SLA-Tree: A Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing

Yun Chi Hyun Jin Moon Hakan Hacıgümüş Junichi Tatemura
NEC Laboratories America. 10080 North Wolfe Rd, SW3-350, Cupertino, CA 95014
{ychi,hjmoon,hakan,tatemura}@sv.nec-labs.com

ABSTRACT

As cloud computing becomes increasingly important in database systems, many new challenges and opportunities have arisen. One challenge is that in cloud computing, business profit plays a central role. Hence, it is very important for a cloud service provider to quickly make profit-oriented decisions. In this paper, we propose a novel data structure, called SLA-tree, to efficiently support profit-oriented decision making. SLA-tree is built on two pieces of information: (1) a set of buffered queries waiting to be executed, which represents the scheduled events that will happen in the near future, and (2) a service level agreement (SLA) for each query, which indicates the different profits for the query for varying query response times. By constructing the SLA-tree, we efficiently support the answering of certain profit-oriented "what if" questions. Answers to these questions in turn can be applied to different profit-oriented decisions in cloud computing such as profit-aware scheduling, dispatching, and capacity planning. Extensive experimental results based on both synthetic and real-world data demonstrate the effectiveness and efficiency of our SLA-tree framework.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous; E.1 [**Data Structures**]: Trees

General Terms

Algorithms, Management, Performance

Keywords

Cloud Computing, Service Level Agreement, SLA-tree, Scheduling, Dispatching, Capacity Planning

1. INTRODUCTION

Accelerating adoption of the cloud computing introduces new challenges to traditional database systems [2, 6, 13]. For example, a database service in the cloud may have to serve much more diverse clients than the traditional enterprise databases usually do. One key observation is that cloud service providers have to optimize their profits while serving diverse clients. Usually, the profit of a cloud service provider is determined by the services the provider offers to a variety of customers, governed by certain service level agreements (SLAs). As a consequence, for a cloud service provider, how to make intelligent profit-oriented decisions is a major challenge and a key to success. In this paper, we present a framework for efficiently supporting SLA-based, profit-oriented decisions in cloud computing.

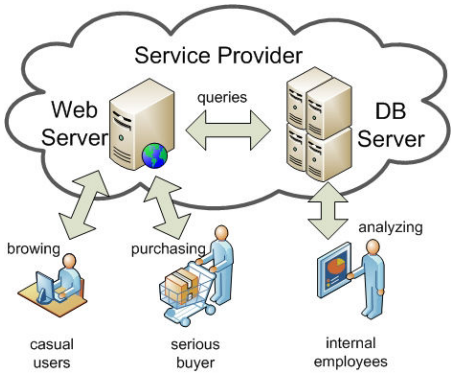


Figure 1: An example of cloud computing system.

To further illustrate the challenges and to motivate our work, we use a real-life scenario. Assume a cloud service provider is hosting an online shopping site, where queries come from different users, as shown in Figure 1. When a query comes from a serious buyer (e.g., a user who is ready to check out, with high margin items in the shopping cart), the potential profit of answering the query can be high and the delay should be short; on the other hand, when the query is from a casual user (e.g., someone who just uses the shopping site's tools to compare features of different products), the potential profit may be low and longer delay is tolerable. Yet another query may come from an internal employee who is collecting some data to make certain business decisions (e.g., whether to put certain products on sale), and in such a case a much longer delay is acceptable up to a certain threshold, after which a penalty may be incurred due to the failure to make the decision. In addition to different profit profiles, another observation from this example is that the workload in a cloud database system can be a mixture

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

of short queries (e.g., OLTP queries from buyers) and long queries (e.g., OLAP queries from internal employees).

In the above scenario, some profit-oriented decisions to be made by the cloud service provider include

scheduling: When there are multiple queries with different profit profiles, which query should we serve first?

dispatching and admission control: When a new query arrives, if there are several servers that can handle the query, to which server should we dispatch the new query? Or should we simply reject the new query due to profit consideration?

capacity planning: Given the current workload and profit situation, should we add a new server?

To address these profit-oriented issues, traditional technologies, we believe, have several weak points. First, most traditional technologies in scheduling and dispatching, focus on *system level metrics* such as database throughput, average query response time, and so on. In contrast, we believe it is in the service provider's best interest to directly optimize the profit while making decisions[1]. Second, although there exist many business analysis tools for profit-oriented decisions, they are often heavyweight and offline. The cloud computing model has to operate in a fast changing environment to provide services to unpredictably diverse set of clients. Therefore, lightweight and real-time decision supporting functionalities are required.

To address these challenges, we propose a framework to efficiently support profit-oriented decisions in database systems in the cloud. The basic idea of our framework is that rather than observing the *output* of a database server for query response time and throughput, we *proactively* look into the query buffer in front of the server for profit information. More specifically, based on the queries buffered in front of the server and their corresponding SLAs, we build a novel lightweight data structure called SLA-tree. SLA-tree not only encodes the information about the profit situation in the near future, but also is able to efficiently infer the potential profit impact if certain conditions are changed in the system. Equipped with such capability, we ask a series of "what if" questions to SLA-tree in real time to make profit-oriented decisions in scheduling, dispatching, capacity planning, and so on. The SLA-tree framework is illustrated in Figure 2.

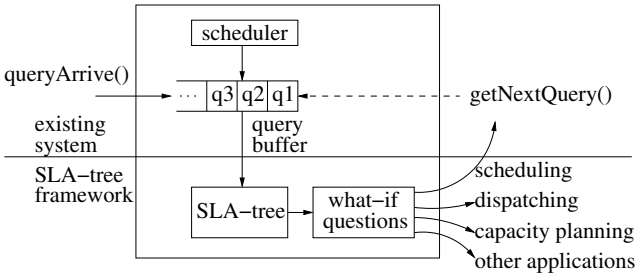


Figure 2: The SLA-tree Framework.

[1]In the scheduling community, however, there exist several cost-based methods that are closely related to profit-oriented decisions, and we will discuss them in related work.

The main contributions of this paper are summarized as follows.

- We propose a novel data structure, SLA-tree, to support profit-oriented decisions in cloud computing. SLA-tree combines a sequence of buffered queries together with the SLA of each query, and it is able to efficiently infer the potential profit change if certain set of queries are postponed or expedited.
- We present how SLA-tree can be used as an efficient framework to improve the profit efficiency of variety of system components, such as scheduling, dispatching, and capacity planning, which are relevant to cloud computing systems.

The rest of the paper is organized as the following. In Section 2, we provide background information. In Section 3, we introduce SLA-tree by using a simple 1/0 profit model. In Section 4, we show how SLA-tree can be extended to handle general profit models. In Section 5, we describe the implementation of SLA-tree in detail and analyze its time and space complexities. In Section 6, we demonstrate how to leverage the information in SLA-tree for profit-oriented tasks in a cloud database system such as scheduling, dispatching, and capacity planning. We show experimental results in Section 7 and discuss some limitations of SLA-tree in Section 8. Finally, we conclude the paper in Section 9.

2. BACKGROUND INFORMATION

In this section, we provide some background information. We start with an introduction to service level agreements (SLAs). Then we describe the system setting in our framework. Finally, we discuss some related work.

2.1 Service Level Agreements (SLAs)

An SLA is a contract between a service provider and its customers. SLAs are used to indicate the profits the service provider may obtain if the service is delivered at certain levels, and the penalty the service provider has to pay if the agreed-upon performance is not met. There exist many forms of SLAs with different metrics (query response time, throughput, availability, security, etc.) and measurement methods (e.g., measured at a per-customer level or a per-query level).

