Distribution-Based Query Scheduling

Yun Chi[†] Hakan Hacıgümüş[†] Wang-Pin Hsiung[†] Jeffrey F. Naughton[‡]

[†]NEC Laboratories America

[‡]Department of Computer Sciences, University of Wisconsin-Madison

{ychi,hakan,whsiung}@nec-labs.com, naughton@cs.wisc.edu

ABSTRACT

Query scheduling, a fundamental problem in database management systems, has recently received a renewed attention, perhaps in part due to the rise of the "database as a service" (DaaS) model for database deployment. While there has been a great deal of work investigating different scheduling algorithms, there has been comparatively little work investigating what the scheduling algorithms can or should know about the queries to be scheduled. In this work, we investigate the efficacy of using histograms describing the distribution of likely query execution times as input to the query scheduler. We propose a novel distribution-based scheduling algorithm, Shepherd, and show that Shepherd substantially outperforms state-of-the-art point-based methods through extensive experimentation with both synthetic and TPC workloads.

1. INTRODUCTION

Query scheduling is a fundamental problem in database management systems [30, 19], and it has been extensively studied in the literature [17, 30, 14, 15, 18]. Recently scheduling has received a renewed attention due to the rise of the "database as a service" (DaaS) model [1, 4] deployed in the cloud. In such deployments, schedulers try to maximize performance with respect to service level agreements (SLAs) that may impose financial penalties for missed deadlines.

While there has been a great deal of work investigating different scheduling algorithms, there has been comparatively little work investigating what the scheduling algorithms can or should know about the queries to be scheduled. Clearly, the ideal would be to know the precise running time for every query in advance of its execution; in some sense this is "perfect" knowledge about the queries to be run and one cannot do better. Most previous work (e.g., [21, 8, 18]) on query scheduling to meet the requirements of SLAs has at least implicitly assumed that such information is available; unfortunately, this ideal is most likely not achievable in many real-world scenarios.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 9
Copyright 2013 VLDB Endowment 2150-8097/13/07... $ 10.00.

The reason this is not achievable is simple: estimating the running time of queries is a notoriously difficult problem, as query running time is a complex function of the query itself (including any run-time parameters), the data over which it is run, and the environment in which it executes (including both the static hardware configuration and dynamic factors such as the contents of the buffer pool and the impact of concurrently running queries.) Despite a great deal of decade-old and recent effort [12, 10, 22, 36, 35], accurate prediction of query running times for dynamic, concurrent workloads remains largely elusive.

Accordingly, in this work we investigate the efficacy of an alternative to knowing the execution times of queries: specifically, we investigate the approach of using a histogram describing the distribution of likely execution times rather than a single, point estimate of the true running time. At first hearing this may sound silly — we have already acknowledged that predicting running times is difficult, and now we are asking for predictions of distributions of running times. However, in actuality, in some common scenarios, building a histogram of expected running times is easier than predicting specific running times. For example, if a workload is generated by instantiating a number of transactions or query templates (like, for example, all of the TPC benchmarks as well as many servlet-based Web applications [11]), one can monitor execution times for instantiations of each template, and "learn" a distribution of running times. This is analogous to many other situations in which one can quickly learn a distribution of some property of a population (for example, by sampling) more easily than one can predict the property for any specific member of the population.

Of course, the fact that learning the distribution of running times is easier than predicting a specific running time raises the important question of whether knowing this distribution is useful for scheduling. In this paper we demonstrate that the answer to this question is a clear "yes." We do so by presenting a new scheduling algorithm, Shepherd[1], that is specifically designed to exploit distributions of expected running times when doing SLA-aware query scheduling. We show through experiments with both simulated and actual workloads that Shepherd is substantially more effective than previous algorithms that simply use the mean of the population as the expected running time for the query, and is much more effective than algorithms that ignore running time altogether.

[1] *Shepherd* stands for "scheduling under probabilistic, histogram-based query time distributions".

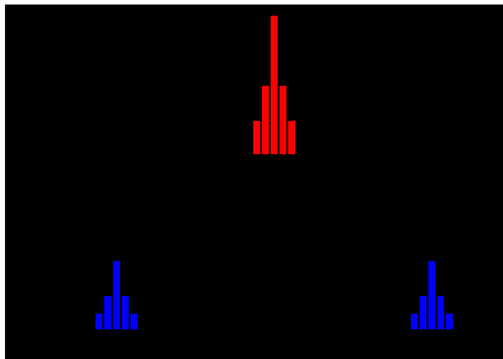

Figure 1: Two queries q_1 and q_2 that have the same expected execution time r, but different execution time distributions.

To gain some insight as to why knowing distributions of query execution times may be more effective than knowing means, we consider a simple example. Figure 1 shows two queries q_1 and q_2 that both arrive at the system at time t, both have a deadline at time d, and both have an expected query execution time r. However, the execution times of the two queries follow different distributions. q_1 has a relatively predictable execution time around r whereas q_2's execution time follows a bimodal distribution determined by some unpredictable factors. In this case, although q_1 and q_2 have the same expected execution time, q_2 is more urgent, because it has higher chance of missing its deadline if not executed right away.

Another factor illustrated by this example is that the absolute and even relative importance of these two queries with the same mean execution time changes over time. For example, assume we revisit the situation after a period of time such that the deadline is shifted to d'. Now running q_1 is urgent because it will almost certainly miss its deadline if it is not executed right away. More interestingly, if we revisit the situation at an even later time, say when the deadline is shifted to d'', then the urgency of q_1 is 0 because its deadline has passed, whereas executing q_2 is still valuable as it still has some chance to meet its deadline. This suggests that an effective, distribution-based scheduling algorithm must revise its priorities constantly over time.

Our contributions

In this paper, we show that probability distributions of execution times can be effectively exploited for scheduling queries in the presence of SLA constraints. We do so by exhibiting one instance of such a distribution-based algorithm, which we term the Shepherd. Shepherd is based on the CBS (*Cost Based Scheduling*) algorithm [25, 27], and also borrows from ideas presented in our previous work on the iCBS (*incremental* CBS) algorithm [8] to increase efficiency in the presence of stepwise SLAs. However, Shepherd is a larger departure from both CBS and iCBS than iCBS is from CBS, in that it builds upon CBS to consider a distribution-based execution time model rather than the traditional point-based execution model. To the best of our knowledge, Shepherd is the first scheduling algorithm designed for database systems that uses distributions to model query execution times.

We present experiments using both simulated performance and actual performance on synthetic and TPC-W workloads. In our experiments we show that it is indeed feasible to learn sufficiently accurate distributions of TPC-W query performance. Moreover, both the simulated and experimental studies verify the effectiveness and efficiency of distribution-based query scheduling, as embodied by Shepherd, in comparison to state-of-the-art point-based approaches.

2. BACKGROUND

In this section, we introduce background information, including the notions of cost-aware scheduling and SLA penalty costs, a previous algorithm (CBS) that inspired our algorithm, and the main weakness of existing cost-aware scheduling algorithms that we address in this paper.

2.1 Cost-aware Scheduling and SLAs

Recently, there has been increasing interest in applying more sophisticated scheduling algorithms in database systems [14, 15, 18]. Such a change is partly due to the new trend of providing databases as a service (DaaS) [1, 4], because offering differentiated service is a key to the success of DaaS providers. A family of scheduling algorithms that are especially effective in this situation are *cost-aware* scheduling algorithms [17, 20, 27, 21, 15]. In such algorithms, instead of optimizing system metrics such as average latency and throughput, queries are scheduled so as to optimize *costs* that are associated with queries. Such costs, from the point of view of a service provider, can be two-fold. On the one hand, executing a query q involves costs such as resource usage (e.g., server utilization) and opportunity cost (e.g., executing q may postpone the execution of other queries [21]). On the other hand, missing q's deadline may have a monetary penalty according to a service level agreement (SLA).

