2017 IEEE 37th International Conference on Distributed Computing Systems

# Robust Multi-Tenant Server Consolidation in the Cloud for Data Analytics Workloads

Joseph Mate
University of Waterloo
jmate@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
kdaudjee@uwaterloo.ca

Shahin Kamali
MIT CSAIL
skamali@mit.edu

***Abstract*—Server consolidation is the hosting of multiple tenants on a server machine. Given a sequence of data analytics tenant loads defined by the amount of resources that the tenants require and a service-level agreement (SLA) between the customer and the cloud service provider, significant cost savings can be achieved by consolidating multiple tenants. Since server machines can fail causing their tenants to become unavailable, service providers can place replicas of each tenant on multiple servers and reserve capacity to ensure that tenant failover will not result in overload on any remaining server. We present the CUBEFIT algorithm for server consolidation that reduces costs by utilizing fewer servers than existing approaches for data analytics workloads. Unlike existing consolidation algorithms, CUBEFIT can tolerate multiple server failures while ensuring that no server becomes overloaded. Through theoretical analysis and experimental evaluation, we show that CUBEFIT is superior to existing algorithms and produces near-optimal tenant allocation when the number of tenants is large. Through evaluation and deployment on a cluster of 73 machines as well as through simulation studies, we experimentally demonstrate the efficacy of CUBEFIT.**

## I. 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 hosting of analytic workloads is experiencing explosive growth. Cloud service providers, such as Amazon Web Services, 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. Service Level Agreements (SLAs) between clients and the service provider define the minimum performance requirement for cloud-hosted client applications. The objective of a service provider is to meet the SLA requirement and, at the same time, minimize the operational cost of providing service. There is generally a trade-off between performance as desired 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 important for green computing and saving on energy-related costs, which account for a significant portion of a data center's ongoing operational costs [3], [9].

In this paper, we present an efficient solution to the problem of server consolidation in the cloud for in-memory multi-tenant data analytics workloads described by the following requirements.

First, to meet the SLA, server consolidation should be performed in such a way that servers are not overloaded. Data centers usually have a large number of server machines providing significant resource capacity [12]. In this context, each tenant has a load defined as the minimum amount of in-memory server compute resources required by the tenant to meet its SLA [8], [10], [12], e.g., a tenant with higher query load places higher in-memory server load. If a server is overloaded, i.e., the total tenant load that it hosts exceeds its capacity, then the SLA 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. However, in practice, tenants appear dynamically, i.e., in an online manner. Thus, the second requirement is that each tenant needs to be assigned to a server without knowledge about forthcoming tenants.

The third requirement is dealing with failure of one or more server machines resulting in hosted tenants suffering from performance degradation or loss of availability. To address this issue, tenants are replicated[1] 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 tenant. The SLA 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.

### A. Contributions

In this paper, we consider a general model of server consolidation for data analytics workloads. We present CUBEFIT, an online algorithm for server consolidation in the cloud

[1]E.g., AWS provides the RDS service at multiple replication levels [1].

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.144

for multi-tenant workloads. Unlike prior work [12] that can maintain an SLA in the presence of only one server failure, CUBEFIT minimizes the number of servers used to host tenants while providing *robustness* by tolerating failure of *any* given number of servers for a particular SLA. In this context, robustness refers to the number of simultaneous server failures that do not result in violation of SLA.

