Scheduling Update and Query Transactions under Quality Contracts in Web-Databases [*]

Χρονοδρομολόγηση Ενημερώσεων και Ερωτήσεων με Συμβόλαια Ποιότητας για Βάσεις Δεδομένων στον Παγκόσμιο Ιστό Πληροφοριών

Huiming Qu [†]
Advanced Data Management Technologies Lab
Department of Computer Science
University of Pittsburgh
huiming@cs.pitt.edu

Alexandros Labrinidis [‡]
Advanced Data Management Technologies Lab
Department of Computer Science
University of Pittsburgh
labrinid@cs.pitt.edu

ABSTRACT
In modern web-database systems, users typically perform read-only queries, whereas all data updates are performed in the background, concurrently with queries. In this paper, we present the concept of Quality Contracts which allows individual users to express their preferences by assigning "profit" values to their expected Quality of Service (QoS) and Quality of Data (QoD). We propose an adaptive algorithm, called QUTS, to maximize the total profit from submitted Quality Contracts, which essentially optimizes the overall user satisfaction. QUTS address the problem of prioritizing the scheduling of updates (crucial to QoD) over queries (crucial to both QoS and QoD) using a two-level scheduling scheme that dynamically allocates CPU resources to updates and queries. We present the results of an extensive experimental study using real data (taken from a stock information web site), where we show that QUTS performs better than baseline algorithms under the entire spectrum of quality contracts; QUTS adapts fast to changing workloads and QUTS' sensitivity to its two parameters is negligible.

ΠΕΡΙΛΗΨΗ
Σε μοντέρνα συστήματα βάσεων δεδομένων στον παγκόσμιο ιστό πληροφοριών, οι χρήστες συνήθως υποβάλλουν ερωτήσεις, ενώ όλες οι ενημερώσεις γίνονται στο υπόβαθρο, ταυτόχρονα με τις ερωτήσεις. Στο άρθρο αυτό, εισάγουμε την έννοια των Συμβολαίων Ποιότητας (Σ.Π.), τα οποία δίνουν τη δυνατότητα στους χρήστες να εκφράσουν τις προτιμήσεις τους αναθέτοντας τιμές κέρδους για την Ποιότητα Εξυπηρέτησης (Π.Ε.) και Ποιότητα Δεδομένων (Π.Δ.) που περιμένουν στις απαντήσεις που θα πάρουν. Συγκεκριμένα, προτείνουμε έναν προσαρμοστικό αλγόριθμο, QUTS, που μεγιστοποιεί το συνολικό κέρδος από τα Συμβόλαια Ποιότητας που έχουν έρθει στο σύστημα και, με τον τρόπο αυτό, μεγιστοποιεί το συνολικό βαθμό ικανοποίησης των χρηστών. Ο αλγόριθμος QUTS δίνει λύση στο πρόβλημα της ανάθεσης προτεραιότητας για τη χρονοδρόμηση των ενημερώσεων (που είναι κρίσιμες για την Π.Δ.) και των ερωτήσεων από τους χρήστες (που είναι κρίσιμες για την Π.Ε.), χρησιμοποιώντας ένα διεπίπεδο σχήμα χρονοδρόμησης, το οποίο αναθέτει δυναμικά υπολογιστικούς πόρους μεταξύ ερωτήσεων και ενημερώσειων. Σε αυτό το άρθρο, παρουσιάζουμε τα αποτελέσματα μιας εκτενούς πειραματικής μελέτης, βασισμένης σε πραγματικά δεδομένα πρόσβασης και ενημέρωσης, και δείχνουμε οτι ο προτεινόμενος αλγόριθμος QUTS έχει πολύ καλύτερη απόδοση από τους αλγορίθμους που χρησιμοποιήσαμε ως βάση αναφοράς. Η υπεροχή του QUTS έναντι των άλλων αλγορίθμων είναι σταθερή σε ολόκληρο το φάσμα των Συμβολαίων Ποιότητας. Τέλος, ο αλγόριθμος QUTS προσαρμόζεται γρήγορα σε μεβαλλόμενα περιβάλλοντα και δεν έχει ευαισθησία στις παραμέτρους του.

[*]Funded in part by NSF ITR Medium Award (ANI 0325353).
[†]Name in Chinese: 屈慧明
[‡]Name in Greek: Αλέξανδρος Λαμπρινίδης

Proceedings of the 5th Hellenic Data Management Symposium (HDMS'06), September 7-8, 2006, Thessaloniki, Greece.

Πρακτικά του 5ου Ελληνικού Συμποσίου Διαχείρισης Δεδομένων (ΕΣΣΔ΄06), 7-8 Σεπτεμβρίου 2006, Θεσσαλονίκη, Ελλάδα.

1. INTRODUCTION

Data-intensive web sites are part of everyday life. Checking sports scores, viewing stock quotes, evaluating realtime warehouse inventory levels, and accessing personalized weather information are all such cases, i.e., of web-database applications that exhibit high update rates and mostly read-only queries. On the one hand, the proliferation of sensor devices and networks is only going to make the volumes of collected data even more massive, resulting in what is being referred to as High-Fan-In systems [9]. On the other hand, the Web is expected to remain the primary user interface for accessing such data, and of course, the demand for online availability of the data will also remain. As such, data-intensive web sites are expected to become even more data-intensive in the near future.

In earlier work, we showed that WebView materialization allows for better scalability in data-intensive web sites, without sacrificing freshness [20]. The key idea was to *decouple* the processing of the queries from the processing of updates. Query results were materialized outside the database (at the web server) and updates were propagated asynchronously in the background (from the DBMS). This setup allows for exploiting the trade-off between performance or *Quality of Service* (QoS) and freshness or *Quality of Data* (QoD), by selecting which query results to materialize [19], and essentially by building an "intelligent" asynchronous cache with eager updates. Efforts towards implementation of similar frameworks have been publicized by most major database vendors [25, 24, 5, 23].

In this paper, we remove many of the assumptions of earlier work, and address the problem of efficiently executing queries concurrently with updates in data-intensive web sites. Specifically, we assume a memory-resident system, such as a main-memory DBMS (as was the case in [3]), or an asynchronous middle-tier cache (as was the case in [19, 5, 23]). We allow queries and updates to run concurrently in the system and address the scheduling of *queries* and *updates* in the presence of *user preferences* in QoS, for example, through a *deadline*, and QoD, for example, through a *freshness requirement*. Our end goal is to maximize the overall user satisfaction (i.e., how close the system matched the prescribed users' preferences).

Motivating example: Our motivating example is that of a web-based stock market information system. In such a system, the web server is receiving queries from users who want to find out current stock prices, perform analysis comparing multiple stocks, etc. The web server (or, rather, the database system) also receives continuous feeds with updates on stock prices, and possibly additional news feeds with articles related to traded companies. Clearly, all users would like to have their queries completed (1) as fast as possible (i.e., have high QoS) and (2) with the most up-to-date data values (i.e., have high QoD). However, this may not be possible most of the times, either because of high query load (for example, a lot of people want to check their stock portfolios during lunch time) or because of high update load (for example, because of a breaking news story). In these situations, many users would be willing to receive slightly stale data, if the data are consistent and the result is returned fast; for example, a query that compares the relative values of two stocks over time can use slightly stale data (i.e. while there are pending updates), without forcing the user to wait for the data to become fresh. Other users would prefer to get data slightly delayed, but with the "promise" that their queries were performed over fresh data.