Cost-aware scheduling has long been studied in areas including computer networking [27], high-performance computing [21, 28], and real-time databases [30]. For example, in computer networking, a video packet can have a cost associated with missing its deadline [26]. As another example, in a real-time service, there is a trade-off between data freshness and the data value [30].

The rapid growth of cloud computing, database as a service (DaaS), and multitenant databases [1, 4] have introduced strong incentives for cost-aware scheduling in database systems as well. Cloud service providers often offer differentiated services (e.g., gold vs. silver customers) in a multitenant environment, and query costs are directly tied to the profit of the service providers. In addition, in the cloud, query cost models are often readily available, due to service level agreements (SLAs) between cloud service providers and their clients. For example, from an SLA for a query, one can obtain the deadline for the query response time and the corresponding cost for missing the deadline [13].

Figure 2 shows two sample SLA functions on query response time. In Figure 2(a), each query has a single deadline d, and missing the deadline will cost an SLA penalty of c. Figure 2(b) shows another SLA with two deadlines d_1 and d_2: missing d_1 will cost c_1, and if d_2 is missed as well, a higher cost c_2 will incur. SLA cost functions such as those in Figure 2 may be obtained from various sources. They can be explicitly written in service level agreements and service level objectives (SLOs) [9], or they can come from particular applications [5]. We will discuss these SLAs in detail in Section 7. In the remainder of this paper, for concreteness

Figure 2: Two examples of SLAs. The query arrives at the system at t and has an execution time r.

we will use the SLA cost functions shown in Figure 2, which have also been used in previous studies [38, 8, 37]. That is, we assume that each query has one or more deadlines, and associated with each deadline, there is a penalty cost value for missing that deadline.

2.2 Cost-based Scheduling (CBS)

There is a rich body of work on cost-aware schedulers [17, 20, 27, 21, 7, 15]. Therefore, instead of inventing a totally new cost-aware scheduling algorithm, we chose to carefully examine several existing state-of-the-art algorithms in terms of characteristics, performance, and applicability to database system requirements. We then built our algorithm on top of an existing algorithm — the cost-based scheduling (CBS) algorithm proposed by Peha [25]. We choose CBS, out of numerous cost-aware scheduling algorithms[2], for two main reasons. First, CBS is effective in terms of cost minimization in an environment of differentiated services. This has been shown in the original work [25] and verified by our recent study [8]. The second reason is that CBS is based on a simple but powerful probabilistic framework and as we show later, our new framework is a natural extension of CBS in this probabilistic sense.

Here we give a brief introduction to CBS, which is based on a very simple intuition:

> At a given time t, for a query q waiting to be serviced, we can either (i) choose q to start execution now (at time t) or (ii) choose a query other than q to execute and hence delay q further. CBS compares these two choices and computes the *expected cost reduction* between the two as the priority score of q at time t.

More specifically, for a query q with execution time r, its CBS priority score at time t is

$$\begin{aligned} p_q(t) &= \frac{1}{r}\left(E[C_{\text{delay}}(t)] - c_{\text{now}}(t)\right) \\ &= \frac{1}{r}\left(\int_{\tau=0}^{\infty} be^{-b\tau}\cdot c(t+\tau)d\tau - c(t)\right), \qquad (1) \end{aligned}$$

where $c(t+\tau)$ and $c(t)$ are the costs for starting the execution of q at time $t+\tau$ and t, respectively; b is a parameter to be set by the algorithm, which we will explain momentarily[3]. Here is an explanation of the CBS score shown in Equation (1). The cost of starting q right now is $c_{\text{now}}(t)$. If on the other hand q is not executed now, then it is assumed that the further delay τ for query q follows an exponential distribution with parameter b, which results in an expected cost of $E[C_{\text{delay}}(t)]$[4]. The factor $\frac{1}{r}$ is used to account for the resource cost of executing q (e.g., occupying the server).

We illustrate the intuition behind CBS by using a simple example in Figure 3. In the example, query q has a single deadline d, where if q can finish its execution by d, there is no SLA penalty cost; otherwise, there is a cost of c.

Figure 3(a) shows how the priority score p_q is evaluated at time t. If we immediately start the execution of q at t, q can be finished at time $t+r$, which is before the deadline d, and so we have $c_{\text{now}}(t) = 0$. If on the other hand we decide to delay the execution of q, then the further delay of q follows an exponential distribution $be^{-b\tau}$. As long as $\tau < d-r-t$, q still can meet its deadline; otherwise, q will miss its deadline and incur a cost of c. The probability of the latter scenario (and $E[C_{\text{delay}}(t)]$) is reflected by the shaded area in Figure 3(a). So overall, $p_q(t) = E[C_{\text{delay}}(t)]$ is proportional to the shaded area in Figure 3(a).

Figure 3(b) shows how the priority score p_q is re-evaluated at time t', which is later than t. At time t', $c_{\text{now}}(t')$ is still 0. That is, if we start the execution of q at time t', we still can meet q's deadline. However, at time t', the deadline of q is more urgent, and therefore a smaller further delay (namely $\tau > d-r-t'$) will make q miss its deadline. This implies that at time t', because q's deadline becomes more urgent, $E[C_{\text{delay}}(t')]$ and the expected cost reduction of the immediate execution of q (reflected by the shaded area in Figure 3(b)) become larger. Therefore, the priority score of q at time t', namely $p_q(t')$, grows larger than its priority score $p_q(t)$ at time t. As can be seen, the priority scores in CBS are *dynamically* changing over time.

In a recent work [8], we developed a specific implementation of CBS, named iCBS. However, iCBS is simply an efficient implementation of CBS under piecewise linear SLAs. Therefore, in our current context, iCBS shares the same weakness as CBS, which we discuss next.

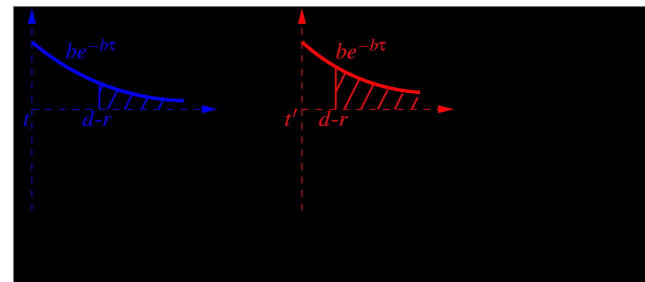

Figure 3: Illustration of CBS and its weakness.

2.3 Weakness in terms of Robustness

Despite its elegant theoretical foundation and superb performance in computer networking applications, CBS (and therefore iCBS) has a weakness that prevents it from being used in the database community. This weakness, we believe, is shared by all cost-aware scheduling policies that rely on knowledge of the *exact* query execution time. We illustrate

[2]In this paper we mainly focus on *non-preemptive* algorithms, where a query execution is not preempted once it gets started, a common scenario in database systems.

[3]Actually, $E[C_{\text{delay}}(t)]$ should be written as $E_\Gamma[C_{\text{delay}}(t)]$, with the underlying random variable Γ representing the further delay if q is chosen to be postponed.

