iCBS: Incremental Cost-based Scheduling under Piecewise Linear SLAs

Yun Chi
NEC Laboratories America
10080 N. Wolfe Rd., SW3-350
Cupertino, CA 95014, USA
ychi@sv.nec-labs.com

Hyun Jin Moon
NEC Laboratories America
10080 N. Wolfe Rd., SW3-350
Cupertino, CA 95014, USA
hjmoon@sv.nec-labs.com

Hakan Hacıgümüş
NEC Laboratories America
10080 N. Wolfe Rd., SW3-350
Cupertino, CA 95014, USA
hakan@sv.nec-labs.com

ABSTRACT

In a cloud computing environment, it is beneficial for the cloud service provider to offer differentiated services among different customers, who often have different cost profiles. Therefore, cost-aware scheduling of queries is important. A practical cost-aware scheduling algorithm must be able to handle the highly demanding query volumes in the scheduling queues to make online scheduling decisions very quickly. We develop such a highly efficient cost-aware query scheduling algorithm, called iCBS. iCBS takes the query costs derived from the service level agreements (SLAs) between the service provider and its customers into account to make cost-aware scheduling decisions. iCBS is an incremental variation of an existing scheduling algorithm, CBS. Although CBS exhibits an exceptionally good cost performance, it has a prohibitive time complexity. Our main contributions are (1) to observe how CBS behaves under piecewise linear SLAs, which are very common in cloud computing systems, and (2) to efficiently leverage these observations and to reduce the online time complexity from $O(N)$ for the original version CBS to $O(\log^2 N)$ for iCBS.

1. INTRODUCTION

In a cloud computing environment, service providers offer vast IT resources to large sets of customers with diverse service requirements. By doing so, the service providers leverage the customer volume to achieve economies of scale. A service provider usually has contracts with its customers in the form of service level agreements (SLAs), where the SLAs can be about many aspects of a cloud computing service such as availability, security, response time, etc. An SLA indicates the level of service agreed upon as well as the associated cost if the service provider fails to deliver the level of service. In this work, we focus on this cost defined by the SLAs. Obviously, optimizing the cost while serving a large number of customers is vital for the cloud service providers. Such an optimization has to take many aspects of a cloud computing system into consideration, such as capacity planning, dispatching, and scheduling. Therefore, a single all-inclusive solution is very difficult to obtain (see Appendix A for more discussions). Instead, in this paper we focus on a local optimization problem, namely to schedule queries from diverse customers in a cost-aware way in order to minimize the SLA cost to the service provider.

Proceedings of the VLDB Endowment, Vol. 4, No. 9

The problem of cost-aware scheduling has been studied in the past in the areas of computer networks [12] and real-time databases [7]. However, it is somewhat discouraging to note that the developed algorithms have not been widely deployed in real applications, mainly due to their high complexity. Such a situation seems to be changing with the advance of the cloud computing. This is evident from the recent new interests in cost-aware scheduling algorithms in the database area [1, 4, 5]. We believe that the cloud computing environment offers two reasons for such a change.

Change in resource usage for scheduling: In order to make scheduling decisions, we have to use system resources (e.g., CPU cycles and memory). In computer networks, for example, the ratio between the resources used by scheduling and those used by processing a network packet has to be very low, because of the limited resource capacity at a network router. This is a main reason why simple policies such as first-come-first-serve (FCFS), shortest-job-first (SJF), and earliest-deadline-first (EDF) were preferred. In comparison, this ratio of resource usages is very different in the cloud computing environment. In the cloud computing environment, a higher resource usage for scheduling is acceptable, as long as doing so is justified by the reduction in the query execution cost.

Availability of cost functions: In traditional computer networks and real-time databases, there is a lack of consensus on what the most reasonable cost function should be. In the cloud computing environment, such cost functions are usually available from the contracts between the service providers and their customers, in the form of service level agreements (SLAs). In addition, the SLAs used by cloud service providers are usually piecewise linear functions because they can be defined by the clauses in business contracts. Such piecewise linear SLAs, as we will show, make many cost-aware scheduling computations more tractable.

Based on the above motivations, we investigate a cost-aware scheduling algorithm, CBS, that has been proposed by Peha and Tobagi [10, 12]. The main reason for us to choose

CBS is its exceptionally good cost performance, which often approaches the lower bound of theoretically optimal total cost [10]. Our own experiments, as will be shown later, verified this claim, showing that CBS outperforms several recently proposed cost-aware scheduling algorithms in the database systems [4] and supercomputing [8] communities. However, CBS suffers from one weak point—its time complexity for making each scheduling decision is $O(N)$, where N is the number of queries to be scheduled. Such a linear time complexity is prohibitive in a cloud computing system, where there can be a very large number of queries to be scheduled and where the online decisions have to be made extremely quickly. We will discuss CBS in detail in the next section. Here we only point out that in CBS, the priorities of the queries are *dynamically* changing over time. On one hand, such dynamically changing priorities capture the change of urgency of queries over time (e.g., as a query approaching its deadline) thereby making CBS superior to algorithms that simply assign *static* priority scores to the queries. On the other hand, because the query priorities have to be re-computed each time a scheduling decision is made, it seems that the linear time complexity $O(N)$ is a necessary price that we have to pay.

However, in this paper, we demonstrate that the linear time complexity of CBS can be avoided, under piecewise linear cost functions (SLAs). The main contributions of this paper are two-fold. First, we observe that in CBS, although the priority scores of the queries change continuously over time, the *relative order* among the priority scores only changes as discrete events. In other words, the relative order changes over time as a series of discrete snapshots. Our second contribution is to develop an efficient scheduling algorithm, iCBS, to exploit the above observation. In iCBS, the above snapshots are incrementally maintained, where from these snapshots, the query with the top priority can be obtained very quickly. Our iCBS algorithm makes *exactly the same* scheduling decisions as CBS. But compared with the $O(N)$ time complexity of CBS, iCBS achieves a time complexity of $O(\log N)$ for *many* piecewise linear SLAs, and $O(\log^2 N)$ for *all* piecewise linear SLAs, by using techniques borrowed from the field of computational geometry. iCBS is also used as the scheduler in our comprehensive data management platform in the cloud, CloudDB [6].

2. BACKGROUND AND RELATED WORK

In this section, we provide background information. We first discuss SLAs, especially various piecewise linear SLAs. Then we survey recent work on SLA-based scheduling algorithms. Finally, we introduce the CBS algorithm in detail.

2.1 Service Level Agreements (SLAs)

A service level agreement (SLA) is a contract between a service provider and its customers. An SLA indicates the level of service agreed upon as well as the associated cost if the service provider fails to deliver the level of service. SLAs can be about many aspects of a cloud computing service such as availability, security, response time, etc. In this paper, we focus on SLAs about query response time. That is, an SLA is a function $f(t)$, where t is the response time of a query and $f(t)$ is the corresponding cost if the response time is t.

Fig. 1 shows several examples of SLAs used in previous work (e.g., [1, 4, 8]). Fig. 1(a) shows a step-shaped cost function, where the cost to the service provider is c_1 if the

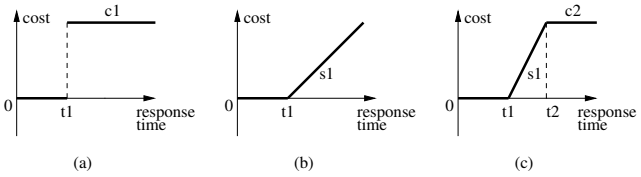

Figure 1: Examples of SLAs used in previous work.

query response time is later than its deadline t_1. Fig. 1(b) shows a cost function that increases proportionally to the tardiness, after a deadline t_1 is missed. In Fig. 1(c), the cost has a bound c_2, which indicates the maximum cost a query can introduce.

It is worth noting that although the SLAs shown in Fig. 1 represent a mapping between system-level metrics (e.g., query response time) and costs, the system-level performance can be dramatically different from the cost performance. For example, assume a service provider offers services to two customers, a gold customer and a silver customer, by using a shared database server. Because the customers have different SLAs, simply from the distribution of query response time for all the queries, we are not able to tell how good the performance is in terms of cost. This is because the latter depends on other factors such as the shape of the SLAs from the gold and silver customers, the workload ratios between the two customers, how the scheduling priority is determined, and so on.

Instead of *cost* functions, SLAs in some previous work are in the form of *profit* functions. We show in Appendix B.1 that these two are equivalent. In addition, in this paper we use *per-query* SLAs instead of *per-customer* or *quantile* SLAs, because we believe the former have more flexibility. For example, under per-query SLAs, queries from the same customer may have different SLAs, depending on the expected execution time and resource consumption of the query, or depending on the budget quota that the customer has used so far. In addition, there exists techniques (e.g., [3]) that directly map per-query SLAs to quantile SLAs.