In this paper, we use the SLA metric on query response time, which is the time between a query is presented to the system and the time when the query execution is completed. For the SLA shape, we choose to use stepwise functions, because they were commonly used in previous studies and they naturally capture business terms stated in the service contracts [21]. More specifically, as illustrated in Figure 3(a), assuming a query is presented to the system at time 0, if the system is able to finish the execution of the query at time t, where t is earlier that t_1, then the service provider obtains a gain of g_1; otherwise, if $t_1 < t \leq t_2$, the gain is g_2, and so on. Finally, if the answer cannot be delivered within a certain delay (t_2 in our example), a penalty p is incurred[2].

[2]In this case, the service provider can actually choose to drop the query, since the penalty p has already happened. However, in this paper we choose not to drop the query, assuming the client may still request the query result even the deadline has passed, which can be seen in many application scenarios.

As an example of the use case for the SLA in Figure 3(a), we use the system described in Figure 1 again. When a buyer issues a query (e.g., by clicking a "check-out" button) to place an order, a response time less than t_1 may indicate a normal check-out time. On the other hand, if the response time is longer than t_1, the buyer may lose patience and attempt to cancel the order. Finally, t_2 may represent the timeout threshold of the client's browser, whereas after t_2, a query result (e.g., the order confirmation) becomes invalid.

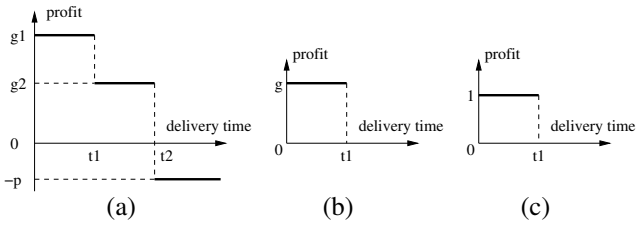


Figure 3: SLAs with (a) general profit-penalty model, (b) g/0 profit model, and (c) 1/0 profit model, assuming the query arrives at time 0.

Other than Figure 3(a), in Figures 3(b) and 3(c) we show two special cases of SLAs with a single step in the stepwise functions. These special cases will be used to facilitate the introduction of our framework.

In addition, we choose query-level SLAs instead of customer-level SLAs. The reason for this choice is that very often, even for the same customer, the service provider can assign different prices for queries within the expectation (e.g., the first 1000 queries during a day) and those beyond budget (e.g., over-use charge).

Profit vs. Revenue From the service provider's point of view, its profit is the difference between the revenue (e.g., determined by SLAs) and the cost(e.g., hardware and software costs). In most part of this paper, when we talk about profit, we refer to the revenue part, which is directly related to SLAs. That is, we consider the cost part to be fixed when comparing the profit of different decisions.

2.2 System Setting

Our high-level system setting is shown in Figure 4. The system consists of multiple servers, where queries arrive to the system dispatcher. The dispatcher dispatches each incoming query to one of the servers (or maybe rejects the query if admission control is in place). When a query is dispatched to a database server, if the server is busy, the query joins a buffer of queries waiting to be executed. When the execution of the current query in the server is completed, the query at the front of the buffer is executed next, or optionally, a scheduler picks a query from the buffer to be executed next according to certain scheduling policies.

We consider the case where there is only one buffer for each server. Furthermore, we focus on online decision making where the query workload is not known beforehand.

2.3 Related Work

In the areas of computer networks and database systems, scheduling and dispatching have been studied for decades [14] and numerous policies have been proposed and studied in various areas such as computer networks [4], database systems [18], and Web services [19].

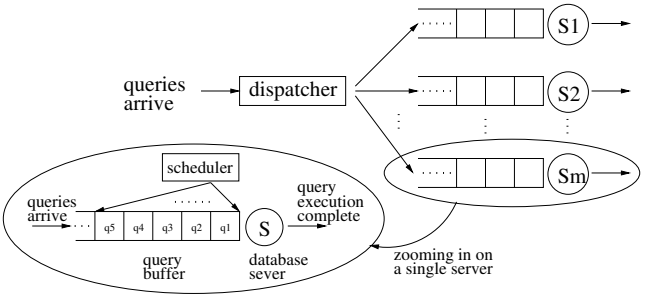


Figure 4: The system setting.

Some well-known scheduling policies include first-come-first-serve (FCFS), shortest-job-first (SJF), earliest-deadline-first (EDF), and so on. Each of these scheduling policies has its own merits. However, these scheduling policies are profit-unaware and they usually optimize certain system-level metrics such as throughput and average response time. More recently, there has been some work on profit-based scheduling. Haritsa et al. [10] proposed a value-based scheduling in database systems, where the value of a query is based both on a fixed priority and a hard deadline. Peha et al. [15] proposed CBS, a cost-based scheduling that schedules queries according to their expected cost (profit) whereas the cost changes over time. Gupta et al. [9] argued that, in a business intelligent system, a performance metric should use a user's perspective instead of a system perspective, and accordingly proposed an end-to-end solution that takes fairness, effectiveness, efficiency, and differentiation into consideration. Similar ideas had appeared in the field of high-performance grid computing. Chun et al. [3] used a user-centric performance metric, where the utility of a job depends on the execution time, and designed scheduling algorithms accordingly. Irwin et al. [11] considered a utility function with a gradually increasing penalty with a bound and demonstrated better performance than that in [3].

For dispatching, other than the simplest Round-Robbin policy, a commonly used approach is least-work-left (LWL), which dispatches a query to the server that currently has the least workload (including the workload left for the query being executed in the server and the expected total workload among all queries waiting in the buffer; see Zhang et al. [20] for a detailed discussion). More recently, Schroeder et al. [17] proposed size-interval based task assignment (SITA), which classifies queries according to their execution time and then dedicates certain number of servers to each query class. However, most existing work on dispatching targeted at balancing the workload, and they are not profit aware.

Another topic related to profit-oriented decision making is to predict the query execution time, because it is needed for making the right decisions. There have been some recent studies that estimate query execution time by using machine learning techniques. Elnaffar et al. [5] used classification techniques to predict if a workload is OLTP or DSS (Decision Support System). Ganapathi et al. [7] used advanced kernel methods and a k-nearest-neighbor approach to predict the execution time of a query, by using the information in the query plan. The encouraging results from these studies show that we can obtain reliable estimation of query execution time and use it in profit-oriented decision making.

3. SLA-TREE FOR 1/0 PROFIT MODEL

Fundamentally, SLA-tree is mainly used to efficiently answer two key questions about potential profit change among queries in the buffer. In this section, we introduce these two questions, discuss why they cannot be answered efficiently by using naive approaches, and show how SLA-tree is built to answer the questions efficiently.

3.1 Two Key Questions to Answer

Assume that at a given time t, there are N independent queries $\{q_1, \ldots, q_N\}$ to be executed sequentially in a database server, and each query has its own SLA. Furthermore, we assume the order of query execution is fixed (either determined by the time when the queries join the buffer, or decided by a certain scheduling policy). Without loss of generality, we assume the query execution order is $q_1 \prec q_2 \prec \cdots \prec q_N$, i.e., q_1 is to be executed first and then q_2, ..., and finally q_N.

The high-level objective of our SLA framework is as follows. By combining the order $q_1 \prec q_2 \prec \cdots \prec q_N$ with the SLAs associated with the queries, we build a data structure, SLA-tree, in order to efficiently answer the following two key questions.

Two key questions: For any m and n such that $1 \leq m \leq n \leq N$, and for arbitrary time interval τ,

postpone(m,n,τ) How much profit will be lost if we postpone queries $q_m, q_{m+1}, \ldots, q_n$ (with respect to their originally scheduled starting time) by a time of τ?

expedite(m,n,τ) How much profit will be gained if we expedite queries $q_m, q_{m+1}, \ldots, q_n$ (with respect to their originally scheduled starting time) by a time of τ?

Then equipped with the capability of answering these two key questions efficiently, we can ask a variety of "what if" questions to support profit-oriented decisions in the system. We will defer the discussion on the application areas of "what if" scenarios in a later section (Sections 6) and instead, in this section and the next two sections, we describe how to answer **postpone(m,n,τ)** and **expedite(m,n,τ)** efficiently by using SLA-tree.