[4]This assumption avoids the NP-hardness of the original scheduling problem by assuming the further delay of each query follows a known exponential distribution in an independent manner.

this in Figure 3(c). At time t'' (which is only slightly earlier than $d - r$), if we immediately start the execution of query q and q takes time r to finish, then the deadline can be met. As a result, we will set the term $c_{\text{now}}(t'')$ to 0 in Equation (1). If on the other hand, q takes $r+\Delta$ instead of r to finish its execution, then q will miss its deadline even if it is immediately started at t'' and as a result, we set $c_{\text{now}}(t'')$ to c. In other words, if we look at how sensitive $p_q(t)$ is with respect to the accuracy of our knowledge about the query execution time r of query q, it can be shown that at time t''

$$\lim_{\Delta \to 0} \frac{p_q(t'', r+\Delta) - p_q(t'', r)}{\Delta} = -\infty. \quad (2)$$

This result implies that a small perturbation to the execution time of q, or small imprecision in the prediction on q's execution time, can dramatically change the priority score of q in the scheduling algorithm. Therefore, we say CBS is not *robust* in the presence of imprecision in query execution time prediction. We believe this weakness is a hurdle to the application of CBS to database systems, in which it is almost impossible to know beforehand the *exact* execution time of a particular query instance. In our experiments, we will demonstrate that this weakness is shared by several state-of-the-art cost-aware scheduling algorithms and it has a non-trivial negative impact on their performance.

While inaccurate estimates for running times create problems for scheduling, we should point out that knowing nothing about r is not acceptable either. Without any knowledge about r, one cannot infer the urgency of each query and as a result, one cannot apply any cost- or urgency-aware scheduling. Recently, there have been many efforts in the database community (e.g., [12, 33, 10]) to reduce the errors in the prediction of r. Moreover, recent studies (e.g., [11]) have shown that although it can be extremely difficult to predict the *exact* execution time of a *particular* query instance, it is relatively easy, e.g., through a brief period of online monitoring, to learn about the *distribution* of the execution time for queries with the same template or from the same family.

In the rest of this paper, we focus on scenarios where r is not known *exactly* but instead can be determined to follow a certain probability distribution represented by a histogram.

3. SHEPHERD IN A SIMPLE SCENARIO

In this and the next section, we present in detail our algorithm, which we name "Shepherd," which explicitly takes into account the distribution of expected query execution times. At a high level, Shepherd makes use of the following insight:

> In Shepherd, instead of relying on a single *point estimation* for the execution time of a query q, we consider *all* the possible values of the execution time of q, together with their consequences in terms of SLA penalty cost.

Note that this is very different from relying on a single estimated mean value for the execution time of q.

3.1 Priority Score for Shepherd

We first describe how a priority score is defined in Shepherd. In Shepherd, we represent the execution time of q not as a constant, but instead, as a random variable R that follows a probability distribution. With this representation, we define the Shepherd priority score in a spirit similar to that of CBS, except that all the possible values of R are evaluated, according to the probability that they occur. Specifically, the Shepherd score is defined as:

$$\begin{aligned} p_q(t) &= \frac{1}{E[R]} \left(E[C_{\text{delay}}(t)] - E[C_{\text{now}}(t)] \right) \quad (3) \\ &= \frac{\int_{r=0}^{\infty} p(r) \left[\int_{\tau=0}^{\infty} be^{-b\tau} \cdot c(t+\tau, r) d\tau - c(t,r) \right] dr}{\int_{r=0}^{\infty} r \cdot p(r) dr}, \end{aligned}$$

where $p(r)$ is the probability density function of R, the random variable representing the execution time of q.

Comparing Equation (3) with the CBS score in Equation (1), we can see several distinguishing points. First, the r term in Equation (1) is replaced by $E[R]$. This is because the query execution time is not fixed but instead is a random variable R with mean value $E[R]$. Second, $c_{\text{now}}(t)$ is replaced by $E[C_{\text{now}}(t)]$. This is because that the query execution time R is a random variable, and therefore even if we start the execution of q immediately, the cost is still a random variable $C_{\text{now}}(t)$, and $E[C_{\text{now}}(t)]$ captures the expected value of $C_{\text{now}}(t)$. In addition, although comparing Equation (3) with Equation (1), the form for the term $E[C_{\text{delay}}(t)]$ does not change, the details have changed because the expectation is now taken over two random variables R and Γ[5].

The Shepherd score defined in Equation (3) is complicated, which naturally raises concerns about its practicality. On the one hand, Equation (3) does achieve our goal of evaluating over all the possible execution times for q; on the other hand, this evaluation requires a double integration. Furthermore, this evaluation (which is time-varying) has to be conducted at *any* time t when a scheduling decision is to be made, for *all* queries waiting to be serviced at time t. In the remainder of this paper, a main theme is to make the Shepherd scores more manageable through analysis, simplifications, and special-purpose techniques. We start by carefully deriving closed-form solutions to the Shepherd score under special scenarios. We first analyze Shepherd scores in a very simple scenario, where the query execution time follows a uniform distribution and the SLA cost is a single-step function. We use this simple scenario to illustrate the desirable properties of the Shepherd score and we postpone the description of more general scenarios to the next section.

3.2 Detailed Derivation of the Priority Score

In this section we focus on cases where R follows a uniform distribution as described in Figure 4. That is, if we start the execution of query q at time t, q's finishing time is uniformly distributed between $t + r_1$ and $t + r_2$. In the figure, obviously $h = \frac{1}{r_2 - r_1}$. In addition, we assume that the SLA cost function is a single-step function with a single deadline d and a jump of height c for the cost of missing the deadline, also shown in Figure 4.

3.2.1 Derivation of $E[C_{\text{now}}(t)]$

Figure 4 shows how $E[C_{\text{now}}(t)]$ is derived. At time t, if we immediately start the execution of q, there can be three cases, depending on the relation between d and ($t+r_1$,$t+r_2$):

[5]Actually, $E[C_{\text{now}}(t)]$ should be written as $E_R[C_{\text{now}}(t)]$ and $E[C_{\text{delay}}(t)]$ should be written as $E_{R,\Gamma}[C_{\text{delay}}(t)]$. This is because $C_{\text{now}}(t)$ only depends on R, the query execution time, while $C_{\text{delay}}(t)$ depends on both R and the further delay Γ, where Γ is also a random variable.

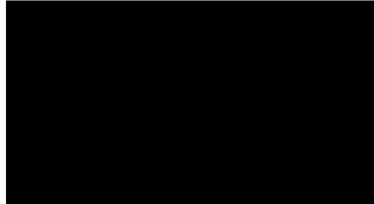

Figure 4: Uniform distribution of execution time, with three possible cases.

case 1, $t + r_2 \leq d$: In this case, q can meet its deadline no matter what its execution time R is (as long as R falls between $t+r_1$ and $t+r_2$, which is guaranteed). Therefore

$$E[C_{\text{now}}(t)] = 0.$$

case 2, $t + r_1 < d \leq t + r_2$: In this case, the expected cost of starting query q immediately at time t is proportional to the probability that R falls after the deadline d, and we can show

$$E[C_{\text{now}}(t)] = (t + r_2 - d)hc.$$

case 3, $d \leq t + r_1$: In this case, there is no way for q to meet its deadline and so we have

$$E[C_{\text{now}}(t)] = (r_2 - r_1)hc.$$

Note that in the last case, since we have $h = \frac{1}{r_2 - r_1}$, we actually have $E[C_{\text{now}}(t)] = c$. However, we keep the current form so that it can be extended in the next section to cases where h is not necessarily equal to $\frac{1}{r_2 - r_1}$.

