of clients evenly distributed over various numbers of tenants. Some client-tenant configurations resulted in the SLA being violated, and others resulted in meeting the SLA. This allowed us to derive the equation of the line that separates the configurations that meet SLA from those that do not, providing us with the values for δ and β. To focus on tail latencies, we set the SLA to be 5 seconds at the 99th percentile.

When a server fails, clients of tenants hosted on that machine execute their queries on the remaining tenant replicas on other servers. This increases the number of concurrent clients the remaining servers need to serve. To meet SLA requirements, a server should not receive more clients from failed tenant replicas than its available capacity so as to ensure that its capacity does not exceed the 99th percentile latency as described above.

Chapter 5

Experiments

In this chapter, we present our evaluation of the CUBEFIT algorithm[1] for server consolidation. Our comparison is threefold: first, we implement CUBEFIT on a real system comprising of 73 machines and compare with the RFI algorithm using the TPC-H workload. (ii) We compare CUBEFIT and the RFI algorithm from [24] through extensive simulations. (iii) We compare CUBEFIT and RFI algorithms with the optimal solution of the mixed integer program formulation from Section B.1 implemented as a solver using CPLEX v12.6.1.0 [4].

5.1 Robust Fit Interleaved

Robust Fit Interleaved (RFI) [24] is a modified version of the Best Fit bin packing algorithm. RFI first searches for the server that would have the least load left over after a tenant is placed on it including having enough reserved capacity for additional load from any single failed server (overload capacity), and a μ value that governs how much of the first server's total capacity to use for interleaving. If no such server is found, a new server is provisioned and the replica is placed there. For the second replica, the algorithm repeats the process but selects a different server machine.

The online algorithm as described in [24] can generate invalid packings. The problem was that placing the second replica could change the overload capacity for the first replica that was already placed. We fix this problem by checking that placing the replica would not create a (worse) load transfer that would overload the server. If it does, we do not consider that server.

[1] We use a slightly optimized version of CUBEFIT described in Appendix D

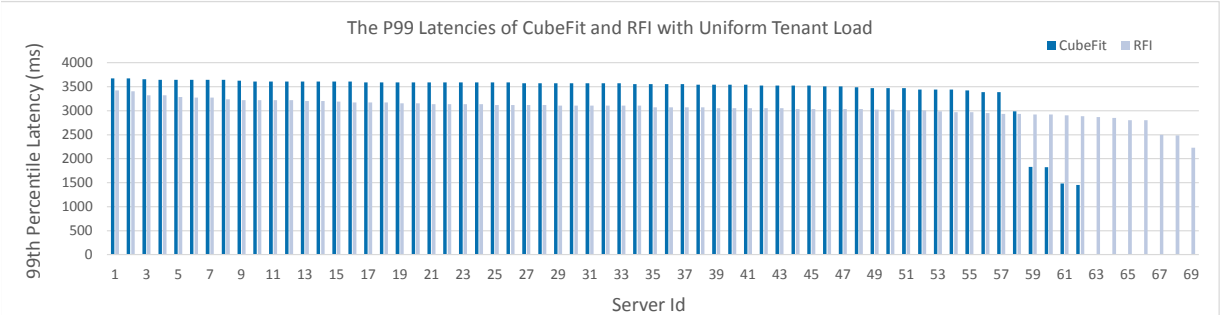


Figure 5.1: CUBEFIT and RFI placing 309 tenants, each with tenant load drawn from a discrete uniform distribution

We also check that there is enough capacity for the first replica's load before placing it on the first server, in addition to capacity to take on a failing replica's load in the event that placing the second replica on a new server puts it in overload capacity failure.

5.2 System Performance

We implemented both CUBEFIT and RFI on a cluster of 73 server machines. We use 69 of the servers to host the tenants and their data. The remaining 4 are used to generate the client query load for tenants. All of the machines were Intel Xeon 2.1 GHz with 12 cores, 32GB of memory and connected over 10G Ethernet. We use the shared DBMS tenant isolation model described in Section 4 on which to run these algorithms. We created a PostgreSQL DBMS instance to hold the data of each tenant executing TPC-H benchmark queries. Each tenant starts out with 100MB of data, which amounts to having up to 10GB of tenant data in the memory of a machine.[2]

While any SLA can be chosen to work with CUBEFIT, our SLA of 5 seconds latency was derived empirically. Per Section 4, we determined that a maximum of 52 concurrent clients can be supported per host machine. For the first experiment, the number of clients per tenant was selected with equiprobability from a discrete uniform distribution of 1 to 15 clients. For the second experiment, the number of clients followed a Zipfian distribution of exponent 3 and the number of clients was sampled from 1 to 52 as mentioned above. We evaluated the CUBEFIT and RFI placements against these distributions. For both experiments, we continued to add tenants until the RFI algorithm used all 69 servers. This favors RFI since it is given the chance to fill up as much as it can before having to open a new bin while the exit condition does not give

[2]Typical tenant sizes in prior work have ranged from 25MB to 204MB [22].

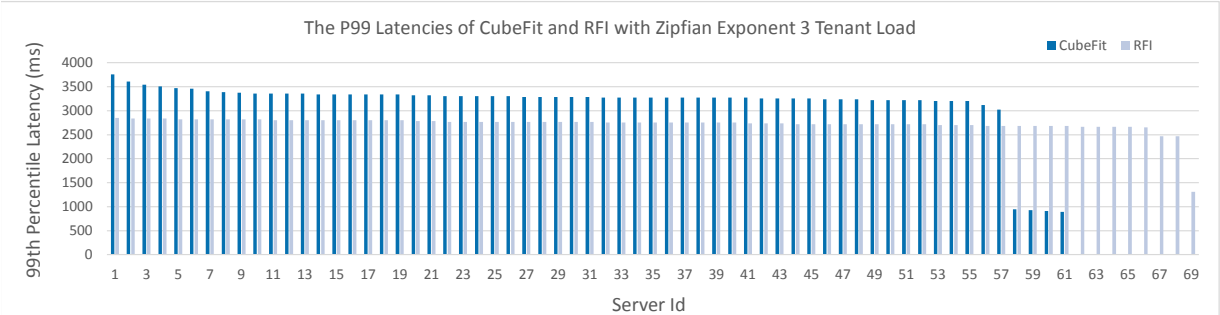


Figure 5.2: CUBEFIT and RFI placing 1573 tenants, each with tenant load drawn from a Zipfian distribution

CUBEFIT this opportunity. Thus, with this bias against CUBEFIT, we expect RFI to use fewer servers than otherwise.

To achieve steady state, we let the system warm-up by running the workload on all tenants for five minutes. This allows the database system to cache all tenants' data in memory. Over the next five minutes after warm-up, we measure system load and latencies.[3] Our measurements were obtained using CUBEFIT configured with 5 classes for both the Uniform experiment and the Zipfian experiment. Unless otherwise mentioned, all of our experiments were run with both of the distributions. For RFI, we used μ equal to 0.85 as recommended in [24].