2.2 Piecewise Linear SLAs

We focus on piecewise linear SLAs, as they exhibit very good model for capturing service contracts. By *piecewise linear* SLA, we mean that the cost function $f(t)$ can be divided into finite segments along the time line and in each segment, $f(t)$ is a function linear in t. We next provide several representative piecewise linear SLAs and show that these SLAs are able to capture various rich semantics.

Fig. 2 shows several examples of piecewise linear SLAs. In the figure, the x-axis represents query response time (for easy illustration, we assume the query arrives to the system at time 0); the y-axis represents the cost corresponding to different response time t. We discuss these cases in detail.

Fig. 2(a) describes an SLA with a linearly increasing cost versus the response time. That is, the cost is proportional to the response time. Such an SLA reflects a target of weighted mean response time. A moment of thought will reveal that under such SLAs, minimizing the total cost among all queries is equivalent to minimizing the weighted response time of all queries, where the weight for each query is proportional to the slope of its linearly increasing SLA.

Fig. 2(b) describes an SLA in the form of a step function, where the cost is c_0 if the response time is less than t_1 and c_1 otherwise. As a special case, when $c_0 = 0$ and $c_1 = 1$ for all queries, the average cost under such an SLA is the same

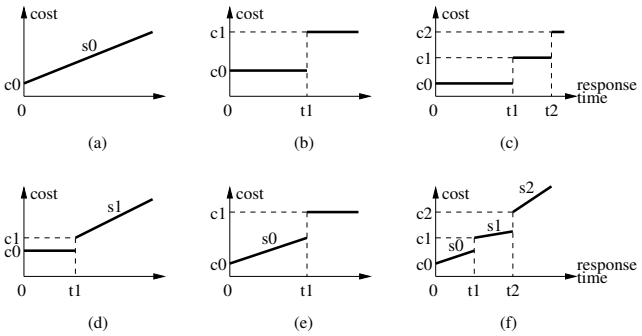

Figure 2: Representative piecewise linear SLAs.

as the fraction of queries that miss their (sole) deadlines. It is worth noting that in this case, and in all other cases in the figure, the parameters such as c_0, c_1, and t_1 can be different for *each individual query*. That is, each query can name its own deadline and corresponding cost for missing that deadline.

Fig. 2(c) describes a staircase-shaped SLA. The cost is c_0 if the response time is less than t_1, c_1 if the response time is between t_1 and t_2, and so on. Finally, the cost becomes a fixed value (c_2 in this example) after the last deadline (t_2 in this example). Such an SLA can be used to capture multiple semantics simultaneously. For example, when checking out a shopping cart at an online e-commerce Web site, a response time less than t_1 may result in good user experience whereas a response time longer than t_2 may reflect certain unacceptable performance (e.g., t_2 can be the time-out threshold of the user's browser, after which the query result becomes invalid).

Fig. 2(d) describes the mixture of a step function and a linear function. That is, the cost remains constant up to a response time t_1 and then grows linearly afterward. This SLA allows a *grace period*, up to t_1, after which the cost grows linearly over time. This example also illustrates that in general, SLAs may contain cost "jumps" (at time t_1 in this example) in the cost functions.

Fig. 2(e) describes another way of mixing the step and linear functions. The cost initially grows linearly, but after time t_1, it becomes a constant. This SLA captures, for example, the concept of a proportional cost with a time-out. That is, after t_1, the damage has been done and so the cost has reached its maximal penalty value.

Fig. 2(f) describes a piecewise linear SLA in the most general form. That is, the slope at each time segment can be different and there can be cost jumps between consecutive segments in the SLA.

Other than capturing various rich semantics, piecewise linear SLAs also make many computations in cost-aware scheduling more tractable, which is a key to our work.

2.3 Related Work

Scheduling is a mature problem and has long been extensively studied in various areas such as computer networks, database systems, and Web services. There exist many different scheduling policies and there is a vast amount of theoretical results and practical analysis on various policies (e.g., see [9]). However, in most existing work on scheduling, the performance metrics are low-level metrics (e.g., average query response time or stretch [5]). Instead, in this subsection, we mainly survey some of the recently proposed cost-aware scheduling strategies. As will be seen, this previous work assumes some specific SLA shapes and designs scheduling policies accordingly, and very often, the SLA shapes are piecewise linear.

Guirguis et al. [4] aim to minimize *tardiness* as shown in Fig. 2(d), which is i) zero if a query is finished before its deadline, or ii) the query's finish time minus its deadline, if the query misses it deadline. They analyze that EDF is a better choice at minimizing tardiness when the system load is low, and SRPT (Shortest Remaining Processing Time) is a better choice when the load is high. They propose ASETS*, a novel scheduling method that combines EDF and SRPT using a heuristic, and show that ASETS* outperforms both EDF and SRPT over a wide range of loads.

Irwin et al. [8] consider SLA cost functions in the form of a gradually increasing penalty with a *bound*, as shown in Fig. 2(e) and Fig. 1(c). In order to maximize service provider's profit, they employ two economic concepts, namely *present value* and *opportunity cost*, and propose a scheduling policy called FirstReward that combines the two concepts. Irwin et al. focus on the case of preemptive execution, where a query can be stopped in the middle of execution and resumed later, which is often the case in the supercomputing context. Note that we assume non-preemptive execution in this paper, which is often the case in data-intensive computing, such as database query execution. Lastly, note that FirstReward's scheduling decision takes $O(N^2)$ time, where N is the number of queries in the queue, and this may cause a significant performance overhead.

Peha [11] proposes a scheduling and admission control method by using a priority token bank. It is assumed that jobs can be grouped into N classes and jobs in each class are treated equally (by using FIFO). Because different classes have different profiles (e.g., guaranteed or best-effort, different costs for missing deadlines, etc.), a prioritization scheme is designed to choose the next class to serve. This method assumes job classes (videos, emails, etc.) and does not differentiate jobs within the same class. Therefore it is much more restricted than the CBS framework, in which each query can have its own cost profile.

In [1], we study staircase-shaped SLAs as shown in Fig. 2(c). We build a data structure, called SLA-tree, to capture the SLAs of the queries in the queue and propose a greedy algorithm to improve a given scheduling algorithm. However, the SLA-tree based scheduling algorithm in [1] can only be used to improve an existing scheduling algorithm and it can only handle staircase-shaped SLAs.

While this previous work focuses on specific types of SLA cost function, CBS (which we will introduce in detail momentarily) supports all types of SLA cost functions. Also, our iCBS supports *all* piecewise linear SLA cost functions, which is much more expressive than the cost functions in most of the previous work. We will compare the cost performance of iCBS with that of some previous work mentioned above in the experiment section.

2.4 Cost-based Scheduling (CBS) in Detail

CBS is a cost-based scheduling algorithm proposed by Peha and Tobagi [10, 12]. The goal of CBS is to schedule queries in the queue, where each query has its own cost SLA in terms of response time, in a way that minimizes the total *expected* cost. However, achieving such a minimal cost, even for the offline case where future arrivals of queries are known beforehand, is an NP-complete problem. The heuris-

tic used by CBS is to compute a priority score for each query q in the queue based only on (1) q's SLA and (2) how long q has been waiting in the queue. Because the priority score of q is computed *independently* from other queries in the queue, and because the query with the highest value for the computed priority is chosen for execution, CBS has a time complexity of $O(N)$ for making a scheduling decision when there are N queries in the queue.

More specifically, for each query q in the queue, CBS computes an *expected cost reduction* of executing q immediately, rather than delaying it further. The expected cost reduction at a given time t (where t is the time when the system schedules the next query to execute) is computed by

$$r(t) = \int_0^\infty ae^{-a\tau} f(t - t_0 + \tau)d\tau - f(t - t_0) \qquad (1)$$

where t_0 is the time at which the query arrives to the system and a is a free parameter (the only parameter) to be set by CBS. For convenience, in the rest of the paper, we refer to "the expected cost reduction $r(t)$ computed by CBS at time t" as the *CBS score at time t*.

To intuitively describe how CBS works, in Fig. 3 we illustrate how the CBS score is computed at time t' and t'' for a query q with a SLA of Fig. 2(b), and we assume the query arrives to the system at time t_0. Fig. 3(a) shows how the CBS score at time t' is obtained. At t', if q is served immediately, the cost will be $f(t' - t_0) = c_0$. If, on the other hand, we decide to postpone q, then it is assumed that q will stay in the queue for an amount of waiting time τ that follows an exponential distribution with mean $1/a$. So the *expected* cost for postponing q is $\int_0^\infty ae^{-a\tau} f(t' - t_0 + \tau)d\tau$. The CBS score is the net gain of choosing to serve q now instead of postponing it further[1].

