Preference-Aware Query and Update Scheduling in Web-databases*

Huiming Qu Alexandros Labrinidis

Advanced Data Management Technologies Laboratory
Department of Computer Science
University of Pittsburgh

{huiming, labrinid}@cs.pitt.edu

Abstract

Typical web-database systems receive read-only queries, that generate dynamic web pages as a response, and write-only updates, that keep information up-to-date. Users expect short response times and low staleness. However, it may be extremely hard to apply all updates on time, i.e., keep zero staleness, and also get fast response times, especially in periods of bursty traffic. In this paper, we present the concept of Quality Contracts (QCs) which combines the two incomparable performance metrics: response time or Quality of Service (QoS), and staleness or Quality of Data (QoD). QCs allows individual users to express their preferences for the expected QoS and QoD of their queries by assigning "profit" values. To maximize the total profit from submitted QCs, we propose an adaptive algorithm, called QUTS. QUTS addresses the problem of prioritizing the scheduling of updates over queries using a two-level scheduling scheme that dynamically allocates CPU resources to updates and queries according to user preferences. 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 QCs; QUTS also adapts fast to changing workloads.

1 Introduction

Living without the Web is unimaginable for most of the developed world today. From checking the scores of the World Cup soccer games, to automatically aggregating & monitoring blogs on a specific topic, to accessing personalized weather forecasts, to checking stock quotes and evaluating investment options, the Web has become an indispensable information portal.

All such data-intensive web applications have a few common characteristics. First of all, they all exhibit relatively *high volumes of user requests*, especially during periods of peak load or flash crowds, for example, during the World Cup Final game. Secondly, user requests are typically *read-only queries*, i.e., users do not perform any updates, for example, all updates to stock prices come only from "official" sources (such as NYSE) and not from user-submitted reports. Thirdly, there is also increased possibility for *high volumes of updates*, for example, a tsunami of stock trades because of breaking news regarding a certain company. Finally, updates arrive in the background and are applied to the back-end database (which is driving the web application), keeping the database as fresh as possible.

In all of these applications, end users want *short response times* (i.e. have their queries answered as fast as possible) and *low staleness* (i.e., have the updates also be applied as fast as possible). However, it may be extremely hard to satisfy both needs, especially in periods of high load. In these cases, the order by which queries and updates are executed is expected to play a crucial role in the resulting query response times and staleness.

To illustrate the impact of scheduling on response time and staleness and the ensuing trade-off, we ran a simple experiment with three naive scheduling policies. We considered (a) the non-preemptive *First In First Out (FIFO)* on the combined query and update queue, where queries and updates are executed according to their arrival times; (b) the *FIFO Update High (FIFO-UH)*, which consists of a dual priority queue (one for updates and one for queries) where the FIFO update queue preempts the FIFO query queue; and, (c) the *FIFO Query High (FIFO-QH)*, which consists of a dual priority queue (one for updates and one for queries) where queries preempt updates. All three scheduling policies used the 2PL-HP (Two Phase Locking-High Priority) concurrency control scheme [2]. The plain FIFO could be seen as the most "fair" policy (to both queries and

*Funded in part by NSF ITR Award ANI-0325353.

1-4244-0803-2/07/$20.00 ©2007 IEEE.

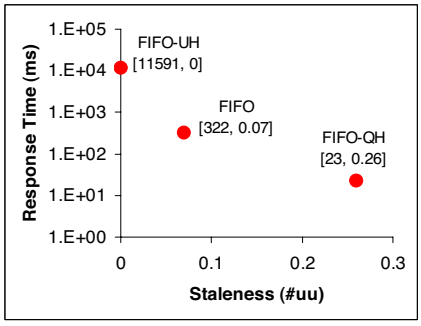


Figure 1. Impact of Scheduling on the Trade-off between Response Time and Staleness.

updates), which, however does not provide any guarantees. The FIFO-UH guarantees zero staleness, since all the updates are applied as soon as possible, and there will not be any pending updates when queries get to execute. Finally, the FIFO-QH is expected to give the best response time for queries among the three policies, since queries always get top priority, running ahead of updates.