Figures 5.1 and 5.2 show the 99th percentile latencies of each host machine used in our experiments. In all cases, CUBEFIT stays within the 5 second latency while still leaving enough capacity in the event of a failure. In the uniform case, CUBEFIT uses an average of 11.3% fewer servers. In the Zipfian case, CUBEFIT uses an average of 13.1% fewer servers. The better utilization of capacity by CUBEFIT is due to restricting the amount of load shared between two servers, reducing the amount of capacity that is reserved. In contrast, RFI is unable to achieve this reduction as it does not have an upper bound on the amount of space it reserves. Additionally, we observed that CUBEFIT packs better than HH in all the scenarios we tested. For instance, when we tried the above Uniform experiment, HH used up all 69 servers while CUBEFIT used only 54, a relative difference of almost 30%. Thus, given the across-the-board superiority of CUBEFIT over HH, we do not consider it further.

[3] We experimentally determined that there was no deviation in results by increasing this measurement interval.

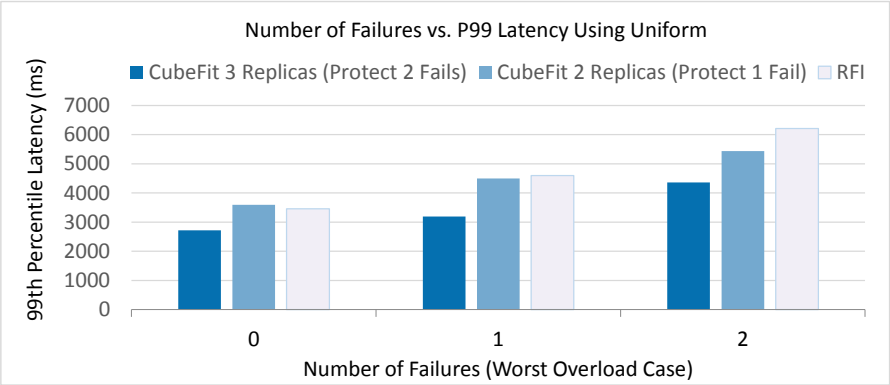


Figure 5.3: 99th percentile latency of CUBEFIT and RFI with discrete uniform distribution tenants

5.2.1 Server Failures

In the next two experiments, we study how CUBEFIT and RFI respond to server failures. In addition to the configurations from Section 5.2, we study CUBEFIT's behavior with a configuration of three replicas that can protect against two failures. Recall that the RFI algorithm from [24] cannot protect against multiple server failures. We keep adding one tenant until CUBEFIT fills up all 69 data store servers. To cause f server failures, we select f servers that result in the distribution of the highest number of clients to a single server (resulting in the highest possible load on a server; we call this the "worst overload case"). When a server fails, the load is distributed as described in Section 4.

With one server failure, the highest 99th percentile latency was 4.49 seconds for CUBEFIT (uniform distribution) that protects against an overload from one server failure. The latency for CUBEFIT (Zipfian distribution) was 4.16 seconds. These results demonstrate that CUBEFIT does not violate SLA in the one server failure scenario.

For the two-failure scenario, the CUBEFIT configuration that protects against an overload from two server failures resulted in a 99th percentile latency of 4.36 seconds for the uniform distribution and 4.29 seconds for the Zipfian distribution. This shows that CUBEFIT is effective in protecting against the failure of 2 servers while the algorithm allows it to be configured to protect generally against k server failures.

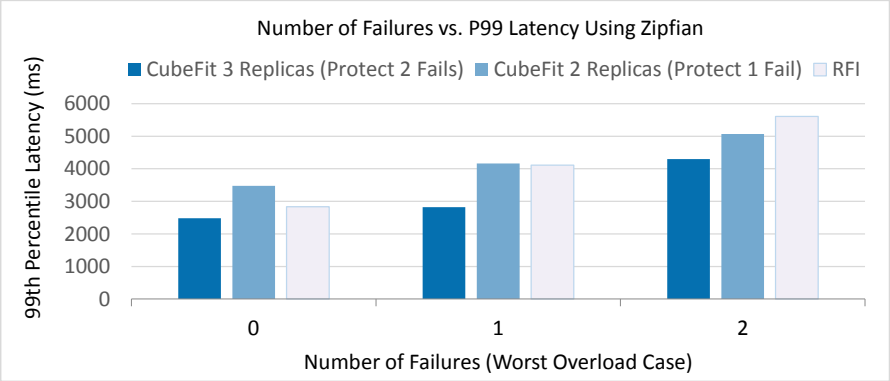


Figure 5.4: 99th percentile latency of CUBEFIT and RFI with Zipfian distribution tenants

Figures 5.3 and 5.4 show the relationship between the number of server failures and the latency. We can see that both load and latency increase more slowly for CUBEFIT. CUBEFIT's superior performance is due to having an upper bound on the load that can be shared between servers. As a result, each additional server failure brings a bounded increase of load. In contrast, RFI is unable to enforce an upper bound on the amount of load shared between servers. Finally, CUBEFIT with 3 replicas is able to provide even more protection against overload by trading a slightly worse packing for the additional protection.

5.3 Simulations

Experimental results from previous sections demonstrate the performance advantage that CUBEFIT has over algorithms like RFI. As shown through theoretical analyses in Section 3, asymptotic performance of the CUBEFIT algorithm is significantly better when there is a large number of tenants to consolidate on a large number of servers, as would be the case in a data center. Since we do not have access to thousands of servers, we further study the behavior of CUBEFIT under large (resource) scale through simulation experiments.

We implemented a simulator which has a suite of distributions generate tenant load sequences and these loads are given to the placement algorithms. Based on the resulting placement, the simulator captures statistics including how many servers were used, amount of time each placement algorithm needs to consolidate tenants onto servers, and the average server utilization. We ran 10 independent simulations each with 50,000 tenants and computed the relative differences of CUBEFIT compared to RFI using the average number of servers used over these 10 runs. Results of these simulation experiments for different uniform and Zipfian distributions are shown in

Figures 5.5 and 5.6, with 95% confidence intervals as whiskers on the bars in these figures. The relative difference (in the average number of servers utilized) is defined as $\frac{RFI - \text{CUBEFIT}}{\text{CUBEFIT}} \times 100\%$ where RFI in this case is the average number of servers used by the RFI algorithm and CUBEFIT in this case is the average number of servers used by the CUBEFIT algorithm. CUBEFIT allows a variable number of classes to be used for any particular configuration. We used 10 classes for both the uniform and Zipfian distributions. We used more classes than in the system experiments presented earlier because more classes provide better performance with larger numbers of tenants. For RFI we used μ equal to 0.85 as recommended by [24]. We graph the results of using various uniform and Zipfian distributions. The Zipfian distribution does not produce values between 0 and 1 on its own so we sample a Zipfian distribution with values 1 to C and divide by C to get normalized values between 0 and 1, where C is the maximum number of clients that a server can support without violating SLA. We set C to 52, similar to the number of clients our cluster can support. In both Figures 5.5 and 5.6, CUBEFIT performs better than RFI across-the-board. When smaller tenants increase, there are more servers belonging to larger classes that reserve less space to prevent overload due to server failure. This results in better server utilization, allowing CUBEFIT to perform increasingly better over RFI.

Finally, we computed the cost savings over one year for the uniform and Zipfian distributions from the simulation results using 50,000 tenants. To compute these costs, we use a cost of $0.822/hour per Amazon EC2's c4.4xlarge machine instances, which are similar in system resources to the machines we used in our experimental results from Section 5.2. As Table 5.1 shows, for continuous server operation, performing server consolidation in the cloud using CUBEFIT can generate substantial yearly cost savings for cloud service providers.