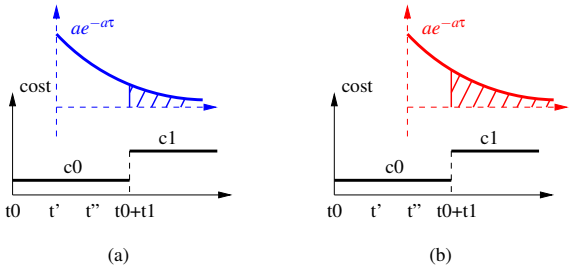

Figure 3: The intuition of CBS: how CBS scores are computed (a) at time t' and (b) at time t''.

So as can be seen from Fig. 3, there are two factors that affect the CBS score of q. The first factor is the SLA of q, i.e., the values of c_0, c_1, and t_1; the second factor is how long q has been waiting in the system. This second factor can be illustrated by comparing Fig. 3(a) with Fig. 3(b). Compared with the situation at time t', at time t'', q has been waiting in the system longer, and its deadline for the next cost jump in the SLA (the jump at time t_0+t_1) is more urgent. As a result, its CBS score, which is reflected by the shaded areas in the figures, has grown higher.

[1]In the original CBS formula [12], the query execution time q_s of a query q is used to derive the real urgency of q (i.e., the time when the execution of q has to be *started*). In addition, the CBS score of q is $r(t)/q_s$ instead of $r(t)$. For convenience, without loss of generality, we describe CBS as if q_s were 0, and use $r(t)$ instead of $r(t)/q_s$. In Appendix B.2, we show that we can compensate these differences by shifting and scaling the SLA of q.

Some additional discussions about CBS are given in Appendix B, in which (1) we offer more details on CBS, (2) we prove that for CBS, without loss of generality we can always set $c_0 = 0$ in piecewise linear SLAs, and (3) we discuss two factors, the multi-programming-level (MPL) and the errors in the estimation of query execution time, that may affect the effectiveness of CBS.

From Equation (1) we can see that CBS is inefficient in that at each time of scheduling, a CBS score has to be computed for each query in the queue and each computation involves an integration. Assuming there are N queries waiting in the queue and assuming no new query arrives, the total number of CBS scores to be computed in order to finish the N queries one by one is $N^2/2$. As a consequence, the average time for online scheduling of each query is $O(N)$. To alleviate such expensive computation of CBS, in the next two sections, we propose an efficient incremental version of CBS, namely iCBS, that has $O(\log^2 N)$ time complexity under piecewise linear SLAs.

3. ICBS WITH $O(\log N)$ COMPLEXITY

We develop our incremental variation of CBS, iCBS, in this section and the next. For certain special forms of piecewise linear SLAs, namely those described in Figs. 2(a)–2(d), iCBS is able to achieve an $O(\log N)$ time complexity (compared with $O(N)$ for original CBS). We postpone iCBS for the general piecewise linear SLAs, namely those described in Figs. 2(e) and 2(f), to the next section.

3.1 CBS Score without Integration

We start by showing a method to compute the CBS scores under piecewise linear SLAs *without* conducting any integration. The method is based on a canonical decomposition of piecewise linear SLAs and on rules of calculus. We provide the derivation details in Appendix C and give the final formula, for the SLA described in Fig. 2(f), as the following

$$r(t) = \frac{1}{a}s_0 + (j_1 + \frac{s_1 - s_0}{a})e^{-a(t_1 - t)} + (j_2 + \frac{s_2 - s_1}{a})e^{-a(t_2 - t)} \qquad (2)$$

where s_0, s_1, and s_2 are the slopes of the SLA segments; j_1 and j_2 are the heights of the jumps at time t_1 and t_2. Note that Equation (2) is valid only for $t < t_1$.

Therefore, when SLAs are piecewise linear, we can efficiently compute CBS scores at any time t by evaluating the exponential values at the places where SLA segments change, without conducting any integration.

3.2 iCBS for the SLA in Figure 2(a)

It can be easily shown that in CBS, if the SLA is a simple linear function, like that in Fig. 2(a), then the CBS score $r(t)$ is a constant over t. The reason is that by reshuffling Equation (1), we have

$$\begin{aligned} r(t) &= \int_0^\infty ae^{-a\tau}[f(t - t_0 + \tau) - f(t - t_0)]d\tau \\ &= \int_0^\infty ae^{-a\tau} b\tau d\tau = b/a \end{aligned}$$

where b is the slope of the linear SLA. So as can be seen, in such a case, each query has a *time-invariant* CBS score and so incremental CBS can be trivially implemented by using a priority queue, with an $O(\log N)$ time complexity.

3.3 iCBS for the SLA in Figure 2(b)

To handle the SLA in Fig. 2(b), we first derive that for a step-function SLA with height c_1, the CBS score is

$$r(t) = c_1 \cdot e^{-a(t_1 - t)} = \beta e^{at}$$

where $\beta = c_1 \cdot e^{-at_1}$, and t is the time when the CBS score is computed. Note that the above formula is valid only for $t < t_1$ and after t_1, $r(t)$ becomes 0. We can see that in this case $r(t)$ is time-varying. However, a key observation is that CBS does not care the *absolute value* for each individual $r(t)$, but instead cares about the *relative order* among the $r(t)$'s for all the queries. This is because that CBS only has to pick the top query with the highest CBS score to be executed next. Therefore as long as the CBS scores among all queries share the same factor e^{at}, their *relative order is time-invariant* and only depends on t_0, when a query arrives, t_1, when the deadline is, and c_1, the cost of missing the query's deadline (recall that without loss of generality, we assume $c_0 = 0$). Therefore, this case of step-function SLA can be handled by using a priority queue ordered by the β values of the queries (where β values are *time-invariant*), again with an $O(\log N)$ time complexity.

3.4 iCBS for the SLA in Figure 2(c)

Before handling SLAs in Figs. 2(c) and 2(d), we first study a more general case as described in Fig. 4(a). Note that in this more general case, the SLA cost is 0 before $t_0 + t_1$ and an arbitrary function (which we denote by a *blob*) afterward. Now we study how the CBS score changes over time. It turns out that between time t_0 and $t_0 + t_1$, the CBS score changes in a particular pattern. To see this, we compute the CBS scores at two time instances, t' and t'', where $t_0 < t' < t'' < t_0 + t_1$. At time t', we have

$$r(t') = \int_0^\infty a e^{-a\tau} f(t' - t_0 + \tau) d\tau - f(t' - t_0)$$

and at time t'' we have (derivation given in Appendix B.5)

$$r(t'') = e^{a(t'' - t')} \cdot r(t') \tag{3}$$

As a consequence we can see that the CBS score for the SLA in Fig. 4(a) grows exponentially with parameter a until time $t_0 + t_1$, as shown in Fig. 4(b).

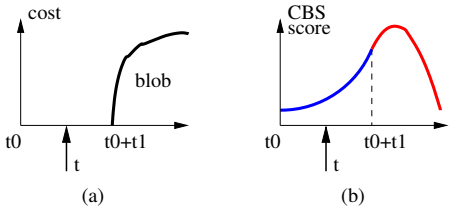

Figure 4: (a) An SLA with zero cost until $t_0 + t_1$ and (b) how the CBS score changes over time exponentially between t_0 and $t_0 + t_1$.

Now we come back to the piecewise linear SLA described in Fig. 2(c). It can be shown that its CBS score changes over time as described in Fig. 5(a). The basic idea is that at each segment of the SLA, we can shift down the staircases to get an SLA similar to that in Fig. 4(a), where the blob is replaced by the remaining staircases. As time elapses, as long as we are still in the same segment, the CBS score grows in an exponential fashion, as we have just shown. In addition,

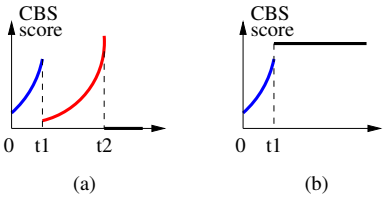

Figure 5: Left: how the CBS score changes over time for the SLA in Fig. 2(c). Right: how the CBS score changes over time for the SLA in Fig. 2(d).

the exponential rate is the same for all the segments and for all the queries, because the rate only depends on a.

Based on all these properties, we develop an incremental CBS by using two priority queues. Both priority queues keep all the queries, but with different priority scores. In the first priority queue, the score for each query q is the CBS score for q if we consider the remaining staircases of q as a blob. Following the same idea as shown in the case of SLA in Fig. 2(b), the relative order among the queries in the first priority queue remains fixed. So the problem is boiled down to detecting when each query changes its SLA segments (and so changes the shape of the blob).