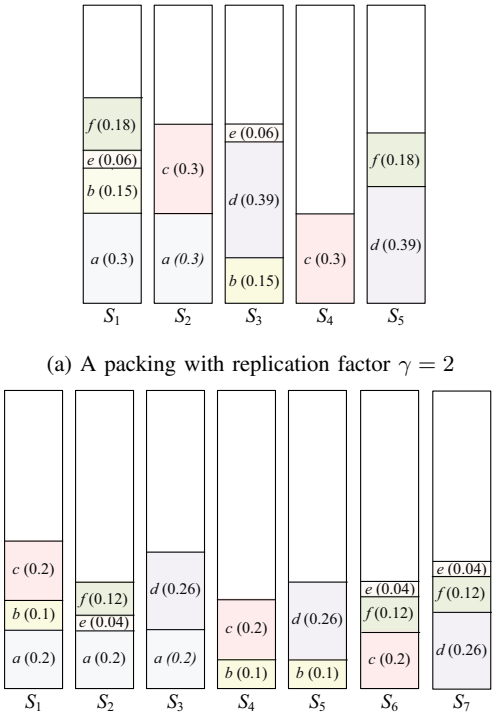
Our CUBEFIT algorithm is based on the idea of placing tenants of the same class into multi-dimensional "cubes" and consolidating smaller tenants into cubes formed by larger tenants which still 'fit' them. We demonstrate using theoretical analysis, real system experiments, and simulations that the CUBEFIT algorithm utilizes the least number of servers for hosting tenants while satisfying a given SLA.

Our system model and experiments address multi-tenant consolidation for the data analytics domain but we expect CUBEFIT and its analysis to apply to other types of server consolidation requiring robust solutions.

## II. PROBLEM

In this section, we formally define the robust tenant placement problem, extending and building on prior work such as [12]. 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 $x$, $\gamma$ replicas of the tenant are created, where $\gamma$ is a parameter of the problem and typically $\gamma \in \{2, 3\}$. To achieve load balancing, the load of a tenant is distributed among the replicas, i.e., a replica of size $x$ has a load $x/\gamma$. A consolidation algorithm needs to *place* these replicas on $\gamma$ different servers. Each replica might be placed on an existing server or the algorithm might *open* (allocate) a new server for it. To meet SLA requirements, the total load of replicas on each server should not be more than the unit capacity of the server.

When a server fails, the load associated with each replica hosted by the server is distributed among servers that host its remaining 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 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 robust 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 replica on $S_j$. To have a robust 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 the reserved capacity of $S_i$. Figure 1 provides an illustration.




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

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

Fig. 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 $0.6 + 0.3 \leq 1$ for $S_2$. Similarly, loads of $b$ and $e$ redirect 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 $a$ redirects to $S_3$, resulting in a total load of $0.46 + 2 \times 0.2 \leq 1$.

The server consolidation problem is closely related to the online bin packing problem in which 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.

## III. CUBEFIT 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 $K = 10$, while for smaller settings, it would be smaller, e.g., $K = 5$. Recall that $\gamma$ 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 $\tau$ (have *type* $\tau$), where $1 \leq \tau < K$. Note that the size of each replica is at most $1/\gamma$. The replicas with sizes in the range $(0, \frac{1}{K+\gamma-1}]$ belong to class $K$. Each bin also has a class (type) which is defined as the class of the first replica placed in the bin. A bin of class $\tau$ $(1 \leq \tau < K)$ is expected to receive $\tau$ replicas of the same class. More precisely, it has $\tau + \gamma - 1$ *slots*, each of size $1/(\tau + \gamma - 1)$, out of which $\tau$ slots are expected to be occupied by replicas of type $\tau$ and $\gamma - 1$ slots are reserved to be empty in anticipation of servers' failure. If $\tau$ slots of a bin of type $\tau$ become occupied, we say the bin is a *mature bin*. There might be empty, non-reserved space in a mature bin which the algorithm uses to place smaller replicas, i.e., replicas belonging to classes larger than $\tau$.

Let $(x_1, x_2, \ldots, x_\gamma)$ denote the $\gamma$ 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, in which 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 2 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. Recall that each bin of type $\tau$ $(1 \leq \tau \leq K)$ is partitioned into $\tau + \gamma - 1$ *slots* of size $1/(\tau + \gamma - 1)$. Out of these slots, $\gamma - 1$ slots are left empty. The other $\tau$ slots

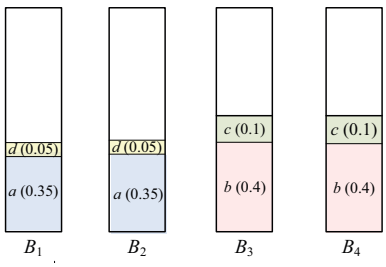



Fig. 2: An illustration of the first stage of the algorithm. There are two replicas per tenant ($\gamma = 2$). Consider sequence $\langle a, b, c, d\rangle$ of tenants. There will be four bins of class 1, opened by replicas of $a$ and $b$. After placing these replicas, the four bins become mature. When 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 $d$.