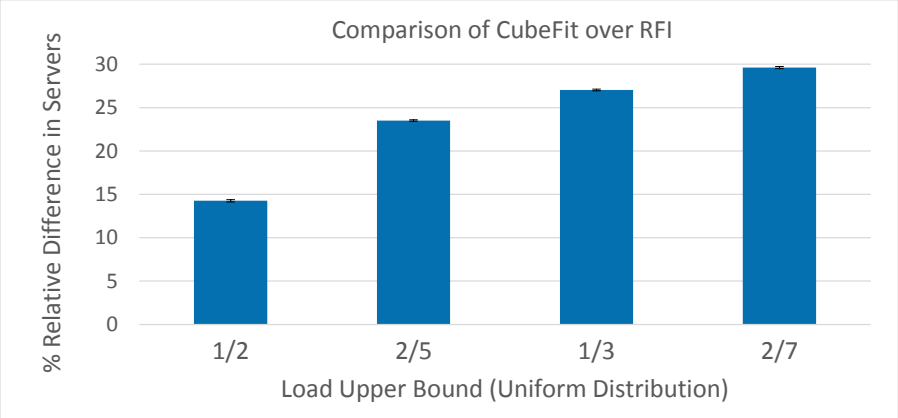


Figure 5.5: % Relative difference of servers used by CubeFit over RFI for various uniform distributions

Distribution	Servers Saved	Dollar Savings
Uniform	2,474	18,045,004
Zipfian	496	3,571,557

Table 5.1: Yearly cost savings of CUBEFIT over RFI

The performance improvement over RFI increases as the size of the tenants decrease. As we have more large tenants, there is less opportunity for better packings. This is because all algorithms place tenants with replica load between (1/3, 1/2] in the same way – one replica of that load per server. We know that a new server is opened for every replica between (1/3, 1/2] so all placement algorithms will converge to the same solution, as shown in Appendix C by Lemma 3. As we increase the number of large tenants, they become responsible for a majority of newly opened servers.

5.4 Comparison with Optimal

We compare the CUBEFIT and RFI algorithms with the optimal solution of the mixed integer program formulation from Section B.1 solved using CPLEX. We converted our constraints into a mixed integer program as follows. First, we removed the max by adding a constraint for all possible server failures instead of just for the maximum one. Second, the multiplication of binary variables was translated to a mixed integer program using the elimination of products of variables technique outlined in [5]. These linearizations were done to allow the use of mixed-integer linear

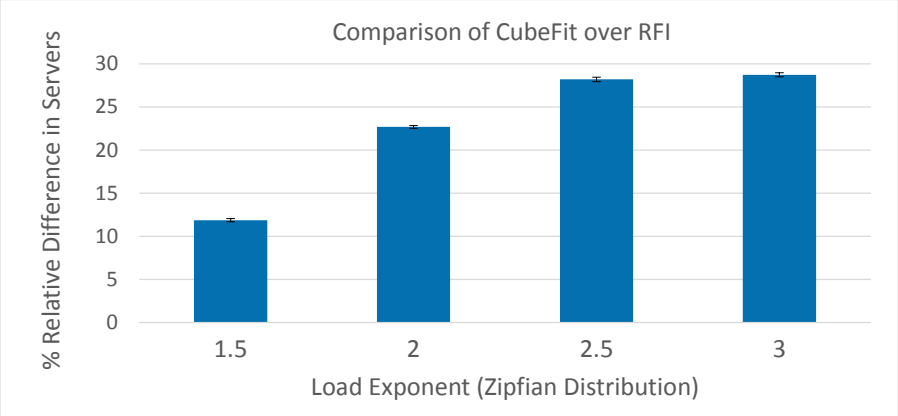


Figure 5.6: % Relative difference of servers used by CubeFit over RFI for various Zipfian distributions

program solvers rather than resorting to slower non-linear program solvers. Appendix B.2 covers this in more detail.

We use the same uniform and Zipfian distributions as in Section 5.3. We repeat each experiment ten times and report the average relative differences and distance ratios with 95% confidence intervals in Table 5.2. The relative difference percentages are calculated by $\frac{X-CPLEX}{CPLEX} \times 100\%$, where $CPLEX$ is the number of servers used by CPLEX and X is the number of servers used by CUBEFIT or RFI. The "distance ratio" percentage is calculated by $\frac{RFI-\text{CUBEFIT}}{RFI-CPLEX} \times 100\%$ where RFI, CUBEFIT, and CPLEX in this case are the average number of servers used by RFI, CUBEFIT, and CPLEX respectively. This measures the percentage performance difference between RFI and CPLEX covered by CUBEFIT. This metric is more useful than a relative difference between CUBEFIT and RFI because CPLEX, given enough time, provides the best possible result and should be used as the lowerbound. We solve instances of the mixed integer program using the CPLEX solver with the upperbound on the number of servers set to half the number of tenants. We give the solver one hour to solve each problem on a machine with two 12-core Intel 3.5GHz Xeon E5-2697 v2 processors and half a terabyte of memory. One hour is a very liberal upper bound on how long the placement algorithm can take to generate a solution *offline*, in contrast to CUBEFIT which generates a solution *online* in a fraction of a second. Since running CPLEX for 1 hour and 10 hours gave the same result with at least three different seeds, we see no hope of significantly improving the result of CPLEX by running it for longer than an hour. We configure CUBEFIT and RFI as they are configured in Chapter 5.2. All three algorithms work on the same tenant inputs.

Table 5.2 shows that CUBEFIT performs better than RFI and comes closest to the optimal solution. At 600 tenants CUBEFIT (with a running time of 23.4 milliseconds) has reduced the "distance" to the average that CPLEX can achieve (in one hour) by 59% for the uniform distribution. Additionally, CUBEFIT used only 11.4% more servers than CPLEX while RFI used 27.9% more. CUBEFIT demonstrates robustness by its ability to work with any number of tenants while CPLEX is unable to find a solution beyond 1000 tenants as it exhausts the half terabyte of memory of the machine it ran on within an hour. If left to run longer, CPLEX would consume even more memory resulting in even fewer hosted tenants. The biggest factor for running out of memory is the number of servers as the problem size is quadratic in the number of servers. For the Zipfian distribution with 2,200 tenants, CUBEFIT (in 42.8 milliseconds) reduced the distance to what CPLEX can achieve (in one hour) by 47.4%. CUBEFIT used only 19.4% more servers than CPLEX, while RFI used 36.8% more. Beyond 2,600 tenants, CPLEX was unable to find a solution because it again ran out of memory. These results show that CUBEFIT is an efficient algorithm, and that it can be used to generate server consolidation solutions that are effective in practical settings.

Distr.	Max Tenants	Servers Used			% Rel. Diff. wrt CPLEX		% Dist. Ratio Reduced
		CUBEFIT	RFI	CPLEX	CUBEFIT	RFI	
Uniform	600	114.1	131	102.4	11.4 ± 1.07	27.9 ± 1.03	59.0 ± 4.45
Zipfian	2200	86.3	98.9	72.3	19.4 ± 3.07	36.8 ± 2.68	47.4 ± 4.97

Table 5.2: Comparison of the number of servers used by CUBEFIT, RFI, and CPLEX with the given distributions using the maximum number of tenants solvable by CPLEX.