3.2 Naive Approaches

Before showing how SLA-tree is built to efficiently answer the above two key questions, we first examine why naive approaches fail to do so.

The most naive approach is to directly calculate the answers to the postponing and expediting questions from the query buffer. More specifically, we first compute the original total profit for queries $q_m, q_{m+1}, \ldots, q_n$, assuming they would be executed on schedule; then we compute the new total profit assuming these queries would be postponed (or expedited) by τ from their originally scheduled starting time; the difference between the two profits is therefore the answer. As can be seen, to answer each question about postponing and expediting, it will take $O(N)$ time. This is too slow if answers to many such questions are needed at the same time. For example, it will take $O(N^2)$ time if answers to N such questions are needed by certain applications.

A more sophisticated approach is to pre-compute the answers to **postpone(1,i,τ)** and **expedite(1,i,τ)** for each query q_i. After this pre-computation, the profit change for postponing or expediting any subset $q_m, q_{m+1}, \ldots, q_n$ can be derived in $O(1)$ time. A moment of thought will reveal that this approach can only handle a fixed constant τ, and it fails if τ is a variable in the questions. However, as will be seen momentarily, this idea of pre-computing cumulative values to facilitate efficient question answering is one key technique used by our SLA-tree data structure.

3.3 Building and Querying the SLA-tree

In this subsection, we introduce our SLA-tree for efficiently handling the two key questions of **postpone(m,n,τ)** and **expedite(m,n,τ)**. To begin with, in this section we focus on the simplest SLA, the 1/0 profit SLA as shown in Figure 3(c), where a profit of 1 is gained if the delay is less than t_1 and 0 otherwise (i.e., there is no penalty). We extend SLA-tree to general SLAs in the next section.

3.3.1 Converting Delay to Slack

To make the discussion easier to follow, we first convert the delivery time in the SLA definition (Figure 3) into another concept of the *slack* time.

In the 1/0 profit SLA model, for each query q_i in the buffer, there are three parameters: t_i^d, t_i^s, and τ_i. t_i^d is the deadline for query q_i. That is, if query q_i is finished before t_i^d, the profit to the service provider is 1, otherwise the profit is 0. Obviously, t_i^d depends on the arrival time and the SLA threshold t_1 for query q_i. t_i^s is the scheduled starting time for query q_i. τ_i is the execution time for q_i in the database server. For convenience, instead of t_i^d and t_i^s, in the following

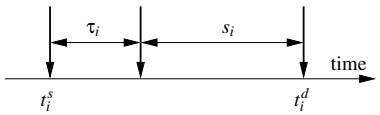


Figure 5: Illustration of how the slack is derived.

discussion we focus on a derived value—the *slack* time s_i for q_i, where $s_i = t_i^d - t_i^s - \tau_i$, as illustrated in Figure 5. In other words, slack s_i is the time that q_i can be further *postponed* (with respect to its originally scheduled starting time) without introducing additional penalties. If the slack is a negative value, we reverse its sign and call the result *tardiness*, which is again a positive value. In other words, if $s_i < 0$, then the tardiness -s_i indicates the time that q_i has to be *expedited* (with respect to its originally scheduled starting time) in order to avoid its profit loss.

3.3.2 Building the Slack Tree S^+

The SLA-tree framework consists of a pair of components—a slack tree S^+ and a tardiness tree S^-. The slack tree S^+ is a complete binary search tree built as follows. First, we compute the *slacks* $s_1, \ldots, s_N$ for all the queries $q_1, \ldots, q_N$ and sort them in the increasing order (without loss of generality, we assume these N slack values are distinct). Then the queries whose slack values are non-negative are used to construct the binary search tree S^+, which is constructed by merging the sorted list in a bottom-up fashion. More specifically, the sorted list of queries (sorted by their slacks, which are positive) are the leaf nodes of S^+, and upward pairwise merges are recursively applied to build the internal nodes of S^+. Each node d of S^+ has two parts: a slack value d_τ and an ordered list of descendants d_l. Here is a recursive description:

- Each leaf node d in S^+ corresponds to a query q_i: its slack value d_τ is the query's slack τ_i and its descendant list contains a single item: i, the id of the corresponding query q_i;
- For an internal node d, its slack value d_τ is the middle point between the largest slack value in its left subtree and the smallest slack value in its right subtree; and its descendant list contains the query id's of all its descendant nodes, sorted by query id.

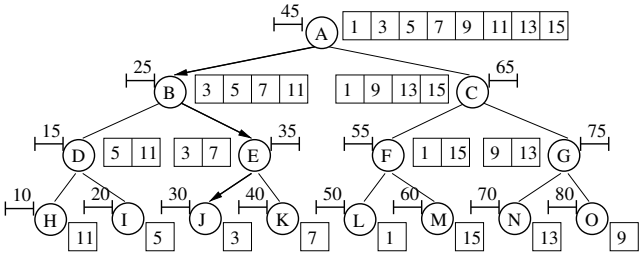


Figure 6: The simple version of a slack tree in the SLA-tree, with 1/0 profit model.

Figure 6 shows an example of slack tree. We will use this as a running example in the rest of the paper. In this example, we assume there are 16 queries q_1 to q_{16}. Without loss of generality, we assume the query execution order is $q_1 \prec q_2 \prec \cdots \prec q_{16}$. In addition, for illustration purposes, in this example we assume queries with odd ids have positive slack values and queries with even ids have negative slack values (positive tardiness values). As a consequence, the slack tree in Figure 6 is constructed by queries $q_1, q_3, \ldots, q_{15}$. In the figure, for each node d in the slack tree, its slack value d_τ is given by a number on top of a line segment and its descendant list is shown as an array in a square box. For example, look at node D: it is an internal node with a slack value of d_τ=15 and a list of two descendants with ids 5 and 11, respectively.

3.3.3 Querying the Slack Tree S^+

After S^+ has been built, we now show how to efficiently answer **postpone(m,n,τ)** by using S^+. We first start with a special case where m=1, i.e., **postpone(1,n,τ)**. To answer **postpone(1,n,τ)**, we traverse S^+ starting from visiting the root in the following way. An accumulator c is used and its initial value is set to zero. For an internal node d, τ is compared with d_τ. There can be two cases:

$\tau \le d_\tau$ If this happens, then we know *none of* (the queries corresponding to) the descendants of d's right child has slack less than τ and so they can be safely ignored, because their profit will not be affected if they are postponed by τ. If this happens, we simply visit the left child of d and ignore the right child of d.

$\tau > d_\tau$ If this happens, then we know *all of* (the queries corresponding to) the descendants of d's left child have slacks less than τ, and so each of them will lose a profit of 1. The only unknown is how many of them have id's less than or equal to n. This can be answered by a binary search for n in the descendant list of the left child of d. Assuming this number of queries is k, then we increase c by k and visit the right child of d.

The traversal will finally arrives at a leaf node d. When this happens, we increment c by 1 if the query id in the leaf node d is less than or equal to n and $\tau > d_\tau$. Finally, the result c represents the number of queries that (1) have id's less than or equal to n and (2) have slacks less than τ. This number c is exactly the profit loss that will be introduced in q_1 through q_n if they are postponed by a time interval of τ.

The arrows in bold in Figure 6 show the traversal path for answering the question **postpone**(1,n=9,τ = 32). (1) Starting at root node A, because $\tau < d_A = 45$, we ignore the right subtree of A and visit node B; (2) $\tau > d_B = 25$ and so we move to node E and increase c by 1, which is the number of ids in the descendant list of node D that are less than n; (3) $\tau < d_E = 35$ and so we ignore E's right subtree and move to J; (4) J is a leaf node with id 3, which is less than n, and $\tau > d_J = 30$, so c is increased by 1 again. The final answer is therefore **postpone**(1,9,32) = 2.

