SWAT: A Lightweight Load Balancing Method for Multitenant Databases

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

ABSTRACT

Multitenant databases achieve cost efficiency through the consolidation of multiple small tenants. However, performance isolation is an inherent problem in multitenant databases due to resource sharing among the tenants. That is, a bursty workload from a co-located tenant, i.e., a noisy neighbor, may affect the performance of the other tenants sharing the same system resources. We address this issue by using a load balancing method that is based on database replica swap. Unlike the traditional data migration-based load balancing, replica swap-based load balancing does not incur data movement, which makes it highly resource- and time-efficient. We propose a novel method of choosing which tenants should be subject to swaps. Our experimental results show that swap-based load balancing effectively reduces the number of SLA violations, which is the main performance metric we choose.

1. INTRODUCTION

Cloud computing has revolutionized the IT industry, with the promise of on-demand infrastructure. Cloud service providers often consolidate small or time-varying workloads on a shared hardware to achieve economies of scale. Database services in the cloud [1, 2] have successfully adopted this strategy using multitenant databases, which has shown to achieve the consolidation ratio of 6:1 to 17:1 [3]. Because of such a potential, multitenant DBs have received a lot of interest from the database research community [4, 5, 6, 7, 8, 9, 10, 3].

Multitenant databases, however, have an inherent problem, namely the lack of performance isolation. When a tenant receives an increased workload, either temporarily or permanently, the neighbor tenants within the same server will suffer from the increased total workload [11]. There can be many causes for the increased workload, where some examples include: i) the growth of a company, leading to a permanent traffic growth, ii) predicted infrequent traffic changes of a tenant, e.g., the bursty query traffic at a Web site dedicated to the World Cup, iii) predicted frequent traffic changes, such as daily or weekly workload pattern of a company, iv) unpredicted traffic spikes by a flash crowd [12], or any combination of these. No matter what the causes or the patterns are, the impact of such overloading can be highly damaging: neighbors of a noisy tenant can immediately see violations on their performance Service Level Agreements (SLAs), and in a severe case, the server and all tenants therein may become completely unresponsive. Hence, load balancing is an important feature to minimize the impact of a heavily loaded tenant on the other co-located tenants[13].

In this paper, we consider a load balancing technique to address this overload problem, which is a major cause of increased SLA violations in many real systems. While load balancing is a popular technique used in general server workload management, here we propose a new method targeted for replicated multitenant databases. To explain our method, we first introduce *database replica swap*, which is used as a building block in our method. In most of today's database deployments in cloud oriented data center environments, such as [1, 2], databases come with one or more secondary replicas for fault tolerance and high availability purposes. Often times, primary replicas[1] serve both read and write queries and secondary replicas receive the updates relayed from the primary replica for the purpose of fault tolerance. Hence the primary replicas receive a larger amount of workload compared to the secondary replicas: in our experiment environment described in Section 4, we observe that a read-only workload incurs zero load on the secondary replicas and a write-only workload incurs about one fourth of the primary's I/O on the secondary. [1, 2] suggests to leverage this difference for workload migration: by swapping the roles of the primary and the secondary replicas, workload can be effectively moved from the primary to the secondary. This is typically fast and lightweight as it involve no data movement. Using this replica swap as a basic operator, we propose a load balancing method, *SWAT*[2], which finds a subset of tenants that should be subject to replica swap to achieve the desired load balancing effect across the system.

We acknowledge that our load balancing method does not always guarantee to eliminate all types of overloads. It is

*Currently with Google

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

[1]We interchangeably use *primary replica* and *master* in this paper, and also *secondary replica* and *slave*.
[2]Swap-based load balancing method.

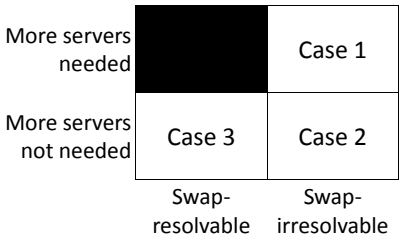


Figure 1: Overload Resolution Problem Space

known that the tenant workloads may change in various ways, such as increased/decreased load, query distribution, access patterns, and data distribution. Real systems employ different methods to monitor and resolve overload situations due to diverse reasons. Here, we present a lightweight and effective method that take advantage of a feature that already exists in most shared-nothing architectures, namely; replicated databases[14, 13]. The method can be used before resorting to heavier weight solutions, such as migration, which is unavoidable in some cases. As we show in the experiments and also observe in the implementation of the method in our real products, the SWAT method works very effectively in numerous situations.

Figure 1 shows the overload resolution problem space. The top row indicates the overload problem instances where additional servers are needed to eliminate the overload, mainly because the total workload is too high compared to the existing cluster capacity. We need a capacity increase, followed by tenant data migration into the new servers. The bottom row refers to the cases where overload can be removed simply by re-balancing the load without a new server. The left column, denoted as *swap-resolvable*, refers to the problem instances where a set of tenant replica swaps can remove overload without any tenant data migration. The right column, denoted as *swap-irresolvable*, refers to the cases where we cannot achieve overload removal with replica swap alone and some tenant data migration is needed to attain it. In this problem space, the goal of SWAT is as follows. Given Case 3, it aims at quickly removing all overloaded servers, or hotspots, through the techniques called *Hotspot Elimination* and *Load Leveling*. Given Case 1 and Case 2, where overload cannot be completely removed without data migration, SWAT aims at minimizing the amount of overload through the technique called *Hotspot Mitigation*, so that the overload becomes less severe until other solutions, such as migration or new capacity addition, are applied for the complete resolution of overload. Hence, we present SWAT as a complimentary technique with migration and capacity planning for overload resolution.

Toward this goal, we make the following contributions in this paper:

Replica swap-based workload migration Previous works [1, 2] have suggested an idea of using replica swap for workload migration, but the details have not been explored. In this paper, we design, implement, and evaluate replica swap for the purpose of workload migration. We also propose a technique for detecting and filtering out slow swaps, which makes the overall load balancing faster.

Swap-based load balancing We present a load balancing method, SWAT, which chooses a subset of tenants to swap to achieve a desired load balancing. SWAT tries to

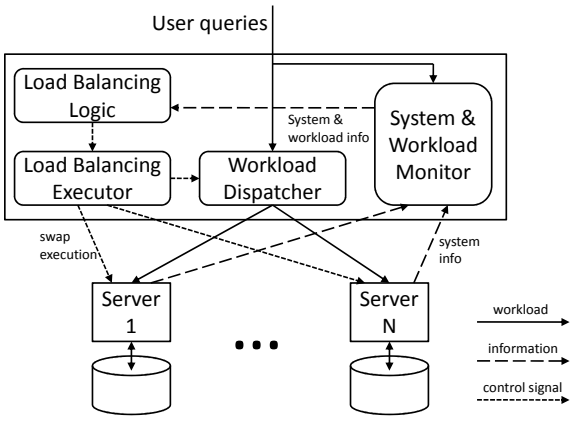


Figure 2: System Architecture

eliminate hotspots, if possible, or minimize it otherwise. We formalize this as three different integer linear programming (ILP) problems and show how SWAT combines the three subproblems into a single framework. We also analyze the scalability characteristics of the ILP solution.

Swap sequencing and parallelization Once we found a set of tenants to swap, we need to execute them in a certain order. This order needs to be carefully designed, otherwise it may create temporary overloads during the execution. Also, it is desired to execute some swaps in parallel when possible. We develop a swap sequencing and parallelization algorithm that minimizes temporary overload during the swap execution using the right level of parallel swap executions.

Evaluation We experimentally evaluate the effectiveness of SWAT, using synthetic and real-world traces under various workload configurations.