In general, the QoS and QoD preferences are expected to vary significantly from one user to another. As our motivating example indicated, some users would probably put QoS first and then worry about QoD, whereas others will prefer high QoD with slightly less QoS. In this paper, we propose a unifying framework for specifying QoS and QoD requirements, which we call *Quality Contracts*. Quality Contracts, or QCs for short, are based on the microeconomic paradigm [35, 8], but effectively "merge" all dimensions of Quality (i.e., multiple, different definitions of QoS and QoD) into a single, unifying concept. Essentially, the user specifies the amount of "worth" to him/her for the query to have a certain QoS and QoD. In this way, users can specify the relative importance of QoS over QoD and can also specify the relative importance among their different queries. The system, on the other hand, can infer the relative importance of different users' queries and allocate its resources to maximize "profits", and as such, maximize user satisfaction. With QCs, we can now cast the problem of scheduling queries and updates into the problem of optimizing the total profit for the system.

In this paper, we present a novel two-level scheduling scheme for scheduling updates and queries under quality contracts. The basic idea behind the proposed scheme, is in deciding on the allocation of resources between queries and updates using the expected "profit gain" to the system: executing queries contributes to QoS (and QoD), whereas executing updates contributes to QoD. We propose, *QUTS*, short for Query-Update Time-Sharing, which adapts resource allocation by monitoring the achieved user satisfaction in QoS/QoD and comparing it to the best-case scenario.

Contributions: In this paper, we make the following contributions:

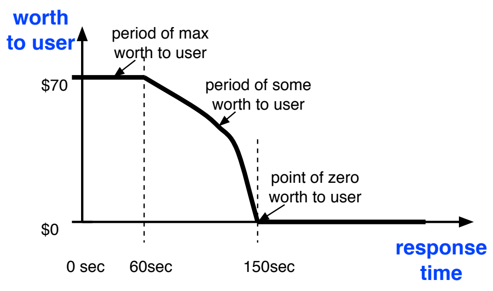
1. We introduce Quality Contracts (QCs), a unifying framework for expressing QoS and QoD user preferences.

2. We propose using a two-level scheduler for scheduling queries and updates in the presence of QCs. We also develop a new (single-level) global scheduling algorithm for comparison.

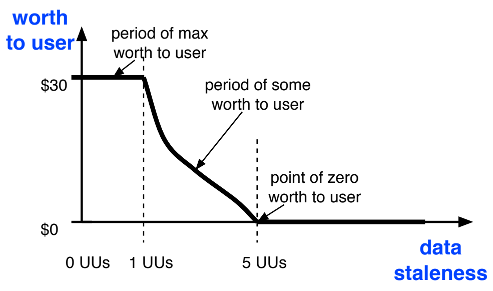
3. We present QUTS, which is a two-level scheduler, that address the problem of prioritizing the scheduling of updates (crucial to QoD) over queries (crucial to both QoS and QoD) using a two-level scheduling scheme. In the presence of QCs, QUTS dynamically allocates CPU resources to updates and queries at a high-level, while allowing for maximum flexibility in prioritizing the queries and updates in the low level.

4. We perform an extensive experimental study, where, using real data (query and update traces from a popular stock market information server) we compare QUTS to current algorithms (which it outperforms) and examine the sensitivity of the algorithm with regards to its two parameters (very little) and also its adaptability in the presence of rapidly changing workloads (very good).

Structure of paper: The paper is organized as follows. We present the Quality Contracts concept and our optimization problem in Section 2. Section 3 summarizes related work and describes the baseline algorithms. We introduce our two-level scheduling algorithm in Section 4. Section 5 describes our experimental setup, whereas Section 6 presents the results of our experimental study. Finally, we conclude in Section 7.



(a) QoS graph ($0 to $70)



(b) QoD graph ($0 to $30)

Figure 1: QC example: combination of QoS and QoD requirements for one ad-hoc query. QoS is measured in response time (seconds), whereas QoD is measured in number of Unapplied Updates (UUs)

2. QUALITY CONTRACTS

For traditional database management systems, QoS is measured in textbook ways: response time, stretch[1], etc. In the case of real-time DBMSs, adherence to deadlines (soft or hard) is an additional measure of QoS [17]. There are also many different ways to measure QoD in traditional DBMSs (see [19] for an overview). Basically, there are three main categories of QoD metrics: *time-based*, where the time since the last update is used as a measure of staleness, *lag-based*, where the number of unapplied updates is used as a measure of staleness, and, finally, *divergence-based*, where the difference of the current value from the most up-to-date value is used as a measure of freshness. Such QoD metrics can be further aggregated (either using average, minimum, or maximum) when we are considering a set of data items together (e.g., HTML fragments as part of a web page).

The most important deficiency of the current approaches to QoS/QoD is that they *do not have strong support for user preferences*. In typical DBMSs (i.e., for ad-hoc queries), quality is simply reported as an overall system property (even if both QoS and QoD are reported as separate measures); user preferences are not even considered. There are a few exceptions to this. Work on real-time databases [18, 32, 2] typically considers user preferences on a single QoS metric (in this case: preference on response time by means of a deadline) while attempting to maximize QoD. Our work on database-driven web servers [19, 22, 21, 20], balances the trade-off between QoS and QoD, while considering user preferences on one of the two measures: given an application-specified QoD requirement, the proposed system adapts to improve the overall QoS. Finally, in prior work [31], we have extended the work of [18] to perform admission control for incoming queries and consider both QoS user requirements (i.e., deadlines) and QoD user requirements (i.e., freshness threshold).

In this paper, we propose Quality Contracts, as a simple, yet powerful, framework to express user preferences for QoS and QoD. Quality Contracts, or QCs for short, are based on the microeconomic paradigm [35, 8], but effectively “merge” all dimensions of Quality (i.e., multiple, different definitions of QoS and QoD) into a single, unifying concept. In the general form of QCs, the user specifies a function over the quality metric of interest, along with the amount of “worth” to him/her for the query to have a certain QoS or QoD when it executes. In this way, users can specify the relative importance of QoS over QoD and can also specify the relative importance among their different queries. We illustrate the concept with an example.

Figure 1 is an example of a Quality Contract (QC) for a query submitted by a user. We assume an ad-hoc query in this example, whose QC consists of two graphs: a QoS graph (Figure 1a) and a QoD graph (Figure 1b). The QoS metric in this QC is response time, whereas the QoD metric is data staleness. Data staleness is measured in number of unapplied updates (UUs), i.e., number of pending updates for the data items that were accessed in order to answer the query. In our work, we assume that the staleness of a collection of data items (accessed by a query) is the highest staleness over all data items in the group (i.e., we get the max(UU) count for the entire query). From the example, we see that *QCs allow users to combine different aspects of quality*. In this example, the user has set the budget for the query to be $100; $70 are allocated for optimal QoS, whereas $30 are allocated for optimal QoD. This allocation is one important feature of the QC framework: *users can easily specify the relative importance of each component of the overall quality by allocating the query budget accordingly.*

Although we are not the first ones to use the microeconomic paradigm (for resource allocation, e.g., [35, 8]) or use a graph/function to describe quality (e.g., schemes in data stream management systems [7, 1]), to our best knowledge, this is the first work that combines these concepts together in a unified framework, also allowing for multiple quality metrics to be expressed at the same time. In fact, we propose to use the Quality Contracts framework for both ad-hoc queries (like those in this work) and also for continuous queries (like those processed by data stream management systems), but this discussion is beyond the scope of this paper. Existing schemes [7, 1], do not allow for a user to specify the relative importance of each component of the overall quality (at best, they simply have a global function which combines these metrics in the same way for all users), and they do not allow for a simple way to distinguish among queries of difference importance (from the same user or from different users). All these issues are elegantly addressed by the QC proposal.