Next, to generalize to arbitrary m and n, it can be easily shown that for a fixed τ, **postpone()** has the following additive property:

$$\textbf{postpone(m,n,}\tau\textbf{)} = \textbf{postpone(1,n,}\tau\textbf{)} - \textbf{postpone(1,m-1,}\tau\textbf{)}.$$

Querying S^+ can be handled in time $O((\log N)^2)$ in this naive implementation of SLA-tree: $(\log N)$ steps of traversal and each step involves at most one binary search in the descendant list (of the left child). However, in Section 5 we show a more sophisticated implementation of SLA-tree using auxiliary information that reduces the query time complexity to $O(\log N)$.

3.3.4 Building and Querying the Tardiness Tree S^-

The tardiness tree S^- is built in the same way as the slack tree S^+, except that S^- is built for those queries with negative slacks (and hence positive tardiness values) and the slack value d_τ is replaced by tardiness values, which are again positive. Then we can easily show that **expedite(m,n,τ)** can be answered by traversing S^- in exactly the same way we previously traversed S^+.

4. SLA-TREE FOR GENERAL PROFIT MODEL

In this section, we describe how SLA-tree can handle the general profit model, as illustrated in Figure 3(a). We start with an intermediate step, the g/0 profit model, as illustrated in Figure 3(b).

4.1 g/0 Profit Model

Recall that when moving to a right child, we ask its left sibling "in your descendant list, how many of them have ids less than or equal to n". Actually, what we really cared about is not "how many" but "how much profit loss". These two happen to be identical in the case of 1/0 profit. So if we have g_i/0 profit, where g_i is the profit loss because of missing q_i's deadline, what we really care about is "among the queries with id's less than or equal to n in your descendant list, what the total profit loss is". Therefore, if we have this total profit loss for each entry in each descendant list, then we can handle arbitrary profit loss. This can be achieved by accumulating and recording these total losses while merging the descendant lists. It can be shown (detailed in the next section) obtaining such losses can be done in an incremental

way and so will not change the time complexity of building the SLA-tree.

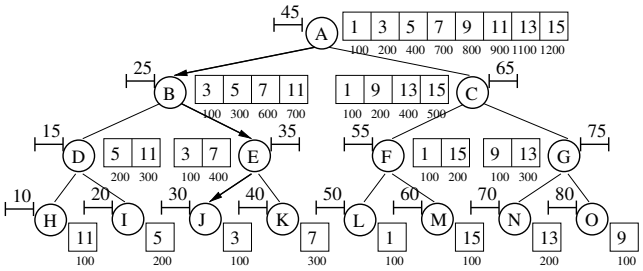


Figure 7: The simple version of a slack tree in the SLA-tree, with g/0 profit model.

In Figure 7, we extend our running example by adding a profit loss to each query (shown at the bottom of the descendant lists) and showing the corresponding SLA-tree with accumulated profit losses. For example, from node D, we can see that among the descendants of D, the total profit loss for queries with ids of 5 or lower is 200, and that for queries with ids of 11 or lower is 300. It can be shown that if we ask **postpone**(1,9,32) again, the answer will be a profit loss of 300.

4.2 General Profit Model

Now we show how SLA-tree can be extended to handle the general profit model as described in Figure 3(a). The key idea here is to decompose a general profit model SLA into the sum of several g/0 profit model SLAs. This is illustrated in Figure 8. If the query execution is completed before t_1, a profit gain g_1 is obtained; otherwise if it is completed before t_2, a smaller profit gain g_2 is obtained; otherwise, a penalty p is incurred. There is an equivalent profit model: for each query, the service provider starts by paying the penalty p upfront; then if the query execution is completed before t_2, a profit gain $g_2' = g_2 + p$ is made; if in addition, the query is completed before t_1, an *additional* profit gain $g_1' = g_1 - g_2$ is made. (It is worth noting that in this equivalent model, p can be excluded from future consideration, for its net effect has been absorbed by g_2'.) With such a decomposition of SLAs, we can extend SLA-tree in the following way to handle SLAs with the general profit model.

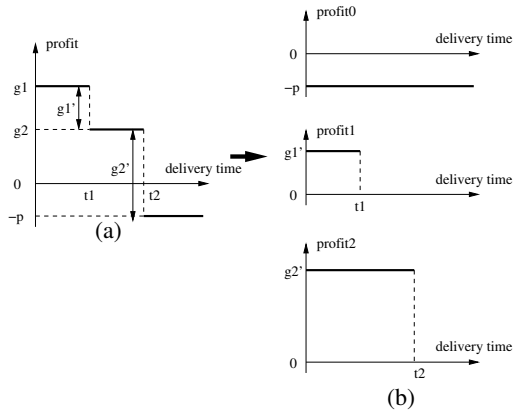


Figure 8: (a) SLA with multiple levels of profits, and (b) the profit decomposition into three g/0 profits.

We start by decomposing the general SLA into the sum of several g/0 profit models, as shown in Figure 8(b). Assuming each SLA-tree has exactly K steps, we can duplicate each query id by *K-1* times (obviously with different slack or tardiness values). Then we can build and query SLA-tree the same way as before. However, there are two changes here: when merging two descendant lists, if two copies of the same query meet, then their accumulative profits are added and they are combined as a single query in the current descendant list. The details are given the next section. Second, because the number of nodes in SLA-tree is now NK (because each of the N queries corresponds to $K-1$ leaf nodes in the SLA-tree), the time and space complexity become $O(NK\log(NK))$, and the time complexity for querying SLA-tree becomes $O(\log(NK))$.

In Figure 9, we show 4 queries with different general SLAs and in Figure 10, we show the corresponding SLA-tree[3].

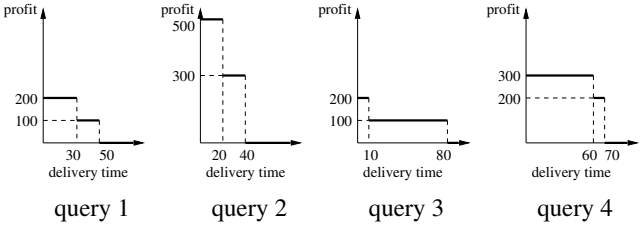


Figure 9: Profits for the 4 queries in the example.

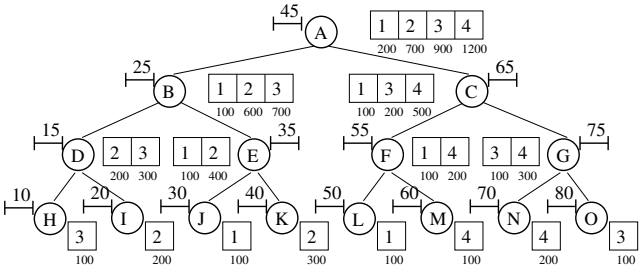


Figure 10: The simple version of a slack tree in the SLA-tree, with general profit model.

5. DETAILS OF SLA-TREE

In this section, we first provide the implementation detail for a sophisticated version of SLA-tree that achieves the $O(NK\log(NK))$ time complexity, and then provide the formal proofs for the space and time complexities of the implementation.

5.1 SLA-Tree Implementation

Figure 11 illustrates the information stored in the sophisticated version of SLA-tree (we only show the slack tree here). We discuss this information in detail.

First, as mentioned before, SLA-tree is a *complete* binary search tree. That is, a node either is a leaf (and so corresponds to a query) or has exactly two children. Therefore, the tree size is 2M-1 where M is the total number of leaf

[3]Note that the SLA of each of the 4 queries in Figure 9 are obtained by joining those of two queries in Figure 7, and hence the two SLA-trees happen to be very similar.