2. SYSTEM MODELING

Figure 2 shows the system architecture. A user workload arrives at the middleware layer, where a workload dispatcher routes it to the right DB server, which employ a shared-nothing architecture [13]. The system and workload monitor constantly observes the workload level for each tenant and each server and sends information to the load balancing logic module, which periodically runs the load balancing algorithms that we present in this paper and sends a sequence of swap operators to be executed to the load balancing executor. The executor runs the given swap operators sequentially or in parallel as specified, achieving the balanced load.

2.1 Multitenancy

In recent years, various multitenant architecture options have been explored, including private virtual machines, private instances, private databases, private tables, and shared tables [6, 15, 3]. While each option has pros and cons, we focus on the private instance model, currently used in many cloud offerings such as Amazon RDS[16]. We consider the system model where each tenant's data size and query workload can be served within a single server, similar to [1, 5]. Note that our system can also serve a bigger multi-server tenant, as long as its data and workload are partitioned across multiple servers so that all queries are served within a single server. In this case, each partition can be considered as an independent tenant and the techniques described here

N	Number of tenants
M	Number of servers
T_i	i-th tenant
T_R	Tenant of replica R
$R_i/R_i^p/R_i^s$	Any/primary/secondary replica of T_i
$\mathbf{R_{S_j}}$	The set of all replicas at server S_j
S_j	j-th server
S_R	Server location of a replica R
$L_{i,r}/L_{i,r}^p/L_{i,r}^s$	Load of $R_i/R_i^p/R_i^s$ on resource r, e.g., CPU, I/O. (Note that we omit r when the discussion holds for any given resource type.)
L_{TH}	Hotspot threshold
x_i	Binary variable for swapping T_i (1: swap, 0: no swap)

Table 1: Notation

can be applied.

2.2 Replication

In our system, similar to [1], each tenant has one primary replica and one or more secondary, asynchronous replicas for the purpose of fault tolerance. The primary replica serves all the queries (both read and write queries), while the secondary replica executes the update logs relayed from the primary replica, in an asynchronous manner. As in [1], we do not use secondary replicas for read query answering, to provide strong consistency. This replicated database model, and its close variations, are widely used in many commercial settings and well-studied in the literature, including transactional and availably properties during switching from the primary to a secondary in the case of failures. We do not make any additional assumptions nor modifications to already implemented product features and protocols in our system.

2.3 Workload

We define the load of a tenant as the amount of server resources needed to serve the tenant's workload, as a percentage of the server capacity. The load information for each tenant replica to be given as an input to the problem.

Table 1 shows the important system parameters used in this paper[3]. Some of important specifications are as follows. First, $N \geq M$ as described above. Second, for a given tenant T_i, its primary replica's load is always greater than or equal to its secondary replica's load, i.e., $L_i^p \geq L_i^s$. Note that $L_i^s = 0$ when T_i has no write query workload.

In this paper, we use a linear additive model for individual resource loads as in [5], e.g., if we co-locate two tenants with CPU loads of 20% and 30%, they consume 50% together on a single machine. We acknowledge that this model is reliable for CPU, but less so for I/O [3]. However, additive model for I/O is successfully used as a reasonable approximation in numerous systems[5], mainly due to the lack of a widely applicable model for I/O behavior. We also find that it works well for the load balancing purpose as shown in our experimental study.

2.4 Hotspot Threshold

To determine system capacity, or hotspot threshold, we use a queueing-theoretic approach: we run a controlled experiment involving a single server, with varying load levels. As we keep increasing the arrival rate, we find a critical point where the query response time explodes, and we use this load level as the hotspot threshold. The corresponding point effectively indicates the overload situation where the queries arrive at the server faster than the speed the server can process them. This method gave us a reasonably accurate information about the system capacity, compared to the one obtained by CPU/I/O utilization report of operating system tools, which were often inaccurate and misleading.

[3]Here, we give the notations with a single secondary replica to keep the formulations concise. It is straightforward to consider multiple secondary replicas, which we considered and tested in the experiments sections.

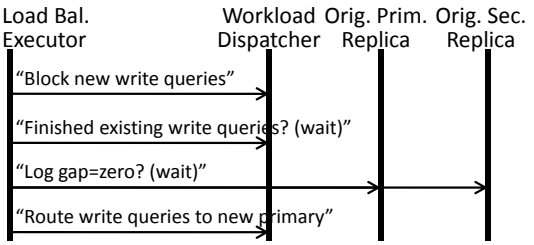


Figure 3: Swap Procedure

3. SWAP-BASED LOAD BALANCING

In this section, we first describe the replica swap operator. We then present a swap-based multitenant DB load balancing method, SWAT. SWAT consists of two subcomponents: i) finding the optimal swap set that eliminates hotspots, and ii) finding the sequencing and parallelization of swap executions.

3.1 Replica Swap Operator

Given a tenant that has a primary replica and a secondary replica, the replica swap involves the following steps, outlined in Figure 3. The middleware layer first temporarily holds the incoming write workloads in its per-tenant queue, and allows the write queries running in the DBMS to finish. After that, it waits until the relay log at the primary replica (i.e., *asynchronous delta*) propagates to the secondary and two replicas are synchronized. This may take varying amount of time, depending on the log gap between two replicas (see below) and the system utilization at the two nodes, while it is quite fast in general from our experimental study. Note that these two steps ensure the correctness of transactions. Finally, the roles of the primary and the secondary replicas are swapped and the workload dispatcher starts to send the read and write queries waiting in the queue to the new primary replica, which used to be a secondary replica before the swap. The new secondary replica starts to fetch log records from the new primary replica in an asynchronous manner. Below we briefly discuss some considerations with the SWAP protocol.

Primary-Secondary Log Gap Limit: Under a write-intensive workload, a secondary replica may not be able to catch up to the primary replica, especially when the secondary replica is located in an overloaded server and cannot catch up to the speed of the primary replica's write query executions. In this case, the *log gap* may keep increasing between the primary and the secondary.

To address this, we first check the log gap of all secondary replicas against their primary, and eliminate those with log gaps higher than a specific threshold from the swap candidate list. We parallelize these checks for tenants to minimize the latency. The load balancing algorithm then uses the tenants that pass the test for finding the load balance solution. While this screening reduces the flexibility of the

load balancing algorithm, it does not seem to affect the load balancing quality much, for the following reason: from our observations, the tenants with high log gaps often have their second replicas in the overloaded servers, which means that swapping those tenants does not directly reduce the load levels of the overloaded servers.

Tenant Resource Usage Estimation: Estimating the exact resource usage of tenants in all placements is an extremely difficult problem[3], mainly due to unpredictable interactions among different sets of tenants in servers. However, it is important to recall that our problem is not the general tenant placement problem. The only options we have to place the primary of the tenant database is one of the secondary replica locations, which is a relatively very small number compared to the general case. Therefore, essentially we need to pick the relatively better secondary replica location, which is much more tractable and error-tolerant problem.

Availability During the Swap: As we stated earlier, we use standard primary-secondary asynchronous database replication with specific database product (MySQL) features without any additional assumptions or modifications. The swap protocol is implemented as an ordinary handover by using those features. Therefore any failure in the primary or in the secondaries is handled by following the standard protocols implemented in the product. The implementation of this setting and failure handling are available in all major database products.

3.2 Finding Optimal Swap Set

As mentioned above, SWAT is composed of two steps, and here we discuss the first one, where it seeks to find the subset of tenants for replica swap that achieves the desired load balancing effect. The solution of this step, if any, is passed to the second step where the execution order of the found swaps are determined. The overall flow is shown in Figure 4.

We divide the swap set finding problem into three subproblems and formally define each of three subproblems, and then combine them within a global framework as in Figure 4. First, we attempt *load leveling* where we seek to eliminate hotspots *and* balance the load, if possible. The success of load leveling is the best case of SWAT, so the algorithm terminates successfully and return the found swap set. When it is not possible to balance the load, but possible to eliminate all hotspots, SWAT tries the second component, *hotspot elimination*, where it tries to simply eliminate overloads at all servers, without trying to balance the load evenly. If this succeeds, SWAT again successfully terminates and returns the solution. When both of the first two components fail to find a solution, it means that there exists no solution for swap-based hotspot elimination, i.e., case 1 or 2 in Figure 1. In such cases, SWAT attempts its best effort solution, which is *hotspot mitigation*, where it tries to *minimize* the overload, rather than eliminating it.