Usability of Quality Contracts: We envision that a system which supports Quality Contracts (QCs) will provide a wide assortment of possible types of QoS/QoD metrics to the users. Making QCs easy to configure is fundamental to their acceptance by the user community. Towards this we expect service providers to support *parameterized versions of QC graphs* that the users can easily instan-

[1]Stretch measures the factor by which a job is slowed down relative to the time it would have taken to execute if it where the only job in the system [26, 29].

tiate. In fact, a simpler scheme is one where the service provider has already identified a "budget" for each type of user (which will translate to a certain class of QCs, a real-life example is the amount of free minutes included in wireless plans today) and a user will simply have to turn a "knob" on whether she prefers higher QoS or higher QoD (the real-life equivalent is choosing for the same price on a wireless plan whether you want more anytime minutes or more free nights & weekends minutes). In this way, service providers can better provision their systems, provide different classes of service, and allow end users to specify their preferences with minimal effort.

Definitions: We assume that there are two kinds of transactions in our system: user query transactions (or simply *queries*) and update transactions (or simply *updates*). The set of all queries is denoted as $Q = \{q_i | 0 \leq i < N_q\}$. The set of all updates is denoted as $U = \{u_j | 0 \leq j < N_u\}$. Note that both query and update sets are ordered by arrival time. The number of queries N_q and the number of updates N_u can be infinity, which means that both query and updates can be unbounded. The database D consists of N_d data items, more formally, denoted as $D = \{d_k | 0 \leq k < N_d\}$. Each user query q_i reads one or more data items, whereas each update transaction u_i updates a single data item.

Optimization Goal: Given multiple quality graphs (or functions), the question remains on how to combine this into a single QC. For our case, we have QoS and QoD, so the issue remains on how to combine them into a single function that the system will optimize. There are four choices, based on whether the two quality metrics are independent of each other or not. The fully-dependent choice would be represented by multiplying the two metrics together, which means that both QoS and QoD constraints have to be met before the system gets any "profit". In our case, we opt for the full-independent case, where essentially we are adding the two metrics up. This means that QoD "profit" can be gained even if the QoS constraint is not meant, and vice versa.

3. RELATED WORK AND BASELINE ALGORITHMS

To our knowledge, no work has been done to optimize the system profit with the presence of both QoS and QoD profit functions. However, some related work has given solutions to part of the problem. In the following, we present the related work and two simple solutions based on those works.

Scheduling Queries: One of the related work is in real time systems where many schemes [2, 6, 14, 30, 13] are proposed to schedule tasks with time critical value functions, which is similar to our QoS profit function in quality contracts. The idea is to consider both deadlines and values (or profit) of the tasks. Haritsa showed that Value over Relative Deadline (VRD) [12], which uses the ratio of value over the relative deadline as the task priority, gives the best performance over all workloads, thus it is a well-suited candidate to assign query priorities. With some changes on the values which was originally only on QoS, a promising priority scheme for queries could be Profit over Relative Deadline (PRD) which uses the ratio of total maximal profit from both QoS and QoD functions over query relative deadline. The maximal profit varies along time: before the query deadline, it is the sum of QoS and QoD maximal profit; but after the query deadline, it only contains QoD maximal profit.

Scheduling Updates: Prioritizing updates is not as straightforward as queries since updates do not have profit functions associated with them. Adelburg has studied some alternatives in [3] which may be useful although it is under a different setup, where queries have QoS profit functions and but no QoD profit function associated. [3] showed that there are two promising schemes: update first (update always have higher priority than queries) and update on-demand (relative updates are executed only when queries find the needed data items are stale). Something missing in [3] is that they did not use any concurrency control which could be tricky with update-on-demand when the the queries get preempted because how to resume the these queries depends on what concurrency control is used. Update first, instead, is more independent with the concurrency control. Moreover, only execute on-demand is shown to diminish the response time by later work[20]. Thus, we use update first to concurrently schedule updates and queries. Since this guarantees the highest QoD has no effect on QoS, there is no need of priority within the updates, so First In First Out (FIFO) is applied.

Concurrency Control (CC): As we just mentioned, [3] did not address the problem of concurrency control. Yet data contention is unavoidable with the preemptive real time scheduling of queries and updates. In conventional databases, there are two prevailing concurrency schemes, Two Phase Locking (2PL) and Optimistic Concurrency Control (OCC). People have extended the concurrency control schemes in both firm-deadline and soft-deadline real time database systems [2, 11, 10, 15, 34, 33, 27, 28]. Earlier studies[11] have shown that OCC based performs better than 2PL based schemes in firm deadline systems because in such systems tasks that miss deadlines are discarded immediately and it is very beneficial to OCC's late conflict resolvent. Whereas [10] showed that 2PL-HP[2] (two phase locking - high priority, which solves any conflict in favor of high priority tasks) outperforms the others with finite resources in soft real time database systems where tasks continue even after deadlines with less value (or profit) generated. In our case, queries have both QoS and QoD profit functions which means even if no QoS profit is possible to earn, the system should still try to finish the query for the QoD profit. Thus, it is more towards soft real time systems and 2PL-HP is the better choice than any other lock based and invalidation based concurrency control.

Along with all the above introduction on related works, we introduce two promising candidate baseline algorithms:

- **UH (Update High)** UH is a preemptive scheduling scheme with two priority queues for updates and queries, respectively. Updates always have higher priorities than queries. Within updates, FIFO (First In First Out) is applied for its simplicity, since with updates always proceeding queries, re-ordering updates have little effects on the actual profit from queries. Within queries, PRD (profit over relative deadline) is applied where queries with higher total maximal profit and tight deadlines have higher priorities. Notice that after query miss its deadline, total maximal profit only contains QoD profit instead of the sum of QoS and QoD profit. 2PL-HP is used for concurrency control.

  The problem with UH is its ignorance of the relative importance between QoS and QoD, and blindly establishes all updates to get the best QoD even if it dramatically deteriorates QoS. Next we generated the second baseline algorithm Global Priority to relieve the problem by pushing behind the updates that are not so contributive to the system profit.

- **GP (Global Priority):** GP is a preemptive scheduling scheme with a single-priority queue. The priority scheme for updates is High QoD Profit (HDP) which uses the sum of QoD maximal profit from the queries which have intersected data set with the updates. Query priority scheme is HP (High Profit) which uses the sum of QoS maximal profit and QoD maximal profit. Unfortunately, to be comparable with the updates, it is hard to have deadlines considered in the query priority. In other words, updates on those data that are needed by more queries tend to have higher priorities. This scheme automatically pushes behind the updates which may not be contributive to the data quality of the queries, and promote the updates for those data with high profit attached through QoD functions of queries. One of the disadvantage is that much overhead is incurred from the computation of update priorities because every time a query is submitted or finished, the priority of relative updates (if any) needs to be adjusted. As with UH, 2PL-HP is used for concurrency control.

  GP sounds promising to automatically prioritize updates and queries. In fact, it might give too much control to the effect of QoD profit to updates. And all these above algorithms are purely heuristic-based which only works for specific scenarios.

4. QUTS SCHEDULING

Would the simple heuristics UH and GP handle stochastic arrival of updates and quality specific queries? We did an experimental test and found they both tend to provide a better QoD than QoS. In other words, the scheduling is always biased towards QoD. Obviously an algorithm that intrinsically favor one of the QoS and QoD will not perform good for general quality contracts. Therefore, we propose Query Update Time Share (QUTS) scheduling algorithm, which is a two-level scheme that can dynamically adjust query and update share so as to maximize overall system profit.