3.2.2 Derivation of $E[C_{\text{delay}}(t)]$

If, on the other hand, we decide to postpone the execution of q, then the cost $C_{\text{delay}}(t)$ is another random variable. We provide the detailed derivation of the expected value $E[C_{\text{delay}}(t)]$ in Appendix A and here just present the result:

case 1, $t + r_2 \leq d$:

$$E[C_{\text{delay}}(t)] = \frac{hc}{b}\left(e^{b(r_2-d)} - e^{b(r_1-d)}\right)e^{bt}$$

case 2, $t + r_1 < d \leq t + r_2$:

$$E[C_{\text{delay}}(t)] = hc(t + r_2 - d) + \frac{hc}{b} - \frac{hc}{b}e^{b(r_1-d)}e^{bt}$$

case 3, $d \leq t + r_1$:

$$E[C_{\text{delay}}(t)] = (r_2 - r_1)hc.$$

3.2.3 The overall priority score

Now we derive the overall priority score for Shepherd. Note that $E[R]$ is a constant that is invariant over time. On the other hand, $E[C_{\text{delay}}(t)]$ and $E[C_{\text{now}}(t)]$, the expected costs of further postponing q and that of executing q right away, both change over time. At a give time t, we have

$$p_q(t) = \begin{cases} \frac{hc}{E[R]\cdot b}\left(e^{b(r_2-d)} - e^{b(r_1-d)}\right)e^{bt} & t + r_2 \leq d \\ \frac{hc}{E[R]\cdot b} - \frac{hc}{E[R]\cdot b}e^{b(r_1-d)}e^{bt} & t + r_1 < d \leq t + r_2 \\ 0 & d \leq t + r_1. \end{cases} \quad (4)$$

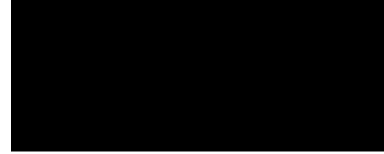

Figure 5: Shepherd score changing over time.

This priority score of Shepherd is displayed in Figure 5. As can be seen from the figure, for this particular case, namely a uniformly distributed execution time and a single-step SLA cost function, the Shepherd score increases exponentially up to time $d - r_2$ (referred to, in the rest of the paper, as the *floating* stage) and then starts to decrease exponentially up to time $d - r_1$ (referred to as the *sinking* stage). After time $d - r_1$, the Shepherd score becomes 0.

3.3 Robustness of the Shepherd Score

The original CBS can be considered as a special case of Shepherd in which there is no sinking stage. That is, in the original CBS, the priority score increases exponentially until time $d - r$, and then suddenly drops to 0.

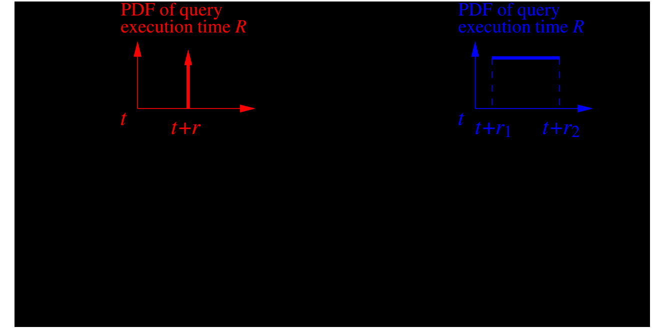

Figure 6: Robustness of (a) CBS vs. (b) Shepherd.

Figure 6 illustrates how the Shepherd score and CBS score change over time. From the figure we can see that compared to the CBS score, the Shepherd score is more *robust* in that a small disturbance or error in the query execution time prediction does not change the priority score too much. Such a robustness is very important especially when we cannot predict r perfectly, which is the case in most of database applications.

4. SHEPHERD IN GENERAL

In this section, we extend Shepherd to handle more general scenarios. First, we extend the execution time distribution from uniform distributions to multiple-bucket histograms. Then we extend the SLA cost function from single-step functions to multiple-step functions. However, instead of giving the tedious technical details, in the discussion in this section, we present intuitive descriptions and omit the detailed derivations. The key observation we leverage to address the general case is the following:

> The main operations we used to compute the Shepherd score, namely expectation, integration, and convolution, are all *linear* operations. Therefore, we can apply the *superposition principle.*

4.1 Histogram-based Query Execution Time

In the general cases, instead of a uniform distribution, we assume the distribution of the query execution time can be represented as a histogram with multiple buckets. A histogram-based representation is practical in many applications. For example, if a query q follows a query template (e.g., issued from a servlet from a Web server), then the execution time of historic queries from the same template can be used to infer q's execution time. The execution time of a large number of historic queries can be succinctly encoded in a histogram. As another example, very often, the prediction obtained from machine-learning approaches is not in the form of a single value (point estimation) but instead is in the form of a probability distribution (e.g., obtained from a Bayesian approach). Such probability distributions can be approximated by histograms with finite number of buckets.

It turns out that Shepherd can be easily extended from execution time following a uniform distribution (which can be considered as a histogram with a single bucket) to query execution time following a general histogram. The key observation is that in Equation (3), both $E[C_{\text{delay}}(t)]$ and $E[C_{\text{now}}(t)]$ are linear functions and as a result, we are able to compute $E[C_{\text{delay}}(t)]$ and $E[C_{\text{now}}(t)]$ for each bucket in the histogram separately and put the results together. ($E[R]$ is a constant independent of t and can be derived from the histogram.) More specifically, we can decompose a general histogram vertically into the sum of individual buckets, as shown in Figure 7, then handle each bucket by using the method in the previous section (recall that in the previous section, we intentionally used h instead of $\frac{1}{r_2-r_1}$). Finally, the results are aggregated to get the Shepherd score.

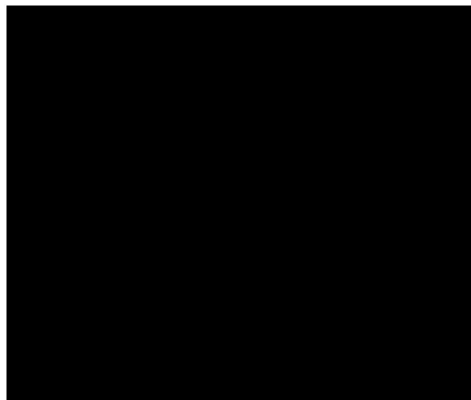

Figure 7: Decomposition of query execution time histogram.

4.2 Multiple-step SLA Cost Functions

Following a similar line of thought, we rely on the linearity of the Shepherd score to extend from single-step SLA cost function to multiple-step SLA cost functions.

Figure 8(a) shows a multiple-step SLA cost function with three deadlines and the corresponding costs. Such an SLA turns out to be decomposable horizontally into the sum of three single-step cost functions, as shown in Figure 8(b). Then we can compute the priority score for each of the single-step cost functions and sum them to get the total priority score for Shepherd.

Figure 8: Decomposition of a multiple-step cost function.

4.3 Putting It All Together

By combining (i) the decomposition of the histogram of the query execution time as shown in Figure 7 and (ii) the decomposition of the multiple-step cost function as shown in Figure 8, it can be derived, in a straightforward way, that the overall Shepherd score can be written as

$$p_q(t) = \sum_{i,j} p_{qij}(t). \tag{5}$$

In the above expression, $p_{qij}(t)$ is the Shepherd score for the special case with a single-bucket (the i-th bucket as in Figure 7) distribution of query execution time and a single-step (the j-th step as in Figure 8) cost function. This special case is exactly what has been solved in the previous section. Therefore, we have demonstrated that the general cases, with multiple-bucket histograms and multiple-step SLA cost functions, are not intrinsically more difficult than the special case in the previous section. We can prove that the Shepherd score for the general cases, as described by Equation (5), consists of various floating and sinking stages. Moreover, $p_q(t)$ is a *continuous* function over time t and therefore the non-robust scenario, as described earlier in Equation (2), will never happen to a Shepherd priority score.

5. EFFICIENT IMPLEMENTATION

So far we have focused on computing the Shepherd score for a *single* query. In real applications, there can be a large number of queries to be prioritized at any given time. In this section, we develop a technique that achieves this prioritization with time complexity logarithmic in N, the number of queries to be prioritized.