Figure 1 shows the average staleness and average response time from running a simulation experiment using a real real stock information web server trace. We measured average staleness as the number of *unapplied updates*, #uu (see Section 2.1 for more details on staleness metrics) and averaged over all queries; the trace consists of about 80,000 user queries with about 490,000 updates arriving during the same time. In Figure 1, the impact of the three scheduling policies on performance is clear: FIFO-UH has the lowest staleness, but the worst response time; FIFO-QH has the lowest response time, but the worst staleness; the plain FIFO policy is somewhere in between the two extremes. It is not clear which of these policies is better, since all three points are dominating points (i.e., for each point, no other point exists with smaller values on both dimensions).

Combining performance metrics In general, if we have two incompatible performance metrics, such as response time and staleness, there are two ways to combine them:

(a) Introduce a constraint on one metric (typically freshness) and optimize on the other metric (typically response time), such as [12, 9]. However, this approach is somewhat limited, as it is "hard-wiring" the metric to optimize and therefore cannot change it according to users' preferences.

(b) Combine them into a single metric and optimize on the aggregate metric, such as [1, 14], where the individual metrics are combined using a set of weights that differentiate multiple metrics. However, this approach does not consider the different preferences of individual users over the importance of response time versus staleness.

Quality Contracts We believe that user preferences on the trade-off between *Quality of Service* (QoS) and *Quality of Data* (QoD) are going to be *different among users*. For example, if it is not possible to have fresh data fast, some users may prefer getting fresh data slightly late (i.e., prefer high QoD), whereas others may prefer getting answers very fast, even if they correspond to slightly stale data (i.e., high prefer QoS). As such, we advocate for a way to extend prior approaches for aggregating QoS and QoD in order to incorporate individual user preferences.

Towards this, 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 [15, 5], which effectively merge all dimensions of Quality into a single, unifying concept. The QC framework allows users to specify their preferences among different quality metrics by assigning the amount of "worth" for the corresponding performance expectation of each query. In this way, users can specify the relative importance of QoS over QoD and 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 the worth to the user, or, the "profit" to the system. With QCs, we can now cast the problem of scheduling queries and updates according to user preferences into the problem of optimizing the total profit for the system.

Scheduling under Quality Contracts Given the QC framework, we propose a novel two-level scheduling (or *meta-scheduling*) scheme for scheduling updates and queries. The basic idea behind the proposed scheme, *QUTS* (short for Query-Update Time-Sharing), is in deciding on the allocation of resources between queries and updates using the expected "profit gain" of the system. In the presence of QCs, QUTS dynamically allocates CPU to updates and queries at the high level, while allowing for maximum flexibility in prioritizing the queries and updates at the low level.

Contributions Our contributions are as follows:

1. We introduce Quality Contracts (QCs), a unifying framework for expressing user preferences for QoS and QoD,
2. We advocate that a single global scheduling policy is not feasible when both QoS and QoD preferences have to be considered, and propose using a two-level scheduler instead,
3. We present QUTS, 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) in the presence of user-specified QCs.

In order to evaluate QUTS, we perform an extensive experimental study, using real data (query traces from a pop-

ular stock market information server and the corresponding update trace from NYSE). We compare QUTS to three baseline algorithms (which it outperforms), evaluate its adaptability in the presence of rapidly changing workloads (very good), and examine the sensitivity of the algorithm with regards to its two parameters (very little).

Structure of paper The paper is organized as follows. In the next section, we describe the system model and introduce the Quality Contracts framework. Section 3 describes the baseline algorithms. We introduce our two-level scheduling algorithm in Section 4 and present the results of our experimental study in Section 5. Finally, we briefly present related work in Section 6 and conclude in Section 7.