5.5 Sensitivity Experiments

In this section we explore the relationship between the performance of CUBEFIT with respect to its class parameter. From the worst case analysis in Section 3.2, as we increase the number of classes we would expect the performance to increase. For each incremental increase in the number of classes, we should also expect to see diminishing returns. To verify this, we run the same simulations as in Section 5.3 but we iterate over the number of classes.

Figures 5.7 and 5.8 show that as you increase the number of classes, you get improved performance for both the uniform and Zipfian distributions. The performance increases more slowly as the number of classes increases. One pattern repeated again from Section 5.3 is that as the distribution has more smaller tenants, the performance increases. Each line being higher than the previous demonstrates this pattern. The performance increases as the number of classes increases because the larger the class, the less space that needs to be reserved for that class. Recall that the amount of space reserved is $\frac{1}{K}$.

Figures 5.9 and 5.10 show the number of tenants needed on average before CUBEFIT always performs better than RFI. The graph shows that there is a trade off for increasing the number of classes. The minimum number of servers needed to improve performance increases as the number of classes increases.

Secondly, this is the only point at which CUBEFIT performs just better than RFI. The number of servers need to achieve the best performance that the number of classes offer will also increase as the number of classes increase. The minimum number of tenants needed to perform well increases as the number of classes increases because a large number of classes results in a large constant factor that needs to be overcome. Consider how servers of class 10 fill up. For the first

10 tenants, the second replicas create 10 new servers. Each server has only 1 replica. 100 tenants later, the 10 servers are utilized to their maximum potential. However, another period of server creation begins from the 101st to 110th tenant creating another 10 new servers for the second replica. Continuing the process, with enough tenants the number of accumulated fully utilized servers greatly exceeds the 10 unfilled servers.

We expected that as the distribution had more smaller tenants, an increased number of tenants would be needed until CUBEFIT performed better than RFI because these the smaller tenants belong to larger classes which takes more tenants to fill up. However, we did not consider that the smaller tenants are also more efficiently packed. As a result, there are two competing forces. The larger classes of the smaller tenants causes an increase in the number of tenants needed, but at the same time the efficiency of the larger classes reduces the number of tenants needed.

There are some data points on the Figures 5.7 and 5.8 with negative values. These indicate that CUBEFIT performed worse than RFI even after 50,000 tenants. Also in Figures 5.9 and 5.10 the data points that never performed better than RFI are not shown. Too few classes performs poorly because they reserve up to $\frac{1}{K}$ of the servers space. Five was the smallest number of classes needed to perform better in all simulations.

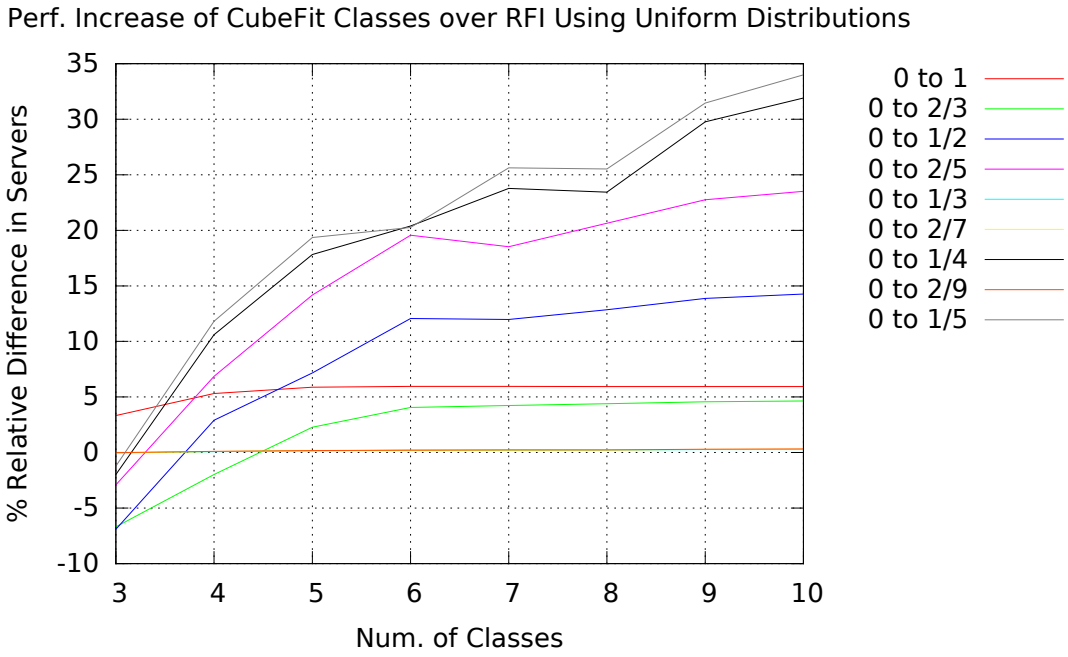


Figure 5.7: % Relative difference of servers used by CubeFit over various class parameters using uniform distribution

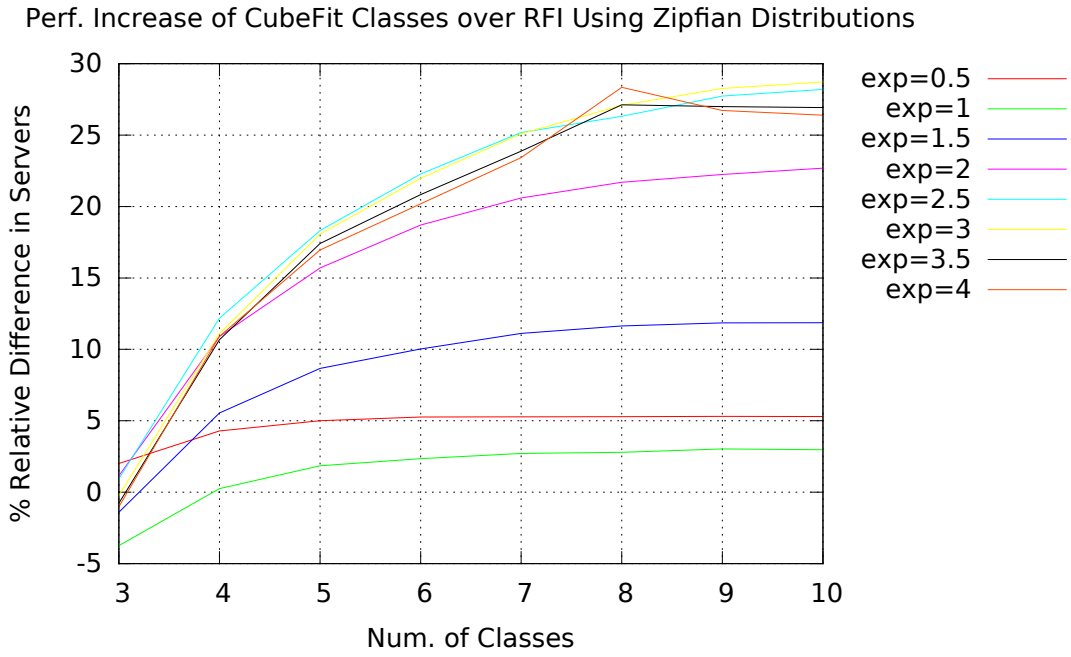


Figure 5.8: % Relative difference of servers used by CubeFit over various class parameters using Zipfian distributions