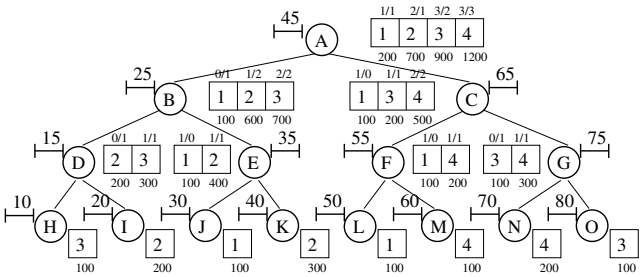


Figure 11: The complete version of SLA-tree.

nodes, i.e., the number of queries (and their duplications generated by the decomposition of SLAs) that have positive slacks. So M is bounded by NK, where N is the total number of queries in the query buffer (where a query can be in the slack tree, or the tardiness tree, or both) and K is the maximal number of "steps" in the stepwise SLA. In Figure 11, nodes A through G are internal nodes, and nodes H through O are leaf nodes.

Second, for each SLA-tree node d, its slack value d_τ is the average between the maximum slack in its left subtree and the minimum slack value in its right subtree. This slack value is the search key for the binary search tree. In addition, the descendant list of d records the ids of all the queries who are decedents of d, and the ids in the list are in an increasing order. Such an order can be automatically obtained during the SLA-tree construction, as shown in detail in Figure 13. For each query id i in d's descendant list, there is a number (indicated by the numbers below the boxes in Figure 11) records the information about *among all queries with ids $\leq i$ in the descendants of d, the total profit loss if they are all postponed by d_τ*. Again as shown in Figure 13, such total profits can be easily obtained during the SLA-tree construction.

Third, the last piece of information in SLA-tree is the left/right pointers for each item in a descendant list. For an internal node d, for each query id i in d's descendant list, i's left pointer records *the index, in the descendant list of d's left child, of the largest query id that is $\leq i$.* The right pointer of i is defined similarly but based on d's right child. A moment of thought will reveal that such a pointer records exactly the location that will be resulted by the binary search in the naive version of SLA-tree (as described in previous sections). Therefore such pointers, because their construction cost does not increase the overall time complexity, are the key to reduce the SLA-tree question answering time from $O(\log^2(NK))$ to $O(\log(NK))$.

In Figures 12 and 13, we show the pseudo code for operations **buildSlackTree()** and **merge()**, the two key steps in SLA-tree construction. In Figure 14, we show the pseudo code for **postpone()**, the key step in question answering by using SLA-tree.

```
Algorithm buildSlackTree(Q)
▷ input: Q, the list of queries with SLAs
▷ output: T, the slack tree
1:  Q1 ← decompose Q according to SLAs;
2:  Q2 ← {q ∈ Q1 : q.slack ≥ 0};
3:  Q3 ← Q2.sortBySlack();
▷ construct T by bottom-up merge
4:  s ← Q3.size();
5:  T ← a complete binary tree with size 2s-1;
6:  insert Q3, in order, into T as leaf nodes;
7:  i ← T.size() − 1;
8:  while (i+1) ≠ T.root do
9:      merge(T[i], T[i+1], T[i/2]);
10:     i ← i-2;
11: return T;
```

Figure 12: Implementation of buildSlackTree().

```
Algorithm merge(l,r,p)
▷ input: left child node l, right child node r
▷ updated: parent node p
1:  p.lBound ← l.lBound; p.uBound ← r.uBound;
2:  p.slack ← (l.uBound + r.lBound)/2;
3:  lPtr, rPtr, lIdx, rIdx ← 0's;
▷ merge
4:  while lPtr < l.desc.size() or rPtr < r.desc.size() do
5:      if l.desc[lPtr] < r.desc[rPtr]
6:          p.desc.append(l.desc[lPtr])
7:          p.profit.append(l.profit[lPtr])
8:          p.lPtr.append(lIdx); p.rPtr.append(rIdx);
9:          lIdx ← lPtr; lPtr ← lPtr+1;
10:     else if l.desc[lPtr] > r.desc[rPtr]
11:         p.desc.append(r.desc[rPtr])
12:         p.profit.append(r.profit[rPtr])
13:         p.lPtr.append(lIdx); p.rPtr.append(rIdx);
14:         rIdx ← rPtr; rPtr ← rPtr+1;
15:     else
16:         p.desc.append(l.desc[lPtr])
17:         p.profit.append(l.profit[lPtr]+r.profit[rPtr])
18:         p.lPtr.append(lIdx); p.rPtr.append(rIdx);
19:         lIdx ← lPtr; rIdx ← rPtr;
20:         lPtr ← lPtr+1; rPtr ← rPtr+1;
21: return;
```

Figure 13: Implementation of merge().

```
Algorithm postpone(T,n,τ)
▷ input: SLA-tree T, id n, time interval τ
▷ output: c, the profit loss due to
          postponing queries 1 to n by τ
1:  c ← 0;
2:  p ← T.root;
3:  i ← position of n in p.desc;
▷ traverse the tree
4:  while p is not a leaf node do
5:      if τ ≤ p.slack
6:          i ← p.lPtr[i];
7:          p ← p.leftChild;
8:      else
9:          c ← c + p.leftChild.profit[p.lPtr[i]];
10:         i ← p.rPtr[i];
11:         p ← p.rightChild;
12: return c;
```

Figure 14: Implementation of postpone().

5.2 Space and Time Complexity

We formally prove the space and time complexity of the above implementation. In the following, by SLA-tree we mean slack tree, because tardiness tree has the same complexities.

THEOREM 1 (SPACE COMPLEXITY). *SLA-tree has a space complexity of $O(NK\log(NK))$, where N is the number of queries in the query buffer and K is the maximal number of steps in the stepwise SLAs.*

PROOF. The number of leaf nodes in SLA-tree is bounded by NK and so the height of the SLA-tree is bounded by $\log(NK)$ and the total number of nodes bounded by $2NK$. The space that each node d requires is linear in the size of d's descendant list. Because each query id occurs at most $K-1$ times at a given tree level, the total size among all descendant lists at each level of the SLA-tree is bounded by NK. So the space complexity for SLA-tree is $O(NK\log(NK))$. □

THEOREM 2 (TIME COMPLEXITY). *The time complexity for building SLA-tree is $O(NK\log(NK))$ and for querying SLA-tree is $O(\log(NK))$.*

PROOF. The initial sorting of slacks takes time $O(NK\log(NK))$. During SLA-tree construction, a merge is called on each internal node d where the time for merge is linear in the size of the descendant list of d's left child and that of d's right child. Because each item in a descendant list at level l of the SLA-tree is visited exactly once during the merge at level *l-1*, the total amortized time for merges at each level of SLA-tree is $O(NK)$ and so the time complexity for constructing the whole SLA-tree is $O(NK\log(NK))$.

The time for a traversal of SLA-tree (i.e., answering one *postpone* question) is the time spent at each node multiplied by the number of nodes visited during the traversal: the former is a constant and the latter is $O(\log(NK))$. So the time complexity for querying SLA-tree is $O(\log(NK))$. □

6. APPLICATIONS OF SLA-TREE IN PROFIT-ORIENTED DECISION SUPPORT

Now by exploiting the two primitives in the SLA-tree framework, i.e., **postpone(m,n,τ)** and **expedite(m,n,τ)**, we are able to efficiently ask a variety of "what if" questions, whose answers can be of great value for supporting profit-oriented optimization of database management tasks. In this section we show three application examples for SLA-tree, namely scheduling, dispatching, and capacity planning[4].

6.1 SLA-Tree for Scheduling

In this subsection, we demonstrate how to use SLA-tree for supporting profit-oriented decisions on query scheduling. We study two cases. In the first case, queries are simply buffered and executed according to their arrival time. In other words, we can consider this case as a first-come-first-serve (FCFS) scheduling policy, whereas it is obvious that such a policy is cost-unaware. In the second case, we assume there already exists a baseline SLA-aware scheduling policy, such as the cost-base-scheduling (CBS [15]). However, even in this case, maximizing the total SLA-based profit, a special case of which is the knapsack problem [8], is an NP-hard problem. So most SLA-aware scheduling policies used certain heuristics (e.g., CBS assumes a distribution of the waiting time for each query, independent of other queries) and further improvement by using SLA-tree is possible.