5.1 Efficient Prioritization

The Shepherd score for a query is continuously changing over time, as we can see from Equation (5). As a consequence, the query having the highest Shepherd score varies with time. On the one hand, this dynamic ordering is a key to the good performance of Shepherd. On the other hand, it seems to impose a time complexity of $\Omega(N)$ for choosing, among the N queries waiting to be serviced, the query currently having the highest Shepherd score — after all, it seems that at a given time t, the Shepherd score for each query (which depends on t) has to be recomputed. This time

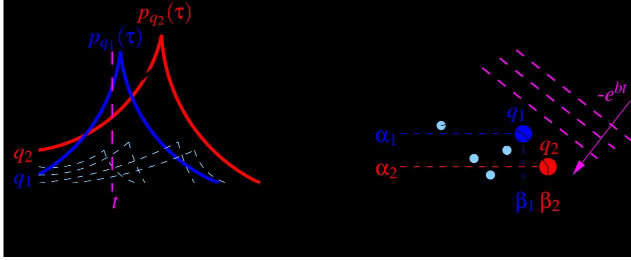

Figure 9: (a) Dynamically changing Shepherd scores and (b) static mapping into the dual space.

complexity is undesirable. To address this issue, we borrow a technique that we previously proposed in [8], which maps the time-varying Shepherd score for a query to a fixed point in the dual space of linear functions. This mapping is based on the following observation on the special form of the Shepherd score — for a query q, its Shepherd score at any given time t can always be written as

$$p_q(t) = \alpha_q + \beta_q e^{bt}, \tag{6}$$

where α_q and β_q are constants (i.e., invariant with respect to t) for a certain period of time. We start with the simple case of a single-bucket histogram and a single-step cost function as shown in Equation (4). As can be seen, $p_q(t) = \alpha_q + \beta_q e^{bt}$ where the (α_q, β_q) pair only change value twice during the lifetime of q — once from the floating stage to the sinking stage, and once from the sinking stage to 0. Similar properties hold in the Shepherd score for the more general cases. As shown in Equation (5), the Shepherd score for the general cases can be written as the superposition of several buckets in the histogram of query execution time and several steps in the cost function. Therefore, it can be shown that with I buckets in the histogram and J steps in the cost function, $p_q(t)$ is always in the form of $\alpha_q + \beta_q e^{bt}$ where the (α_q, β_q) pair only change value finite times during the lifetime of q — it happens only when the left or right border of a bucket in the histogram "hits" a new step in the cost function, which occurs $(I + 1)J$ times in total[6].

By using Equation (6), we can map each query q to a point with coordinate (α_q, β_q) in the dual space of linear functions. Then at time t, to find the query with the highest Shepherd score at t, we "sweep" the dual space by using a line with slope $-e^{bt}$, as illustrated in Figure 9(b). It can be shown that (i) the first point hit by the sweeping line corresponds to the query with the highest Shepherd score at time t, and (ii) such a point is guaranteed to be on the convex hull of all the points in the dual space. In this method, the *time-varying* (relative) priorities among the queries are reflected by two types of dynamics in the dual space. First, the coordinate (α_q, β_q) for a query q can shift several times during the lifespan of q. Second, depending on time t, we use a sweeping line with different slope $-e^{bt}$ and therefore may hit different "corners" on the convex hull (and therefore pick different queries). We refer the interested readers to [8] for further details.

[6]This is one reason why we focused on cost functions in the form of step functions. It can be shown that for general piecewise linear cost functions, as given in [8], the Shepherd score is in the form of $p_q(t) = \alpha_q + \beta_q \cdot e^{bt} + \gamma_q \cdot t$.

5.2 Implementation Details

For the above technique to work, we need an algorithm to maintain a planner convex hull incrementally. In our previous work [8], we pointed out that such algorithms with logarithmic time complexity exist (e.g., the $O(\log N)$ algorithm in [6] and the $O(\log^2 N)$ algorithm in [24]), without actually implementing any of them. In this paper, we implemented the algorithm proposed by Overmars and von Leeuwen [24], which has an $O(\log^2 N)$ time complexity. For the implementation, a special data structure named *concatenable queue* [2] is needed. We implemented the concatenable queue by using a 2-3 tree [2], and we implemented the 2-3 tree by extending the left-leaning-red-black-tree (LLRB) [32]. In total, our implementation of the dynamic convex hull algorithm consists of less than 1000 lines of Java code.

6. EXPERIMENTAL STUDIES

In this section, we evaluate the effectiveness and efficiency of distribution-based scheduling by investigating the performance of Shepherd using both simulations and real system experiments. The main metric we use to measure performance is the *average SLA penalty cost per query*. That is, after each query is finished, we look at its query response time, compare the response time with the deadline(s) in its SLA cost function, and derive its penalty cost.

6.1 Settings for the Experiments

Our simulator and TPC-W testbeds

We conducted experiments on two testbeds, both implemented in Java. The first testbed is a comprehensive discrete-event simulator developed at NEC Labs. This simulator allows us to evaluate different scheduling algorithms under different workload characteristics and SLAs. With the simulator, we can control the distribution of query execution time and even study hypothetical (and practically unattainable) *oracle* cases, where the exact query execution time is available to the scheduling algorithms before each query is executed.

Our second testbed is a deployment of the TPC-W benchmark[7], with the following details. The server machine has Intel Xeon 2.4GHz quad-core CPUs and 16GB memory. We used the 600MB data size for TPC-W; the database was MySQL 5.5 using InnoDB 1.1.3, with 1GB memory as the InnoDB buffer pool. We implemented a middleware-based workload manager, which handled the execution of the scheduling algorithms, generated the query workloads, and collected relevant statistics. Separate machines with the same specifications were used as clients to issue queries. The multiprogramming level (MPL) was set to 20, which was also the case for the simulator testbed. We experimented with other MPLs and observed similar performance, hence those results are omitted.

We achieved MPL 20 by creating 20 database working threads, where each of the threads maintained its own database connection. The workload manager generates SQL statements and pushes them into a queue shared by all 20 working threads; these working threads with individual JDBC connections then grab the SQL statements from the queue and execute them. The SQL queries were uniformly generated from 5 TPC-W query templates, which are a mixture of

[7]http://www.tpc.org/tpcw/

short running (< 1ms) and relatively long running (> 50ms) queries. However, to get a fair comparison among different algorithms, for each test, once a workload was generated and used by the first algorithm under study (usually FCFS), we recorded the arrival time and the SQL statement for each query; then we reused the same workload trace when running the other algorithms in the same test.

Two types of workload traffic

We used two types of workload traffic: *static* and *bursty*. For static traffic we used Poisson arrivals, where the arrival rate was tuned to achieve the target system load required by each experiment. Here, by *system load*, we mean the ratio of (1) the total service time required by all the queries that arrive during an experiment and (2) the total service time that can be provided by the database server. A system load greater than 100% corresponds to an overload situation, which is of interest to us as well.

For bursty traffic we used the well-known 1998 World Cup trace [3], which describes the visitor arrival rate to a webserver over the course of several hours. The trace is plotted in Figure 10, and again we scaled the arrival rate to achieve the target system load. Both types of traffic belong to the category of open-workload [31], where queries arrive independently of the status of previous queries.

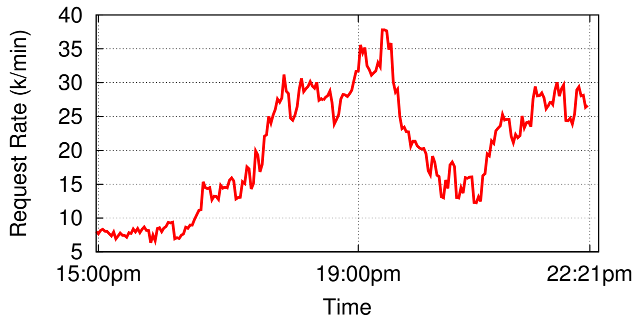

Figure 10: The World Cup trace, with the main event happened at 19:00pm.