In the following, we discuss the details of individual components and show how each component is formulated as an integer linear programming problem. Note that we first present hotspot elimination, which is most straightforward and ideal for an illustration purpose. We then move on to hotspot mitigation and load leveling.

3.2.1 Hotspot Elimination

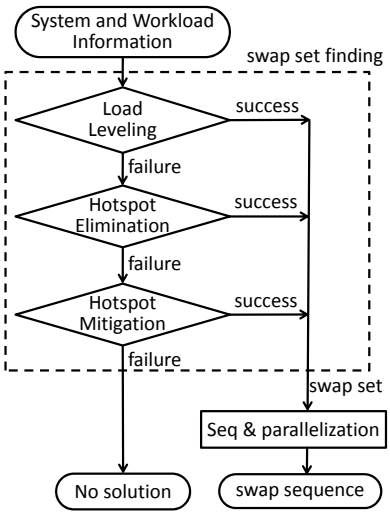


Figure 4: SWAT Overview

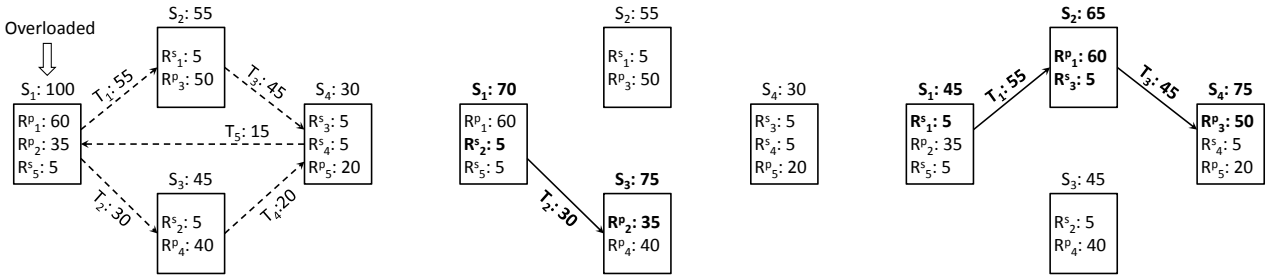
Hotspot elimination aims at eliminating all overloaded servers, i.e., servers whose aggregate load is above the hotspot threshold, through replica swap. To be more specific, it finds a set of swaps that remove overloads at all servers, if possible, while minimizing the total amount of workload being interrupted by the swap.

We illustrate the key idea by using the example in Figure 5(a). Five tenants, T_1 through T_5 are placed on four different servers, S_1 through S_4. Each tenant T_i has a primary replica, R_i^p, and a secondary replica, R_i^s. For brevity, we consider a single resource example here, and the number next to the server and replica labels represent their respective load, e.g., I/O, out of 100, the server capacity. The labels next to the edges (e.g., T_1:55) represent the potential load transfer when the swap on the corresponding tenant is executed. We consider a server as a hotspot, if its total load is higher than a threshold a *hotspot* and try to reduce the loads of hotspots through load balancing. In the example, the hotspot threshold in Figure 5(a) is set to 80.

Figure 5(a) shows that the server S_1 (with the load 100) is a hotspot to eliminate, according to the hotspot threshold of 80. One of the possible solutions is to swap the two replicas of T_2 at S_1 and S_3, effectively transferring the load difference of 30 from S_1 to S_3. If this swap is adopted, the after-swap snapshot is shown in Figure 5(b), where all servers' loads are below the hotspot threshold of 80.

Next, for the sake of explanation, we assume that for some reasons it is not possible to swap the replicas of the tenant T_2, e.g., due to the primary-secondary log gap limit criteria mentioned above. Under such a restriction, another solution is possible, as shown in Figure 5(c). The two replicas of T_1 at S_1 and S_2 are swapped, and also the two replicas of T_3 at S_2 and S_4 are swapped. After swapping T_1's replicas, we eliminate the hotspot of S_1 (i.e., from 100 to 50), but we get another hotspot at S_2 (i.e., from 55 to 110). In order to eliminate this *new* hotspot, we also swap the replicas of T_3, effectively transferring the load difference of 45 from S_2 to S_4. Note that S_4's new total load of 75 is below the threshold, which makes it a valid solution.

Now assume that we have a tighter threshold value, 70. In this case, it is more difficult to find the hotspot elimination



(a) Before load balancing (b) A solution (c) A solution with unswappable T_2

Figure 5: An Example for Hotspot Elimination (Hotspot Threshold=80)

plan than before: Solutions in Figure 5(b) and Figure 5(c) are not feasible now since the highest loads are 75 in both solutions, which are greater than the new hotspot threshold value, 70. There exists, however, a solution: three swaps that include the two in Figure 5(c) and an additional swap on T_5, between S_1 and S_4. Interestingly, in this solution there exists a cycle among S_1, S_2, and S_3.

Note that we have shown solutions that minimally affect the tenant workloads in the sense that the sum of workload interrupted by the swap is minimized. This is important because a swap requires the asynchronous replica's delta catchup for full synchronization and therefore a swap introduces a short service interruption during the full synchronization. The level of such impacts is proportional to the amount of write workload, which is equivalent to the secondary replica's load. Hence, we formulate the optimal hotspot elimination problem as follows.

DEFINITION 1. *Hotspot elimination problem* finds a set of swaps that resolve the hotspots given a threshold, while minimizing the sum of secondary replicas' workload for the swapped tenants.

We also define a term *swap load impact* as follows, used for the problem formulation below.

DEFINITION 2. Swap load impact $LoadImpact(T_i, S_j, r)$ *is the potential load difference at the server* S_j *caused by swapping the replicas of a tenant* T_i *at* S_j*, on the resource* r*, e.g., CPU or I/O.* $LoadImpact(T_i, S_j, r) = L^s_{i,r} - L^p_{i,r} \leq 0$*, if* S_j *has a primary replica of* T_i*, and* $L^p_{i,r} - L^s_{i,r} \geq 0$*, if* S_j *has a secondary replica of* T_i*.*

For example, assume a tenant T_1 has a primary replica of CPU load 60 located at S_1, and a secondary replica of CPU load 20 at S_2. Then, $LoadImpact(T_1, S_1, CPU)$=-40 and $LoadImpact(T_1, S_2, CPU)$=40.

In the optimal swap set problem, we try to find a binary variable assignment for x_i, $(1 \leq i \leq N)$, where $x_i = 1$ means T_i is swapped and $x_i = 0$ means otherwise, so as to

$$\min \sum_{i=1}^{N} L^s_{i,max} \times x_i, \quad \text{where} \quad L^s_{i,max} = \max_r L^s_{i,r} \qquad (1)$$

subject to

$$\sum_{R \in \mathbf{R_{S_j}}} [x_R * LoadImpact(T_R, S_j, r) + L_R] < L_{TH} \qquad (2)$$

for each resource type r, where $x_i \in \{0, 1\}$

Eqn 2 says that there should be no hotspot in any of servers. We use x_R to refer to the x_i of the replica R's tenant T_i.

Example Following is the problem formulation for the example given in Figure 5(a):

$$min(5x_1 + 5x_2 + 5x_3 + 5x_4 + 5x_5)$$

$$\begin{bmatrix} -55 & -30 & 0 & 0 & 15 \\ 55 & 0 & -45 & 0 & 0 \\ 0 & 30 & 0 & -20 & 0 \\ 0 & 0 & 45 & 20 & -15 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} 100 \\ 55 \\ 40 \\ 30 \end{bmatrix} < \begin{bmatrix} 80 \\ 80 \\ 80 \\ 80 \end{bmatrix}$$

$$x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}$$

While this binary integer linear programming problem (BILP) is NP-complete, in practice it often can be solved using highly efficient open source ILP solvers, such as lp_solve that we use in our experiments, up to several thousand variables. Moreover, since tenant migration (swapping) algorithms are not interactive algorithms, their running times are not as critical. Even with extremely large number of tenants, typically the tenants are clustered and the migrations are executed on the clusters rather than the whole data center. Hence, the scalability of the ILP solution should generally be acceptable. We present a more detailed analysis in the experiments section.