Based on SLA-tree, we propose a very simple scheduling algorithm by directly looking at the profit change in the following way. For each query q_i in the query buffer, we ask "what will the profit gain be if we rush q_i, instead of q_1, to be executed next, with all other conditions unchanged." Then instead of picking q_1, which was in the front of the buffer, we select q_i, whose immediate execution brings in the maximum profit gain, to be executed next. Such a simple SLA-tree based method is expected to make SLA-unaware baseline policies to become SLA-aware, and improve already SLA-aware baseline policies by taking all the queries in the buffer into consideration.

Now the problem boils down to how to efficiently compute the profit gain of rushing each q_i to the front of the buffer. By expediting q_i, we change the total profit in two ways. First, there will be a potential profit gain because q_i is executed earlier than it was originally scheduled (e.g., originally q_i was not supposed to be completed on time, but it can be completed on time if it is rushed to be executed first). Second, there will be a potential profit loss because queries q_1 through q_{i-1} will be postponed by τ_i, the execution time of q_i, from their original schedule. That is, the potential profit gain of rushing q_i is

$$p_i = p_{gain}(q_i) - p_{loss}(q_1, \ldots, q_{i-1}).$$

The first part can be easily obtained by looking at q_i and the second part is exactly **postpone(1,i-1,τ_i)**, where τ_i is the execution time of q_i.

Our SLA-tree scheduling policy will pick q_j to be executed next where

$$j = \arg\max_i \left(p_{gain}(q_i) - p_{loss}(q_1, \ldots, q_{i-1})\right).$$

This scheduling policy takes $O(NK\log(NK))$ time by using SLA-tree: building the SLA-tree from scratch takes time $O(NK\log(NK))$; for each i such that $1 < i \leq N$, we call **postpone()** once for query q_i to answer "what if q_i is rushed to be executed first", and each question takes $O(\log(NK))$ time to answer.

6.2 SLA-Tree for Dispatching

So far we have focused on the single buffer/single server case. As long as in the system each database server has its own query buffer and the scheduling decision is made locally (i.e., independent of other servers), the SLA-tree framework can be applied to each database servers individually. However, there do exist issues that cannot be handled independently, where dispatching is such an example.

Assume we have a centralized dispatcher that assigns a query q to one of the several database servers that can execute the query. A naive dispatching policy, such as Round-Robbin or least-work-left (LWL), will dispatch queries regardless of the current *profit* conditions of the database servers. A more profit-aware dispatching policy can ask the "what if" question to each database server S_i, i.e., "What will be S_i's expected profit change if q is dispatched to S_i?" The profit change here depends on the profit brought by q itself as well as the potential profit loss brought by inserting q to the query buffer of S_i, which affects other queries in the buffer. After obtaining the answers to these "what if" questions, a profit-aware dispatcher can then dispatch the query

[4]Due to the space limitations, we could not include other applications that make use of **expedite()**.

q to the server that results in the maximal profit gain, or drop q if profit impacts for all the servers are negative and if admission control is activated.

Such profit change turns out to be answerable by SLA-tree in the following way. We first build an SLA-tree for queries in S_i's query buffer without inserting q and from the SLA-tree and read the total profit. And then we ask SLA-tree the potential impact on the profit of S_i if q is inserted in the middle of S_i's buffer (according to the scheduler) in $O(\log(NK))$ time. However, building an SLA-trees from scratch just to answer such a "what if" question is inefficient. Actually, it can be shown that as long as the scheduling policy satisfies a very minor condition, constructing a new SLA-tree is not needed. Instead, we can derive from the most recent SLA-tree, assuming it has been saved, in S_i to answer the dispatching related "what if" question in $O(\log(NK))$ time. Details are skipped due to the space limitations.

It is worth noting that in our SLA-tree based dispatching, the servers can be heterogeneous (e.g., in terms of processing power), because each server has its own SLA-tree. That is, the potential impact on the profit of S_i, if q is inserted in the middle of S_i's buffer, is computed based on the execution time of q on S_i, where such an execution time can be different for different servers. In addition, our SLA-tree based dispatching is scalable to the number of servers—the "what if" questions can be computed in parallel among the servers by using the SLA-trees at the servers.

6.3 SLA-Tree for Capacity Planning

Capacity planning is crucial for the success of a cloud database service provider. In this section, we show a case how SLA-tree can help online resource allocation. We focus on the following fundamental question "what is the profit margin of adding an additional database server?" An accurate answer to this question, or even a good approximation, is very valuable for a service provider.

Assuming currently there are m database servers, in total making a profit of $\$a$ per day. A naive estimation is that adding another server will bring in additional $\$a/m$ profit per day. Obviously, such an estimation may not be accurate. Consider two extreme cases. First, the system is over-provisioned, hence most of the time the database servers are idle. In such a case, adding another server may not improve profit proportionally, because the current capacity achieves the best profit already. In the second case, the system is heavily loaded. In such a case, adding another server, in addition to handling more queries, may help reduce the waiting time of queries in existing servers and therefore improve the profit in a super-linear rate.

Here we describe a way to use SLA-tree to get an approximation of the profit margin of an additional server. When an incoming query q arrives, the dispatcher asks the potential profit gain (or loss) of inserting q to each server and then picks the server S_i that reports the largest potential gain (or smallest loss), which we indicate by g_i. At the same time, the dispatcher asks the same "what if" question to a fictitious idling server and gets an answer g_0. Then $g_0 - g_i$ is the current profit margin of an additional server, and such a profit margin will be achieved by an execution time τ (of q) at the fictitious new server. Accumulating such potential margins over time, we can get an approximate answer to the profit margin of adding one additional server to the system.

7. EXPERIMENTAL STUDIES

In this section, we conduct experimental studies. We first focus on the effectiveness of SLA-tree in terms of supporting decisions on scheduling, dispatching, and capacity planning. Then, we investigate the efficiency and robustness of the SLA-tree framework.

7.1 Experiment Setting

In the experiments, we use three data sets that cover different data statistics. The first data set is a synthetic data set with query execution times following an exponential distribution, which has been commonly used and analyzed in previous studies [16], with a mean execution time of 20ms. For the second set of experiments, we used job traces derived from the published parameters of real-life systems [9], where query execution times follow a Pareto (i.e., long tail) distribution. Such a distribution represents a typical mixed workloads in real systems. Following [9], we set the minimum value of the Pareto to be 1ms and the Pareto index to be 1. The resulting workloads have mean values around 25ms. The histograms of query execution time for these two data sets are given in Figure 15 (note the logarithm scale for Pareto).

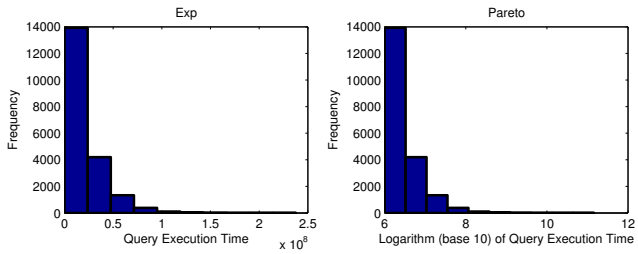


Figure 15: The histograms of query execution time for the first and the second data sets.

The third data set is the Star Schema Benchmark (SSBM) [1], which is a data warehousing benchmark derived from TPC-H. There are 13 queries in SSBM benchmark and we generate workloads by uniformly sampling among them. For the query execution time, we exactly followed the figures reported by Abadi et al. [1]. The query execution time is given in Table 1. For query arrival rate, we adopt the Poisson distribution. For convenience we refer the three data sets as Exp, Pareto, SSBM, respectively.