2 System Model

We believe that a main-memory database system is the most suitable type for the applications that we are interested in this work, i.e., highly scalable information portals with read-only queries and write-only updates. The memory-residency of such systems eliminates the problems with complex buffer management and I/O scheduling that are crucial on traditional, disk-based database systems. Instead, in our system, CPU scheduling is the primary means of improving performance. Beyond CPU scheduling, concurrency control is also expected to play an important role in determining performance, as is the case with traditional database systems, however, developing new concurrency control schemes for such a system is outside the scope of this paper. In the next paragraphs, we describe the other assumptions and details of our system.

Data Model Our database D consists of N_d data items which are hash-based accessed. Data items are updated aperiodically by external sources, responsible for maintaining the master copy and the whole history of updates on each data item. For example, in the stock information server case, our system corresponds to an information portal, with the entire history of updates stored at the NYSE servers. Furthermore, we assume that data items are independent of each other and database D only needs to keep the most recent update (i.e., data items are independently refreshed).

2.1 Transaction Model

There are two kinds of transactions in our system: read-only user query transactions (or simply *queries*) and write-only update transactions (or simply *updates*).

Queries: Queries in our system can be selection, projection, join, or aggregation queries on multiple data items. Each query has user preferences attached to it in the form of a *quality contract*, which we describe in Section 2.2.

Symbol	Description
rt_{max}	maximum response time
uu_{max}	maximum number of unapplied updates
qos_{max}	maximum QoS profit
qod_{max}	maximum QoD profit
QOS_{max}	$\sum qos_{max}, \forall$ queries
QOD_{max}	$\sum qod_{max}, \forall$ queries
Q_{max}	QOS_{max} + QOD_{max}
$QOS_{max}\%$	QOS_{max} / Q_{max}
$QOD_{max}\%$	QOD_{max} / Q_{max}
QOS	total gained QoS profit for all queries
QOD	total gained QoD profit for all queries
Q	QOS + QOD

Table 1. Symbol Table

Updates: Updates in our system are assumed to be *"blind"*, since the update stream is coming directly from external sources. Each update refreshes one data item. Users are only interested in the most recent value, thus, we do not need to process all updates. The arrival of a new update automatically invalidates any pending update on the same data item. This is done by maintaining an *update register table* where each entry has hash-based access on the data item and an update identifier. Invalidated updates are simply dropped from the system without violating data consistency.

Staleness Metrics: Data will easily get stale if updates are not applied on time. In general, staleness can be measured by the number of unapplied updates (#uu), as well as the time differential (t_d) or value distance (v_d) between the current and the most up-to-date data items. Although both #uu and t_d can be used in our system, we believe #uu is more appropriate for systems that push all updates to replicas as soon as the main replica changes.

Concurrency Control: With the assumption of read-only queries and blind updates, most canonical concurrency problems [16] disappear. For example, the *lost-update* problem, which usually happens for two incremental updates, will not appear with blind updates. However, if a query needs to read a data item repeatedly, we must guarantee that the data item is consistent (unchanged) during the query's execution.

In this paper, we use Two Phase Locking - High Priority (2PL-HP) [2] concurrency control. With 2PL-HP, when there is a read-write conflict, the lower priority transaction will restart and release the lock to the higher priority transaction. On the other hand, for a write-write conflict, the older update will be dropped from the system, since only the most up-to-date update is needed.

2.2 Quality Contracts (QC)

We propose *Quality Contracts (QCs)* as a unifying framework for specifying QoS and QoD preferences. In the general case of Quality Contracts, users specify a number

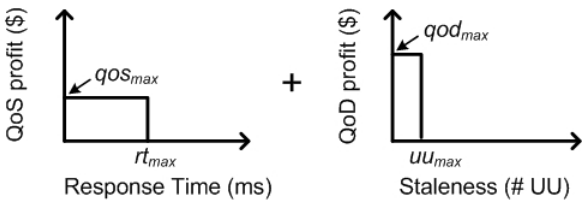


Figure 2. Quality Contracts Example - Step Function (qos_{max} = \$1, rt_{max} = 50ms, qod_{max} = \$2, uu_{max} = 1)