are each filled with one replica of type $\tau$. 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 $\gamma$ groups of bins, as in Fig. 3, for each type $\tau \leq K - 1$. Each group is formed by $\tau^{\gamma-1}$ bins of type $\tau$. The $\tau$ slots in these $\tau^{\gamma-1}$ bins can be arranged to form a cube of size $\tau$ in the $\gamma$-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 $cnt_\tau$ which is initially 0. After placing replicas associated with a tenant of type $\tau$ in the second stage of the algorithm, the counter $cnt_\tau$ is updated to $(cnt_\tau + 1) \mod \tau^\gamma$. Note that the value of $cnt_\tau$ is always in the range $[0, \tau^\gamma - 1]$, i.e., it can be encoded as a number of $\gamma$ digits in base $\tau$. Let $I_\tau$ indicate this number before placing replicas of tenant $x$ of type $\tau$. The algorithm places replicas of $x$ in the slots indicated by the $\gamma$ cyclic shift value of $I_\tau$. In other words, the $\gamma$ digits of $I_\tau$ are used to address the slot at which the replica is placed at. For example, if $\tau = 3$, $\gamma = 2$, and $I_3 = (21)_3$, 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, $I_3$ is updated to $(22)_3$. As another example, if $\tau = 3$, $\gamma = 3$, and $I_3 = (001)_3$, the first replica of $x$ is placed at slot $(0, 0, 1)$ of the first 3-dimensional cube, the second replica at slot $(1, 0, 0)$ of the second cube, and the third replica at $(0, 1, 0)$ of the third cube. After this, $I_3$ is updated to $(002)_3$. Figure 3 provides an illustration and pseudocode for CUBEFIT is shown in Algorithm 1, which considers tenant loads larger than $\frac{1}{K+\gamma-1}$.

Since each replica of a given tenant is placed in a different dimension in each of $\gamma$ 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 $j$th 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 $\tau$ are placed in both $B_1$ and $B_2$. Since replicas of $x$ and $y$ are placed in $B_1$, the value of $I_\tau$ 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 $I_{\tau,x,B_2}$ and $I_{\tau,y,B_2}$ also differ in only the least significant digit. Also, $I_{\tau,x,B_1}$ and $I_{\tau,x,B_2}$ are cycle shifts of one another as well as $I_{\tau,y,B_1}$ and $I_{\tau,y,B_2}$. This implies that $I_{\tau,x,B_2}$ and $I_{\tau,y,B_2}$ differ in some digit other than the least significant. This contradicts the previous fact that $I_{\tau,x,B_2}$ and $I_{\tau,y,B_2}$ differ only in the least significant digit. □

We have described how to place tenants up to class $K - 1$. To place the tenants in class $K$ in the second stage of the algo-