In the following, we first analyze the problem in Section 4.1 and then describe the QUTS algorithm in details.

4.1 Problem Analysis

Here we motivate QUTS through a discussion of the weakness of UH and GP. UH and GP are both one-level scheduling algorithms with different preferences: (1) UH always favors updates than queries. If updates keep coming, UH is not able to process any queries. (2) GP improves UH by letting a global priority decides the order of queries and updates. As a sacrifice to have updates with QoD property inflated priorities, queries priorities can only be QoS property inflated and lose the information on deadlines. Whereas deadline information is crucial to success rate when system workload is light. Another concern is if updates are highly inflated by QoD profit and keep coming, GP will still starve the queries, as a result, system can get neither QoS nor QoD profit.

Therefore, we need an algorithm with the following two properties:

- It can dynamically adjust the shares of queries and updates with no starvation;
- It can use different strategies for updates and queries.

Next we explain how QUTS, our proposed algorithm, achieves both goals.

4.2 Algorithm overview

QUTS is a two-level scheduling scheme with two priority queues for queries and updates, respectively. At higher level, it achieves the first goal by dynamic *CPU allocation* to either query queue or update queue according to a profit-guided percentage ρ. At lower level, queries and updates have their own priority queues. More specifically, queries are scheduled via PRD and updates are scheduled via HDP, which are the best choices as we have discussed in Section 3. Multiversion is applied to allow the maximal concurrency and flexible query QoD choices. Table 2 summarizes the scheduling of both baselines schemes and QUTS.

The rest of the section mainly focus on the higher level schedule, which is the central component of the algorithm.

4.3 Priorities Between Updates and Queries

We first introduce the objective function for prioritizing updates and queries. Essentially, the *CPU allocation* decides how we choose from updates and queries to have higher priorities in higher level.

Suppose the total CPU can be allocated is 1, queries share ρ, and updates share the rest, $1-\rho$. The goal is to have the right ρ such that the total profit P is maximized. The total profit P depends on the QoS profit P_S and the QoD profit P_D (i.e., $P = P_S + P_D$). QoS profit depends on 1) the sum of maximum QoS of all queries in the system S and 2) the success query percentage r (i.e., $P_S = S \cdot r$). A leap of faith is that more CPU allocation leads to higher success percentage. For example, if all CPU cycles are given to process queries, QoS profit should be maximized. Thus P_S can be approximated by $S \cdot \rho$. Similarly, QoD profit relies on (1) the sum of maximum QoD of all queries in the system D, (2) the percentage of queries that meets their QoD constraint which depends on both query and update execution. Thus P_D can be approximated by $D \cdot p \cdot (1-p)$. Note that QoD profit has stricter constraint because it demands CPU allocation to both updates and queries.

Finally, the (normalized) total profit can be modeled as

$$P \approx S \cdot \rho + D \cdot (1-\rho) \cdot \rho, \quad 0 \leq \rho \leq 1. \tag{1}$$

4.4 Optimizing ρ

Here we explain how to find the optimal ρ. QUTS tries to find the optimal ρ periodically. The *adaption period* ω decides how often ρ is adjusted. The default value for ω is 1000 milliseconds. Sensitivity test on ω is given in experimental examination in Section 6.3.

Equation 1 is a quadratic function with linear constraints, which usually requires expensive quadratic programming to find the optimal solution. Since there is only one variable ρ in the function, we can simplify it into a gradient descent problem. The optimal solution is:

$$\rho = min(\frac{S}{2 \cdot D} + 0.5,\ 1) \tag{2}$$

Notice that since both S and D are positive, the minimal value of ρ is actually 0.5, which indicates that we should always keep more than 50% of time giving queries higher priorities than updates under this model. A further improvement of QUTS is to use the aging scheme [16] to alleviate the possible random variation which is

High Level	for each adaptation period ω Adjust ρ according to Equation 3, 4 (see Section 4.4)	
	for each atom time period τ (or the current running queue is empty) Generate a random number $\xi \in [0, 1]$ if $\xi < \rho$ query queue is chosen. else update queue is chosen. (see Section 4.5)	
Low Level	query priority queue: PRD (Profit over Relative Deadline) (see Section 3)	update priority queue: HDP (High QoD Profit) (see Section 3)

Table 1: QUTS Two-Level Scheduling with Multiversion Concurrency Control

similar to standard conjugate gradient optimization:

$$\rho_{new} = min(\frac{S_{k-1}}{2 \cdot D_{k-1}} + 0.5,\ 1) \quad (3)$$

$$\rho_k = (1 - \alpha) \cdot \rho_{k-1} + \alpha \cdot \rho_{new} \quad (4)$$

where S_{k-1} is the maximal sum of submitted QoS values during the previous adaptation period. Note that in general α should be a small value[2], but the exact α does not matter much.

4.5 How to establish ρ?

Now we know the guideline is that the probability of query running (or queries have higher probabilities) is ρ and the probability of update running (or updates have higher probabilities) is $1-\rho$ within current ω. A simple way to establish ρ is that at every CPU time we throw a token (a random number from 0 to 1.0) and see where it drops. If it drops in the query territory $[0 \sim \rho)$, we pick the head of the query priority queue to run, and if drops in the update territory $[\rho \sim 1]$, we pick the head of update priority queue to run. If either queue is empty, the head of another queue is picked automatically.

The problem is if we throw it too often, not only we have a lot of overhead making this decision, but also the contention between data reads and writes could range from very bad to prohibitive. On the other hand, we also cannot afford to wait too long that the system may lose a big trunk of QoS profit with time critical queries. We define an *atom time* τ to be the minimal time we keep running queries or updates if both queues are nonempty. Specifically, there are two possible *states*: if queries have higher priorities than updates in τ, we call it *query state* and label it with $\tau \cdot q$, otherwise, we call it *update state* and label it with $\tau \cdot u$. Each time when τ expires, the system throws the token and chooses from queries and updates for the next τ. A state change may happen every τ time, or the picked queue is empty at any instant of time.

4.6 Concurrency Control

Notice that with QUTS, the more often the state changes, potentially the more unfinished transactions there will be, thus the more data contention could be generated. In the following, we will explain the concurrency control scheme applied with QUTS beginning with an example.

[2]We use $\alpha = 0.1$ in our experiment.

Figure 2 shows an example of queries and updates over the same data set, arriving along time. Suppose that they all arrive within a single adaptation period ω_k; q1, q2 and q3 have increasing query priorities respectively; and the two updates are on the same data item. Look at Figure 3, at the beginning of atom time $\tau_1 \cdot q$, q1 starts execution, but later on, is preempted by q2. At the end of $\tau_1 \cdot q$, both q1 and q2 have not committed yet. At the time of $\tau_2 \cdot u$, it happens that updates take over the CPU. As the transaction with highest priority in the system, u gets executed and tries to commit with the writing on the data item that q1 and q2 have not released yet. With any lock or invalidation based concurrency control schemes that avoids priority inversion[3], both q1 and q2 will be restarted since the committing update holds the highest priority among all transactions. Although the restart is good for the QoD values of q1 and q2, it wastes a lot of time (in the example, all the execution in $\tau \cdot q$, and q1's execution in $\tau \cdot q$ since q1 will be preempted by a higher priority query q3) which may diminish the system's QoS profit. In addition, it is not mandatory for queries to use perfectly fresh data if their QoD constraint is loose (e.g., more than one unapplied updates are allowed).

