AdaptiveSLA: A Two-Stage Scheduling Framework for SLA Profit Maximization in Multi-tenant Database

YANG Ji, ZHANG Lin, TANG Rong
State Grid Chongqing Electric Power Company, Chongqing, 400014
Zhkwang21@163.com

Abstract. The requirement of multi-tenant applications is continuously increasing in cloud computing epoch, and they usually have larger data volume compared with traditional applications. Considering the quality of managing these data, SLAs (Service Level Agreement) are usually defined between multi-tenant database service providers and tenants. The providers will get revenues if SLAs are met, otherwise they will pay penalties. In order to maximize the profit, this paper presents AdaptiveSLA, a two-stage scheduling policy for multi-tenant database. In the first stage, AdaptiveSLA detects performance crises and leverages sliding window algorithm to mitigate crises in distributed multi-tenant database. The crises may derive from heterogeneity of requests or performance degradation of nodes. In the second stage, AdaptiveSLA makes requests' execution sequence. This stage aims at completing requests as many as possible. The executing process is constrained by SLA, which is modeled as slack time. Extensive experimental results demonstrate the effectiveness and efficiency of AdaptiveSLA scheduling policy.

1. Introduction
Database service provider is critical to efficient work of SaaS application. With the rapid expansion of tenant scale, database service provider maintains larger amounts of data and faces more unpredictable requests such as queries and transactions sent by application servers. SLA - an agreement signed by tenant and service provider - guarantees the effectiveness of multi-tenant database service [1]. More specifically, response time is in the service provider's best interest to directly optimize the profit [2]. Hence, it is very important to make profit-oriented schedule for database service providers according to request's response time. In fact, the problem requires both monitoring resources and making execution sequence. (1) Resources monitoring. When a node detects performance crises, target nodes should be chosen timely to execute its subsequent workloads. Crises may derive from heterogeneity of requests or performance degradation of the nodes, and they can extend request's response time if not eliminated in time. (2) Execution sequence making. Requests' execution sequence has directly correlation with the final profit. The more requests providers finished before deadline, the more possible they maximize the holistic profit.

Taking the problems and challenges into consideration, a research presents Delphi to mitigate crises using hill-climbing algorithm[3]; while another study designed SmartSLA, addressing the issue of how to intelligently manage the resources among different tenants to maximize its own profits[4]. However, their policies have several weak points. Firstly, they focused on system level metrics such as database throughput, average query response time. But the granularity they used is too rough to leverage the resources efficiently. Second, they neglected the significance of making execution sequence online, the cloud computing model has to operate in a fast changing environment to provide...
services to unpredictably diverse set of tenants, providers should develop lightweight and real-time decision policy.

In order to maximize the profit, this paper presents AdaptiveSLA, a fine-grained two-stage scheduling framework aiming at dispatching analysis queries. It detects and mitigates performance crises in the first stage to make good use of the cluster's resources (e.g., CPU, Mem, and I/O, this paper don't take I/O bandwidth into consideration for the reason of connection pooling in LAN). Compared with other methods AdaptiveSLA just migrate the following requests to nodes who have the same tenant data. Then the second stage makes profit-effective execution sequence for admitted requests, the constrain of scheduling is represented by slack time which describes the SLA constrain. The major contributions are as following:

A self-managing peer-to-peer multi-tenant DBMS architecture. This paper design and implement AdaptiveSLA, which eliminates the bottleneck of master node. Every node of it can execute requests or dispatch requests to other nodes without directions of a master.

Crisis migration mechanism. AdaptiveSLA uses sliding window algorithm to help find targets when nodes detect crises. The principle not only considers the status of free resources of target nodes, but also thinks about their resource consumption rate in case of causing new crises in target nodes.

Requests execution sequence. When making execution sequence of requests, decisions are made in a holistic fashion by synthetically considering the profit and slack time. The slack time model expresses the time a request can be further postponed without violating SLA constraint.

The rest of the paper are organized as follows. Section 2 provides background of database multi-tenancy and overview of AdaptiveSLA. Sections 3 presents the detailed implementation of AdaptiveSLA and explains how it works. At last, this paper presents the experiment results in Section 4.