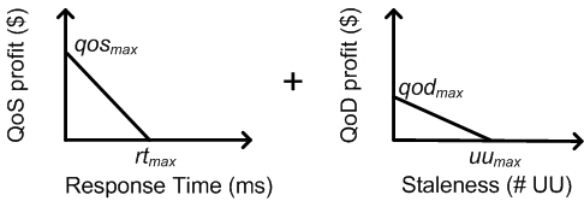


Figure 3. Quality Contracts Example - Linear Function (qos_{max} = \$2, rt_{max} = 50ms, qod_{max} = \$1, uu_{max} = 2)

of non-increasing functions over the QoS/QoD metrics of interest, along with the amount of "worth" to them, for the query to have a certain QoS or QoD when it finishes. In this way, users can specify the relative importance of QoS over QoD as well as the relative importance among their different queries. Although QCs can be defined with any non-increasing functions, we look into two types in this work: (a) step functions and (b) linear functions.

Figure 2 has an example of QCs with step functions (or *step QCs* for short), with only two functions specified: one for QoS, using response time, and one for QoD, using staleness (i.e., the number of unapplied updates). We can uniquely identify such QCs using four parameters:

- qos_{max}, is the **maximum QoS profit** that the server can possibly get from executing this query,
- rt_{max}, is the **maximum response time** (i.e., the relative deadline) that the query can have for the server to get any (QoS) profit from executing this query,
- qod_{max}, is the **maximum QoD profit** that the server can possibly get from executing this query,
- uu_{max}, is the **maximum number of unapplied updates** that the query can have for the server to get any (QoD) profit from executing this query.

Figure 3 has an example of QCs with linear functions (or *linear QCs* for short), with the same setup as in Figure 2. In this paper, we consider both step QCs and linear QCs.

Having described the individual profit functions for QoS and QoD, the question remains on how to combine these into a single overall profit for the system. There are two practical ways to do this for our target environment:

- **QoS-Dependent**: any QoD profit is considered only if the QoS profit is more than zero (i.e., the query commits within the maximum response time), and
- **QoS-Independent**: QoD profit is considered regardless of QoS profit, but the query still has to be completed by a *maximum lifetime* deadline (to avoid keeping queries in the system forever).

In both cases, the overall profit is computed by adding the individual QoS and QoD profits when allowed. In this paper, we consider QoS-Independent QCs.

Usability of Quality Contracts We envision that a system which supports Quality Contracts 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 instantiate. In fact, a simpler scheme is one where the service provider has already identified a certain class of QCs for each type of user (such as a pre-determined cell phone plan) and a user will simply have to turn a "knob" on whether she prefers higher QoS or higher QoD (a local plan with more minutes or a national plan with fewer minutes under the same budget). In this way, using QCs service providers can better provision their systems, provide different classes of service, and allow end users to specify their preferences with minimal effort.

3 Baseline Algorithms

As we have shown with the three naive algorithms in the introduction, there are two basic ways [3] to schedule updates and queries: single priority queue (e.g., FIFO) and dual priority queue (e.g., FIFO-UH and FIFO-QH). Let's look at these two categories respectively.

3.1 Single Priority Queue

With a single priority queue, the simplest scheduling policy is FIFO. However, since each user query has a preference on QoS and QoD (with corresponding profit functions) and our optimization goal is to maximize the system profit, the question is whether we can do better than FIFO.

Query Priority: QoS functions in quality contracts are similar to utility functions, or soft/firm deadlines and rewards, which have been studied extensively in real time systems [4, 7, 13, 6]. The idea is to consider both dimensions (time constraints and profit) of the QoS functions. However, deadline and profit pressure are only helpful to prioritize queries and maximize the profit from QoS functions.

Update Priority: Updates determine data freshness. Thus they have an indirect impact on query freshness and

therefore on QoD profit. Suppose we let updates inherit the QoD functions associated with the corresponding queries, then the update priority should consider both dimensions (staleness constraints and profit) of the QoD functions.