3.2.2 Hotspot Mitigation

While it is desired to eliminate all hotspots, there are cases where this is simply not possible to achieve. Consider the example in Figure 6(a). There are two servers with the loads 160 and 20. Given a hotspot threshold of 80, there is no feasible solution for eliminating the hotspot without an additional server. In this case, our hotspot elimination problem would report that there is no feasible solution.

However, it is still valuable to decrease the overload degree of the hotspot server. In Figure 6(a), one tenant can be swapped, reducing the load of hotspot from 160 to 105, while keeping the other server still non-overloaded, at the load of 75. This action is beneficial since the swapped tenant would not be affected by the overload anymore, and the remaining tenants there would also see reduced overloading, which would somewhat improve the performance.

To achieve this, we modify the problem formulation of hotspot elimination as follows. First, we drop the hotspot removal *constraints*, i.e., Eqn 2, for the overloaded server-resources, while keeping the constraints for the original non-

overloaded server-resources. Second, we change the objective function of Eqn 1 into the sum of load impacts of all overloaded server-resources. With these changes, we minimize the overloading, rather than strictly requiring the overloading to be removed. Below is the modified problem formulation.

$$\min \sum_{R \in \mathbf{R_{S_j}}} [x_R * LoadImpact(T_R, S_j, r) + L_R] \qquad (3)$$

for all *currently overloaded* resource type r of S_j, subject to

$$\sum_{R \in \mathbf{R_{S_j}}} [x_R * LoadImpact(T_R, S_j, r) + L_R] < L_{TH} \qquad (4)$$

for all *non-overloaded* resource type r of S_j, with binary x_i.

3.2.3 Load Leveling

In the above two problems, we discussed a *reactive* load balancing solution, where actions are taken only when there are hotspots. It may, however, be desirable to take a *proactive* action even before a hotspot is detected: whenever there is some load unbalance among the servers, try to balance them. To achieve such load leveling, we design an algorithm as outlined in Algorithm 1.

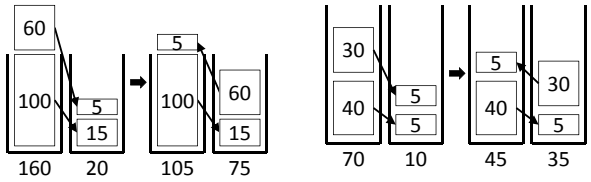
First, we see if there is a significant load unbalance, using a *load skew limit*, e.g., 50%: if the load of a server is greater than the average load of all servers by the load skew limit or more, or lower than the average by the load skew limit or more. Consider the example in Figure 6(b). Two servers originally have the loads of 70 and 10, respectively. With the average load of 40 and load skew limit of 50%, we observe load unbalance, i.e., 70/40=1.75>1.5 and 10/40=0.25<0.5.

Second, we try to minimize the load skew as follows. We run the ILP solver in a very similar manner as above, but with a minor difference on the constraints, i.e., Eqn 2. Rather than using a hotspot threshold as the constraint right hand side, we use the average load times the *target load skew limit*, e.g., 1.1, 1.3, or 1.5. When this constraint is greater than the hotspot threshold, we use the hotspot threshold instead. We try with a tight target value first, e.g., 1.1, and apply the solution if the solver succeeds in finding a solution. When it fails we move on to a more relaxed target, 1.3 and 1.5, until the solver succeeds. In Figure 6(b), there is no solution that can satisfy the target load skew limit of 1.1, but 1.3 is satisfiable: by swapping one tenant, we reduce the load skew as follows: 45/40=1.125 and 35/40=0.875.

Note that there's a cost-benefit tradeoff involved in load leveling, as in any proactive or preventive action in general: we take an action to avoid or lessen a problem, before it actually happens. The action itself has a small cost associated, and so the benefit of the load leveling action depends on the likelihood of the overload on the currently high- (but not over-) loaded servers, which in turn depends on the specific workload pattern. In our case, the swap operator has a very small cost in terms of SLO violations induced by it, so we believe proactive approach is a good thing to do in general. We evaluate the effectiveness in the experimental study.

3.2.4 Flexible Replica Configuration

In our problem definitions above, we assumed that each tenant has exactly one secondary replica. In real systems, this may not be true, because a DB service provider may use more secondary replicas or a tenant may be allowed to choose its own number of secondary replicas at different service prices.



(a) Hotspot Mitigation (b) Load Leveling

Figure 6: Load Balancing Examples

With a higher number of secondary replicas, replica swap-based load balancing becomes more powerful as there are more swap choices for each tenant. We support this with the following extension of problem formulation. Instead of using one swap decision variable x_i for each tenant T_i, we can use a variable $x_{i,k}$, for k-th secondary replica $R_i^{s,k}$ of Tenant T_i. $x_{i,k} = 1$ means swapping $R_i^{s,k}$ with R_i^p and $x_{i,k} = 0$ means no swapping for $R_i^{s,k}$.

Since the primary replica can be swapped with only one of the secondary replicas, the following constraint is added for each tenant:

$$\sum_{k=1}^{K_i} x_{i,k} \leq 1 \qquad (5)$$

where K_i is the number of secondary replicas of tenant T_i.

Also we need to replace x_R in the original problem formulation's constraints with $x_{i,k}$ if R is a secondary replica, or $\sum_{k=1}^{K_i} x_{i,k}$, if R is a primary replica.

3.3 Swap Sequencing and Parallelization

Once we have found the optimal swap set, we need to execute them in a certain sequence. However, a random execution order may create undesirable temporary hotspots during the swap executions. For instance, in Figure 5(c), consider swapping T_1 and then T_3. Between the two executions, there will be a temporary hotspot in the server S_2. Hence, T_3 should be swapped first and then T_1, since otherwise it would create a temporary hotspot in the server S_2. Another issue is parallel execution. Since a sequential execution of all swaps may take too long to finish load balancing, parallel execution is desired. However, some swaps, when executed in parallel, may again create some temporary hotspot as above, in a non-deterministic fashion.

Our goal here is to minimize, or avoid if possible, the temporary hotspots during the operator execution, while minimizing the number of sequential steps through parallelization. The key idea is to execute all swap operators in parallel that will not make their destinations (i.e., the server with the secondary replica) overloaded.