2. Database Multi-tenancy and Overview of AdaptiveSLA
In this section, this paper first introduces the background of database multi-tenancy, describing how data is organized in database. Then it provides an overview of AdaptiveSLA, including several models used when making the schedule. At the end of the section the slack time model is explained in detail.

2.1 Database Multi-tenancy
In multi-tenant applications, data of one tenant is not so much compared with the overall tenants, so it is of big advantage to manage data in tenant fashion. Many companies have studied a lot in designing multi-tenant database. For instance, Frederick Chong identified three distinct approaches for creating data architectures, such as separating database, sharing database but separating schema and sharing database and schema architecture [6]. Stefan Aulbach compared the three approaches, and proved that...
numerous tables can intensify the competition in node resources [11], which increased the risk of degrading of the performance of the node or even crashing it. Therefore, nowadays sharing table attracts more and more attention when maintaining data for several tenants. Every request submitted by application server were marked with tenant property. In response to the solution, Force.com's optimized metadata-driven architecture for multi-tenant applications, however metadata node had a latent visit bottleneck. AdaptiveSLA eliminated these limitations by developing the system with peer-to-peer (P2P) architecture, then focused on maximizing the whole SLA profit.

2.2 Overview of the AdaptiveSLA Models

2.2.1 Node Model. AdaptiveSLA introduces node model NMi to model behaviors of node Ni. In practice, it has two functions:(1) Execute requests. It introduces a concept to define the request execution unit at a node model in this paper.

Definition 1(Execution Unit): Unit \( U_{i,j} \) means a request execute unit at \( NM_i \) that can handle requests \( q_{kj} \) about \( T_j \).

As can be seen from Figure 1, \( NM_1 \) maintains data of \( T_1 \) and \( T_2 \), so it can execute requests about \( T_1 \) and \( T_2 \) using \( U_{1,1} \) and \( U_{1,2} \) separately. Besides, units coexist at a node may have different resource demands and enjoy respective SLA profits, node model allocates different CPU, Mem, I/O resources to different units according to their tenant resource consumption and SLA profit [5].

(2) Mitigate crises. To provide well resource provision, node models must mitigate performance crises in time. In order to show the process this paper introduces a vector to express resource state and define a concept of crisis.

Definition 2(Resource Vector): The resource vector \( V = [CPU; Mem; I/O] \) is the resource state of CPU, Mem and I/O. Respectively, \( V_i \) expresses the free resource of \( NM_i \) and \( V_{i,j} \) expresses the resources \( NM_i \) needs to hold on \( U_{i,j} \).

Definition 3(Crisis): For each dimension \{CPU, MEM, I/O\} ∈ a crisis is detected to \( U_{i,j} \) at \( NM_i \), if \( V_{i,j}[D] - \eta V_i[D] \leq 0 \), (\( \eta \) is a coefficient of relaxation generated by experiments).

The crisis of \( U_{i,j} \) means the resources demand for \( U_{i,j} \) exceeds the supply ability of \( NM_i \), then should look for targets to relieve its workload pressure, which will be described in more detail latter.

2.2.2 Tenant Model. One tenant's data is replicated to several nodes, they compose a tenant model to collaborate for the tenant's requests. Tenant model \( TM_j = \{NM_i | \forall NM_i \rightarrow T_j \} \) means a set of node models who have the same data replica of \( T_j \) and can execute requests about \( T_j \).

Definition 4(Collaborative Node): Node models in a tenant model maintain the same tenant data replica and can provide the same database service, we call them collaborative node for each other.

Table 1. Profile in NM1

|          | NM2 | NM3 | NM4 | NM5 |
|----------|-----|-----|-----|-----|
| TM1      | 1   | 1   | 0   | 0   |
| TM2      | 0   | 0   | 1   | 1   |