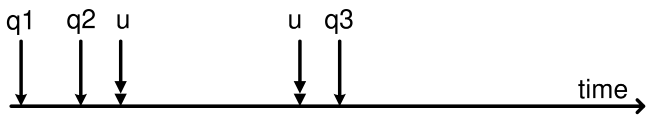


Figure 2: Example Execution: Arrival of Queries and Updates

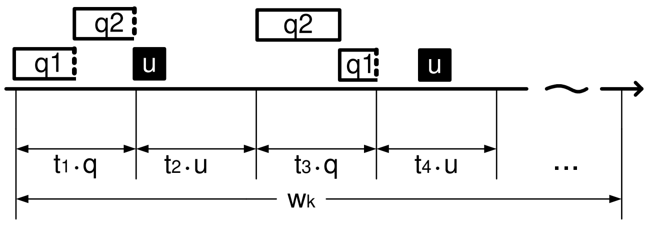


Figure 3: Example Execution with Lock or Invalidation Based Concurrency Control

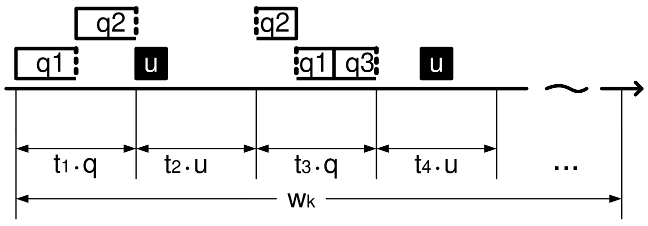


Figure 4: Example Execution with Multiversion Concurrency Control

Under various QoD requirements, we should have a flexible scheme to allow transactions to use a stale (but still acceptable) data item. As a result, we keep a multiversion [4] table for each data item. The table entry consists of the time stamp, read locks, and value. The benefits are two fold: (1)queries and updates can proceed without conflicts and queries are allowed to read the stale data to minimize

[3]Priority inversion happens when higher priority transactions wait on lower priority transactions because of resource contention.

	UH (Update High) Section 3	**GP (Global Priority) Section 3**	**QUTS Section 4**
Query Priorities	PRD (Profit over Relative Deadline)	HP (High Profit)	PRD
Update Priorities	FIFO (First In First Out)	HDP (High QoD Profit)	HDP
Queries vs. Updates	Updates always proceed queries	Naturally comparable	Profit guided ρ
Concurrency Control	2PL-HP (2 Phase Lock - High Priority)	2PL-HP	Multiversion

Table 2: Algorithms

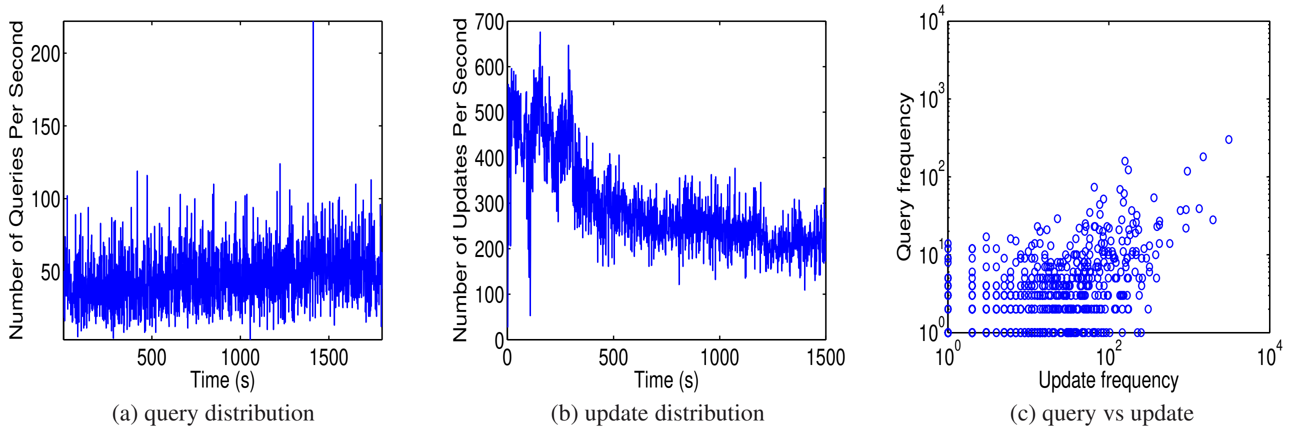


Figure 5: Data characteristics: (a) query distribution is more stable ; (b) while the update has a downward trend over time; (c) the number of updates and queries are roughly positively correlated.

the waste of time on restarts for the sake of its QoS profit. As shown in Figure 4, now q2 can finish without being preempted by q3. (2)multiversion also facilitates the computation of QoD (which depends on the number of unapplied updates of the data) by comparing the time stamp queries are holding and the time stamp of newest intended update. Otherwise, the computation may be difficult when unfinished queries are holding data with different staleness. Nonetheless, multiversion also comes at a cost of maintaining the table. We do the garbage collection on unused versions to keep the table small. The sensitivity test in Section 6.3 shows that with careful choice of atom time τ, the overhead can be negligible.

5. EXPERIMENTAL SETUP

We have managed to acquire access traces from a popular stock market information web site which we will refer to as *Stock.com*[4], which we combined with update traces from the NYSE. These traces enabled us to accurately generate both query and update workloads for our experiments, without having to resort to generating synthetic data.

Our goal is to evaluate how well the proposed methods perform under the entire spectrum of quality contracts, and also gauge the adaptability and sensitivity of our proposed algorithm.

5.1 Query Traces

We used real trading queries from Stock.com for the date of April 24, 2000. Query types include but not limited to: (1) look-up, (2) computing moving average of stock prices, (3) comparison among stocks. They are all read-only. Each query has an arrival time and query stock name. Query execution time ranges from 5 to 28 milliseconds. Our source, Stock.com, is an online trading platform which provides various types of real-time queries and data analysis tools on stock information. The server works 24 hours a day accepting queries from users. However, most activity is occurring during normal trading hours (9:30am - 4:00pm). Thus, we concentrate on queries during those hours for our experiments, when the server is challenged by the floods of stock updates as well as the queries from jittery investors. The results presented in the paper are based on a 30-minute (9:30am-10:00am) interval with over 82,000 queries on more than 4,000 different stocks.

5.2 Update Traces

To match the queries, we extracted the actual trades on all securities listed in NYSE during 9:30am-10:00am on April 24, 2000[5]. The update trace includes the stock ticker symbol, record date, trade time, and trade price per share. Update execution time ranges from 1 to 5 milliseconds. In particular, there are over 496,000 updates on different stocks which share the same indexing scheme with query traces, the stock ticker symbol. Figure 5(a) and (b) show the query and update distributions over time, respectively. Figure 5(c) presents the number of updates and queries over all the stocks (each point corresponds to a stock). Notice that many of the updates occur on stocks with very few queries. We can definitely see that these updates could be reduced (i.e., applied less frequently) to save processing time without diminishing the QoD.

5.3 Experiment Setup: Quality Contracts

As part of the experimental setup, we attach a quality contract to every real query before it is submitted to our system.

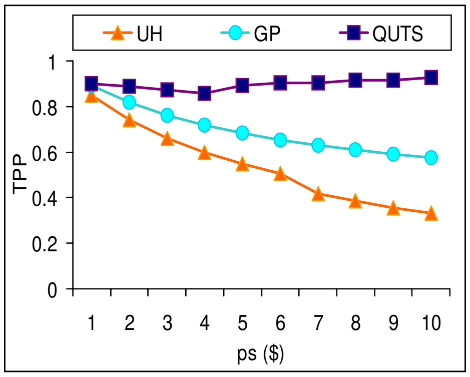
Although QUTS works for generic profit functions, simple functions like step and linear decreasing functions. in these experiments

[4]We cannot disclose the identity of the site because of a confidentiality agreement.