For this, we use the second priority queue. In the second priority queue, the score for each query q is the time when the next cost jump will happen in q's SLA, or in other words, the time when q's current SLA segment will expire. At each time t when we need to select the next query to execute, we first take out all those queries whose SLA segments expired before t. For these queries, the old blobs are not valid anymore and so for each such query q, we locate q's current SLA segment that overlaps t, get q's new time-of-expiration of current SLA segment, and reinsert q into the second priority queue. At the same time, we update q's CBS score in the first priority queue to reflect its new score. This second priority queue actually makes sure that all the queries are in their first segment of (probably revised, if the old first segments have expired) piecewise linear SLAs. That is, for all queries, $t < t_1$ is always true in $r(t)$, and therefore Equation (2) is always valid. As can be easily shown, the numbers of de-queue and en-queue operations for a query q in the first and second priority queues are both bounded by the number of segments in q's SLA, which we assume to be bounded by a constant. So the amortized time complexity is still $O(\log N)$ for scheduling each query.

3.5 iCBS for the SLA in Figure 2(d)

For SLAs in Fig. 2(d), it can be shown that its CBS score changes over time as described in Fig. 5(b), where initially it grows exponentially with rate a and after t_1, it becomes a constant. So we can use the similar idea as before, except that we need three priority queues. The first and the second ones are the same as before, except when a query is popped out from the second queue, it is never inserted back; instead, it is removed from both the first and the second priority queues and is inserted into the third priority queue. This is because at the time of being popped out of the second priority queue, q has no further SLA segment changes and so its CBS score becomes time-invariant. When scheduling the next query to execute, after updating queries in the first and second priority queues, we compare the query with the highest CBS score in the first priority queue and that in the third queue, and pick the one with the higher score to execute. The time complexity clearly is still $O(\log N)$.

4. ICBS WITH $O(\log^2 N)$ COMPLEXITY

In this section, we investigate the incremental version of CBS for general piecewise linear SLAs, as described in Figs. 2(e) and 2(f). We will analyze why these cases are more difficult than the previous cases. Then we describe an efficient method by using a dynamic convex hull algorithm in computational geometry.

4.1 Why the Difficult Cases Are Difficult

Starting from Equation (2), with certain derivation we can show that the CBS score at time t for a query q, who has a general piecewise linear SLA, can be written as

$$f(t) = \alpha + \beta e^{at}, \tag{4}$$

where α and β are constant values totally determined by q's arrival time and its SLA.

It is obvious from Equation (4) why an incremental version of CBS is hard to get. Fig. 6(a) illustrates for three sample queries with different α's and βs', how their CBS scores change over time. As can be seen from Fig. 6(a), the relative order of the CBS scores of the three queries changes over time. In other words, in general, the order among the CBS scores of the queries is different depending on the time t when we ask about that order. Such a property poses difficulties to an incremental version of CBS in the time space. To address such difficulties, in the following we investigate a different view of the problem: we study the problem in the *dual space* of linear functions.

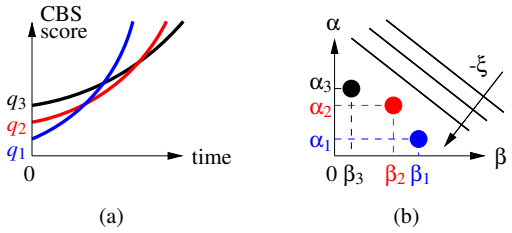

Figure 6: (a) How the SLA scores of three fictitious queries change over time. (b) The three queries in the dual space, with the affine-lines for scheduling.

4.2 A Dual-Space View of CBS

Now we provide a new perspective of viewing the CBS score $f(t) = \alpha + \beta e^{at}$. First, we define a new variable $\xi = e^{at}$ (where ξ can be considered as a *warped* timeline) to get

$$f(\xi) = \alpha + \beta\xi, \tag{5}$$

where ξ is time-varying but α and β are constants. Note that there is a one-to-one mapping between ξ and t. Then, we put each of these score functions into the *dual space* of linear functions. That is, each function $f(\xi) = \alpha + \beta\xi$ is mapped to a point in the dual space with coordinate (α, β), as shown in Fig. 6(b).

Next, we examine at a given time t', how to determine the order among the CBS scores. With $\xi' = e^{at'}$, at time t', we can treat ξ' as fixed and look for the query that maximizes $f(\xi') = \alpha + \beta\xi'$. If we consider α and β as variables, then in the dual space, those points (α, β) that satisfy $f(\xi') = \alpha + \beta\xi' = c$ are on an affine line $\alpha = c - \xi'\beta$, as shown in Fig. 6(b). So at a given time t', here is a way we can get the point in the dual space that corresponds to the query with the highest CBS score at time t'. We first get $\xi' = e^{at'}$ and construct an affine line with slope $-\xi'$. Then starting from the upper-right corner of the dual space, we move the affine line toward the lower-left corner of the dual space while keeping the slope fixed at $-\xi'$. The first point hit by the moving affine line will be the query that has the highest CBS score at time t'. Figs. 7(a) and 7(b) show the same set of three queries, whose CBS scores are compared at two different time, t' (with $\xi' = e^{at'}$) and t'' (with $\xi'' = e^{at''}$), where $t' < t''$. As can be seen, at early time t', the (absolute) slope of the affine line is smaller and so the point (α_3, β_3) has the highest score; as time going, at time t'', (α_1, β_1) has the highest score because it has a higher β value.

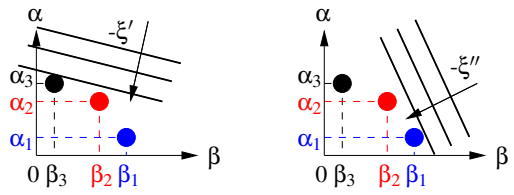

Figure 7: Selecting the query with the top CBS score (a) at time t' and (b) at time t''.

In summary, the advantage of this dual-space approach is that in the dual space, we fix the (α, β) location of each query as *time-invariant*; and over the time we change the slope of the scheduling affine line to select the query with the highest CBS score at any given moment. This dual-space perspective, as we show in Appendix D.1, also offers insights about why the cases in Section 3, i.e., those SLAs in Figs. 2(a)–2(d), can be handled with time $O(\log N)$.

4.3 A Convex Hull based Algorithm

In this subsection, we develop iCBS under general piecewise linear SLAs that takes advantages of this dual-space view. The first question is whether we can efficiently maintain the position of a query q in the dual space over time. The answer is affirmative, as the position of q in the dual space can only change finite number of times (which turns out to be bounded by the number of segments in q's SLA). We formally prove this in Appendix D.2.

Second, it can be seen from Fig. 8(a) that a necessary condition for a point q to be hit first by the moving affine line is that q is on the (upper) convex hull of all points in the dual space. Furthermore, it can be shown that if the convex hull is maintained properly, at any time t, finding the point hit first by the affine line of slope -ξ (i.e., with $\xi = e^{at}$) can be done in $O(\log M)$ time, where M is the number of points on the convex hull (obviously, $M \leq N$). More specifically, if the points of the convex hull are sorted by their β values, as shown in Fig. 8(b), then the angles of the edges on the convex hull are in an monotonically decreasing order. At time t, we have a fixed slope -ξ, and the condition for q on the convex hull to be the point hit first by the affine line of slope -ξ is that "the slope -ξ is between the angles of the two edges that incident to q". This can easily be done by conducting a binary search on the convex hull.

Therefore, it follows that we can develop iCBS as long as we can (1) efficiently detect when a query changes its segment in its SLA and (2) incrementally maintain a convex hull that supports adding and deleting points dynamically. For item (1), as discussed before, we can maintain a priority

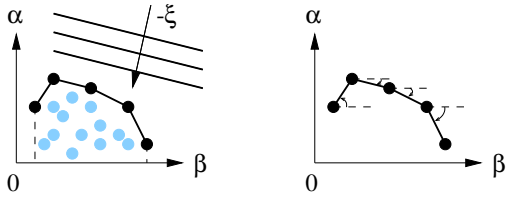

Figure 8: (a) The queries with the highest score can only be on the upper convex hull in the dual space. (b) The angles of the edges on the convex hull follow a monotonic decreasing order.

queue of queries ordered by the time of the expiration of the current SLA segment. At any time t, we can quickly locate those queries whose current SLA segments have expired before t, and update their new current SLA segments together with new expiration time of the current SLA segments. For item (2), we adopt, from the field of computational geometry, a dynamic convex hull algorithm for points in a 2-D plane that has $O(\log^2 N)$ time complexity [13]. The detailed implementation of the iCBS algorithm, as well as the time and space complexity analysis, are given in Appendix D.3.