Review the example of Figure 1, tenant models \( TM_j = \{NM_i | \forall NM_i \rightarrow T_j \} \); \( NM_2 \) and \( NM_3 \) are collaborative nodes in TM1. Here are two reasons why to design this device. For one reason, tenant model evaluates economic results of executing requests. Requests that have the same tenant property share the same SLA profit model assigned by the tenant model, e.g. the profit out-coming of executing requests of \( T_1 \) is decided by \( TM_1 \). For another, tenant model records how the tenant data distributed in system, it represents dispatching candidates when a node model of it detects performance crises. Every node model likes \( NM_i \) is configured with a profile as show in Table 1 in order of distinguishing collaborative nodes, value 1 means collaborative relation and value 0 doesn't, e.g., \( profile[TM_1; NM_2] = 1 \) means if \( NM_1 \) detects crises to \( U_{1,1} \), \( NM_1 \) can dispatch the coming requests about \( T_1 \) to \( NM_2 \), it is tenant model who responses for finding targets by using sliding window algorithm.
2.2.3 Slack Time Model. It is a key problem to make a rational execution sequence for requests in maximizing the profit. To make an accurate decision, this paper adopts a concept of the slack time.

![Slack Time Model](image1)

**Definition 5 (Slack Time):** For each submitted request \( q_k^j \) at a node model, AdaptiveSLA defines four parameters for it: \( t_k^b, t_k^e, t_k^e, \) and \( t_k^{SLA} \). \( t_k^b \) is arrival time and \( t_k^{SLA} \) is the response time under SLA constrain. \( t_k^e \) is the execution time for \( q_k^j \), the slack time \( s_k \) can be calculated by \( s_k = t_k^{SLA} + t_k^b - t_k^e \).

As illustrated in Figure 2, Slack \( s_k \) is the time that \( q_k^j \) can be further postponed without introducing additional SLA penalties, the scheduling policy is made based on this constrain. The fact is that, to calculate \( s_k \) of \( q_k^j \), we just need to know \( t_k^e \), which is predicted by using machine learning technology.

Both TYPE (a version of Gatekeeper implemented by Tozer et al. with load shedding added) and Q-Cop approaches start by predicting the execution time of a query using the number of currently running queries as the feature in their model for each query type. This paper uses the same method with boosting approach method for it iteratively obtains weak learners (namely, learners that do not necessarily have good performance individually) and combines the weak learners to create a strong learner which reduces the prediction error. The features for each query is the number of predicates and cardinality for each predicate. Algorithms used in this paper are all from the off-the-shelf machine learning package WEKA.

2.3 SLA Profit Model

![SLA Profit Model](image2)

In multi-tenant database, we assume there is an associated SLA profit model for each tenant, requests of one tenant enjoy the same profit model. Different studies have explored many profit modes in various shapes, AdaptiveSLA believes that a segmented function as shown in Figure 2 is a more commonly choice used in the real-world. The figure denotes that if the response time \( t \) of request \( q_k^j \) is shorter than \( \tau \), the service provider obtains a revenue \( A \), otherwise, pays a penalty back to the tenant according to the final response time. In addition, what has not been illustrated is if a request rejected up-front, the service provider has to pay a constant penalty \( P_j \) according to its tenant. The profit formulation is defined as:

\[
\text{profit}(q) = \begin{cases} 
A, & \text{if } t \leq \tau \\
 f(t), & \text{if } t > \tau \\
 -P_j, & \text{if } q \text{ is rejected}
\end{cases}
\]  

(1)
Once a request finished at a node model, the tenant model will report its economic result to the service provider by using Equation (1).

3. The Two-Stage Scheduling Policy

AdaptiveSLA designed to have no single point of failure, nodes in this system have equal role of providing database service. To maximize SLA profit, this paper decomposes the policy into two stages as shown in Figure 4. Firstly, AdaptiveSLA mitigates performance crises using sliding windows algorithm to provide effective resource provisioning, secondly, it makes SLA-oriented request's execution order for every node model according to request's slack time.

3.1 The First Stage: Dispatching Requests

Every node model can execute requests or dispatch them to other node models, this stage determines in which node model requests will be executed.