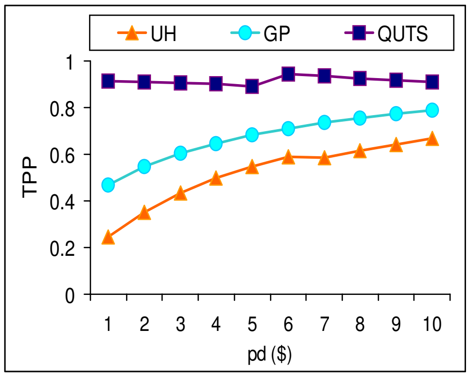
[5]The original trace was acquired from Wharton Research Data Services of the University of Pennsylvania.

Varying	ps	pd	rd	uu
ps	$\{\langle 1:\$10\sim\$19\rangle, \ldots, \langle 10:\$100\sim\$109\rangle\}$	$\langle 5:\$50\sim\$59\rangle$	50ms	0
pd	$\langle 5:\$50\sim\$59\rangle$	$\{\langle 1:\$10\sim\$19\rangle, \ldots, \langle 10:\$100\sim\$109\rangle\}$	50ms	0
rd	$\langle 5:\$50\sim\$59\rangle$	$\langle 5:\$50\sim\$59\rangle$	{20, 30, ..., 100, and 200}	0
uu	$\langle 5:\$50\sim\$59\rangle$	$\langle 5:\$50\sim\$59\rangle$	50ms	{0, 1, 2, 3, 4}

Table 3: Quality Contracts Used in Performance Comparison. Notice that $\langle 5 : \$50 \sim \$59\rangle$ represents that ps or pd is uniformly distributed within \$50 to \$59 and it is labelled by 1 in the plots.



(a) vary QoS max (ps)



(b) vary QoD max (pd)

Figure 6: QUTS performs best for all settings. UH performs worst because too many unnecessary updates execute and starve most of the queries; GP avoids those unnecessary updates, however, due to the global prioritization, starvation on queries or updates can easily occur, which jeopardizes the performance. QUTS works best because the time share scheme successfully avoids the starvation problem of GP.

we only use a "step" profit function for both QoS and QoD. Thus, a quality contract can be defined by four parameters, **ps** (the QoS profit if the query is return before deadline), **rd** (relative deadline which is the difference between deadline and query arrival time), **pd** (the QoD profit if the query is return with data meet the freshness requirement), and **uu** (number of unapplied updates which measures the maximal staleness allowed). The values of these parameters for the quality contracts will be discussed in the presentation of the experiments.

5.4 System Parameter Setup: ω and ρ

System parameters consist of (1) the atom time τ (i.e., the minimal time quantum before QUTS switches the priority between the query queue and update queue), and (2) the adaptation period (i.e., the minimal time before a rescheduling occurs). The default values of τ and ω are 10 and 1000 milliseconds respectively. As we will show in Section 6.3, the exact values of these two parameters do not influence the performance much of our algorithm.

5.5 Performance Metric

Intuitively, we want to maximize the system profit. However, the total actual profit[6] very much depends on the profit value attached to each QC. To bypass this dependency, we normalize the profit and instead use another metric called the *total profit percentage* (TPP),

query execution time	5 ~ 28ms
update execution time	1 ~ 5ms
number of queries	82129
number of updates	496892
number of stocks	4608
default atom time	10ms
default sampling time	1000ms

Table 4: Workload Information and Parameter Setup

which is the actual gained profit by the maximal possible profit, as shown in Equation 5. In the next section, we will illustrate the TPP as function of different parameters to understand the performance of the proposed methods.

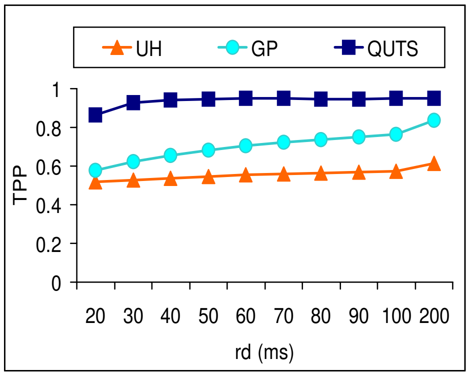
$$TPP = \frac{\sum_{i\in N_q} P_i}{\sum_{i\in N_q} max(P_i)} \qquad (5)$$

where N_q is the number of queries.

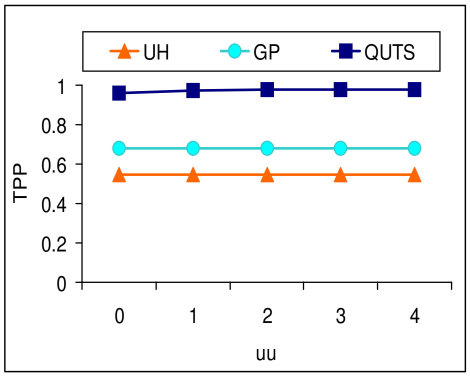
6. EXPERIMENTS

In this section, we show that QUTS performs consistently better than existing methods under the entire spectrum of quality contracts. In Section 6.2, we study the adaptability if QUTS, and show that the proposed algorithm reacts quickly to changing workloads.

[6]When showing the experiment result, we use \$ if we really need to give the actual profit. ms is short for millisecond and s for second.



(a) vary relative deadline (rd)



(b) vary max staleness (uu)

Figure 7: QUTS performs best across all the spectrum of different relative deadline and maximum staleness settings.

Finally, we study the sensitivity of QUTS to its two parameters (adaptation period and atom time) in Section 6.3.

6.1 Performance Comparison

We want to compare the three different algorithms UH, GP and QUTS (Table 2) under various quality contracts. Since there are four parameters in a quality contract, we vary one and fix the others to median values to see how the TPP changes.

6.1.1 *Varying QoS (ps) and QoD (pd)*

Experiment Design: In order to change the ps and pd, we generate quality contracts from ten possible profit ranges: $\langle 1 : \$10 \sim \$19 \rangle$, $\langle 2 : \$20 \sim \$29 \rangle$, ..., $\langle 10 : \$100 \sim \$109 \rangle$. Specifically, $\langle 1 : \$10 \sim \$19 \rangle$ means the QoS or the QoD profit of quality contracts is uniformly distributed within \$10 to \$19 and we use 1 to label it in the plots. Similar notation applies on the other ranges. In this set of experiments, we fix QoS (or QoD) to the median, which is $\langle 5 : \$50 \sim \$59 \rangle$, and varying another. rd is fixed to 50ms. And uu is fixed to 0, which means no stale data is allowed. This QC setup is also summarized in the first two entries of Table 3.

Result: Figure 6(a) shows the TPP as a function of ps. QUTS performs the best among the three which almost reaches the maximum TPP. Note that the performance gap is bigger for smaller pd (when the QoS constraint is more important than the QoD constraint), which is usually of more interests in real applications. Likewise, Figure 6(b) plots TPP vs. pd. Again, QUTS outperforms all others.

6.1.2 *Varying Relative Deadline (rd) and Un-applied Updates(uu)*

Experiment Design: We evaluate the performance of the three algorithms with rd and uu, while the other two parameters in the quality contracts were constant. Please refer to the last two entries in Table 3 for the setup details.

Results: Figure 7(a) shows the TPP as a function of relative deadline. QUTS performs consistently better than the other two. In general, the smaller the relative deadline, the harder the query demands will be. Even for "difficult" queries with small deadline, QUTS achieves over 80% TPP and 2X better than the others.