Table 1: Query execution time for queries in SSBM benchmark, as reported by Abadi et al. in [1].

query	q_1	q_2	q_3	q_4	q_5	q_6	q_7
time (ms)	1.0	1.0	0.2	15.5	13.5	11.8	16.1
query	q_8	q_9	q_{10}	q_{11}	q_{10}	q_{13}	Average
time (ms)	6.9	6.4	3.0	29.2	22.4	6.4	10.2

In terms of SLAs, for obvious business reasons, there are not many publicly available numbers. Therefore, we choose to use several case studies to illustrate the potential profit benefits of the SLA-tree framework. The first case, which we refer to as SLA-A, is a 1/0 profit SLA as given in Figure16(a). Results for case SLA-A offer some intuitive interpretation—the profit loss is the same as the fraction of customers who missed their (sole) deadline. The second case, which we

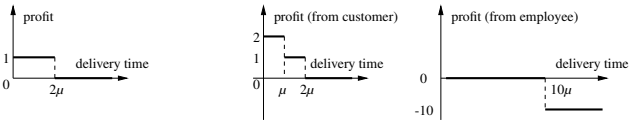
Table 2: Scheduling: average profit loss per query for different algorithms.

SLA Type	SLA-A									SLA-B								
Workload	Exp			Pareto			SSBM			Exp			Pareto			SSBM		
System Load	0.5	0.7	0.9	0.5	0.7	0.9	0.5	0.7	0.9	0.5	0.7	0.9	0.5	0.7	0.9	0.5	0.7	0.9
FCFS	.332	.454	.601	.280	.404	.540	.368	.494	.634	.865	1.08	1.36	.713	1.04	1.41	.714	.950	1.22
FCFS+SLA-tree	.274	**.341**	**.416**	**.258**	.371	.492	.307	**.370**	**.437**	.755	.896	1.04	.662	.959	1.28	.615	.753	.908
CBS	.275	.347	.425	.259	.371	.495	.308	.374	.445	.752	.897	1.03	.662	.956	**1.27**	.616	.752	.897
CBS+SLA-tree	**.273**	.342	.418	**.258**	**.370**	**.491**	**.306**	**.370**	.439	**.750**	**.890**	**1.02**	**.660**	**.952**	**1.27**	**.614**	**.742**	**.893**

Table 3: Dispatching: average profit loss per query for different algorithms.

SLA Type	SLA-A									SLA-B								
Workload	Exp			Pareto			SSBM			Exp			Pareto			SSBM		
server #	2	5	10	2	5	10	2	5	10	2	5	10	2	5	10	2	5	10
LWL/CBS	.381	.300	.225	.408	.266	.218	.412	.339	.283	.964	.821	.689	1.06	.671	.527	.804	.674	.562
LWL/SLA-tree	.375	.296	.223	.404	.260	.209	.408	.336	.282	.958	.817	.686	1.05	.657	.513	.794	.670	.561
SLA-tree/SLA-tree	**.300**	**.242**	**.208**	**.220**	**.056**	**.048**	**.341**	**.282**	**.252**	**.869**	**.743**	**.654**	**.613**	**.202**	**.168**	**.717**	**.568**	**.485**

refer to as SLA-B, is a mixture of two SLAs as given in Figure16(b). We use case SLA-B to simulate the case where there are two types of customers with different profit profiles: regular buyers (left) and internal employees (right). Furthermore, when applying SLA-B to data sets Exp and Pareto, we assume the query execution time and query SLA are uncorrelated. In addition, we assume queries from buyers are 10 times more frequent than queries from internal employees. For the SSBM data set, we make the query execution time and query SLA correlated in the following way: if query execution time is more than 20ms, then we assume it is from an internal employee, otherwise, it is from a regular buyer; and then we assign corresponding SLAs to the query. For each experiment, we generate 20K queries and use the first 10K to warm up the system and the second 10K to measure the performance. For each test, we repeat 10 times with different random seeds and report the average.



(a) SLA-A (b) SLA-B

Figure 16: The SLAs used in the experiments, where μ is the mean query execution time for the corresponding workload.

As we consider the revenue part of the profit but not the cost (e.g., hardware cost) part, we choose to report the average profit loss for each customer with respect to the ideal world, where all the customers' first deadlines are met. The lower this average profit loss number is, the better the performance is. In the case of 1/0 profit SLA, this average profit loss is the same as the fraction of customers that miss their deadline.

7.2 Experiments on Scheduling

We first study the performance of our scheduling algorithm based on SLA-tree. We study the case where the queries are executed in a first-come-first-serve (FCFS) order and the case where a sophisticated cost based scheduling (we chose CBS by Peha et al. [15]) is used to determine the original order of query execution.

Table 2 reports the average profit loss of FCFS and CBS, as well as our new algorithms of combining SLA-tree with FCFS and CBS. In the table we report the performance for different data sets under different system loads and different SLAs. From the results we have several observations. First, using SLA-tree *always* improves over the original scheduling, whereas the improvement on FCFS is more significant. The improvement over CBS, albeit marginal[5], is due to the fact that CBS assigns a priority score to each query *independent of* the current system condition (e.g., when computing the priority score for a query q, CBS does not care whether or not there are other queries waiting in the buffer), whereas SLA-tree uses *all* the current queries in the buffer to analyze a series of what-if scenarios. Second, using SLA-tree to adjust the naive FCFS gives performance similar to that of the sophisticated CBS. In addition, recall that scheduling is only one of the application areas for SLA-tree, whereas CBS is a scheduling policy and cannot deliver other information management capabilities of SLA-tree.

7.3 Experiments on Dispatching

For dispatching, we compare SLA-tree dispatching with the well-known least-work-left (LWL) algorithm, which has shown state-of-the-art performance [12]. Table 3 reports the performance comparison (using CBS as the baseline scheduling). As can be seen, the profit-aware SLA-tree dispatching improves the profit significantly, especially for the Pareto workload. In addition, in the table we also report the case where LWL is used with CBS-SLA-tree scheduling, where the profit improvement is much less significant, and this verifies that the profit gain *does* come from the profit-oriented dispatching.

7.4 Experiments on Capacity Planning

For capacity planning, we conduct the experiments in the following way. We first generate a workload of queries and run the system with n server, and then use *the same* workload trace to run the system with *n+1* servers. The profit change between the two cases is considered as the ground truth, namely the real profit margin of adding one server to the current system with n servers.

Table 4 reports the performance in terms of estimation error for $n = 2, \ldots, 10$. As can be seen, SLA-tree provides a practical estimation compared to ground truth.

7.5 Experiments on Robustness

One requirement for the SLA-tree framework is that the

[5]The relatively good performance of CBS confirms the claim in [15] that CBS performs almost as well as the oracle algorithm.

Table 4: Profit gain estimation performance for SLA-A, with the system load set to 0.9.

Server #		2	3	4	5	6	7	8	9	10
Exp	ground truth	.115	.086	.070	.063	.056	.050	.047	.045	.039
	SLA-tree estimation	.170	.097	.066	.050	.038	.030	.026	.021	.018
Pareto	ground truth	.148	.063	.034	.018	.021	.009	.013	.013	.010
	SLA-tree estimation	.081	.032	.019	.013	.010	.008	.007	.006	.005
SSBM	ground truth	.126	.094	.080	.067	.056	.054	.053	.048	.047
	SLA-tree estimation	.184	.104	.073	.053	.042	.034	.029	.024	.020

Table 5: Robustness of scheduling: average profit loss per query for different algorithms, with the system load set to 0.9.

SLA Type	SLA-A									SLA-B								
Workload	Exp			Pareto			SSBM			Exp			Pareto			SSBM		
σ^2 of error scale	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0
CBS	.425	.467	.550	.495	.529	.582	.445	.491	.581	1.04	1.07	1.21	1.26	1.35	1.51	.897	.992	1.26
CBS+SLA-tree	.418	.467	.551	.491	.528	.582	.439	.488	.581	1.04	1.08	1.21	1.26	1.35	1.51	.893	1.02	1.28

exact query execution time has to be known in advance, which may be difficult to obtain accurately. In this experiment, we test the robustness of SLA-tree, assuming the query execution time is estimated with certain errors.