Baseline algorithms

For baseline scheduling algorithms, we implemented two cost-unaware algorithms: first-come-first-serve (**FCFS**) and (**SJF**), where SJF is aware of query execution time but not SLA penalty cost. For more sophisticated cost-aware scheduling policies, we implemented two state-of-the-art algorithms, FirstReward and iCBS, described below. We chose these algorithms because they have cost-aware properties similar to Shepherd, namely they dynamically prioritize queries in real time based on query execution time and SLAs.

FirstReward [21] is a sophisticated algorithm that considers the benefit and opportunity cost of each scheduling choice. FirstReward needs as input the execution time of the queries to be scheduled. For the parameters *alpha* and *discount rate*, we use the suggested values of 0.3 and 0.01.

iCBS [8] is our previous implementation of CBS. For the parameter b, we follow the suggestion in [25] and set $1/b$ to be four times the mean query execution time. The same b is used in our **Shepherd** algorithm.

6.2 Evaluation on Effectiveness & Robustness

6.2.1 Effectiveness using synthetic data

We start with a simple experiment using the simulator. We generated queries with mean execution time of $r = 30$ ms using static traffic. The Poisson arrival rate was controlled so that the target system load, e.g., 95%, was achieved under the given MPL. However, when assigning the exact query execution time at run time, we randomly picked half of the queries and assigned their execution times using the histogram shown in the upper panel of Figure 1 (centered at 30 ms, with 3 buckets each with a width of 0.5 ms); the other half of the queries had execution times following the bimodal histogram described in the lower panel of Figure 1 (centered at 1ms and 59ms). The baseline algorithms only use the the means of query execution time, while Shepherd additionally takes into consideration the histograms. No algorithm (other than the oracle) knows the *exact* query execution time before a query is executed.

In terms of SLAs, we start with the simple one shown in Figure 2(a), namely with a single deadline. We set the deadline d at $5r = 150$ ms after the arrival of each query and we set the cost simply at $c = 1$. In terms of performance, we report the average SLA penalty cost per query. This average penalty, when used together with the simple SLA (i.e., single deadline and $c = 1$), turns out to be equivalent to the fraction of queries that miss their deadlines. For each system load and each scheduling algorithm, the simulation was run three times with different random seeds, each time using 100K queries. The average performance is reported in Table 1, under "SLA-1". As can be seen, Shepherd clearly outperforms its non-robust counterparts, incurring only between one third and one half of the SLA cost penalties of the best baselines.

Next, we randomly picked half of the queries (independent of the distribution of their execution time) and changed their SLAs to the one shown in Figure 2(b). More specifically, we set $c_1 = 1$, $c_2 = 10$, d_1 as $5r$ and d_2 as $10r$ after query arrival time, respectively. In other words, among all queries, half of them (e.g., from gold customers) have a second deadline, with a penalty cost much more severe than that of the first deadline. We use such a mixture of two SLAs to demonstrate that cost-aware scheduling algorithms can provide differentiated service. The results are shown in Table 1, under "SLA-2". As can be seen, the performance of the cost-unaware algorithm SJF suffers, while the cost-aware algorithms handle the new SLA quite well.

In addition, to find the performance degradation due to imprecise query execution time prediction, we ran an "oracle" version of iCBS, where the *exact* query execution time was used in the algorithm, and report the result in the last row of Table 1. As can be seen, the performance degradation from the oracle cases (iCBS-oracle, fed with the *exact* query execution time) to the practical cases (iCBS, fed with *mean* query execution time) can be up to 500%.

Table 2 reports the performance of the same set of tests using the World Cup traffic. As can be seen, although the performance improvement is less dramatic, Shepherd consistently outperforms the best baselines by around 30%.

6.2.2 Robustness using synthetic data

To compare the robustness of Shepherd to that of iCBS, we conducted the following simple test. In the test, the exe-

Table 1: **Average SLA cost per query, for different algorithms under different SLAs and system loads (static traffic).**

SLA Type	SLA-1			SLA-2		
load	95%	99%	105%	95%	99%	105%
FCFS	0.087	0.955	1.00	0.192	5.15	5.51
SJF	0.055	0.106	0.149	0.184	0.435	0.676
F-REWARD	0.015	0.057	0.121	0.026	0.134	0.305
iCBS	0.022	0.085	0.176	0.021	0.092	0.184
Shepherd	**0.008**	**0.030**	**0.067**	**0.008**	**0.035**	**0.077**
iCBS-oracle	0.004	0.016	0.031	0.005	0.022	0.047

Table 2: **Average SLA cost per query, for different algorithms under different SLAs and system loads (World Cup traffic).**

SLA Type	SLA-1			SLA-2		
load	95%	99%	105%	95%	99%	105%
FCFS	0.796	0.984	0.996	4.29	5.40	5.49
SJF	0.297	0.330	0.362	1.58	1.76	1.94
F-REWARD	0.221	0.234	0.250	0.767	0.825	0.883
iCBS	0.232	0.248	0.265	0.586	0.628	0.689
Shepherd	**0.173**	**0.187**	**0.200**	**0.408**	**0.464**	**0.516**
iCBS-oracle	0.099	0.108	0.117	0.281	0.313	0.341

cution time of the queries comes from the bimodal histogram described in Figure 11, with a mean of $r = 30$ ms and a gap of $2 * e * r$ between the two modes (e can be considered as the imprecision level in the knowledge about the query execution time). Also, all queries have the same SLA, with a single deadline at $5r$ and a cost of $c = 1$. Then, we conducted a series of tests with different e values. Under each e value, we ran Shepherd, iCBS, and iCBS-oracle. We report (i) the ratio of the performances of Shepherd and iCBS-oracle, and (ii) the ratio of the performances of iCBS and iCBS-oracle. As can be seen, as e increases, the imprecision of the query execution time increases. As a consequence, both Shepherd and iCBS suffer, in terms of their performance vs. the performance achievable by iCBS-oracle. However, the performance degrades much more gracefully with Shepherd, which suggests that Shepherd is much more robust to such imprecision.

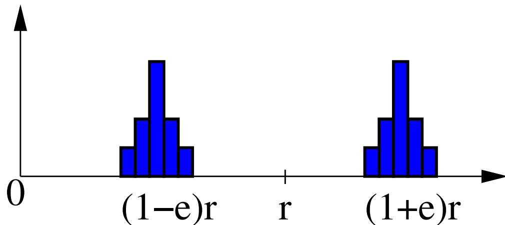

Figure 11: Histogram used in the robustness test.

6.2.3 *Effectiveness using TPC-W data*

For the experiments on the real TPC-W testbed, we chose a subset of five query templates, referred to as Q1,...,Q5, from the TPC-W benchmark specification. These five templates, whose details are skipped due to the space limit,

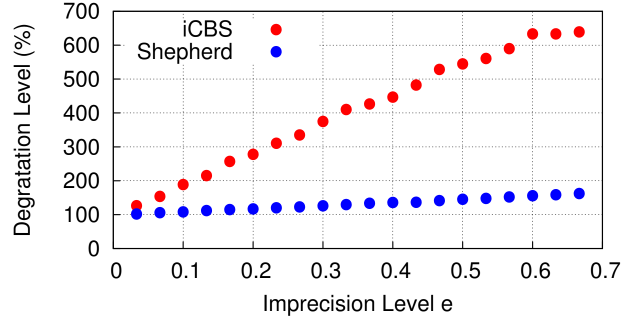

Figure 12: The robustness test.

are a mixture of (relatively) long-running queries and short-running ones. In the experiments, we used single-step SLAs and set the deadlines for each of the long-running queries (from query templates Q2, Q4, and Q5) to five times its mean execution time. For query templates Q1 and Q3, whose queries usually take less than 1 ms, we simply set their deadlines to be 10 ms after query arrival. For the SLA penalty cost, we assign the cost to be proportional to the mean query execution time. Note that in order to set up the deadlines and the SLAs, we need to know the mean execution time of queries from the five templates beforehand. For this purpose, we run a pre-processing step to get the profiles of the query templates, which are shown in Figure 13.