An intuitive interpretation of iCBS is that it incrementally maintains (1) the snapshot of the CBS scores of all the queries in the dual space and (2) the convex hull of the snapshot. At each time t of making a scheduling decision, iCBS quickly brings up-to-date both the snapshot and the convex hull, from which the query with the top CBS score at time t can be located very efficiently.

5. EXPERIMENTAL STUDIES

The experiments are composed of two sets. First, we verify the good performance of CBS (iCBS[2]), in terms of cost optimization, by comparing it with several other state-of-the-art cost-aware scheduling algorithms. Second, we compare the running time of iCBS with that of the original CBS under various workloads. Some additional results are given in Appendix E.

5.1 CBS Scheduling Effectiveness Verification

In this subsection, we verify the performance of CBS (iCBS) using two different types of SLA cost functions. We compare iCBS with two pieces of previous work, ASETS* [4] and FirstReward [8], as described at the beginning of this paper. For the query execution time we use an exponential distribution with mean μ =30ms; for the arrival rate we use a Poisson arrival with rate $1/\lambda$, which is determined by μ and the required load. The data set contains 20K queries, where the first 10K are used for system warm-up.

The first SLA cost function is shown in Fig. 1(b), which is also known as weighted tardiness [4]. We set the deadline t_1 for each query as 10 times its execution time. Following the experiment design by [8], we choose the slope value from one of two classes, high and low, where half of queries get high slope values and half get low slope values. Within each class, we choose the specific slope value from a normal distribution with a mean and stddev=0.2*mean. The ratio between high mean and low mean is controlled by *decay skew factor*, which we vary in the experiment. Note that ASETS* [4] is explicitly designed for this first SLA cost function.

The second SLA cost function is shown in Fig. 1(c). Each query incurs a cost after the minimum response time t_1, and there is a bound at 10 times the execution time (t_2), which differentiates it from the weighted tardiness above. The bound value is chosen from two normal distributions as described above, and the ratio of high mean and low mean is called *value skew factor*. Note that FirstReward [8] is explicitly designed for this second SLA cost function.

Table 1(left) shows the result on the first SLA cost function under different decay skew factors as described above. When the system load is 0.8, the three cost-aware scheduling algorithms already significantly outperform the cost-unaware counterparts, namely FCFS and SJF. The improvement become more striking when the system load is increased to 0.95. Among the three cost-aware scheduling algorithms, iCBS clearly outperforms the other two, especially when the system load and the decay skew factor are high (exactly when differentiated services are needed the most). For example, under system load of 0.95 and decay skew factor of 128, iCBS outperforms the second best algorithm by 45%.

Table 1(right) shows the result on the second SLA cost function under different valued skew factors as described above. (We do not compare the performance with ASETS*, because ASETS* cannot handle this SLA.) As can be seen, although the margin is smaller, iCBS still consistently outperforms FirstReward, which was designed specifically for this second SLA, by about 10%.

5.2 Scheduling Efficiency Study

In this subsection, we investigate the performance of iCBS in terms of running time[3]. As for the baseline, for fair comparison, we implement CBS so that it computes CBS scores under piecewise linear SLAs without conducting integration, as described in Section 3.1. All the running times are reported on a Xeon PC with 3GHz CPU and 4GB memory, running Fedora 11 Linux.

When the system has light load, the number of queries to be scheduled is very small in most of the time with exponentially distributed execution times. Therefore, in this experiment we focus on the case when the system load is 0.95, to study the situation when the system is borderline overloaded. Fig. 9 shows the running time vs. the number of queries to be scheduled for CBS and iCBS. As can be seen, while CBS exhibits a clear $O(N)$ behavior, the online scheduling time of iCBS is almost unaffected by N. It is worth noting that FirstReward on average needs a couple of seconds to make each scheduling decision, which makes it impractical for many applications.

Finally, we study a case that emulates the scenario with a mixture workload with OLAP and OLTP queries. Query execution time in such a case can be captured by a Pareto (e.g., long-tail) distribution. Following [5], we set the minimum value of the Pareto to be 1ms and the Pareto index to be 1. We use a load of 0.8 because even with such light load, the number of queries to be scheduled can grow very

[2]In terms of the parameter a used in CBS, Peha [10] showed that the CBS performance is not sensitive to the exact value of a as long as a is in a reasonable range. So following Peha's suggestion, we simply used $a = 10\mu$ where μ is the mean execution time of a query.

[3]We implemented a naive version of incremental convex hull instead of using the sophisticated version in [13], and in our experiments it is never a bottleneck. Details are skipped due to the space limit.

Table 1: Average cost per query, for different algorithms under different skew and load factors.

SLA Type	SLA-1								SLA-2							
Workload	0.80				0.95				0.80				0.95			
Skew Factor	2	8	32	128	2	8	32	128	2	8	32	128	2	8	32	128
FCFS	0.062	0.186	0.821	2.859	0.569	1.769	6.943	19.70	0.013	0.038	0.139	0.543	0.028	0.087	0.316	1.284
SJF	0.008	0.024	0.118	0.364	0.103	0.324	1.289	3.662	0.008	0.025	0.092	0.358	0.014	0.043	0.156	0.628
ASETS*	0.005	0.012	0.051	0.175	0.066	0.109	0.237	0.619	-	-	-	-	-	-	-	-
F-REWARD	0.005	0.013	0.049	0.176	0.069	0.102	0.201	0.490	0.007	0.018	0.059	0.223	0.010	0.023	0.077	0.296
iCBS	0.005	0.012	0.045	0.158	0.066	0.092	0.146	0.273	0.007	0.016	0.053	0.200	0.010	0.022	0.068	0.260

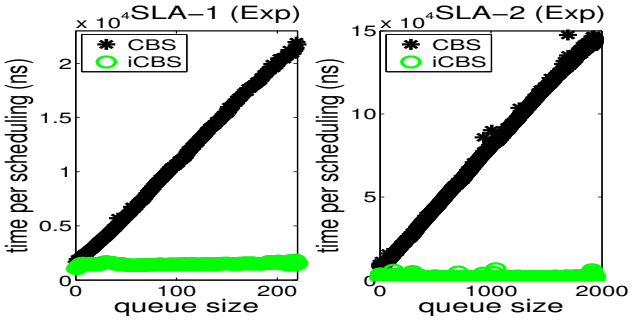

Figure 9: Average scheduling time per query for the Exponential case with Load=0.95.

large (because of OLAP queries may occupy the database server for rather long time). Note that for this experiment, FirstReward was not able to finish within reasonable time (hours). Fig. 10 shows the performance comparison and we can draw similar conclusions as before in terms of the running time of iCBS. One observation is that in this case, while the time to make a scheduling decision for iCBS is extremely fast (less than 0.01ms), that for the original CBS can become tens of milliseconds, an overhead unacceptable in many database applications.

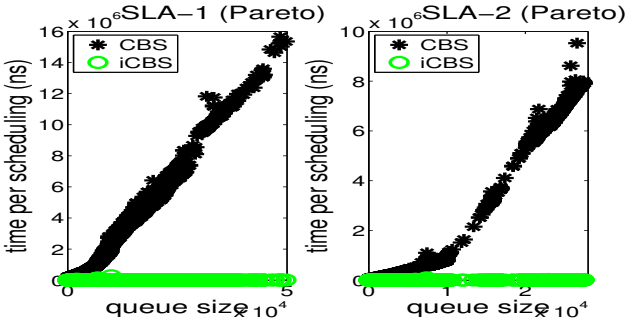

Figure 10: Average scheduling time per query for the Pareto case with Load=0.8.

6. CONCLUSION

We investigated a cost-based scheduling algorithm, CBS, which has superior performance in terms of SLA cost and therefore has great potentials in cloud computing. The main focus of this paper was to implement CBS in a more efficient way, where the cost is determined on query response time by using piecewise linear SLAs. To achieve this goal, we developed an incremental variation of CBS, called iCBS, that greatly reduces the computation time during online scheduling. iCBS can handle any type of piecewise linear SLAs, which are very common in cloud computing contracts, with a very competitive time complexity.

7. ACKNOWLEDGMENTS

We thank Jeffrey Naughton for the insightful discussions and his contributions.

8. REFERENCES

[1] Y. Chi, H. J. Moon, H. Hacigumus, and J. Tatemura. SLA-tree: A framework for efficiently supporting SLA-based decisions in cloud computing. In *EDBT*, pages 129–140, 2011.

[2] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.

[3] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive quality of service management for enterprise services. *TWEB*, 2(1):1–46, 2008.

[4] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In *ICDE*, pages 357–368, 2009.

[5] C. Gupta, A. Mehta, S. Wang, and U. Dayal. Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In *EDBT*, pages 696–707, 2009.