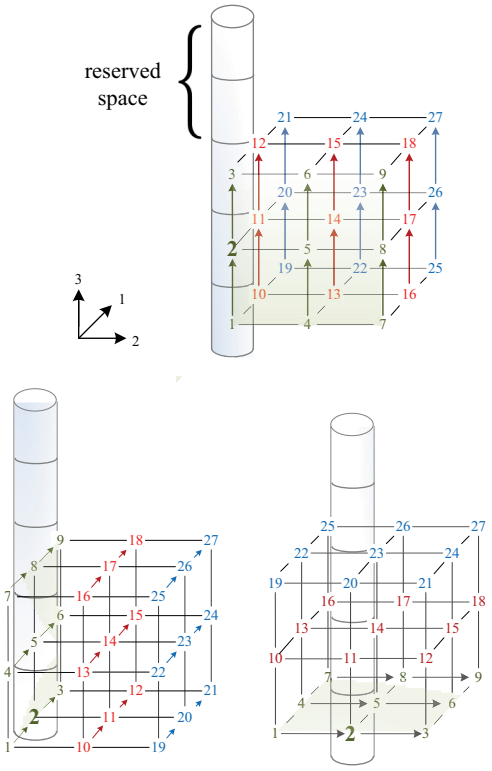



Fig. 3: The idea behind CUBEFIT for placing replicas of the same type. In this example, there are three replicas per tenant ($\gamma = 3$), each placed in one of the three cubes. Also, replicas have type $\tau = 3$, i.e., the load of each is in the range $(1/6, 1/5]$. Tenants are labelled from 1 to 27. Each cube includes 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 $(1, 0, 0)$ of the second cube, and $(0, 1, 0)$ of the third cube. Any pair of the servers associated with these slots share only tenant $x = 2$.

rithm, consider the largest integer $\alpha_K$ so that $\alpha_K^2 + \alpha_K < K$. This ensures that $\frac{1}{\alpha_K} - \frac{1}{\alpha_K+1} > \frac{1}{K}$; consequently, the algorithm can group sets 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 $\gamma$ *active* multi-replicas at each stage of the algorithm, each associated with one of the $\gamma$ cubes (initially, they are empty sets of replicas). For placing the $i$th replica of a tenant of class $K$ $(1 \le i \le \gamma)$, the algorithm checks whether adding the replica to the $i$th 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 $\alpha_K - \gamma + 1$. Throughout the paper, 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, $\gamma$ replicas of $x$ are placed in $\gamma$ different cubes as described above.

Using Lemma 1, we prove the validity of CUBEFIT.

**Theorem 1.** *In the schemes resulting from* CUBEFIT *no bin is overloaded in case of failure of at most $\gamma - 1$ servers.*

**Proof of Theorem 1.** Consider an arbitrary bin $B^*$ of type $\tau$ in the packing of CUBEFIT. Also, consider an arbitrary set $S = \{B_1, \ldots, B_{\gamma-1}\}$ 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 $\tau$. 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 first stage of the algorithm (i.e., placed after $B^*$ became mature) are ensured to m-fit in $B^*$. This implies that the extra load resulting from these replicas do not cause an overflow. □

### A. Worst-case Analysis

In this section, we provide upper bounds for the competitive ratio of CUBEFIT, which reflect the worst-case behavior. Recall that there are $\gamma$ replicas for each tenant $x$ that we denote with $x_1, \ldots, x_\gamma$, and each replica $x_j$ $(1 \le j \le \gamma)$ has a load $x/\gamma$. To prove an upper bound of $r$ for the 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 is at least 1. (II) The total weight of replicas in each server in an optimal packing scheme is at most equal to a constant $r$. The above statements imply a competitive ratio of at most $r$ for CUBEFIT. This is because (I) implies that CUBEFIT$(\sigma) \le W(\sigma)$ and (II) implies OPT$(\sigma) \ge W(\sigma)/r$ where $W(\sigma)$ denotes the total weight of all replicas of all tenants in $\sigma$. This gives us Theorem 2. Note that no online algorithm can have a competitive ratio better than 1.42 [4].

**Theorem 2.** *The competitive ratio of* CUBEFIT *with replication factor $\gamma = 2$ and $\gamma = 3$ approach 1.59 and 1.625 respectively for large values of $K$.*

**Proof of Theorem 2.** We define the weight of each replica $x$ in the following manner. If $x \in (1/(i+1), 1/i]$ for some positive integer $i$ $(\gamma \le i \le K + \gamma)$, then the weight of $x$ will be $1/(i - \gamma + 1)$. Recall that $x_j$ belongs to class $\tau = i - \gamma + 1$

**Algorithm 1:** CubeFit Algorithm

**input :** An online sequence $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ of tenant loads, positive integers $K$ (no. of classes), and $\gamma$ (no. of replicas per tenant)