We also vary the number of maximum allowed un-applied updates (see Figure 7(b)). GP and UH are not affected by that (since they are biased towards updates), while QUTS actually performs slightly better with uu increasing (i.e., the QoD constraint is relaxed). However this is hard to see from the plots since QUTS is too close to the optimal: with uu of 1, TPP is 96.1% and with uu of 4, TPP is 97.6%. 100% means maximum "profit" was received from all queries.

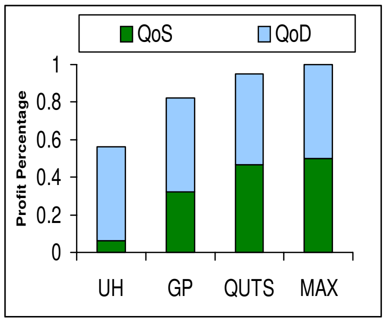
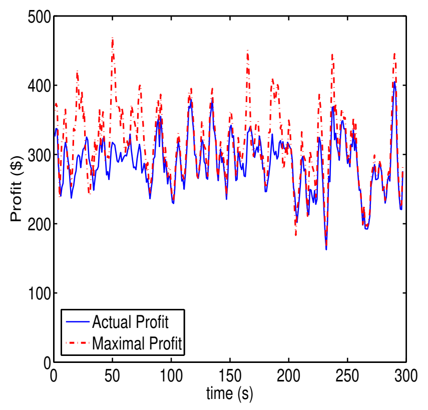


Figure 8: The performance comparison over various quality contracts. QUTS performs much better than the competitors and is very close to the maximal possible profit.

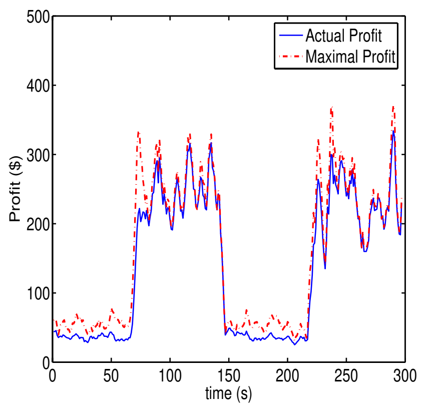
6.1.3 *Mixed QCs*

Experiment Design: Next, we use a hybrid setting where the QoS and QoD functions traverse all ten ranges listed in Table 3 (10×10 choices). This means the total maximal QoS profit and QoD profit submitted to the system is equal despite the randomness. Of course, this is a completely unrealistic scenario, and is meant to simply assess the entire spectrum of choices. In reality, we expect QoD to contribute a much smaller part of the overall quality, with QoS getting the lion's share. In this set of experiments, rd and uu remain constant at 50ms and 0 respectively.

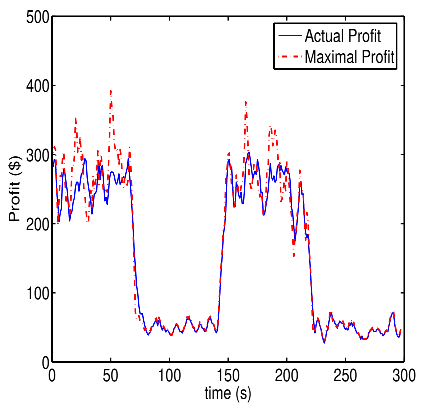
Results: Figure 8 illustrates the overall performance of the three methods followed by the maximal possible profit percentage. The profit is divided into two parts: QoS profit and QoD profit. Notice in the maximal possible profit percentage, QoS and QoD occupy half and half as expected. We can see that although UH and GP perform very well on the QoD profit component, they both lose a significant amount of QoS profit (i.e., they have "unrealized" profit), whereas QUTS performs very close to the optimal.



(a) total profit



(b) QoS profit



(d) QoD profit

Figure 9: The system profit is very close to the maximum.

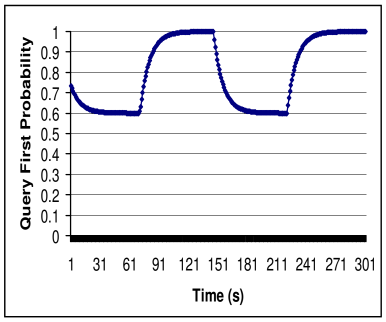


Figure 10: ρ: quickly adapts to the QC changes.

6.2 Adaptability to User Preferences

Having shown that QUTS is much better and robust than UH and GP in the general cases and even when the QoD profit dominates, we now gives some insight into why QUTS can achieve that. In particular, here we illustrate that QUTS can quickly adapt to the changing workloads.

Experiment design: Using real query and update traces, we vary the QoS and QoD in the quality contracts over time. More specifically, we divide the experiment period into 4 intervals evenly. With rd and uu fixed at 50ms and 0, we vary the QoS and QoD ratios as 1:5 or 5:1 (i.e., QoD is 5 times more important than QoS (1:5), or vice versa). Note that we intentionally create the sudden changes on user preferences during small time intervals (75 seconds) in order to test the performance of QUTS in a challenging scenario. The goal is to show (1) how quickly the ρ adapts to the changes, and (2) how good the system profit is over time.

Results: Figure 9(a)-(c) plot the actual and maximal profit of submitted queries over time. As expected, the maximal line in (b) shows the QoS profit trend along time: low-high-low-high, and the maximal line in (c) shows the QoD profit trend along time: high-low-high-low. The maximal line in (a) shows the total maximal profit which is the sum of the profits from (b) and (c). The solid line in all three figures are actual profit "made" by QUTS, which is closely following the maximal line (sometimes higher due to the late completion of former submitted queries.) Note that the figure is plotted after applying a low-pass filter with the moving-window size of 6s, to smoothen the data. The actual incoming queries with different quality contracts are much more bursty. Despite of the burliness, QUTS performs very close to the optimal one.

Figure 10 shows the ρ vs. time. ρ is the current probability of queries having higher priority than updates. According to the solution to optimize the total actual profit given by Equation 2, ρ should be a number between 0.5 and 1. It should increase with the total maximal QoS profit increasing, and decrease with the total maximal QoD profit increasing. In Figure 10, it is very easy to observe four regions where the ρ follows the QoS profit trend: low-high-low-high, and it ranges from around 0.6 to around 1. When ρ equals 1, it does not mean the system will not execute updates at all, it is because with QoS profit extremely higher than QoD profit (5 times higher in this case), the system chooses to run updates only when no queries are waiting. This automatic adaptation behavior agrees with the actual scenarios.

6.3 Sensitivity Test

Experiment Design: In this section, we evaluate the impact of two system parameters, *atom time* and *adaptation period* of QUTS. We use the same trace which was used for the QUTS adaptability test.

The default settings for *atom time* was 10ms, and for the *adaptation period* was 1,000ms. In this section, we give the result of the sensitivity test on these parameters and provide guidelines of how to set them.

ω	100ms	1,000ms	10,000ms	100,000ms
TPP	.81	.92	.89	.88

Table 5: adaptation period

6.3.1 *Sensitivity of* adaptation period ω

The adaptation period determines how often the top-level rescheduling of QUTS occurs. Intuitively, if the adaptation period is too small, there will not be enough queries and updates for QUTS to make statistically sound decisions, which may lower the performance (see Table 5 at *adaptation period* = 100 ms). Likewise, if too large, the rescheduling will happen too infrequent to adapt to the current workload (see Table 5 at *adaptation period* = 100 s).