To solve this problem, we first construct a *workload transfer graph* as follows: we model the servers as vertices in a graph and an operator that swaps a primary replica at server S_i and a secondary replica at server S_j as an edge from S_i to S_j. The label of the edge is the corresponding load difference, i.e., primary replica's load minus secondary replica's load. We traverse the workload transfer graph in a reverse order: in each iteration, we take all the sink nodes of the graph and execute all the swaps that transfer workload into

```
Input  : Workload information and current tenant placement
Input  : Hotspot threshold L_TH
Input  : Load skew limit F, e.g., 1.5
Input  : A list of load skew limit target values F_tgt, e.g., 1.1,
         1.3, 1.5
Output: A set of swap operators
L_avg ← average of all servers loads
flag ← false foreach S_j of all servers do
    if L_Sj > F_tgt × L_avg then
        flag ← true
    end
end
if flag is false then
    return 'no action needed'
end
foreach F_t of F_tgt do
    Solution ← Hotspot elimination solution with Eqn 2's rhs
    as min(L_avg × F_t, L_TH)
    if Solution is feasible then
        return Solution
    end
end
return 'no feasible solution'
```

Algorithm 1: Load Leveling

```
Input  : Swap operators SW_i, 1 ≤ i ≤ N_SW that form DAG
Input  : A set of servers S_sink involved in SW_i, 1 ≤ i ≤ N_S
Output: SwapOpSequence[j]
for j ← 1 to N_S do
    if S_j has no outgoing-edge swap then
        Add S_j to the set S_sink
    end
end
while S_sink ≠ ∅ do
    foreach S_j in S_sink do
        Pick an incoming-edge swap SW_i and append it to
        SwapOpSequence[j]
        Remove SW_i from its destination server S_j and the
        source server S_j2.
        If S_j has no incoming edge, remove it from S_sink.
        If S_j2 has no outgoing edge, add it to S_sink.
    end
    Append a null to SwapOpSequence[j], which means a
    parallel execution boundary.
end
```

Algorithm 2: Swap Sequencing and Parallelization

the sink nodes, since the sink nodes do not get overloaded even with these load increases. Also all these swaps can be executed in parallel as they do not create overload. We move on to the next iteration with the newly created sinks by removing the swap-edges executed in the previous iteration, and keep traversing the graph toward the original hotspots, resolving them in the end. The algorithm is shown in Algorithm 2, which works for DAG (Directed Acyclic Graph).

When there are cycles within the graph, we first find and remove cycles, and then apply the DAG swap sequencing. Cycles are detected using Tarjan's algorithm [17]. For each cycle, we compute the expected total load L_{new} of each server node comprising the cycle when all incoming-edge swaps are executed. Among those nodes, we find the one with the lowest L_{new} and apply all its incoming-edge swaps, and break the cycle.

Swap Execution In the end, the algorithm creates a load balance solution instance, which consists of a sequence of swap batches. Each swap batch contains one or more swap operators that run in parallel. When all swap operators within a batch finish, the next swap batch starts execution. When all batches finish, the load balancing solution instance is finished.

4. EXPERIMENTAL STUDY

In this section, we evaluate the effectiveness and the scalability of our replica swap-based load balancing by systematically varying comprehensive set of relevant dimensions in the system settings as follows: **Workloads:** 1) Benchmark workloads, 2) Real workload traces. **Spikes:** 1) Controlled spikes to test under different conditions, 2) Real workload spikes. **Read/Write Ratios:** Varying read/write ratios for all tenants, 2) Different read/write ratios among the tenants. **Resource Types:** 1) Single bottleneck resource type, 2) Multiple bottleneck resource types. We executed the experiments with multiple secondary replicas. We also report on the individual features of SWAP and the scalibilty of the ILP solution.

4.1 Experiment Setup

Systems We use six machines with Intel Xeon E5620 2.4GHz processors with 16GB of memory. Five are used as database servers and one as a client machine. MySQL 5.5 is used with InnoDB and a 1GB bufferpool. Each tenant replica uses a private MySQL instance. Each tenant has its own separate query queue in the middleware with the simple FCFS scheduling policy, and four connections per tenant are used to execute queries on the private MySQL process, i.e., MPL=4. MySQL asynchronous replication is used, with one master and one or two slaves for each tenant.

Database and Queries: We use TPC-W database, generated using 100,000 items and 300 emulated browsers (EB's), which makes 1GB of raw data size. We create 50 tenants, each has two to three replicas and 1GB data per replica. Tenants' workload consist of one read and one updated query from the TPC-W queries. We first use 50-50 mix of read and write queries as the default, and later also vary the ratio in some of the experiments to show the effect of varying read/write mix ratios.

Traffic Trace: For a meaningful evaluation of load balancing methods, it is important to use a realistic multitenant workload trace that has traffic changes and spikes over time. Given that we use TPC-W dataset, it would be ideal to use either TPC-W workload pattern, or a real-world traces from an e-commerce site. The former is not suitable since TPC-W workload is too smooth and does not capture fluctuating real-workload pattern that we aim to address [18].

We used two setups as follows. First, we employ a synthetic spike generator [12] to generate volume spikes of individual tenants. This allows us to control several parameters and create different spike patterns, which characterize the increased workload and noisy neighbor situations discussed before. The second is to use a real-world trace captured from Yahoo Video site[4], which is described in [19]. We obtained the video item access counts over time, and use this as a traffic trace for multiple tenants, such that a video item represents a tenant. The main value of this trace is the multi-item popularity distribution and spikes in the real-world that is a distinguishing characteristic of a multi-tenant DB. These two traces provide arrival rate during a small time period (e.g., 10 second), and we generate individual query arrival time using the open-system model [20], and Poisson arrival.

Placement: We place 50 tenants, each with two to three replicas, over 5 servers in a random fashion, while ensuring that a single server contains at most one replica of a tenant.

[4]http://video.yahoo.com

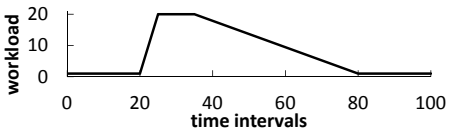


Figure 7: Spike Workload Multiplication Factor

Solver: We use lp_solve 5.5.2.0[5], an open source mixed integer linear programming (MILP) solver. It is written in C, and we used Java wrapper provided at the lp_solve website to link it to our Java-based middleware. To make a fast load balancing action, we limit the solver running time to one second, which is the smallest possible timeout that this specific solver allows. If there is an optimal or a suboptimal solution found within the time limit, it is used. Otherwise, we consider it infeasible. As we report in Section 4.3.4, lp_solve finds high quality solutions even with this short time budget.

Evaluation Metric: By considering the nature of cloud-based service offerings, we focus on Service Level Agreement (SLA) performance in terms of query response time. Therefore the desired level of service, i.e., response time, defines the provider's Service Level Objective (SLO) and the provider tries to conform to the SLOs. We use the SLO achievement/violation percentage as our main performance metric to show the effectiveness of our load balancing technique in reducing the SLO violations. To be more specific, we count how many queries had a response time greater than the SLO of x second, where we experimented with a range of x values[6].

4.2 Trace Type 1: Synthetic Spikes

In this subsection, we use synthetically generated spike traces, based on [12]. We first decide baseline traffic intensity by multiplying a constant so that 50 tenants on 5 servers without a spike will be 50% CPU-saturated. Then we create volume spikes, whose general pattern is shown in [12], which is reproduced here in Figure 7 for convenience. The spikes' peak height is decided by a parameter *SpikeHeight*, in the range of 10 to 30, in the following experiments. These spikes provide a multiplication factor, which is multiplied to the baseline traffic to generate spike workload.

For a given tenant, spikes are generated based on a parameter, i.e., *SpikeProbability*, which is defined as the probability of having a spike at a randomly chosen time instance. We use the range of 10% to 30% for the variable in the experiments below. We also control the duration of individual spikes, using the parameter *SpikeDuration*, which takes the value of 2 to 12 minutes. For the queries, We use the workload of 50-50 read-write mix with TPC-W queries mentioned above.

Note that all tenants have the same baseline traffic and spike characteristic, to keep it easy to analyze the workload pattern, load balancing behavior, and query performance. In the next subsection, where we use Yahoo Video trace, we consider the case where different tenants have different popularities. Also note that we use one primary and two secondary replicas here, i.e., total of three replicas per tenant.

Results with varying spike probability is shown in Figure 8(a), where SpikeHeight and SpikeDuration are fixed at 20 and 6 minutes. It shows average query response time and SLO violation with 5 second SLO, for no load balancing and swap. When the spike probability is 10% and the overall load is low, both no load balancing, referred to as noLB, and swap have very low SLO violation and response time. When the spike probability is 15 to 30%, and the servers are mostly saturated, swap effectively reduces SLO violation and response time, compared with noLB. Figure 8(b) shows similar results as above, where we vary spike height, with SpikeProbability and SpikeDuration fixed at 20% and 6 minutes. We also control SpikeDuration in Figure 8(c). In case of no load balancing, longer spikes have a smaller impact on performance, mainly due to the data caching effect. Under longer spikes, swap makes a bigger improvement compared to noLB, as it benefits better from individual load balancing actions under the longer-persisting load patterns.