During the test, the baseline algorithms were provided with (i) the template for each query and (ii) the mean query execution time of the queries from each template (collected and updated in real time from the most recent 100 queries from the same template). In order for Shepherd to learn the histogram of query execution times in real time, we built the histogram, with five buckets, from the most recent 100 queries as well.

The performance under the static traffic is reported in Table 3, and those under bursty traffic is reported in Table 4. The performance is obtained by averaging over three runs (with different random seeds used by the workload manager), where each run consists of about 5,000 query instances. As can be seen, again, Shepherd consistently outperforms the best baseline by up to 50%.

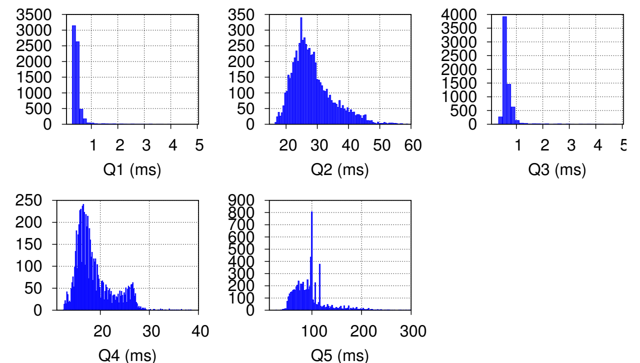

Figure 13: Execution time histograms for the five TPC-W query templates.

Table 3: TPC-W: average cost for static traffic.

load	55%	65%	75%	85%	95%	105%	130%
FCFS	5.57	5.70	5.11	7.29	42.3	70.2	70.5
SJF	4.21	4.81	5.49	7.30	21.0	26.7	30.6
F-REWARD	3.68	4.02	**3.51**	5.60	9.22	12.6	24.0
iCBS	3.51	4.47	4.27	4.61	5.93	16.2	24.4
Shepherd	**3.20**	**3.99**	3.70	**3.42**	**4.98**	**8.4**	**13.2**

Table 4: TPC-W: average cost for World Cup traffic.

load	55%	65%	75%	85%	95%	105%	130%
FCFS	44.4	52.5	60.7	62.5	63.9	67.2	67.3
SJF	18.8	21.5	25.1	26.65	28.3	28.9	30.7
F-REWARD	8.62	9.67	13.8	17.8	19.2	23.3	29.3
iCBS	9.98	11.8	12.2	13.4	16.1	22.5	24.6
Shepherd	**5.01**	**7.36**	**8.57**	**11.3**	**13.8**	**15.8**	**18.9**

6.2.4 *Robustness using TPC-W data*

To study of the level of performance degradation due to imprecise query execution time, we conducted the following test. First, after an experimental run, we collected from the log file the ground truth of the real execution time of each query from each of the five query templates. Then, using the simulator, we ran exactly the same queries with arrival times as recorded in the log file. For each scheduling algorithm, we ran the test two times: (i) first without knowing the exact query execution time (using template means in the baselines) and (ii) with the exact query execution time known before the query execution (oracles). In other words, each algorithm is compared to *its own* oracle, because the exactly same workload trace is used by the simulator and because the simulator can enforce the "true" execution time of each query.

We report the ratio between the performances from (i) and those from (ii) in Figure 14. Note that a degradation level of 100% implies the performance is *not* affected by imprecision in query execution time, whereas a larger level implies worse degradation. From the results we can see that cost-aware scheduling algorithms are not robust in the presence of imprecise query execution time — the real performances can be worse than the ideal (oracle) cases by a factor of two to three, especially when the load is high.

6.3 Evaluation of Efficiency

To study the efficiency of our Shepherd implementation, we compared a naively implemented version of Shepherd,

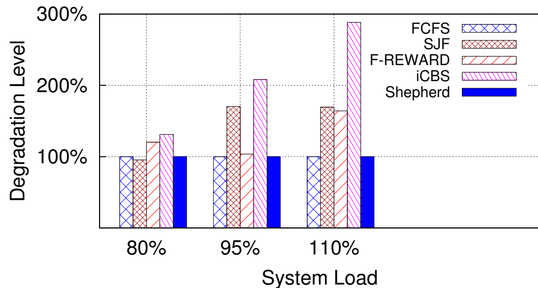

Figure 14: Performance degradation of different algorithms due to imprecise query execution time.

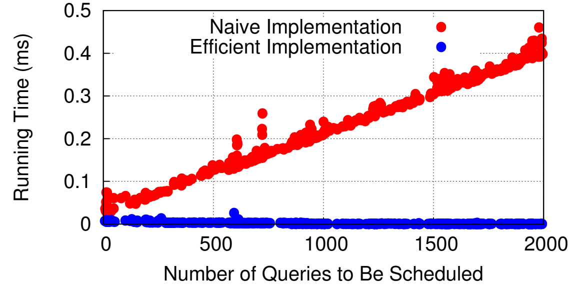

Figure 15: The running time under different numbers of queries to schedule.

where the Shepherd scores of all queries are re-computed each time, with our implementation as described in Section 5. To study the scalability of the algorithm, we introduced a large number of queries waiting to be serviced in the following way. We used an SLA with a single deadline and we set the deadline to a very large value, and at the same time we increased the system load to borderline overload. Figure 15 is a scatter plot that shows how the naive implementation of Shepherd and our implementation scale with respect to the number of queries waiting in the queue to be serviced. We can see that in the naive implementation, the time taken to make a scheduling decision is proportional to the number of queries waiting to be serviced; in comparison, in our efficient implementation, a couple thousands of queries can be handled in negligible time. This result illustrates that Shepherd can be implemented in an efficient way so as to be able to handle large number of queries.

7. RELATED WORK

In this section, we survey literature in related areas.

Scheduling and cost-aware scheduling

Scheduling is a well-studied research topic and is ubiquitous in many applications [7]. Most analytical results in queueing theory are obtained by assuming that query arrival rates and query execution times follow identical independent distributions (i.i.d.). On the other hand, when per-query scheduling decisions are made where each query has a known execution time and hard deadlines, most problem instances become NP-complete [34]. Furthermore, the situation is not improved when the hard deadlines are replaced by minimization of the number of deadline violations [25]. Therefore, most scheduling algorithms adopt certain heuristics. One family of such algorithms are cost-aware or value-based scheduling algorithms. In these algorithms, the decisions on scheduling are made so that certain *costs* are optimized. The costs could be defined in different ways: they could be fixed or time-varying values assigned to different queries [17, 21]; they could be the costs of rolling back and restarting transactions [20]; they could be about other metrics such as fairness [15] and result quality [18]. More commonly, the costs of queries under different query response time are derived from SLAs [23, 38, 28]. However, in most of the approaches described above, a common weakness is that they assumed the exact query execution time to be available beforehand, which is not practical in most database applications.

SLA penalty costs

SLA penalty costs may come from various sources. In service oriented architectures such as cloud computing, SLAs may be readily available. For example, Amazon's Dynamo system guarantees the 99.9-percentile query response time under certain peak workload [9]. The penalty cost can also come from the requirements of particular applications. In the field of Web service design, it has been shown there exists strong correlations between query latency and user-perceived Quality of Service (QoS) and user satisfaction [5]. Such correlations have been extensively used in Web service QoS modeling and design [23, 18]. In computer networking, packets from different applications (e.g., emails vs. real-time voice) are usually associated with different urgency [26]. In real-time systems, query deadlines may be determined by characteristics of physical systems and controls, QoS requirements, human sensory perceptions, and so on [29].