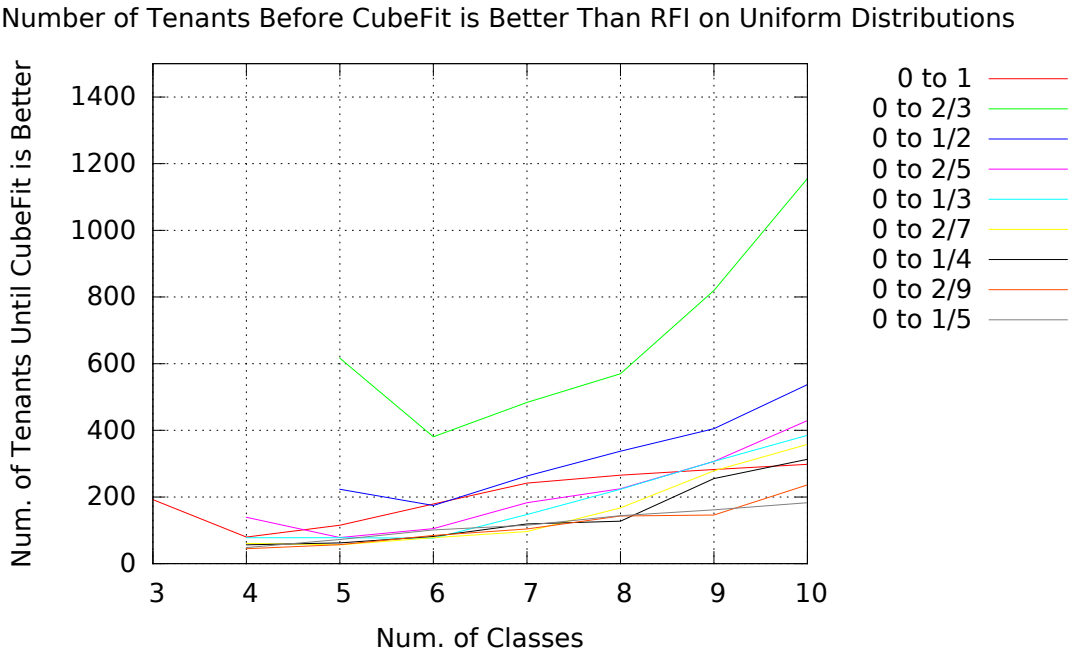


Figure 5.9: The minimum number of Uniformly distributed tenants needed until CubeFit always performed better than RFI for various classes

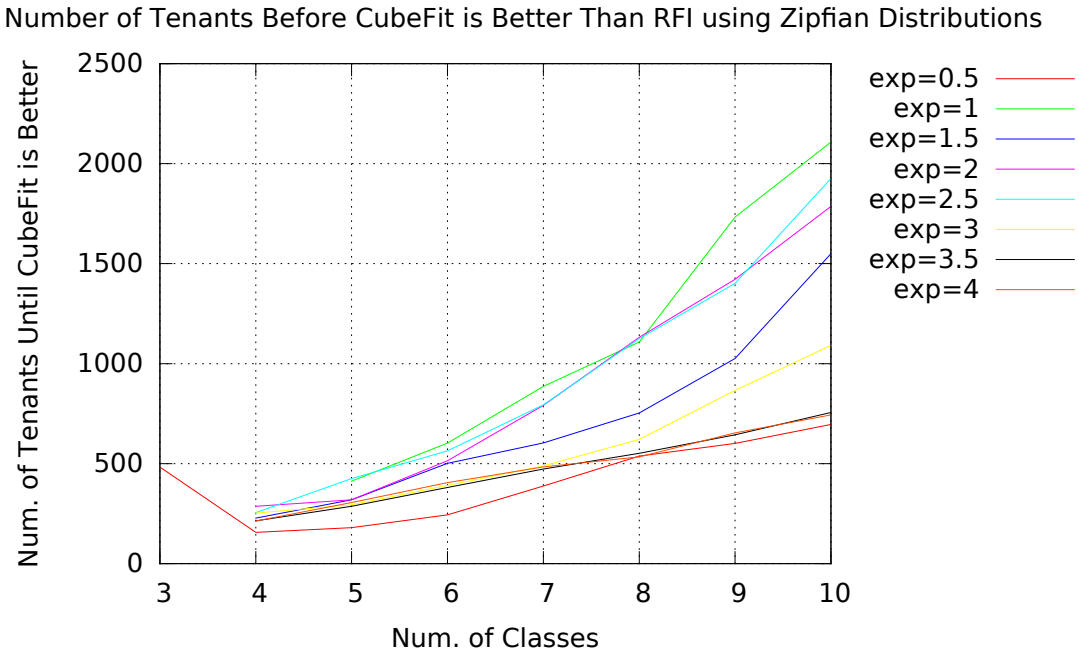


Figure 5.10: The minimum number of Zipfian distributed tenants needed until CubeFit always performed better than RFI for various classes

Chapter 6

Conclusion and Future Work

In this thesis, we presented CUBEFIT, an efficient algorithm for multitenant database server consolidation in the cloud. Through extensive theoretical and experimental analyses, we showed that CUBEFIT is more efficient than its counterparts as it produces superior server packing of tenants and achieves near-optimal allocation on cloud database servers. Importantly, CUBEFIT is also more robust as it can protect against multiple server failures, which none of the previous proposals can provide. Our evaluation through theoretical analysis, system measurements, simulations and comparison to optimal show that CUBEFIT is the best available choice for online multitenant database server consolidation and can generate significant cost savings over existing approaches.

Investigating modifications to CUBEFIT to support high proportions of writes to reads is important future work. Two modifications must be made. First, when computing the replica's load from the tenant load, instead of dividing the entire tenant load by γ, we divide only the read portion by γ. The write portion is not divided because the write must occur on all replicas to keep the data consistent. Second, when reserving space to protect against overload from server failure, only enough capacity for the read portion should be reserved because only the reads from a failing replica are redirected. To reserve different amounts of space for different proportions of reads to writes, we can add a second parameter to classes. This new parameter would be the proportion of reads to writes, allowing CUBEFIT to more intelligently reserve space for reads.

Finally, the assumptions made by CUBEFIT are not limited to analytical workloads for databases. Systems like in-memory caches, web application servers, and video streaming can make the same assumptions.

References

[1] Amazon AWS. http://aws.amazon.com/application-hosting/, Accessed: 2015-11-19.

[2] Analytics cloud. http://www.microsoft.com/en-ca/server-cloud/products/analytics-platform-system/, Accessed: 2015-12-09.

[3] Microsoft analytics platform system. http://www.microsoft.com/en-ca/server-cloud/products/analytics-platform-system/, Accessed: 2015-12-09.

[4] *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*. IBM Corp., 2014. http://ibm.com/support/knowledgecenter/SSSA5P, Accessed: 2015-11-19.

[5] Johannes Bisschop. *AIMMS Optimization Modelling*. Paragon Decision Technology, 2012. http://www.aimms.com/aimms/download/manuals/aimms_modeling.pdf, Accessed: 2015-11-19.

[6] Edward G. Coffman, Michael R. Garey, and David S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation algorithms for NP-hard Problems*. PWS Publishing Co., 1997.

[7] Edward G. Coffman Jr., János Csirik, Gabor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: survey and classification. In Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham, editors, *Handbook of Combinatorial Optimization*, pages 455–531. Springer, 2013.