3.1.1 Crisis Detection. At first, application server submits analysis queries to AdaptiveSLA, and they will be routed to the nearest node model which maintains their tenants' data. Then, the node model periodically checks its resource status using crisis detecting mechanism, if the current resource state is sufficient to support every request execute unit of it, which means no crises are detected, so requests will be sent to the queue of relative execute unit according to the tenant property. Otherwise, the current resource state can’t meet the need of executing requests of some request execute unit, therefore, this node model should search for target node models to help mitigate crises by dispatching the following requests of crisis unit to other node models.

\[ \exists V_{more}^j > V_{left} (\forall T_j \rightarrow NM_j) \]  

The goal of Equation 2 is to find a node who has the most sufficient resources to execute these requests among all the collaborative nodes of NM1, the measurement is defined as the distance of resource vector \( V_i' \) and \( V_{ij} \), \( V_i' \) shows the current resource status of target nodes and \( V_{ij} \) means how much \( U_{ij} \) need to run its requests. Besides, it has two constraint conditions. The first one restrains the resource consumption ratio of the target node, one resource vector \( V_i \) is recorded by NM1, which means the nearest resource status of targets, another vector \( V_i' \) is got by a communication mechanism, which means the current resource status of targets, every resource vector \( V \) is marked with a timestamp \( T \). The threshold value is also an empirical vector gotten by several experiments, which evaluates that the target node has little possibility to detect crises. The second constrain condition indicates that the CPU, Mem and I/O resources are all enough to handle requests dispatched by \( U_{ij} \).
3.1.2 Crisis Mitigation. When crises are detected to a node model, AdaptiveSLA uses sliding window here rather than using the last resource status, providing a more confident view about shifts in behaviour. AdaptiveSLA maintains a per-node sliding window of last two resource vectors, one records the latest history resource status, another shows the current resource status, each V is marked with timestamp. For instance, assume NM_i detects crises on U_i,j. NM_i records resource status V_k of all NM_k (NM_k ∈ TM_j) except itself at the beginning. NM_i then fetches new resource status vector V_k′ to fill the second window. AdaptiveSLA will choose those who have abundant resource and low consumption rate by comparing all collaborative nodes' sliding results. After choosing, V_k takes place of V_k and the window slides forwards.

![Figure 5. Overview of Sliding Window Algorithm](image)

Take Figure 5 as example, at first, NM_1 detected a crisis on U_1,1, and the following requests about TM_1 can be dispatched to NM_2 or NM_3 by scanning the configurable table of NM_1. NM_1 recorded their initial resource status before and then send message to get their current resource status. The sliding window algorithm used the two windows (resource vector) of every candidate to calculate its potential and chose the best one. After choosing the window of every candidate, each window slides forward to wait for next crisis. Conclusively, the compare process can be formulated as:

\[
\text{target} = \arg \max_{i=2,3} (||V_i - V_{1,1}||)
\]

s.t. \( \frac{V_i - V_{1,1}}{T_i - T_1} < \bar{\theta}, \) \( (\bar{\theta} \text{ is an empirical vector}) \)

V_{1,1}(D) < V_i(D) × \eta, \ D \in \{CPU, MEM, I/O\}

The first stage eliminates crises automatically, aiming at utilizing resources rationally. If this stage can't find a target for some node model, it means the current cluster is unable to withstand the workload pressure, new servers should be added in, AdaptiveSLA handles this event for capacity planning by a human administrator.

3.2 The Second Stage: Planning Execution Order

The second stage makes execution sequence for requests waiting to be executed. Requests in a queue share a common SLA profit model, to maximize the profit, requests should be finished as many as possible under their SLA constrains, the constrain used here is modeled as slack time defined above, which means the latest time a request can be delayed. Suppose a request q_i about tenant T_i is admitted by NM_1 at a moment t^i_b, We can also get its SLA constrain time t^{iSLA}_i according to its tenant property from TM_i, besides, the execution time t^{i}_e of q_i can be predicted by ML technology, so its slack time can be calculated by \( st_i = t^{iSLA}_i + t^{i}_b - t^{i}_e \).

3.2.1 Machine Learning Technology. As execution time is so important in our decision, this paper take the model with the least prediction error. Here uses a boosting approach called additive regression in
WEKA package and also uses the regression trees and linear regression as the weak learners. Then evaluates two potential feature vectors: one based on the SQL text of the query such as number of nested subqueries, total number of selection predicates, number of equality selection predicates, number of non-equality selection predicates, the other describes the relationship between the input parameters (the normalized CPU share, memory size, number of database replicas, request rate). Experiments show that the model prediction error is further reduced with the boosting.