In Figure 8(a), we implemented and compared with a migration-based load balancing method. While there are many different types of data migration methods as suggested in [15], we consider a simple and fast method, stop-and-copy: database migration with service interruption and data transfer through file copy. We also use the greedy method proposed in [22]: from the most overloaded server, we choose a tenant with the highest load-divided-by-data-size, which is called in BSR (Bandwidth-to-Size-Ratio), in the original paper. We then move it to least-loaded server that can serve this tenant without getting overloaded beyond the hotspot threshold.

The results show that migration performs worse than noLB in general, within the short time frame of 10 min run that we use here. One exception is the case of SpikeProbability=30%, where migration makes an adjustment of heavy load imbalance that pays off. These results are consistent with the expectations from the migration and our overload resolution problem space discussed before. Therefore, in the following experiments, we focus on the performance comparison between noLB and swap as migration tends to perform worse than noLB under the evaluation scenarios used.

4.3 Trace Type 2: Yahoo Video Access

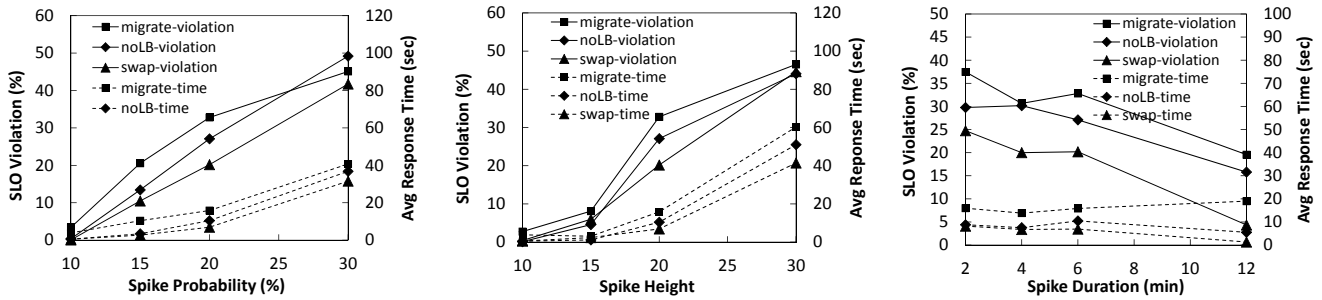
Now we turn to another type of trace, the item access count at Yahoo video site [19]. We use the 5 day worth of the data set, between Oct. 3, 2007 4:30pm and Oct. 8, 2007, 4:30pm, collected by a crawler. It extracted the visitor counts of individual video items every 30 mins, and took the difference between two consecutive measurements to get the video access count during the interval. Then, the length of each video item is multiplied to its access count, to estimate the server load, as done in [19].

We use the top 50 video items with the highest average load to get the workload traces of 50 tenants. Later, we also use the top 500 and the top 3000 to set up bigger size problem instances for scalability evaluation. We use one primary and one secondary replica per tenant in the following experiments. We tune the level of overall traffic intensity, i.e., query arrival rate, so that the maximum aggregate load would slightly overload the entire cluster under no load balancing, where we get about 2 to 5 % of queries violating the SLO of 5 second response time, in a similar fashion to [9].

Since the observation is done every 30 minutes, there are 5*48=240 observations, or 240 intervals, over 5 days of wall-clock time. Exact replay of the original trace would take 5

[5]http://lpsolve.sourceforge.net/5.5/

[6]A survey on general user behaviors under the various system response times can be found in[21].



(a) Varying Spike Probability (b) Varying Spike Height (c) Varying Spike Duration

Figure 8: Results for Spike Trace Workload (50% Write, 50 Tenants, Three Replicas, and 5 sec SLO)

days to run a single experiment, which is too long. So we scale down the experiment by playing the workload of 30 minute interval during 30 seconds. With this 60-to-1 time scale reduction, one run of the experiment takes 2 hours. We repeat three runs and report the mean, and also the standard deviation as error bars, whenever appropriate.

4.3.1 50-50 Read-Write Ratio

In this experiment, we use a 50-50 mix of read queries and update queries. We observe that this specific mix uses CPU and I/O capacities similarly, a bit heavier on the I/O. Given that I/O is the dominant resource here that creates the bottleneck, we run our load balancing based I/O only in this experiment. Note that later in Section 4.3.3, we run experiments where multiple resources may become a bottleneck, and use load balancing based on multiple resources.

Summary Figure 9 shows the SLO performance with a various SLOs. Focusing on 5-sec SLO, noLB violates SLO for 2.3% of queries. Swap-based load balancing effectively lowers the SLO violation down to 0.30% with 5-sec SLO, achieving 7.7 times reduction of violation when compared with noLB. As expected, main strengths of swap operators are: i) small latencies for individual swap operations and ii) little resource contention with the regular workload queries. Across all SLOs, swap significantly reduces the violation count compared with noLB. Also note that no load balancing has the average response time of 399 msec, while it is 169 msec for swap.

Figure 10 shows the cumulative distribution of swap operator durations, swap batch durations, and swap-based load balancing instance durations, which are defined in Section 3.3. It shows a long tail pattern, where most of them finish quickly, while small portion takes a longer time to finish mainly for the resource contention on CPU and I/O with the regular workload. 95% of individual swap operators take 613 ms or less, and 95% of a single load balancing instance take 1642 ms or less.

Fine-granularity Performance Here we present fine granularity experiments by zooming in to specific parts of the workload. These parts are good representatives of many common workload types that can be observed in real application environments. Figure 11 shows three different parts of read-write mix workload results. The X-axis shows the real time since the beginning of the trace. The line of query count indicates the total query count that arrived during each 5 second interval. The lines of noLB violation and swap violation refer to the respective numbers of SLO violations, out of the given query count. Swap detects a hotspot,

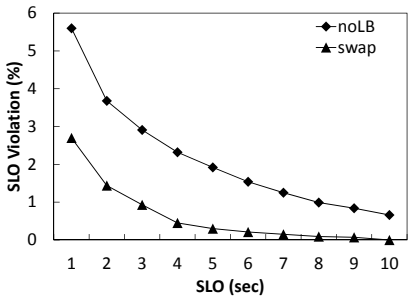


Figure 9: Yahoo Video Trace 50% Write Workload

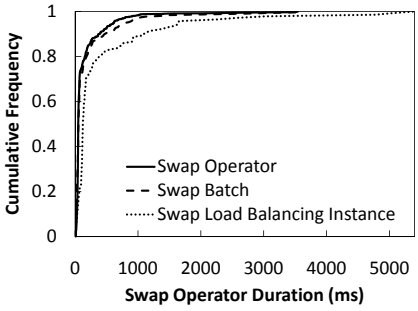


Figure 10: Yahoo Video Trace Swap Duration

and based on its hotspot mitigation, it executes 11 replica swaps, which takes 761 ms. After this quick resolution, the single overloaded server gets a reduced load, which is still slightly over the hotspot threshold, but low enough to avoid actual SLO violation. Figure 11(b) shows the case where both noLB and swap experience violations. Here the overall traffic is so high that load balancing cannot help. Swap applies a solution of hotspot mitigation here, running two swaps during 905 ms. This drops the loads somewhat on the three overloaded servers, by transferring some workload to the non-overloaded servers. As a result, the graph area below swap curve is a bit smaller than that below noLB curve.