[8] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, pages 313–324, 2011.

[9] CyrusOne executive report. Build vs. buy: Addressing capital constraints in the data center. 2013.

[10] Khuzaima Daudjee, Shahin Kamali, and Alejandro López-Ortiz. On the online fault-tolerant server consolidation problem. In *26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 12–21, 2014.

[11] Khuzaima Daudjee, Shahin Kamali, and Alejandro López-Ortiz. Online fault-tolerant server consolidation problem. In *SPAA*, pages 12–21, 2014.

[12] Aaron J. Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *SIGMOD*, pages 517–528, 2013.

[13] Avrilia Floratou and Jignesh M. Patel. Replica placement in multi-tenant database environments. In *International Congress on Big Data*, pages 246–253, 2015.

[14] Rohit Gupta, Sumit Kumar Bose, Srikanth Sundarrajan, Manogna Chebiyam, and Anirban Chakrabarti. A two stage heuristic algorithm for solving the server consolidation problem with item-item and bin-item incompatibility constraints. In *IEEE SCC*, pages 39–46, 2008.

[15] David S. Johnson, Alan J. Demers, Jeffrey D. Ullman, Michael R. Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3:256–278, 1974.

[16] Rachel King. Dannon finds amazon web services redshift, cheap, easy and fast. March 2014. http://blogs.wsj.com/cio/2014/03/31/dannon-finds-amazon-web-services-redshift-cheap-easy-and-fast/, Accessed: 2015-12-11.

[17] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. Towards multi-tenant performance slos. In *ICDE*, pages 702–713, 2012.

[18] Chung-Chieh Lee and Der-Tsai Lee. A simple online bin packing algorithm. *J. ACM*, 32:562–572, 1985.

[19] Ziyang Liu, Hakan Hacıgümüş, Hyun Jin Moon, Yun Chi, and Wang-Pin Hsiung. Pmax: Tenant placement in multitenant databases for profit maximization. In *EDBT*, pages 442–453, 2013.

[20] Maribel Lopez. Turning big data into business context with sap hana vora. September 2015. http://www.forbes.com/sites/maribellopez/2015/09/02/turning-big-data-into-business-context-with-sap-hana-vora/, Accessed: 2015-12-11.

[21] Hyun Jin Moon, Hakan Hacıgümüş, Yun Chi, and Wang-Pin Hsiung. Swat: A lightweight load balancing method for multitenant databases. In *EDBT*, pages 65–76, 2013.

[22] Jan Schaffner, Benjamin Eckart, Dean Jacobs, Christian Schwarz, Hasso Plattner, and Alexander Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, pages 1264–1275, 2011.

[23] Jan Schaffner and Tim Januschowski. Realistic tenant traces for enterprise dbaas. In *ICDE*, pages 29–35, 2013.

[24] Jan Schaffner, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael J. Franklin, and Dean Jacobs. Rtp: Robust tenant placement for elastic in-memory database clusters. In *SIGMOD*, pages 773–784, 2013.

[25] Peter W. Shor. The average-case analysis of some online algorithms for bin packing. *Combinatorica*, 6:179–200, 1986.

[26] Daniel Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28:202–208, 1985.

[27] Benjamin Speitkamp and Martin Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE T. Services Computing*, 3(4):266–278, 2010.

[28] Chandrasekar Subramanian, Arunchandar Vasan, and Anand Sivasubramaniam. Reducing data center power with server consolidation: Approximation and evaluation. In *HiPC*, pages 1–10, 2010.

[29] Paul Taylor. Let's get real about real-time data. September 2015. http://www.forbes.com/sites/sap/2015/09/28/lets-get-real-about-real-time-data/, Accessed: 2015-12-11.

APPENDICES

Appendix A

Details of the Robust Fit Interleaved Algorithm

To use Robust Fit Interleave algorithm in our comparisons, we had to fix a small issue with the algorithm which resulted in invalid packings in which some servers become overload. Here, we describe this issue and explain how we fixed it. For placing each replica, the algorithm searches for the server with the least left over space, while still being able to fit the replica and have enough space reserved in the event of a failure. In doing so, it assumes a load of $\mu \leq 1$, instead of 1.0, for bins when placing the first replica ($\mu \leq 1$). To be more precise, assume that a replica x is being placed on server S. Let L denote the load (total size of replicas) placed in S. Also, let *reserved* denote the total size of replicas shared between S and any other server; the algorithm maintains an empty space of size *reserved* to avoid overload in case of another server's failure. For placing the first replica, the algorithm ensures L + *reserved* $+x \leq 1 - \mu$; this way, it ensures that there is no overload in case of the failure of any of the *existing* servers. However, there will be a future server S' which hosts the partner of x. The shared load between S and S' will be x; so, the algorithm needs to also check that $L + 2x \leq 1 - \mu$. This is not captured in the original algorithm, and consequently it does not always reserve enough space in case of a failure. As an example, assume a sequence of tenants with loads $0.73, 0.43, 0.21$, and 0.34 arrives. Figure A.1 shows the packing of RFI with parameter $\mu = 0.85$ after placing replicas associated with these tenants. Now consider placing the next tenant with load 0.96, i.e., each replica x of the tenant has load, 0.48. The forth server has leftover of 0.66. Even after deducting $1 - \mu = 0.15$ from it, it has renaming capacity of 0.51, which is enough space to hold 0.48 load. So, the algorithm places the first replica of x in the forth server S_4. Assume the second replica is placed in a server S' (a new empty server in this case). The shared load between S_4 and S' will be 0.48, i.e., in case of failure of S', there will be an overload in S. However, if we perform the addition check, we

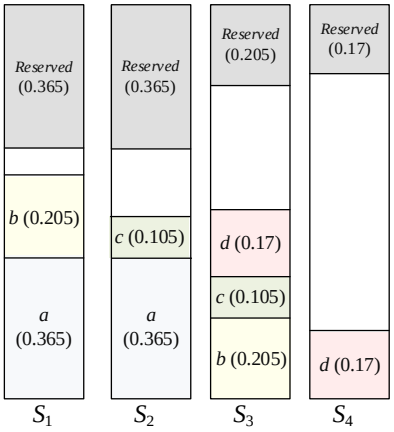


Figure A.1: Using RFI algorithm for placing a sequence of tenant replicas

would have $L + 2x = 0.17 + 0.96 > \mu$; hence the algorithm will open a new bin for x.

We propose a simple fix to RFI algorithm by performing two additional checks on top of the original algorithm. The first check is that the server the first replica is placed on needs to have enough space for the replica, plus enough reserved space for the partner replica. Secondly, we check that by placing the replica x on the second server, the shared load between the two servers after placement (which also includes x) is no more than their remaining space.

Data Structure 1: Data structure maintained for each server

```
Struct Server contains
    float load          // The current load of the server
    float reserved      // The reserved space to avoid overload in case of a server's failure
    list tenIds         // List of tenants (ids) placed on the server
    list tenLoads       // List of the loads of tenants on the server
    place (x , i ){     // Placing replica of size x with i to the server
      Add i to tenIds;
      Add x to tenLoads;
      load ← load+x;
    }
end
```

Algorithms 3 and 4 show the details of the algorithms, while Data Structure 1 shows the data fields maintained for each server. All aspects of these algorithms are similar to the original RFI

algorithm. Algorithm 4 places replicas one by one in the servers recommended by Algorithm 3, and updates the data fields of servers accordingly. The only difference is in the last *if* statement in Algorithm 4 in which the extra check is performed to ensure a valid packing.