3.2.2 Sequence Making Algorithm. The Algorithm 1 is a re-constructive job sequencing problem, it shows how a request be arranged to execute. A candidate can be appended to execution queue Q if its slack time is longer than total execution time of the execute queue, otherwise reverse the execute queue and replace the first request whose execution time is longer than it. A request will be rejected if it misses the deadline or can’t be added in the execution queue.

Algorithm 1: Enqueue Algorithm

Input:
- \( q_k \); # a new request
- \( Q \); # a request queue waiting to be executed
- \( T(Q) \); # total execution time of requests in \( Q \)

1: if \( t_k^e > T(Q) \) then
2: \( Q \rightarrow \text{append}(q_k) \);
3: \( T(Q) := T(Q) + t_k^e \);
4: else
5: if \( t_{\text{current}} > t_k^{sl-a} + t_k^b \) then
6: for all \( q_p := \text{reverse}(Q) \) do
7: if \( t_p^e > t_k^e \) then
8: swap(q_p; q_k); # exchange the place of \( q_p \) and \( q_k \) and enqueue \( q_k \)
9: \( T(Q) := T(Q) + t_k^e \);
10: end if
11: end for
12: end if
13: reject(q_k);
14: end if
15: return \( Q \);

The Algorithm 2 shows that the execution unit just executes requests of the execution queue \( Q \) in turn.

Algorithm 2: Dequeue Algorithm

Input:
- \( Q \); # a request queue waiting to be executed
- \( T(Q) \); # total execution time of requests in \( Q \)

1: if \( Q \neq \phi \) then
2: \( q := Q \rightarrow \text{pop}() \);
3: \( T(Q) := T(Q) - \);
4: end if
5: return \( q \).
To prove the correctness of this algorithm, first suppose requests of a tenant are ordered by deadline considering their arrival orders, which is concluded by the formulation deadline = \( t^{\text{SLA}} + t^b \), requests of a tenant have the same SLA constrain, and different begin time. For a coming request \( q_k \), if \( t^b_k > T(Q) \), executing this request after executing all requests before will not violate its SLA constrain, it can work normally if it is appended to the tail of the queue. Otherwise, replace it with \( q_p \) whose execution time is longer than it. That is if \( e^p > e^k \), and deadline(\( t^p_k \)) > deadline(\( t^p_p \)), so \( t^b_k > t^b_p \), \( q_k \) can work well without violation after replacing and the total execution time of execution requests in \( Q \) reduced, which not only has no bad impact on executing other requests in \( Q \) but also benefits to arrange more requests to execution.

4. Experiment
In this section, this paper deployed and evaluated a system on a cluster of 10 nodes dedicated to database processed. For the test bed, the database server run MySQL v5.0 with InnoDB storage engine on CentOS generated by openstack, each is configured with an Intel Xeon E312xx processor, 4G memory, and 20GB Disk. The following section describes data set and benchmark used for workload generation, and a detailed evaluation of AdaptiveSLA.

4.1 Benchmark Description
Existing benchmarks provide little support for evaluating the effects of multiple tenants’ database, such as the TPC suite, focusing on testing the performance and limits of a single high performance DBMS dedicated either for transaction processing (TPC-C) or for data analysis (TPC-H) [3]. To adapt to multiple tenant environment, AdaptiveSLA changed TPC-C, systematically generating workloads and data set for 20 tenants. Tenant's data size varies from 100MB to 2GB.

This benchmark is centered on the principal activities (transactions) of an order-entry environment. The workloads generator maintains a configuration file describing how tenants’ data distributed in the system, it generated tenants’ workloads with different rate to simulate heterogeneous tenant pressure. In the training phase, we executed requests in an idle node with maximum possible resources, then calculated the average response time as the SLA standard, besides, we experimented the resource consumption of every execute unit. During the working phase of the system, the arrival rate of requests followed a Poisson distribution with the rate set in each test, for the reason that it is widely used to model independent arrival requests to a website [4]. Every node model gathered CPU and Mem status using Linux tool **top** and gathered the I/O usage with the tool **iostat** per minute. If crises were detected, node model sent requests to others by TCP pipelines, otherwise, the node model used relative execute unit to execute requests. After finishing a request, the node model reported its profit. This paper measured the total resource consumption and the final profit under 5 random parameters of the workload setting.