Features of Swap-based Load Balancing We evaluate the effectiveness of individual features in the swap-based load balancing as in Figure 12. The left four bars show the effect of load leveling of Section 3.2.3 and hotspot mitigation of Section 3.2.2. As we compare default (i.e., using both load leveling and hotspot mitigation) and noLL (i.e., default, but without load leveling), we see that load leveling does not affect the average SLO violation, but reduces vari-

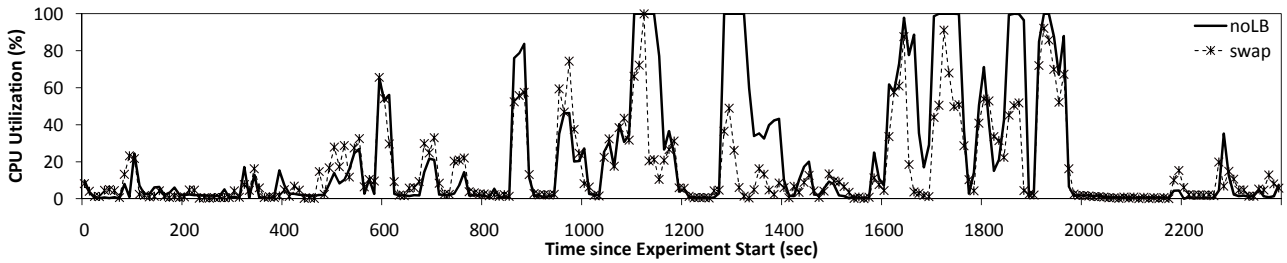
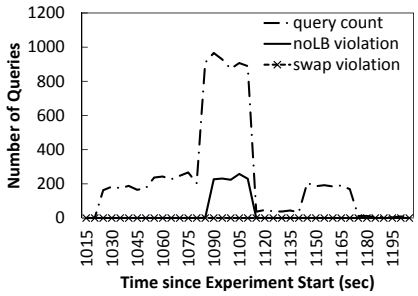
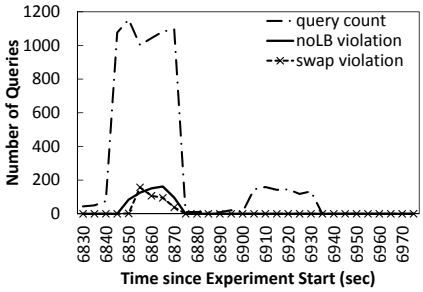


Figure 14: Yahoo Video Trace CPU utilization (Server 2)



(a) Zoom-in case 1



(b) Zoom-in case 2

Figure 11: Yahoo Video Trace Workload Zoom-in

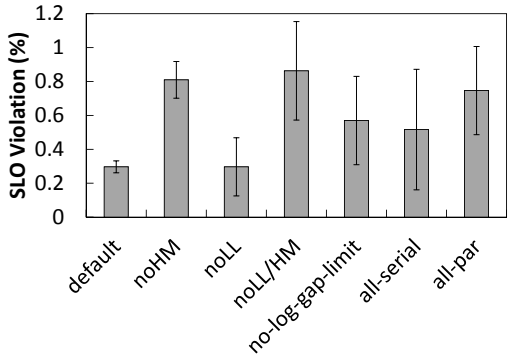


Figure 12: Features of Swap-based Load Balancing

ance across multiple runs in our experiment. It seems that load leveling contributes to the stable performance results due to its load balancing action even before the actual overload and performance degradation. default vs. noHM (i.e., default, but without hotspot mitigation) shows that hotspot mitigation significantly contributes to the reduction of SLO violation, as expected. default vs. noLL/HM (i.e., default, but without load leveling and hotspot mitigation) shows the

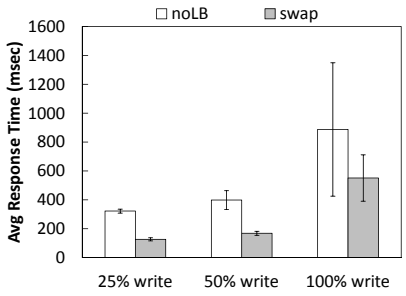


Figure 13: Yahoo Video Trace with Varying Read-Write Ratios (3 runs with 5 sec SLO)

combined effect of missing load leveling and hotspot mitigation, which is increased average violation and increased variance.

Default vs. no-log-gap-limit shows the effect of primary-secondary log gap limit of Section 3.1. Note that we use 10,000 statements as the log gap limit between the primary and the secondary replicas, which seems to work well. Clearly, performance improves when we exclude the tenants with the high log gap from the swap candidate list.

Default, all-serial, and all-parallel show the effect of batch parallel execution discussed in Section 3.3. Note that full sequential execution (serial-exec) use the same swap sequencing mentioned in Section 3.3 except parallelization, and single-batch full parallel (full-par-exec) runs all swap operators in parallel. Both serial-exec and full-par-exec increase SLO violations: the former takes too long to finish the swap, and the latter adds temporary hotspots on some server nodes. Our batch-parallel scheduling shortens the entire duration, while ensuring that each server node does not get temporarily overloaded during the swap.

4.3.2 *Varying Read-Write Ratios*

We now vary the write query ratio in the workloads to see the effect of I/O on the load balance performance. Figure 13 shows the results for three different write ratios, 25%, 50% (reported earlier), and 100%. We note that real OLTP workloads often have the write ratio between 5 and 50%, and 100% workload is a rather extreme case that we use to stress-test our method. Here we use a load balancing based on the single dominant resource, namely CPU for 25% write, and I/O for 100% write.

Swap-based load balancing consistently outperforms the others. However, the improvement margin reduces with a high write ratio. Through close-up observation, we reveal the following reason for that. When we stop sending the write workload to the master node, it suddenly becomes a quiet MySQL instance, other than that it now needs to act as a slave node and execute the relay log received from the

new master. This work is handled by MySQL *slave thread* and the *master thread* that handles the regular workload suddenly becomes idle. With the master thread being idle, MySQL kicks in dirty page flushing from memory to the disk, to better utilize the *idle* I/O cycle. However, this is not true in our multitenant system, because there are other MySQL instances running within the same server, which are often busy with their own I/O workloads. Therefore, MySQL temporarily creates I/O resource contention within the server, which our swap-based load balancing algorithm does not take into consideration. Because of this, we tend to encounter some unexpected SLO violations.

We envision two possible solutions for this. One is to extend the framework of swap-based load balancing to cope with such temporary I/O spikes right after the swap operation, which is an interesting future work to pursue. Another option is to consider a different multitenancy architecture that fundamentally avoids this problem, namely shared instance multitenancy. We plan to investigate these directions in the future.

4.3.3 Tenants with Different Read-Write Ratios

In the previous experiments, we have used various workload mixes across different experiments, while all tenants shared the same mix within an experiment. Hence, hotspots were caused by a single dominant resource type within an experiment, e.g., I/O for the write-only mix. We now consider a case where different tenants have different workload mixes, i.e., one half of tenants run the read-only workload and another half run the write-only workload. Now different resources can create hotspots at different times, which we verified by checking the resource usage statistics discussed below. Swap again effectively reduces SLA violations, from 3.22% to 1.86%.

Here we also examine the low-level resource consumption on the servers. Due to the space limitation, out of five servers used, we pick one server that experiences the most frequent overloads and report its resource utilization during the first third of the experiment run, i.e., first 2400 sec, in Figure 14. Note that we show only CPU here, due to the space limitation, while the I/O statistics are omitted. Each point is a 10 second average CPU utilization. The figure shows the load placed on the server by the initial placement, relative to its capacity (i.e., noLB). Without load balancing the initial placement hits 100% utilization, e.g., at 1100 sec, which leads to SLO violations. Swap quickly detects the overload situations and manages to drop the CPU utilization, thereby reducing SLO violations.

4.3.4 ILP Solver Scalability

SWAT formulates load leveling, hotspot elimination, and hotspot mitigation as ILP. Since ILP is an NP-complete problem, we accept the solution if a solver finds one within the time limit. Otherwise we declare the problem infeasible. In this subsection, we examine the impact of various timeout budgets and the effect of increasing number of tenants on the solution quality. We use three different problem scales: we sort all 3087 tenants available in the trace data, and pick the top 50 tenants, 500 tenants, and 3000 tenants, respectively, to create three different scales of experiments. In each case, we use 5 servers, 13 servers, and 15 servers, respectively, to keep the per-server average workload at the similar level. Note that the first case is the same setup used in the previous experiments.