Algorithm 3: findBestServer(): Find server with the least left over space that will not result in overload

input : *serverList*, servers sorted in decreasing order of loads;
x the load of the replica being placed;
firstServer, the server the first replica (partner of x) was placed on; and parameter μ
output: $bestServer$: a server from *serverList* to place x at. If no suitable server found then $bestServer$ will be $\emptyset$

// for the first replica we add on $1 - \mu$ load to improve interleaving
if *firstServer* $= \emptyset$ **then**
  serverCap $\leftarrow 1.0 - \mu$
else
  serverCap $\leftarrow 1.0$
// find the best fitting servers sorted by least space remaining first
fitableServers $\leftarrow$ list of servers like S ($S \neq$*firstServer*) such that S.*load*+ S.*reserved*$+x \leq$ *serverCap*
bestServer $\leftarrow \emptyset$
// Checking servers in sorted order to find the first server which suits x
for S *in fitableServers* **do**
  sharedLoad $\leftarrow$ Total load of replicas shared between S and *firstServer*
  // Check placing x in S does not cause an overflow in case
  // of failure of *firstServer* or S
  if S.*load* + *haredLoad* + $2x \leq$*serverCap* & *firstServer.load* + *sharedLoad* + $x \leq$*serverCap* **then**
    $bestServer \leftarrow S$
    break
end
return bestServer

Algorithm 4: Robust Fit Interleaved

input : A sequence $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ of items (clients), parameter μ: the server that the first replica is placed on can use only up to $\mu < 1$ load (instead of 1.0)
output: A fault-tolerant packing of σ

serverlist $\leftarrow$ set of server objects sorted by load+reserved
for $i \leftarrow 1$ *to* n **do**
  $x \leftarrow a_i$ // x is the current tenant
  $x_1, x_2 \leftarrow x/2$ // two replicas of x
  firstServer $\leftarrow \emptyset$
  for $r \leftarrow 1$ *to* 2 **do**
    // find a suitable server for x_r
    target $\leftarrow$ findBestServer(*serverList*,*firstServer*,i,σ_i,μ)
    if *target*$= \emptyset$ **then**
      // no server found, opening new bin
      target $\leftarrow$ new empty *server*
      Insert *target* to *serverList*
    end
    // Next, we insert x_r into *target* and update its data fields
    target.place(x_r,i)
    if *firstServer* $\neq \emptyset$ **then**
      shared $\leftarrow$ Total load of replicas shared between *target* and *firstServer*
      shared $\leftarrow$ *shared*$+x_r$
      if *shared* $>$ *firstServer.reserved* **then**
        firstServer.reserved $\leftarrow$ *shared*
      if *shared* $>$ *target.reserved* **then**
        target.reserved $\leftarrow$ *shared*
    end
    if *firstServer* $= \emptyset$ **then**
      firstServer $\leftarrow$ *target*
  end
end
return *serverList*

Appendix B

Conversion of the Server Consolidation Problem to a Mixed Integer Program

B.1 Integer Program Formulation

Server consolidation, as described above, can be formulated as a mixed Integer Program (IP). Here, we describe the IP formulation for $\gamma = 2$. This is used in our performance evaluation where we compare the CUBEFIT algorithm with the optimal solution achieved by solving the IP formulation. For $\gamma = 2$, we have the following simplified IP formulation for the problem based on the one from [24]:

$$\text{Minimize} \quad \sum_{i \in N} s_i \quad \text{subject to}$$

$$\sum_{i \in N} y_{t,i}^k = 1 \qquad \forall t \in T, \forall k \in \{1,2\} \quad (1) \qquad \text{(B.1)}$$

$$\sum_{k \in \{1,2\}} y_{t,i}^k \leq 1 \qquad \forall t \in T, \forall i \in N \quad (2) \qquad \text{(B.2)}$$

$$p_i = \max_{j \in N, j \neq i} \sum_{t \in T} \sum_{k,k' \in \{1,2\}} \frac{l(t)}{2} y_{t,i}^k y_{t,j}^{k'} \qquad \forall i \in N \quad (3) \qquad \text{(B.3)}$$

$$\sum_{t \in T} \sum_{k \in \{1,2\}} \frac{l(t)}{2} y_{t,i}^k + p_i \leq s_i \qquad \forall i \in N \quad (4) \qquad \text{(B.4)}$$

$$y_{t,i}^k \in \{0,1\} \qquad s_i \in \{0,1\} \qquad l_t \in (0,1]$$

In the above system, T and N denote the set of tenants and servers, respectively. The boolean variable s_i indicates whether the i^{th} server is active, i.e., whether at least one replica has been placed there. The boolean variable $y_{t,i}^k$ is 1 if tenant t's k^{th} replica is placed on the i^{th} server. Variable $l(t)$ denotes the load of tenant t. Constraint (B.1) ensures that each replica is assigned exactly once. Constraint (B.2) ensures that no two replicas of a tenant are on the same server. Equation (B.3) computes the worst case server failure for server i when there are two replicas for each tenant. Constraint (B.4) ensures the load of all the tenant replicas plus the maximum load directed to each server (in case of one server's failure) does not exceed the capacity of the server (1.0). Note that the load of each replica is half the load of its tenant because the load of the tenant is evenly distributed between the two replicas.

B.2 Conversion to Mixed Integer Program

This section provides the details for converting the minimization problem from Section B.1 into a mixed integer program (MIP). Equations (B.1) and (B.2) are straightforward to convert. While iterating the variables on the right hand side of the constraint, for each iteration, constraints of the left hand side are added to the solver.

Equations (B.3) and (B.4) cannot be directly implemented in MIP because the max function is unsupported. The max can be eliminated from Equation (B.3) by creating a constraint in (B.4) for all values inside the maximum instead of just the maximum value, p_i, given by Equation

(B.3). Secondly, a MIP does not support multiplication of two decision variables so we need to introduce a dummy variable to replace it ("Elimination of products of variables" [5]).

Below is the dummy variable introduced:

$$h_{t,i,j} = 1 \qquad \text{if tenant t is on server i and server j}$$

But now we have to introduce all these dummy constraints which ensure that $h_{t,i,j} = 1$ if and only if tenant t is on server i and server j.

$$h_{t,i,j} \leq \sum_{k \in R} y_{t,i}^{k} \qquad \forall t \in T, \forall i \in N, \forall j \in N : j \neq i \tag{B.5}$$

$$h_{t,i,j} \leq \sum_{k \in R} y_{t,j}^{k} \qquad \forall t \in T, \forall i \in N, \forall j \in N : j \neq i \tag{B.6}$$

$$h_{t,i,j} \geq \big(\sum_{k \in R} y_{t,i}^{k} + y_{t,j}^{k}\big) - 1 \qquad \forall t \in T, \forall i \in N, \forall j \in N : j \neq i \tag{B.7}$$

Notice that the sum in the parentheses in Equation (B.7) is 2 if servers i and j share tenant t. The sum is 1 if i or only j has tenant t, not both. Lastly it is 0 if neither server i and j have tenant t.