4.2 Experiment Results
We compared AdaptiveSLA's performance with the previous works, to view the results, we plot the distribution of the outcome with average statistical result. We firstly proved that our system can make good use of the resource by mitigating crises. Take the example of CPU and Mem consumption, Figure 6 shows their utilization ratio of the whole cluster, the data is summarize with several experiments. In each sub figure, the x-axis represents the system running time, during which we kept sending requests to nodes with different rate, and the y-axis represents the resource utilization ratio, it indicates the load-balancing capacity of the system. From the plot of Figure 6, we can see clear distinguish of resource usage ratio between AdaptiveSLA and a none-control schedule that doesn't use any schedule policy. With the workload increasing, the resource consumption of none-control fashion is substantially worse than AdaptiveSLA, the experiment result shows AdaptiveSLA improves 30% to 35% in CPU usage, and 15% to 20% in Mem usage compared with none-control method. For one reason that heterogeneous workloads will cause wasting of resource, but AdaptiveSLA balances workloads to free nodes if some nodes detect crises, for another, AdaptiveSLA can arrange more requests to execute, so its resource consumption is higher. What's more, the resource usage in none-control fashion is more susceptible to the skew of requests. However, as can be seen, the rate of change of Mem is not as obvious as CPU, it may because that the Mem resource is more abundant than CPU in our system. Besides, the max consumption of AdaptiveSLA is confined such as the max CPU using ratio is near 70%, it is because the date set is fixed before experiment, sometimes, workloads from a tenant is too heavy, but they can just be dispatched to limited nodes who have the same tenant data, we cannot leverage the resource of other free nodes.

We then compared the final profit gained by different methods in our experiment. Figure 7 demonstrates the profit gained by AdaptiveSLA, Q-Cop and a none-control schedule. As can be seen,
(1) At first, the workload isn't heavy, so all the three methods gained the similar profit because these systems are underloaded. (2) With the workload pressure increase, some nodes have performance crises, the outcome is different. The profit increasing rate of AdaptiveSLA is the highest, while none-control fashion becomes slow. The final statistic shows AdaptiveSLA achieved 35% to 40% more profit than none-control method, for one reason we have proved above, AdaptiveSLA used the total resources effectively when facing the same workloads, for another, the second stage of AdaptiveSLA can arrange more requests to execute under their SLA constrains. AdaptiveSLA also achieved 20% to 25% more profit compared with Q-Cop. The result shows AdaptiveSLA makes more reasonable schedule policy. (3) We carefully analyze this phenomenon and conclude that our schedule policy does well in maximizing the profit. In summary, the experiment results in this section demonstrate that AdaptiveSLA, using a two-stage schedule policy, can make good use of resources of the cluster, and make a well profit-oriented execution plan, the final profit of service provider is improved.

Acknowledgments
First and foremost, I would like to show my deepest gratitude to my supervisor, a respectable, responsible and resourceful scholar, who has provided me with valuable guidance in every stage of the writing of this thesis. Without his enlightening instruction, impressive kindness and patience, I could not have completed my thesis. My sincere appreciation also goes to the colleagues from the State Grid Chongqing Electric Power Company, who participated this study with great cooperation. Last but not least, I'd like to thank all my friends, for their encouragement and support.

References
[1] Lehner W, Sattler K U, Sattler K U. (2013) 'Web-scale data management for the cloud’, pp.137-145
[2] Abadi, D. J., Madden, S., Hachem, N. (2008) Columnstores vs. row-stores: how different are they really. Sigmod
[3] Elmore A J, Das S, Pucher A, et al. (2013) Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. SIGMOD
[4] Xiong P, Chi Y, Zhu S. (2011). Intelligent management of virtualized resources for database systems in cloud environment ICDE
[5] Popovici F I, Wilkes J. (2005) Profitable services in an uncertain world.Processings of the 2005 ACM/IEEE conference on Supercomputing. IEEE Computer Society
[6] Azeez A, Perera S, Gamage D, et al. (2010) Multi-tenant SOA middleware for cloud computing.