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

```
mature-bins ← empty set of bins
for τ ← 1 to K do
    Cnt_τ ← 0    // a counter used in the second stage
    Group_τ^γ ← array of τ^{γ−1} empty bins
    // A γ-dimensional cube of τ^γ slots of type τ.
end
// 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 ← ⌊1/x⌋ − γ        // type of the replicas
        for j ← 1 to γ do
            P ← The I_γth slot of the bin B, where B is the bin at index (I_1, I_2, ..., I_{γ−1})_τ of Group_τ^j
            Place x_j in slot P
            mature-spot[τ] ← mature-spot[τ] ∪ {P}.
            if B includes τ replicas of type τ then
                mature-bins ← mature-bins ∪ {B}
            I_τ ← cyclic shift-right of 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
```

in this case and $i - \gamma + 1$ replicas of this type are placed in each bin of type $\tau$ (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 $\gamma$ 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 is $\frac{1}{i}$, 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 $\epsilon$. 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 $0 < M \leq m_T$. We maximize the total weight of replicas while satisfying the empty space condition, by solving 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:

$$regularSize + tinySize + reservedSpace \leq 1$$

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

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

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 $K$. Solving the program for $\gamma = 2, 3$ and large values of $K$ completes the proof. □

## IV. System Model

In our system model, each server hosts multiple tenants and has a data store which is shared between tenants that it hosts, as shown in Figure 4. A tenant's load is generated by a 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 its hosted tenants. We use a shared data system multi-tenant model in which each tenant resides as a data instance on the single data system running on a server. Studies have shown that a shared data system environment has better performance than virtualization [2] and several multi-tenant environments have used it [2], [10]. As in [12], the analytic workload of a tenant is shared between its $\gamma$ replicas.

To demonstrate the placement algorithm's viability, we use a practical load model that has also been used in [11], [12] in which the in-memory load that the tenant places on a server is input to the algorithm. As shown in [11], load from multiple tenants on the same server is additive and a linear model accurately predicts latency.

We model tenant utilization using a linear relationship between a tenant's properties [11]. In our experiments the load of a tenant is defined as $\delta c + \beta$ where $c$ is the number of clients, $\delta$ is the amount of capacity each client takes up on the server, and $\beta$ is the overhead each tenant places on the server. This function produces a load value larger than 1.0 when the server's capacity is over-utilized. The value of $\delta$ and $\beta$ are specific to a hardware configuration but can be generalized or made specific for other types of configurations. We determined the values for $\delta$ and $\beta$ by running varying numbers of clients distributed over various numbers of tenants on an Intel Xeon 2.1 GHz machine with 12 cores and 32GB of memory. We used the TPC-H benchmark as our analytics workload of read-mostly queries while supporting writes which are simply executed against all replicas to provide consistency.

We created a PostgreSQL database instance to hold the data of each tenant executing the TPC-H benchmark queries. Each tenant runs multiple concurrent client threads that independently iterate through the TPC-H queries submitting them to the PostgreSQL system on the tenant's host machine.

Some client-tenant configurations resulted in the SLA being violated while others met 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 $\delta$ and $\beta$. To focus on tail latencies, we set the SLA to be 5 seconds (at the 99th percentile), which corresponds to a load of 1.0 per the load function described above.

When a server fails, clients of tenants hosted on it execute their queries on the remaining tenant replicas on other servers.

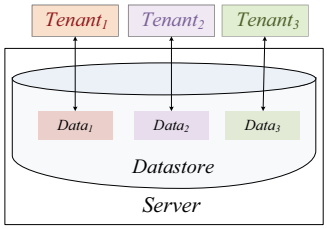



Fig. 4: Shared data system model: Tenants 1, 2 and 3 share the data store on the server.

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.

## V. Performance Evaluation

In this section, we present our experimental evaluation of the CubeFit algorithm for server consolidation. Our comparison is twofold: first, we implement CubeFit on a real system comprising of 73 machines and compare with the RFI algorithm using the TPC-H benchmark.

Our experiments in the upcoming sections will serve to demonstrate that in the presence of one and two server machine failures that result in the shifting of load to replicated servers, CubeFit meets the 99th percentile SLA. We demonstrate that at cloud scale, CubeFit uses significantly fewer servers, generating dollar savings for cloud service providers.