[6] H. Hacıgümüş, J. Tatemura, W. Hsiung, H. J. Moon, O. Po, A. Sawires, Y. Chi, and H. Jafarpour. CloudDB: One size fits all revived. In *IEEE World Congress on Services (SERVICES)*, pages 148–149, 2010.

[7] J. R. Haritsa, M. J. Carey, and M. Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2:117–152, April 1993.

[8] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *HPDC*, pages 160–169, 2004.

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

[10] J. M. Peha. *Scheduling and dropping algorithms to support integrated services in packet-switched networks*. PhD thesis, Stanford University, 1991.

[11] J. M. Peha. Scheduling and admission control for integrated-services networks: the priority token bank. *Computer Networks*, 31(23-24):2559–2576, 1999.

[12] J. M. Peha and F. A. Tobagi. A cost-based scheduling algorithm to support integrated services. In *INFOCOM*, pages 741–753, 1991.

[13] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

APPENDIX

A. CLOUD COMPUTING ARCHITECTURE

Fig. 11 shows several components that we believe are crucial in a cloud computing system. A capacity planning component plans (offline) and manages (online) the replication and system resources; a dispatching component dispatches queries to appropriate servers in a cost-aware way; locally at each server, a cost-aware scheduling algorithm makes best-effort scheduling to minimize the SLA cost among queries dispatched to the server.

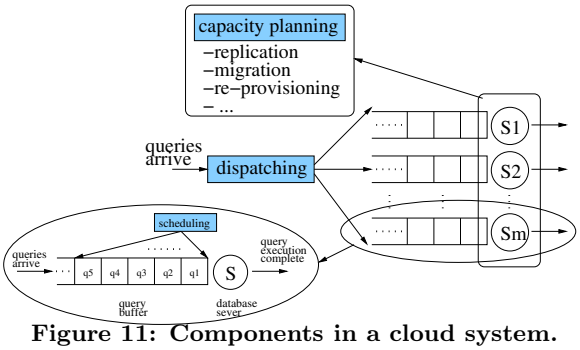

Figure 11: Components in a cloud system.

Ideally, we prefer a comprehensive optimization by considering all the above components in a global way. However, we have concerns about such a global optimization in terms of its adaptability to quick changes and scalability to large numbers of clients in the cloud. Therefore, we choose to use a modular approach where local optimization is made in each component, and in this paper we only focus on the cost-aware scheduling component.

B. MORE DETAILS ABOUT CBS

In this appendix, we discuss more details about CBS, including the equivalence between cost SLAs and profit SLAs, more details of CBS, setting $c_0 = 0$ in piecewise linear SLAs for CBS, some factors that may affect the cost performance of CBS, and the detailed derivation of Equation (3).

B.1 Cost SLAs vs. Profit SLAs

In our paper, to be consistent with the terminology used in the original CBS algorithm, instead of using query *profit*, we use query *cost*. These two are equivalent due to the following reason. As shown in Fig. 12, we can decompose a profit SLA into the difference between a constant profit SLA and a cost SLA. That is, we can assume for each query the service provider obtains a profit of g_1 up front, and pays back certain cost depending on the query response time, where the maximal payback is $g_1 + p_1$. So the total profit among all queries is the difference between the total profit up front and the total cost. From the point of view of a scheduling algorithm, the total upfront profit can be considered as a constant, depending on the shape of SLAs and the workload, which is beyond the control of the scheduling algorithm; the total cost is the part that a scheduling algorithm can control, by changing the priority among queries, where minimizing the total cost is the final objective.

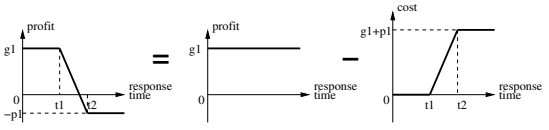

Figure 12: Equivalence of profit and cost SLAs.

B.2 More Details about CBS Score $r(t)$

There are several things we want to point out about Equation (1). First, the CBS score $r(t)$ is time-varying. This makes sense because the urgency of a query depends on how long the query has been waiting in the queue so far and as a result, it depends on the time t when the CBS score is computed. Second, the $f(t - t_0)$ term in $r(t)$ is the cost for executing query q right now (i.e., at time t). Third, inside the integration, the dummy variable τ indicates the *further delay* for query q and $f(t-t_0+\tau)$ is the corresponding cost.

In the original CBS, the CBS score for query q is $r(t)$ divided by q_s, the execution time of q. However, equivalently we can scale the SLA of q by $1/q_s$ and therefore use $r(t)$ computed from the scaled SLA as the CBS score. In addition, for simplicity of discussion, in this paper we illustrate the examples as if the execution time q_s were 0. A non-zero execution time q_s can be handled by shifting q's SLA to the left by an amount of q_s. These adjustments are actually implemented in all our algorithms mentioned in this paper.

B.3 Why We Can Set $c_0 = 0$ under CBS

One observation that can be obtained from Equation (1) is that without loss of generality, to compute CBS score, we can assume $f(t_0) = 0$. This is because (as shown in Fig. 13)

$$r_g(t) = \int_0^\infty ae^{-a\tau} g(t - t_0 + \tau)d\tau - g(t - t_0)$$
$$= \int_0^\infty ae^{-a\tau}[f(t - t_0 + \tau) + c_0]d\tau - [f(t - t_0) + c_0] = r_f(t)$$

where the last step relies on the fact that $ae^{-a\tau}$ is a probability density function and therefore $\int_0^\infty ae^{-a\tau}c_0 d\tau = c_0$. This result reflects a key feature of CBS, where the damage done so far can be ignored. Because of this property, in this paper, without loss of generality, we assume $f(t_0) = 0$, and therefore all the c_0's in Figs. 2(a)–(f) can be set to 0.

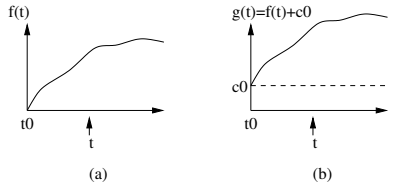

Figure 13: The illustration of why without loss of generality, we can assume $f(t_0) = 0$ in all the SLAs.

B.4 Factors That Affect CBS

There are many factors that potentially affect the effectiveness of CBS. One of the factors is the multi-programming-level (MPL) of the database. However, CBS should not be very sensitive to MPL, because the CBS score of a query q only depends on q's SLA cost function and q's urgency. The former is not related to MPL and the latter is approximated in CBS by using an exponential distribution, which is not sensitive to the exact MPL of the database. Our experimental studies, skipped due to the space limits, have verified this conjecture.

Another factor that affects the effectiveness of CBS is how accurate we can estimate the query execution time. We argue that a fully cost-aware scheduling algorithm has to take query execution time into consideration (in this sense EDF is not fully cost-aware, because it only considers query deadlines but not query execution time). Otherwise, the urgency

of a query is not accurately captured. To estimate query execution time, recently work (e.g., [2]) has started using machine learning techniques and obtained encouraging results. In addition, our own investigation (partly reported in [1]) demonstrate that CBS can tolerate errors in the estimation of query execution time to certain levels. Specifically, when the standard deviations of the errors are 0.2 and 1.0, the performance degraded by 10% and 30%, respectively.

B.5 Derivation of Equation (3)

Here we provide the derivation of Equation (3) and discuss some technical subtleties.

We have

$$r(t') = \int_0^\infty ae^{-a\tau}[f(t'-t_0+\tau) - f(t'-t_0)]d\tau$$

and

$$\begin{aligned} r(t'') &= \int_0^\infty ae^{-a\tau}[f(t''-t_0+\tau) - f(t''-t_0)]d\tau \\ &= \int_0^\infty ae^{-a\tau}[f(t'-t_0+\tau+t''-t') - f(t''-t_0)]d\tau \\ &= e^{a(t''-t')}\int_{t''-t'}^\infty ae^{-a\eta}[f(t'-t_0+\eta) - f(t''-t_0)]d\eta \\ &= e^{a(t''-t')}\cdot r(t') \end{aligned}$$

where in the derivation we defined a new dummy variable $\eta = \tau + t'' - t'$ and in the last step, we used the fact that $f(t'-t_0+\eta) - f(t''-t_0) = 0$ for $0 \le \eta \le t''-t'$.

This feature of CBS is actually due to the *memory-less* property of the exponential distribution $ae^{-a\tau}$, namely at any given time, the conditional distribution of the *remaining* waiting time τ' for a query in the queue is *the same* as the original distribution of the total waiting time.

In addition, technically, the blob in Fig. 4 is not really arbitrary: the cost function has to make the integration in the CBS score finite. However, such a condition is satisfied by any piecewise linear function with finite segments.