Following [9], we apply SLA-tree algorithms by using the *estimated* query execution time while at the running time, we scale the estimated execution time of each query by a random number to get the *real* execution time. We studied two cases where the scaling factor follows different Gaussian distributions: the scaling factor following N(1,0.2), for a smaller error, and following N(1,1.0), for a larger error.

Table 5 compares the performance of CBS and CBS+SLA-tree scheduling policies, under a system load of 0.9. As can be seen, both CBS and SLA-tree suffer from the inaccurate estimation in a similar way and this is reflected in the bigger profit losses. As the error gets larger, i.e., from N(1,0.2) to N(1,1.0), the profit loss for both scheduling policies gets bigger. Table 6 compares the performance of LWL and SLA-tree dispatching algorithms, with the number of servers set to 5. As can be seen, again, both LWL and SLA-tree suffer from the inaccurate estimation and this is reflected in the bigger profit losses. However, it is interesting to observe that the fact that SLA-tree significantly outperforms LWL still remains valid in all the cases.

7.6 Experiments on Running Time

Finally, we study the scalability of the SLA-tree algorithm. We conduct the study by logging, during the execution of SLA-tree scheduling, the time needed for building and querying SLA-tree under different query buffer lengths.

To analyze the time complexity, we pushed the system into the limits as follows. First, we set the system load to be 0.99, the borderline of saturation; second, we set the threshold in SLA-A to very large values in order to force SLA-tree algorithm to build and query very large slack trees (because a lot of queries will have positive slacks).

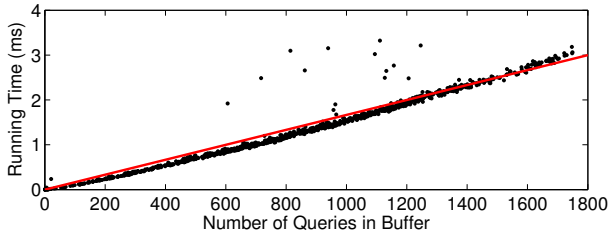


Figure 17: The scatter-plot for the running time of building and querying SLA-tree under different query buffer lengths.

In Figure 17 we show the scatter-plot of the running time of the SLA-tree scheduling algorithm (including both building and querying the SLA-tree) versus the number of queries waiting in the buffer. As can be seen, even up to thousands of queries to be scheduled, the running time of SLA-tree (again, including both the time of building the SLA-tree from scratch and that of querying the SLA-tree) behaves in a rather linear manner[6]. Such running time, e.g., less than 1ms for scheduling 500 queries (on a Xeon PC with 3GHz CPU), demonstrates that the SLA-tree framework is practical for real applications.

8. SOME LIMITATIONS OF SLA-TREE

In this section, we discuss some limitations of the SLA-tree framework.

8.1 SLA-tree Requires An Existing Query Execution Order

An underlining assumption of the SLA-tree framework is that there exists a known order of execution among the queries in the buffer. This is a main limitation of the SLA-tree framework. The reason for this assumption is that in the SLA-tree framework, we have to know what will happen in the near future, in order to improve performance and to answer various "what if" questions. Fortunately, however, in real applications, such an existing order of execution is not very uncommon. For most commonly used static prioritization policies, such an order can be easily derived, e.g., from the query arrival time (FCFS), query deadline (EDF), query execution time (SJF), or query cost (CBS).

8.2 SLA-tree Based Scheduling is A Greedy Algorithm

The SLA-tree based scheduling is a greedy algorithm and cannot guarantee the schedule to be globally optimal. This is true even in the offline case, where all the queries are known beforehand. Table 7 shows a simple example with three queries q_1, q_2, and q_3, each of which has a single deadline and corresponding profit. From the table we can see that starting from the original execution order (q_1, q_2, and then q_3), rushing either q_2 or q_3 to the front of the buffer

[6]The abnormalities in the plot are most likely due to OS factors such as cache missing.

Table 6: Robustness of dispatching: average profit loss per query for different algorithms, with the number of server set to 5.

SLA Type	SLA-A									SLA-B								
Workload	Exp			Pareto			SSBM			Exp			Pareto			SSBM		
σ^2 of error scale	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0	0.0	0.2	1.0
LWL/CBS	.300	.330	.414	.266	.261	.344	.339	.374	.458	.821	.840	.954	.671	.640	.830	.674	.711	.858
LWL/SLA-tree	.296	.294	.413	.260	.259	.340	.336	.371	.455	.817	.838	.956	.657	.634	.841	.670	.710	.865
SLA-tree/SLA-tree	.242	.294	.353	.056	.097	.130	.282	.328	.380	.743	.796	.887	.202	.262	.351	.568	.685	.849

will decrease the total profit. Therefore, our SLA-tree based scheduling will miss the schedule with higher total cost, i.e., rushing q_2 first and then rushing q_3.

Table 7: A simple example of scheduling.

	execution time (sec)	deadline (sec)	profit ($)
q_1	1	1	1
q_2	0.5	1	0.6
q_3	0.5	1	0.6

However, there are two points we want to make. First, such a scheduling problem, with weighted profits and deadlines, is NP-complete [15]. Second, by using an induction argument, we can easily show that in an offline setting, the SLA-tree based scheduling always guarantees a total profit that is higher than (or at least the same as) that of the original schedule.

9. CONCLUSION AND FUTURE WORK

We proposed a framework that uses a novel data structure, SLA-tree, to efficiently support various SLA-based, profit-oriented decisions for databases in cloud computing. Both theoretical analysis and experimental studies demonstrated that SLA-tree could efficiently provide valuable and accurate information that can be used by cloud database service providers for profit-oriented decisions such as scheduling, dispatching, and capacity planning. Therefore, we believe our SLA-tree based framework has great potentials in cloud computing.

For future work, we are investigating how to implement an *incremental* version of SLA-tree to further reduce its time complexity.

10. ACKNOWLEDGMENTS

We thank Michael Carey, Hector Garcia-Molina, and Jeffrey Naughton for the insightful discussions and comments. We thank Evaggelia Pitoura for valuable suggestions. We thank Oliver Po and Wang-Pin Hsiung for their great help with the experimental studies.

11. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, 2008.

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

[3] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CCGRID*, 2002.

[4] G. R. Dattatreya. *Performance Analysis of Queuing and Computer Networks*. Chapman and Hall, 2008.

[5] S. Elnaffar, P. Martin, and R. Horman. Automatically classifying database workloads. In *CIKM*, 2002.

[6] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[7] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, 2009.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[9] C. Gupta, A. Mehta, S. Wang, and U. Dayal. Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In *EDBT*, 2009.

[10] J. R. Haritsa, M. J. Carey, and M. Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2(2), 1993.

[11] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *HPDC*, 2004.

[12] 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. In *WWW*, 2008.

[13] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD*, 2010.

[14] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., 2004.

[15] J. Peha and F. Tobagi. Cost-based scheduling and dropping algorithms to support integrated services. *IEEE Trans. on Comm.*, 44(2):192–202, 1996.

[16] F. I. Popovici and J. Wilkes. Profitable services in an uncertain world. In *ACM/IEEE SC*, 2005.

[17] B. Schroeder and M. Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. *Cluster Computing*, 7(2):151–161, 2004.

[18] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, 2006.

[19] N. Ye, E. S. Gel, X. Li, T. Farley, and Y.-C. Lai. Web server qos models: applying scheduling rules from production planning. *Comput. Oper. Res.*, 32(5), 2005.

[20] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena. Resilient workload manager: taming bursty workload of scaling internet applications. In *ICAC*, 2009.

[21] L. Zhang and D. Ardagna. Sla based profit optimization in autonomic computing systems. In *ICSOC*, 2004.