Nevertheless, the performance varies very small for a wide range of *adaptation periods*.

6.3.2 *Sensitivity of* Atom Time τ

Atom time is the minimal time unit before the system can switch between the query queue and the update queue. The experiments in this subsection aim to answer the following two questions: (1) How does the atom time change affect the profit earning of the system; and (2) How does the atom time change affect the overhead associated with multiversion concurrency control?

Question (1) is answered in Figure 11. We vary the atom time from 1 to 1000 ms. The TPP begins to decrease very slowly as we increase the atom time (see Figure 11). We see that the smaller the atom time, the more flexible the scheduling is. However, extremely small atom times may incur a lot of overhead for multiversion concurrency control. The reason is due to many unfinished queries holding extra versions of data while updates are processed. As we answer Question (2) in Figure 12, we can see, fortunately, that the overhead drops quickly as the atom time increases. With more than 4000 data items, keeping around 100 total extra versions (when τ is 10ms to 50ms) is a negligible overhead. Thus, the rule of thumb is that atom time should be set at least larger than the average query execution time (12ms in the experiment). However, the exact value of atom time does not affect the performance much.

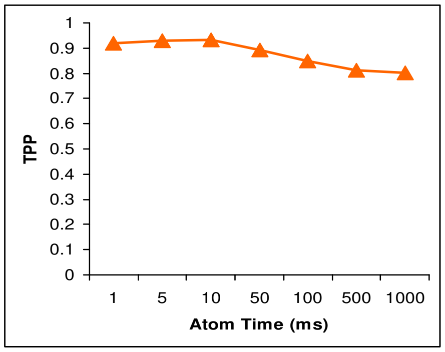


Figure 11: As atom time increases the TPP drops very slowly.

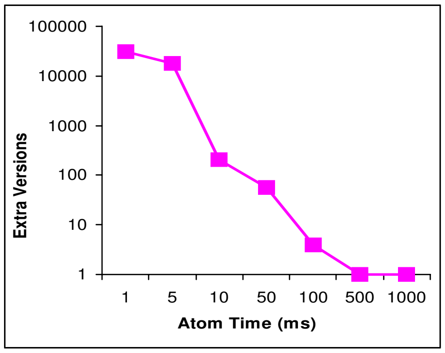


Figure 12: Small atom time may incur huge overhead on the multi-version concurrency control.

7. CONCLUSIONS

In this paper we addressed the problem of scheduling queries and updates in a data-intensive web site. We introduced the concept of Quality Contracts, which is a powerful unifying framework for specifying user preferences over multiple quality metrics. In the presence of QCs, we have proposed a two-level scheduling algorithm, QUTS, that allocates CPU resources in order to maximize the overall system profit (and as such, the overall user-satisfaction). We compared QUTS with a global priority policy and the update-high policy, using real traces collected from a popular stock market information web site. Our extensive experimental study has shown that QUTS outperforms all competitor algorithms under the entire spectrum of Quality Contracts, it has very little sensitivity to its parameters and adapts very well under changing workloads.

8. FUTURE WORK

In the future, we can extend our work from three aspects. First, on concurrency control. We use the existing multiversion concurrency control to combine with our CPU scheduling. Further exploration on a new concurrency control could be the next step. Second, on QoD metrics. We adopt the lag-based metric. Whereas divergence-based and time-based are also widely used QoD metrics and deserve to look into. Third, on combining QoS/QoD functions in quality contracts. We use the *fully-independent* to combine the two functions for the sake of maximal flexibility. In the future, we may investigate more varieties of combinations.

9. REFERENCES

[1] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *VLDB '88: Proceedings of 14th International Conference on Very Large Data Bases*, 1988.

[3] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 245–256, New York, NY, USA, 1995. ACM Press.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] Christof Bornovd, Mehmet Altinel, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. "DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 662, 2003.

[6] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The meaning and role of value in scheduling flexible real-time systems. *J. Syst. Archit.*, 46(4):305–325, 2000.

[7] D. Carney, U. Getintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.

[8] Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh, and Yechiam Yemini. Economic models for allocating resources in computer systems. pages 156–183, 1996.

[9] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*, 2005.

[10] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. Dynamic real-time optimistic concurrency control. In *RTSS '90: Proceedings of the 11th Real-Time Systems Symposium*, 1990.

[11] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Data access scheduling in firm real-time database systems. *Real-Time Syst.*, 4(3):203–241, 1992.

[12] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2(2):117–152, 1993.

[13] Jayant R. Haritsa, Miron Livny, and Michael J. Carey. Earliest deadline scheduling for real-time database systems. In *IEEE Real-Time Systems Symposium*, pages 232–243, 1991.

[14] D. Hong, T. Johnson, and S. Chakravarthy. Real-time transaction scheduling: a cost conscious approach. *SIGMOD Rec.*, 22(2):197–206, 1993.

[15] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, and Donald F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[16] V. Jacobson. "Congestion Avoidance and Control". In *Proc. of ACM SIGCOMM*, pages 314–329, 1988.

[17] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *RTSS*, 1985.

[18] K-D. Kang, S. H. Son, and J. A. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *TKDE*, 16(10):1200–1216, 2004.

[19] A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *VLDB J.*, 13(3):240–255, 2004.

[20] Alexandros Labrinidis and Nick Roussopoulos. "WebView Materialization". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, 2000.

[21] Alexandros Labrinidis and Nick Roussopoulos. "Adaptive WebView Materialization". In *Proc. of the ACM SIGMOD Workshop on the Web and Databases (WebDB'2001)*, Santa Barbara, California, USA, June 2001.

[22] Alexandros Labrinidis and Nick Roussopoulos. "Balancing Performance and Data Freshness in Web Database Servers". In *Proc. of the 29th VLDB Conference*, Berlin, Germany, September 2003.

[23] Paul Larson, Jonathan Goldstein, and Jingren Zhou. "Transparent Mid-Tier Database Caching in SQL Server". In *Proceedings of the Twentieth International Conference on Data Engineering*, pages 177–189, Boston, MA, USA, April 2004.

[24] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. "Middle-tier Database Caching for e-Business". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 662, 2002.

[25] Qiong Luo, Jeffrey F. Naughton, Rajasekar Krishnamurthy, Pei Cao, and Yunrui Li. "Active Query Caching for Database Web Servers". In *Proc. of the ACM SIGMOD Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, USA, May 2000.

[26] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *VLDB*, 1993.

[27] Arif Merchant, Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. Performance analysis of dynamic finite versioning for concurrent transaction and query processing. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 103–114, New York, NY, USA, 1992. ACM Press.

[28] C. Mohan, Hamid Pirahesh, and Raymond Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 124–133, New York, NY, USA, 1992. ACM Press.

[29] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *FOCS*, 1999.

[30] HweeHwa Pang, Michael J. Carey, and Miron Livny. Multiclass query scheduling in real-time database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):533–551, 1995.

[31] Huming Qu, Alexandros Labrinidis, and Daniel Mosse. "UNIT: User-centric Transaction Management in Web-Database Systems". In *Proceedings of the Twenty-Second International Conference on Data Engineering*, Atlanta, GA, USA, April 2006.

[32] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *IEEE*, 82(1):55–67, 1994.

[33] Lui Sha, Ragunathan Rajkumar, and John P. Lehooczky. Concurrency control for distributed real-time databases. *SIGMOD Rec.*, 17(1):82–98, 1988.

[34] Lui Sha, Sang Hyuk Son, Ragunathan Rajkumar, and Chun-Hyon Chang. A real-time locking protocol. *IEEE Trans. Comput.*, 40(7):793–800, 1991.

[35] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.