Equations (B.3) and (B.4) become the single equation:

$$\big(\sum_{t \in T} \sum_{k \in R} \frac{l(t)}{2} y_{t,i}^{k}\big) + \big(\sum_{t \in T} \frac{l(t)}{2} h_{t,i,j}\big) \leq s_i \qquad \forall i \in N, \forall j \in N : j \neq i \tag{B.8}$$

This equation is the same as Equation (B.3) subbed into p_i of Equation (B.4). The first sum is the load of the tenants on server i. The second sum is the load of the tenants on server j that failed that also have a replica on server i. Also notice that previously in Equation (B.4), there was only a constraint for each server. Instead, Equation (B.8) has a constraint for each server pair which is equivalent to the max from (B.3).

Appendix C

Small Improvement Due to Large Tenants

Lemma 3 *All server consolidation placement algorithms with replication factor 2 can place at most one replica with load between (1/3, 1/2] on a server.*

Proof of Lemma 3. We proceed by contradiction. Assume there exists a packing algorithm that can co-locate at least two tenant replicas of load $t_1, t_2 \in (\frac{1}{3}, \frac{1}{2})$ onto the same server s. Let t_o be the load from the tenants with the exception of tenants 1 or 2; we call these remaining tenants. Let r be the load reserved to protect against overload from server failure.

Consider the load of the server.

$$serverLoad = t_1 + t_2 + t_o + r$$

By ignoring the load of the remaining tenants, t_o, the load of the server must be at least or equal to the resulting equation.

$$serverLoad \geq t_1 + t_2 + r$$

Since we know that the load of the replicas of tenant 1 and 2 are strictly greater than 1/3:

$$serverLoad > 1/3 + 1/3 + r = 2/3 + r$$

Additionally, we must consider r, the reserved space needed to protect against overload from a server failure. There needs to be sufficient space reserved in case the second replicas of tenant

1 or 2 fail on a server different from s. As a result, we know we must reserve strictly greater than 1/3 capacity.

$$serverLoad > 2/3 + 1/3 = 1$$

The server load is greater than 1, which means the server is overloaded. This is a contradiction as we assumed that this algorithm satisfied the constraints of the server consolidation placement problem with replication factor 2. □

Appendix D

CUBEFIT Optimizations

Initial experiments revealed opportunities for improving the CUBEFIT algorithm. As a result, we optimized CUBEFIT by making two modifications. First, instead of grouping tiny tenants which are smaller than $\frac{1}{K+1}$ into multi-replicas and placing them in the α_K class, we treat all tiny tenants as if they were in the K^{th} class. Second, on the K^{th} class, we do slot maturation, instead of server maturation. In server maturation, the server has to fill all its slots before the left over space becomes available for mature fit. In slot maturation, once a slot is occupied, the slot's left over space becomes available for mature fit. To evaluate the optimization, we compare the optimized CUBEFIT to placing the multi-tenants in the α_K class and also the largest class, K. We use the same simulation configurations as in Chapter 5.3.

Figures D.1 and D.2 show the optimized CUBEFIT's improvement over placing multi-tenants into the α_K class. The optimized version outperforms the α_K version in all cases. Recall that the server reserves $\frac{1}{\alpha_K}$ space. As a result, small values of α_K waste a large portion of space on tenants that could be more efficiently packed in classes larger than α_K. To improve upon α_K, we place the multi-tenants in the largest class making the reserved space as small as possible.

Figures D.3 and D.4 show the optimized CUBEFIT's improvement over placing the multi-replicas into the largest class. Once again, the optimized version outperforms the largest class version in all cases. Optimized CUBEFIT handles the tiny tenants better by placing them in the same way as K class tenants and doing slot maturation instead of server maturation. Also, slots become mature faster than servers, allowing leftover space to be re-used sooner, improving the performance ever further when there are fewer tenants.

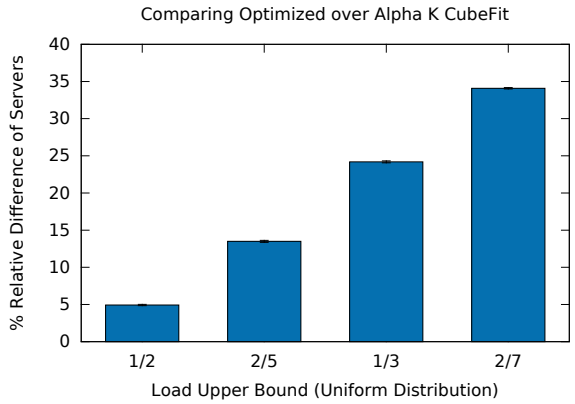


Figure D.1: % Relative difference of servers used by optimized and α_K versions of CubeFit using uniform distributions

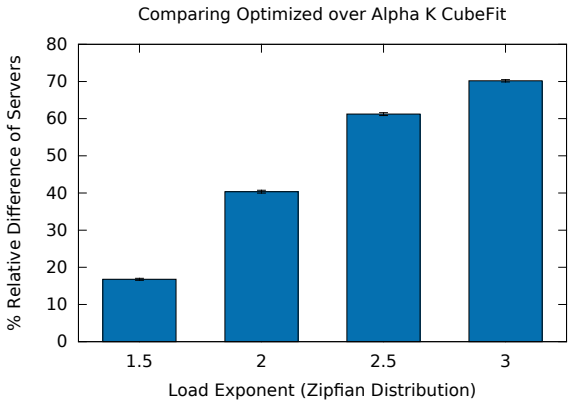


Figure D.2: % Relative difference of servers used by optimized and α_K versions of CubeFit using Zipfian distributions

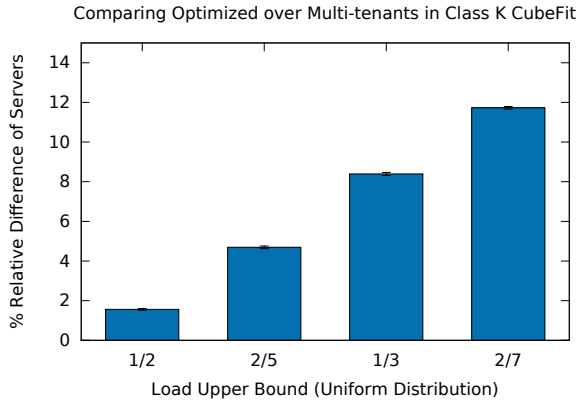


Figure D.3: % Relative difference of servers used by optimized versus multi-tenants in the last class versions of CubeFit using uniform distributions

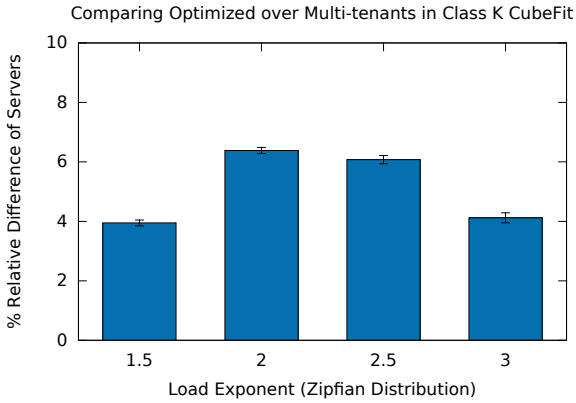


Figure D.4: % Relative difference of servers used by optimized versus multi-tenants in the last class versions of CubeFit using Zipfian distributions