C. FAST COMPUTATION OF CBS SCORE UNDER PIECEWISE LINEAR SLAS

In this appendix, we derive an efficient approach for computing CBS scores of queries with piecewise linear SLAs. This approach, by applying a canonical decomposition of piecewise linear SLAs and by using rules of calculus, avoids the expensive integration in the computation of CBS scores. Such an efficient approach, other than making it fast to compute CBS scores, reveals important insights about CBS under piecewise linear SLAs. These insights turn out to be a key to iCBS, our incremental implementation of CBS.

C.1 Decomposition of Piecewise Linear SLAs

We start with a canonical decomposition of a general piecewise linear SLA. We illustrate the decomposition by using the SLA described in Fig. 2(f), which is reproduced on the left in Fig. 14 as f_1. As shown in Fig. 14, the SLA can be partitioned into three segments: that before t_1, that between t_1 and t_2, and that after t_2. Starting from f_1, we decompose the SLA into the sum of f_2, which is a continuous function, and f_3 and f_4, which are step functions that capture the *jumps in the costs* of the SLA. Next, we take the derivative of f_2 to get f_2', which in turn can be decomposed into the sum of f_5, which is a constant function corresponding to the initial slope of the SLA, and f_6 and f_7, which are step functions that capture the *jumps in the slopes* of the SLA. (Note that $f_2'(t)$ is not defined at t_1 and t_2, which is fine because $f_2(t)$ is continuous and finite.) It can be shown that in Equation (4), the α value of a CBS score for a piecewise linear SLA is determined by f_5 in Fig. 14, and the β value is determined by f_3, f_4, f_6 and f_7 in Fig. 14.

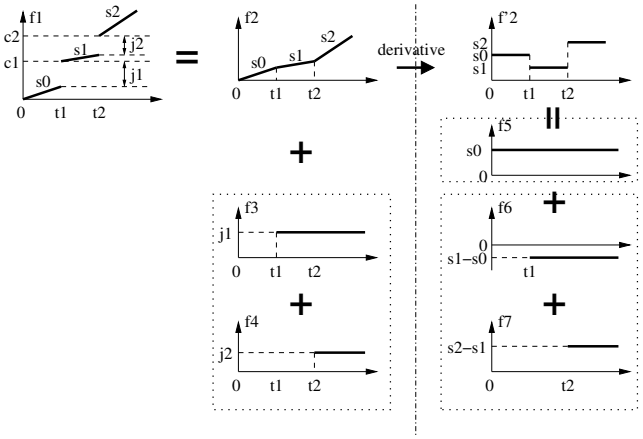

Figure 14: The decomposition of a general piecewise linear SLA f_1 into the sum of f_2, f_3, and f_4, whereas f_2', the derivative of f_2, is further decomposed into the sum of f_5, f_6, and f_7.

C.2 Fast Computation of the CBS Score

It turns out that with the help of the above canonical decomposition, we are able to compute the CBS scores under piecewise linear SLAs *without* any integration. We derive this result by using the decomposition shown in Fig. 14. The CBS score $r(t)$ for $t < t_1$ is computed as

$$r(t) = \int_0^\infty ae^{-a\tau}f_1(t-t_0+\tau)d\tau - f_1(t-t_0) \qquad (6)$$

where without loss of generality we set $f_1(t_0)$ to be 0.

For the step function $f_3(\tau)$, it can be easily shown that

$$\int_0^\infty ae^{-a\tau}f_3(t-t_0+\tau)d\tau = j_1\cdot e^{-a(t_1-t)},$$

where t_1 is the time when f_3 jumps from 0 to a value of j_1. We can obtain similar result for f_4. Because integration is a linear operation, we have

$$\begin{aligned} r(t) &= \int_0^\infty ae^{-a\tau}f_1(t-t_0+\tau)d\tau - f_1(t-t_0) \\ &= \int_0^\infty ae^{-a\tau}f_2(t-t_0+\tau)d\tau - f_1(t-t_0) \\ &\quad + j_1\cdot e^{-a(t_1-t)} + j_2\cdot e^{-a(t_2-t)} \\ &= \int_0^\infty ae^{-a\tau}[f_2(t-t_0+\tau) - f_2(t-t_0)]d\tau \\ &\quad + j_1\cdot e^{-a(t_1-t)} + j_2\cdot e^{-a(t_2-t)} \end{aligned} \qquad (7)$$

where we used the fact that $f_1(t-t_0) = f_2(t-t_0)$ for $t < t_1$.

The main technique we use next is the well-known rule of *integration by parts* in calculus:

$$\int_0^\infty g'(\tau)f(\tau)d\tau = f(\tau)g(\tau)\Big|_0^\infty - \int_0^\infty g(\tau)f'(\tau)d\tau. \qquad (8)$$

Comparing Equation (8) with the integration part of the CBS score in Equation (7), and by setting $g(\tau) = -e^{-a\tau}$ which implies that $g'(\tau) = a \cdot e^{-a\tau}$, we have

$$\begin{aligned}&\int_0^\infty g'(\tau)[f_2(t-t_0+\tau) - f_2(t-t_0)]d\tau \\ &= g(\tau)[f_2(t-t_0+\tau) - f_2(t-t_0)]\Big|_0^\infty \\ &\quad - \int_0^\infty g(\tau) f_2'(t-t_0+\tau)d\tau \\ &= \frac{1}{a}\int_0^\infty a \cdot e^{-a\tau} f_2'(t-t_0+\tau)d\tau.\end{aligned}$$

Now from f_5, we have

$$\frac{1}{a}\int_0^\infty ae^{-a\tau} f_5(t-t_0+\tau)d\tau = \frac{1}{a}s_0,$$

and it can be easily shown that from f_6 and f_7,

$$\begin{aligned}\frac{1}{a}\int_0^\infty ae^{-a\tau} f_6(t-t_0+\tau)d\tau &= \frac{s_1-s_0}{a}e^{-a(t_1-t)}, \\ \frac{1}{a}\int_0^\infty ae^{-a\tau} f_7(t-t_0+\tau)d\tau &= \frac{s_2-s_1}{a}e^{-a(t_2-t)},\end{aligned}$$

which give the final result

$$r(t) = \frac{1}{a}s_0 + (j_1 + \frac{s_1-s_0}{a})e^{-a(t_1-t)} + (j_2 + \frac{s_2-s_1}{a})e^{-a(t_2-t)}.$$

D. MORE DETAILS ABOUT ICBS

In this appendix, we provide more details about iCBS, including insights obtained from the dual-space view, the correctness of an incremental maintenance in the dual space, the detailed iCBS algorithm, and its complexity analysis.

D.1 Why the Easy Cases Are Easy

With the dual-space interpretation, now we can revisit the easy cases in Section 3 and see why they are easy to handle.

The queries with SLA of Fig. 2(a) are located in the dual space on the α-axis, as shown in Fig. 15(a). So no matter when the CBS scores are checked (i.e., no matter with what slope we hit the points using an affine line), the best one is always the one with the highest α value. Similarly, the SLAs of Figs. 2(b),(c), and (d) are shown in Fig. 15(b), where they are all on the β-axis in the dual space. In this case, no matter using what slope, the affine line always hits the one with the highest β value. All these clearly illustrate that for cases of SLAs in Figs. 2(a)–(d), the CBS scores are *order-preserving* over time (except when an SLA segment expires), and so they are easy to handle.

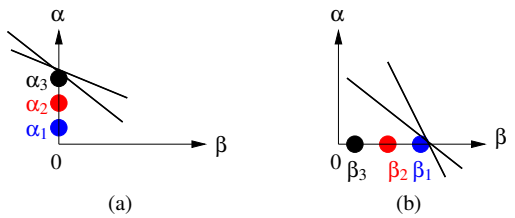

Figure 15: The queries (a) in α-stage and (b) in β-stage in the dual space.

For ease of discussion, in the remainder of paper (appendices), we refer to the queries in Fig. 15(a) as queries in α-stage and those in Fig. 15(b) as queries in β-stage. Along the same line, for those queries that are neither in α-stage nor in β-stage (i.e., those with $\alpha \neq 0$ and $\beta \neq 0$), we refer to them as in $\alpha\beta$-stage.

D.2 Incremental Maintenance Is Practical

We prove that it is practical to incrementally maintain the convex hull in the dual space for queries with piecewise linear SLAs. The key is the following property of CBS scores under piecewise linear SLAs.