RFI [12] 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 $\mu$ 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.

### A. System Setup

We implemented the system model from Section IV to evaluate the performance of CubeFit and RFI on a cluster of 73 Intel Xeon server machines connected over 10 gigabit Ethernet. We use 69 servers to host the tenants and their data. The remaining 4 are used to generate the client query load for tenants. We scaled the TPC-H workload to have 95% read queries and 5% update queries from the benchmark as our focus is on analytic workloads though we also support database writes. Each tenant starts out with 100 MB of data, which amounts to having up to 10 GB of tenant data in the memory of a machine.

As in [12], we focus on tail (99th percentile) latency for our SLA. For the TPC-H workload, we chose an SLA of 5 seconds that was derived empirically and corresponds to unit server

load. Per Section IV, 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.

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.[2] Our measurements were obtained using CUBEFIT configured with 5 classes for both the uniform experiment and the zipfian experiment. The number of classes is a configurable parameter that can be used to tune the algorithm. For example, as the number of servers is increased, increasing the number of classes will yield better performance. Empirically, tenants in the largest class (with replica load between 0 and $\frac{1}{K+\gamma-1}$) are best placed in class K-1 (instead of $\alpha_K$). Then, as an optimization, the first stage of the algorithm re-uses the left over space of server slots in the K-1 class instead of maturing the server. Unless otherwise mentioned, all of our experiments were run with both of the distributions and 5 classes. For RFI, we used $\mu$ equal to 0.85 as recommended in [12].

### B. Server Failures

We conducted experiments on our cluster of machines to study how CUBEFIT and RFI respond to server failures. We studied CUBEFIT's behavior when there are two replicas per tenant that can protect against one server failure and three replicas per tenant that can protect against two server failures. Recall that the RFI algorithm from [12] cannot protect against multiple server failures.

We keep adding tenants 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 IV.

Figure 5 shows the relationship between the number of server failures and latency. With one server failure, the 99th percentile latency results demonstrate that none of the CUBEFIT configurations violate SLA (shown by the horizontal red line).

For the two-failure scenario, only CUBEFIT with 3 replicas stays (well) within the SLA (delineated by the red line) while the other algorithms violate the SLA. The CUBEFIT configuration of 3 replicas that protects against an overload from two server failures resulted in a 99th percentile latency of 4.27 seconds for the uniform distribution and 4.19 seconds for the zipfian distribution. CUBEFIT is effective in protecting against the failure of 2 servers while the algorithm flexibly allows it to be configured to protect generally against $k$ server failures.

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

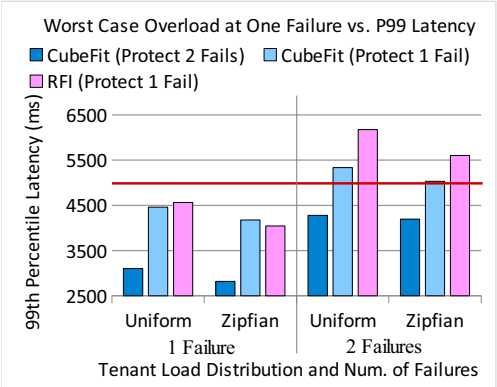



Fig. 5: 99th percentile latency of CUBEFIT and RFI with the worst case overload of 1 failure for uniform and zipfian distribution of tenants

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. CUBEFIT with 3 replicas is able to provide more protection against overload by trading off consolidation for the additional protection.

### C. Server Consolidation

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 with thousands of servers. To realize this level of performance experimentally, we study the large-scale behavior of CUBEFIT through simulation experiments. We use the model described in Section IV on which to run these algorithms.

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 Figure 6, with 95% confidence intervals as whiskers on the bars in this figure.

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. Thus, this relative

difference metric allows us to compute the percentage savings in number of servers used by CUBEFIT over RFI. 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. 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, the number of clients our cluster can support.