Query execution time prediction

In the past few years, much research work has been done to address the problem of predicting the execution time of a query before the query is actually executed. Solutions proposed so far cover both OLTP [12] and OLAP [22, 36, 35] queries, as well as both stand-alone [12, 36] and concurrent [10, 35] queries. Because most of these solutions are based on machine-learning approaches, and because real data contain noise and inherent uncertainties, it is reasonable to assume there exists unavoidable imprecision in the predicted query execution time. Such imprecise query execution time is explicitly addressed by the Shepherd algorithm in this paper.

8. CONCLUSION

In database systems, when the predictions on query execution time are imprecise, there are two choices: either we can ignore the imprecision and use a typical execution time such as the mean value, or we can explicitly model and take such imprecision into account in scheduling decisions. In this paper, we show that the former leads to performance degradation, and accordingly we propose a novel cost-aware scheduling algorithm Shepherd, which we believe to be the first one belonging to the latter. Shepherd has rigorous theoretic underpinnings as well as potential in real applications; it is robust to imprecision in query execution time prediction; and it is highly scalable. Such robustness and efficiency are both supported by our empirical studies.

However, we do not claim Shepherd to be the best solution to scheduling under imprecise query execution time. Perhaps the most interesting aspect of this paper is that by presenting Shepherd, we have demonstrated that it is both necessary and manageable to explicitly treat imprecise estimates of query execution time in scheduling decisions. We believe such an explicit treatment of imprecise query execution times is both crucial to practitioners and fertile ground for future research.

9. ACKNOWLEDGMENTS

We thank Jon M. Peha for his helpful suggestions, and Hyun Jin Moon for his help in building the testbeds. The work described in this paper is part of the CloudDB project [16] at NEC Laboratories America.

10. REFERENCES

[1] A. Aboulnaga, K. Salem, A. A. Soror, U. F. Minhas, P. Kokosielis, and S. Kamath. Deploying database appliances in the cloud. *IEEE Data Eng. Bull.*, 32(1):13–20, 2009.

[2] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[3] M. Arlitt and T. Jin. A workload characterization study of the 1998 World Cup Web site. *Network, IEEE*, 14(3):30–37, 2000.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[5] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. *Comput. Netw.*, 33(1-6):1–16, June 2000.

[6] G. Brodal and R. Jacob. Dynamic planar convex hull. In *FOCS*, 2002.

[7] P. Brucker. *Scheduling algorithms*. Springer, 5th edition, 2007.

[8] Y. Chi, H. J. Moon, and H. Hacıgümüş. iCBS: Incremental cost-based scheduling under piecewise linear SLAs. *PVLDB*, 4(9):563–574, 2011.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[10] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, 2011.

[11] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *WWW*, 2004.

[12] 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*, 2009.

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

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

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

[16] H. Hacıgümüş, J. Tatemura, W.-P. Hsiung, H. J. Moon, O. Po, A. Sawires, Y. Chi, and H. Jafarpour. CloudDB: One size fits all revived. In *SERVICES*, 2010.

[17] 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.

[18] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *SOCC*, 2012.

[19] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[20] D. Hong, T. Johnson, and S. Chakravarthy. Real-time transaction scheduling: a cost conscious approach. In *SIGMOD*, 1993.

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

[22] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[23] Z. Liu, M. S. Squillante, and J. L. Wolf. On maximizing service-level-agreement profits. In *ACM Conference on Electronic Commerce*, 2001.

[24] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.

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

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

[27] J. M. Peha and F. A. Tobagi. Cost-based scheduling and dropping algorithms to support integrated services. *IEEE Transactions on Communications*, 44(2):192–202, 1996.

[28] F. I. Popovici and J. Wilkes. Profitable services in an uncertain world. In *Supercomputing*, 2005.

[29] K. Ramamritham. Where do deadlines come from and where do they go? *Journal Of Database Management*, 7(2):4–10, 1996.

[30] K. Ramamritham, S. H. Son, and L. C. Dipippo. Real-time databases and data services. *Real-Time Syst.*, 28(2-3):179–215, 2004.

[31] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, 2006.

[32] R. Sedgewick. Left-leaning red-black trees. In *Dagstuhl Workshop on Data Structures*, 2008.

[33] S. Tozer, T. Brecht, and A. Aboulnaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE*, 2010.

[34] J. D. Ullman. Np-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.

[35] W. Wu, Y. Chi, H. Hacıgümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. In *PVLDB*, 2013.

[36] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacıgümüş, and J. F. Naughton. Predicting query execution time: are optimizer cost models really unusable? In *ICDE*, 2013.

[37] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacıgümüş. ActiveSLA: A profit-oriented admission control framework for database-as-a-service providers. In *SOCC*, 2011.

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

APPENDIX

A. DERIVATION OF $E[C_{\text{DELAY}}(T)]$

To simplify the discussion, we introduce a new random variable F to represent *the extra time, with respect to t, that is needed for q to finish its execution*[8]. Notice that such extra time is the sum of q's queueing delay and its execution time. If we use random variable W to denote q's queueing delay and R to denote q's execution time, then we have $F = W + R$. Because F is the sum of the two random variables W and R, the PDF $f(\tau)$ of F is the *convolution* of $w(\tau)$ and $r(\tau)$. That is

$$\begin{aligned}
f(\tau) &= \int_{\xi=0}^{\infty} w(\xi)\cdot r(\tau-\xi)d\xi \\
&= \begin{cases} 0 & \tau \le r_1 \\ \int_0^{\tau-r_1} be^{-b\xi}\cdot hd\xi & r_1 < \tau \le r_2 \\ \int_{\tau-r_2}^{\tau-r_1} be^{-b\xi}\cdot hd\xi & r_2 < \tau \end{cases} \\
&= \begin{cases} 0 & \tau \le r_1 \\ h\cdot\left(1-e^{-b(\tau-r_1)}\right) & r_1 < \tau \le r_2 \\ h\cdot\left(e^{-b(\tau-r_2)}-e^{-b(\tau-r_1)}\right) & r_2 < \tau \end{cases}
\end{aligned}$$

Figure 16 shows the distribution of F. At time t, if we decide to postpone the execution of q, obviously the finishing time of q is $t + F$.

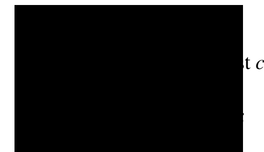

Figure 16: Probability distribution of finishing time if a query is postponed.

With the distribution of the finishing time $t+F$ given, we can derive the expected value of $C_{\text{delay}}(t)$.

case 1, $t + r_2 \le d$:

$$\begin{aligned}
E[C_{\text{delay}}(t)] &= \int_{\tau=d-t}^{\infty} h\cdot\left(e^{-b(\tau-r_2)}-e^{-b(\tau-r_1)}\right)cd\tau \\
&= \frac{hc}{b}\left(e^{br_2}-e^{br_1}\right)e^{-b\tau}\Big|_{\infty}^{d-t} \\
&= \frac{hc}{b}\left(e^{b(r_2-d)}-e^{b(r_1-d)}\right)e^{bt}
\end{aligned}$$

case 2, $t + r_1 < d \le t + r_2$:

$$\begin{aligned}
E[C_{\text{delay}}(t)] =& \int_{\tau=d-t}^{r_2} h\cdot\left(1-e^{-b(\tau-r_1)}\right)cd\tau \\
&+ \int_{\tau=r_2}^{\infty} h\cdot\left(e^{-b(\tau-r_2)}-e^{-b(\tau-r_1)}\right)cd\tau \\
=& hc(t+r_2-d)+\frac{hc}{b}-\frac{hc}{b}e^{b(r_1-d)}e^{bt}
\end{aligned}$$

case 3, $d \le t + r_1$: $E[C_{\text{delay}}(t)] = (r_2 - r_1)hc$.

[8] It can be shown that F is time-invariant, i.e., independent of t.