PROPERTY 1. *Under piecewise linear SLAs, in the dual space, a query q is either in α-stage, or β-stage, or $\alpha\beta$-stage. Moreover, the stage of q does not change within the same segment of the piecewise linear SLA of q.*

PROOF. Here we give a sketch of the proof based on the decomposition of piecewise linear SLAs. Fig. 16 shows that a general piecewise linear SLA for a query q can be decomposed into a linear part and a blob part, where the linear part determines the α value and the blob part determines β value of q. Therefore, we either have $\alpha = 0$ (and so q is in β-stage), $\beta = 0$ (and so q is in α-stage), or $\alpha \neq 0$ and $\beta \neq 0$ (and so q is in $\alpha\beta$-stage). In addition, from the figure we can easily see that such a decomposition remains fixed within the same segment of the SLA of q. □

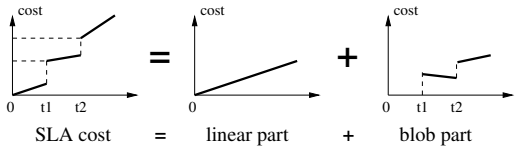

Figure 16: Decomposition of a general piecewise linear SLA into a linear part, which determines α, and a blob part, which determines β.

With such a property we can see that a query q can only change its stage and hence its position in the dual space (q can, e.g., change from $\alpha\beta$-stage back to $\alpha\beta$-stage.) for finite number of times. And this number is bounded by the number of segments in q's piecewise linear SLA.

D.3 iCBS Algorithm in Detail and Its Complexity Analysis

Fig. 17 and Fig. 18 give the pseudo code for the iCBS algorithm. At each time t of scheduling the next query to execute, algorithm `scheduleNext()` is called with t, the current time stamp, and *newQ*, those queries that have arrived before t but after the last time when `scheduleNext()` was called. In the first step of `scheduleNext()`, algorithm `update()` is called in order to incrementally update the internal data structures.

The internal data structures maintained by iCBS (see Fig. 18) include (1) a-Q: a priority query containing queries either in β-stage or in $\alpha\beta$-stage, where for each query q in a-Q, its priority is the $q.\epsilon$, the time when the current SLA segment of q will expire, (2) α-Q, β-Q, and $\alpha\beta$-Q: priority queues for those queries in α-stage, β-stage, and $\alpha\beta$-stage, respectively, and (3) CH: the convex hull in the dual space for all the queries in $\alpha\beta$-stage.

When `update()` is called, it first detects those queries whose current SLA segments have expired before t (line 1). For these queries, their α and β values (as well as stages) may have changed since last time `update()` was called. So these queries are removed from the corresponding internal data structures and these queries are appended to the end of *newQ* so that their α and β values will be recomputed (lines 2–7). Then for all the queries in *newQ* (both those newly arrived and those that need updates), their α and β

Algorithm `scheduleNext(t,newQ)`

▷ **input:** time t, newly arrived queries $newQ$
▷ **output:** q, the next query to execute
1: call `update`(t,$newQ$);
2: $\xi \leftarrow e^{at}$;
3: $q_\alpha \leftarrow \alpha$-Q.top();
4: $q_\beta \leftarrow \beta$-Q.top();
5: $q_{\alpha\beta} \leftarrow CH$.get(-$\xi$);
6: **if** q_α has the highest CBS score
7: $q \leftarrow \alpha$-Q.pop();
8: **else if** q_β has the highest CBS score
9: $q \leftarrow \beta$-Q.pop();
10: remove q_β from a-Q;
11: **else**
12: $q \leftarrow q_{\alpha\beta}$;
13: remove $q_{\alpha\beta}$ from $\alpha\beta$-Q, CH, and a-Q;
14: **return** q;

Figure 17: Implementation of `scheduleNext()`.

values are computed, their stages are determined, their new current SLA segments and expiration time are updated, and they are inserted into the corresponding data structures.

After the internal data structures have been updated, `scheduleNext()` checks q_α, q_β, and $q_{\alpha\beta}$, the three queries that have the highest CBS scores in the α-Q, β-Q, and $\alpha\beta$-Q. To get q_α and q_β, it is suffice to peek at the tops of α-Q and β-Q; to get $q_{\alpha\beta}$, it is suffice to do a binary search on the convex hull CH by using the slope $-\xi = -e^{at}$. Then the best one among q_α, q_β, and $q_{\alpha\beta}$ is selected and removed from the corresponding data structures (assuming this query is actually executed next).

The time and space complexities of the overall iCBS are given as the following.

PROPERTY 2. *Our iCBS implementation has an $O(\log^2 N)$ time complexity for scheduling each query, and it has an $O(N)$ space complexity.*

Algorithm `update(t,newQ)`

▷ **input:** time t, newly arrived queries $newQ$
▷ a-Q: priority queue on expiration time (ϵ)
▷ α-Q: priority queue for queries in α-stage
▷ β-Q: priority queue for queries in β-stage
▷ $\alpha\beta$-Q: priority queue for queries in $\alpha\beta$-stage
▷ CH: convex hull for queries in $\alpha\beta$-stage
▷ First, handle queries that change SLA sections
1: **foreach** q in a-Q such that $q.\epsilon < t$ **do**
2: remove q from a-Q;
3: append q to the end of $newQ$;
4: **if** q was in β-Q
5: remove q from β-Q;
6: **else if** q was in $\alpha\beta$-Q
7: remove q from $\alpha\beta$-Q and CH;
▷ Second, (re)insert updated queries
8: **foreach** q in $newQ$ **do**
9: compute q's α, β, ϵ values;
10: **if** q is in α-stage
11: insert q to α-Q;
12: **else if** q is in β-stage
13: insert q to β-Q and a-Q;
14: **else** /* q is in $\alpha\beta$-stage */
15: insert q to $\alpha\beta$-Q, CH, and a-Q;

Figure 18: Implementation of the `update()`.

PROOF. In `scheduleNext()`, lines 3,4,7,9, and 10 all have time complexity $O(\log N)$, by using an addressable priority queue or a balanced binary search tree. Line 5 has time complexity $O(\log N)$ by using a binary search. Line 13 may involve the update of the convex hull and so has a time complexity of $O(\log^2 N)$.

For *update()*, each query q can enter the loops of lines 1–7 and lines 8–15 at most K times where K is a constant that represents the number of segments in q's SLA. Each step in the two loops has an $O(\log N)$ time complexity other than lines 7 and 15, in which it takes $O(\log^2 N)$ time to add or remove q from the convex hull dynamically.

In the above analysis, we assumed K to be a constant. If this is not the case, then the time complexity becomes $O(K \log^2 N)$ for iCBS and $O(KN)$ for CBS.

In terms of space complexity, we notice that each query q occurs at most once in a-Q and once in either α-Q, β-Q, and $\alpha\beta$-Q, and in addition, the size of the convex hull CH is bounded by the size of $\alpha\beta$-Q. So we can see that the space complexity for iCBS is $O(N)$. □

E. ADDITIONAL EXPERIMENTS

In this appendix, we show the running time of CBS vs. iCBS under general piecewise linear SLAs. The SLAs are in the shape that is described in Fig. 2(f), namely with three segments where s_0, s_1 and s_2 are non-zero. Fig. 19 shows the performance comparison between CBS and iCBS, in terms of running time under different queue sizes, for query execution time following exponential distribution (Fig. 19-left) and Pareto distribution (Fig. 19-right).

Several observations can be obtained from the results. First, the running time for CBS obviously scales linearly with respect to the queue size and for iCBS, the running time is relatively insensitive to the queue size. Second, because there are more queries in $\alpha\beta$-stage, the running time for iCBS is higher than that in Fig. 9 and has higher variations. This is more distinct in Fig. 19-left, where the running time for CBS is low due to smaller queue sizes. However, even for Pareto distribution, the number of queries in their $\alpha\beta$-stage is not very high (in tens). This is because when a large OLAP query blocks the server, most queries will be in their last SLA segment, i.e., in α-stage. In other words, the sophisticated incremental convex hull approach is not warranted in this case and the main benefit comes from queries in α and β stages. However, even this is the case, the dual-space view is still crucial for obtaining the benefit of iCBS.

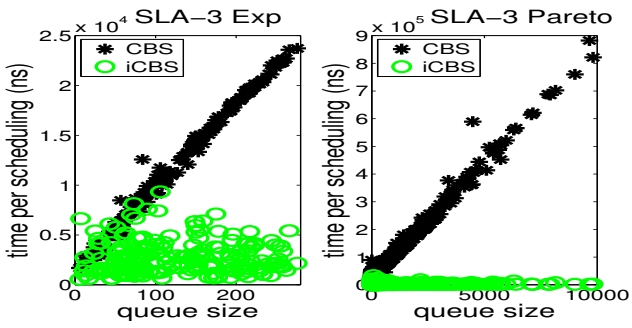

Figure 19: Average scheduling time per query for the Exponential (left) and Pareto (right) cases under general piecewise linear SLAs, with Load=0.95.