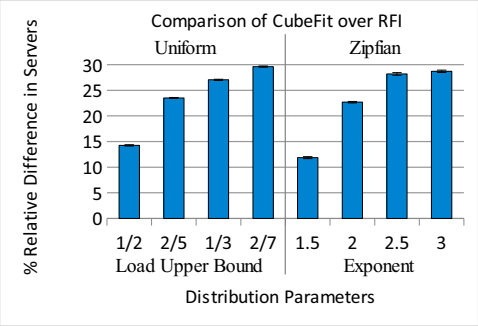



Fig. 6: Percentage savings of servers (relative difference) used by CubeFit over RFI for various distributions

In Figure 6, CUBEFIT performs better than RFI across-the-board. The gains amount to about 30% fewer machines utilized by CUBEFIT. 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 yearly cost savings for the uniform and zipfian distributions using 50,000 tenants. To compute these costs, we use a cost of $0.822 per hour per Amazon EC2's c4.4xlarge machine instances, which are similar in system resources to the machines we used to derive the system model (from Section IV) on which our simulations are based. As Table I shows, for continuous server operation, performing server consolidation in the cloud using CUBEFIT can generate substantial yearly cost savings for cloud service providers.

| Distribution | RFI Servers | CubeFit Saved | Dollar Savings |
|---|---|---|---|
| Uniform | 10,951 | 2,506 | 18,045,004 |
| Zipfian | 2,218 | 496 | 3,571,557 |

TABLE I: Yearly cost savings of CUBEFIT over RFI

## VI. RELATED WORK

The RTP algorithm [12] does not protect from multiple server failures while CUBEFIT is superior performance wise and protects tenants against the failure of multiple servers.

The remaining related work do not protect servers from becoming overloaded due to failure of other servers. For example, [6] focuses on handling situations where it is not possible to accurately estimate server utilization of tenants while we focus on preventing servers from becoming overloaded due to taking on the load of failed servers.

Kairos analyzes tenants' resource usage [2] and minimizes the number of servers on which to place them by using an optimization algorithm similar to CPLEX but does not provide fault tolerant server consolidation. PMAX [8] considers the cost of SLO violations besides the cost of servers using a modified version of the best fit bin packing algorithm to approximate a solution. In contrast, CUBEFIT ensures there are no load violations, thereby avoiding performance degradation.

Lang et al. propose to use an algorithm [7] to search for a mixture of tenant classes on the machine type that gives the lowest cost but they do not provide for fault tolerance. SWAT performs load balancing and load leveling by swapping the servers that maintain primary and secondary replicas [10] but does not consider the efficient online packing of tenants onto servers. Delphi-Pythia uses a machine learning algorithm to determine placement of tenants on a server [5] but unlike CUBEFIT, their approach does not consider fault tolerant server consolidation.

## VII. CONCLUSION

We showed that CUBEFIT can effectively protect against multiple server failures, which none of the previous proposals can deliver. Our evaluation through theoretical analysis, system experiments and simulations show that CUBEFIT is the best available choice for robust online multi-tenant server consolidation for data analytics workloads and can generate significant cost savings over existing approaches.

## REFERENCES


[1] DB instance replication. http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.Replication.html.

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

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

[4] K. Daudjee, S. Kamali, and A. López-Ortiz. On the online fault-tolerant server consolidation problem. In *SPAA*, 2014.

[5] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *SIGMOD*, 2013.

[6] A. Floratou and J. M. Patel. Replica placement in multi-tenant database environments. In *International Congress on Big Data*, 2015.

[7] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance SLOs. In *ICDE*, 2012.

[8] Z. Liu, H. Hacıgümüş, H. J. Moon, Y. Chi, and W.-P. Hsiung. PMAX: Tenant placement in multitenant databases for profit maximization. In *EDBT*, 2013.

[9] P. Mckenna. Can we stop the internet destroying our planet? *New Scientist*, Jan. 2008.

[10] H. Moon, H. Hacıgümüş, Y. Chi, and W.-P. Hsiung. SWAT: A light-weight load balancing method for multitenant databases. In *EDBT*, 2013.

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

[12] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. J. Franklin, and D. Jacobs. RTP: Robust tenant placement for elastic in-memory database clusters. In *SIGMOD*, 2013.