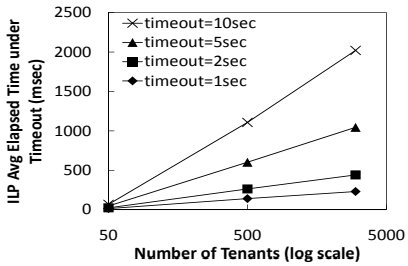


Figure 15: ILP Avg Elapsed Time under Timeout

Figure 15 shows the average elapsed time of the solver. 50 tenant scale runs the fastest, with 14.6, 24.1, 47.4, and 68.6 msec under the timeout of 1 sec, 2 sec, 5 sec, and 10 sec, respectively. As the problem scale grows, the solver time also grows, in a logarithmic fashion.

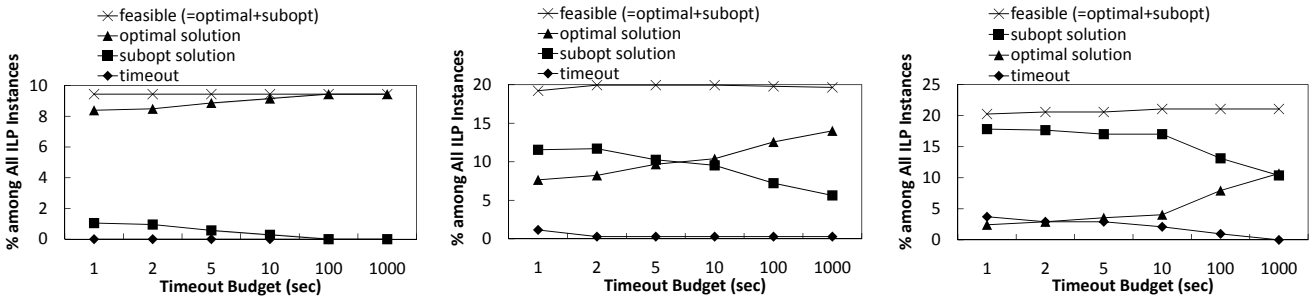
Figure 16 shows the solution types with the varying levels of timeouts. There are four different solution types: i) there is no solution that satisfies the given ILP (infeasible), ii) the ILP is feasible, and an optimal solution is found, iii) the ILP is feasible, and a suboptimal solution is found due to the timeout, and iv) the solver could neither declare the infeasibility nor find any solution within the timeout (timeout). 50 tenant case in Figure 16(a) shows i) zero timeout cases under a very short timeout budgets, e.g., 1 sec, ii) high chances of finding optimal solutions (about 90% among feasible cases) even under the short timeout budgets like 1 second. With the larger scales in Figure 16(b) and Figure 16(c), we get a small portion of timeouts and also the portion of suboptimal solutions among feasible cases increase. Note that the increased timeout improves the solution, but only in an slow, incremental fashion.

To further examine the quality of suboptimal solutions, we look at the value of the objective functions from the obtained solutions. While we omit the detailed result due to the space limitation, the highlight is as follows: the solver finds solutions of reasonably good quality under short timeout of 1 second, which incurs less than 40% of extra workload interruption due to the swaps, compared to the solution with a very large time budget of 1000 sec, at the large scale of 3000 tenants.

It is important to note that as tenant migration (swapping) algorithms are not interactive algorithms, their running times are not as critical. Even with extremely large number of tenants, typically the tenants are clustered and the migrations are executed on the clusters rather than the whole data center. Hence, the scalability of the ILP solution should generally be acceptable.

5. RELATED WORK

Load Balancing on Shared-Nothing Recently, database migration has been studied in the context of multitenant databases [15]. The main focus is on migration options and costs within various multitenant architectures. Also, [14] discusses migration-based load balancing in the context of multitenant DBs, suggesting an approach of on-demand migration, where the migration lazily moves the data as it is needed at the destination. [13] proposed a novel and efficient live migration method, where both source and destination nodes are used for executing transactions during the migra-



(a) 50 tenants (b) 500 tenants (c) 3000 tenants

Figure 16: ILP Solution Optimality under Various Timeout Budgets

tion. This type of live migration can be employed for the case where SWAT cannot completely eliminate hotspots.

Multitenant DBMS Microsoft SQL Azure [1, 2] and Amazon RDS support multitenant DBMS as a service, where each tenant is replicated and runs on a private instance, which are very similar to our system architecture. Notably, [1, 2] suggest that replica swap can be used for load balancing in multitenant databases, but they do not discuss any specific load balancing problem definition or a solution.

6. CONCLUSION

In this paper, we have proposed SWAT, a novel load balancing method for multitenant databases. Using replica role-swap and an ILP-based load balance method, it quickly finds and realizes load balancing in a lightweight manner. We have shown the effectiveness of SWAT using real trace data from Yahoo Video site, where our method significantly reduces the SLO violations compared with no load balancing and migration-based load balancing.

7. ACKNOWLEDGEMENT

We thank Michael Carey, Hector Garcia-Molina, and Jeffrey Naughton for the insightful discussions and the contributions.

8. REFERENCES

[1] D. G. Campbell, G. Kakivaya, and N. Ellis, "Extreme scale with full sql language support in microsoft sql azure," in *SIGMOD Conference*, 2010.

[2] P. Bernstein, I. Cseri, N. Dani, N. Ellis, G. Kakivaya, A. Kalhan, D. Lomet, R. Manne, L. Novik, and T. Talius, "Adapting microsoft sql server for cloud computing," in *ICDE*, 2011.

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

[4] H. Hacigumus, J. Tatemura, W.-P. Hsiung, H. J. Moon, O. Po, A. Sawires, Y. Chi, and H. Jafarpour, "Clouddb: One size fits all revived," in *IEEE SERVICES*, 2010.

[5] F. Yang, J. Shanmugasundaram, and R. Yemeni, "A scalable data platform for a large number of small applications," in *CIDR*, 2009.

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

[7] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold, "A comparison of flexible schemas for software as a service," in *SIGMOD Conference*, 2009.

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

[9] 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.

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

[11] A. Williamson, "Has amazon ec2 become over subscribed? http://alan.blog-city.com/has_amazon_ec2_become_over_subscribed.htm," 2010.

[12] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *SoCC*, 2010, pp. 241–252.

[13] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi, "Zephyr: Live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD*, 2011.

[14] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: a database service for the cloud," in *CIDR*, 2011.

[15] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi, "Who's driving this cloud? towards efficient migration for elastic and autonomic multitenant databases," *UCSB CS Dept Technical Report*, 2010.

[16] "Amazon Relational Database Service. http://aws.amazon.com/rds/."

[17] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972. [Online]. Available: http://link.aip.org/link/?SMJ/1/146/1

[18] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting realistic burstiness to a traditional client-server benchmark," in *ICAC*, 2009.

[19] X. Kang, H. Zhang, G. Jiang, H. Chen, X. Meng, and K. Yoshihira, "Understanding internet video sharing site workload: A view from data center design," *Journal of Visual Communication and Image Representation*, vol. 21, no. 2, pp. 129 – 138, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/B6WMK-4WP47N9-1/2/807b970ad06411a4026c2e6d33dbe600

[20] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Closed versus open system models: a cautionary tale," in *NSDI*, 2006.

[21] B. Shneiderman, "Response time and display rate in human performance with computers," *ACM Comput. Surv.*, vol. 16, pp. 265–285, September 1984. [Online]. Available: http://doi.acm.org/10.1145/2514.2517

[22] V. Sundaram, T. Wood, and P. J. Shenoy, "Efficient data migration in self-managing storage systems," in *ICAC*, 2006.