Combining Query and Update Priority: Now the problem is that the query priority (based on time and profit) is not really comparable to the update priority (based on staleness and profit) because staleness (measured in number of unapplied updates[1]) is not comparable to response time. On the other hand, if we only consider the profit, which is commonly expressed across all metrics, we lose the time information for queries and the staleness information for updates. Thus, it is impossible to have a global priority scheme that considers all the information provided by the QCs. In other words, *query and update priorities are not directly comparable under the QC framework, or any other framework that combines user preferences on QoS and QoD*. Our baseline algorithm for a single priority remains FIFO:

- **First In First Out (FIFO)** orders transactions according to their arrival time. FIFO may perform poorly on QoS profit, because of its ignorance of the QC information. However, because of the random arrival and interleaving of queries and updates, the QoD profit under FIFO should be fair.

3.2 Dual Priority Queue

The benefit of a dual priority queue is that updates and queries can have their own priority scheme and we only need to compare the query queue and update queue instead of individual queries and updates. We present two baseline algorithms with dual priority queue:

- **Update High (UH)** UH forms a preemptive dual priority queue, where updates have higher priority than queries. For queries, we use Value over Relative Deadline (VRD) [6] which, with our QC framework, equals to the ratio of the query's total maximal profit over its maximal response time, or $\frac{qos_{max}+qod_{max}}{rt_{max}}$. For updates, we adopt FIFO for its simplicity, because the priority of updates can hardly affect the queries' performance with separate priority queues. UH guarantees zero data staleness, but if a surge of updates arrives, it will push behind all queries without distinction.
- **Query High (QH)** QH forms a preemptive dual priority queue, with queries having higher priorities than updates. Similar to UH, VRD is used for queries and FIFO is used for updates. QH is in favor of query execution, thus is expected to have better QoS performance than UH. Yet, its delayed execution of updates may accumulate too many unapplied updates for data items, and thus hurt query staleness.

The deficiency of UH and QH is that they have fixed priorities between queries and updates, which leads them to either always favor QoS or always favor QoD. However, not all users will have the same preferences, which may also change over time, thus making these two policies unsuitable for the general case.

4 QUTS Scheduling

The discussion in the last section reveals that it is impossible to have a single priority queue for both queries and updates because the QoS and QoD profit functions (i.e., the metrics on time and staleness) are fundamentally incomparable. On the other hand, the baseline algorithms with dual priority queues focus exclusively on either QoS or QoD, because of the fixed priority between update queue and query queue. Thus, we need a policy with a dual priority queue that adapts the priority between the two queues according to user preferences on QoS and QoD.

Specifically, we propose the *Query Update Time Sharing (QUTS)* scheduling algorithm. QUTS is a two-level scheme that at the high level, dynamically adjusts the query and update share of CPU, so as to maximize overall system profit, and at the low level, allows queries and updates to have their own priority queues. This means that QUTS can utilize any priority scheme that considers both time and profit constraints for queries and staleness and profit constraints for updates. Similarly to the baseline algorithms, queries are scheduled via VRD and updates are scheduled via FIFO.

The rest of the section mainly focuses on the high level scheduling, which is the central component of the algorithm. Essentially, we want to answer the following two questions:

- Theoretically, how much CPU allocation should we assign to queries to optimize the system profit from QCs?
- Practically, how to establish the CPU allocation?

4.1 Theoretical CPU Allocation

In order to see when (or for how long) we should have the priority of queries higher than that of updates, we must find out the relationship between CPU allocation and the total profit that the system can gain, and furthermore, to determine the CPU allocation that maximizes the total profit.

Query CPU allocation ρ and total profit: Suppose the total CPU to be allocated is 1, queries share ρ ($0 \leq \rho \leq 1$) of the CPU, and updates share the rest, $1 - \rho$. The goal is to have the right ρ such that the total profit Q is maximized. Let's look at the QoS and QoD profits respectively.

[1]Even if we use time since last update to measure data staleness, this time value will still not be comparable with query response time.

Total QoS profit, depends on (1) the maximum QoS profit for each query, and (2) the response time for each query, r. With linear QCs, higher query CPU allocation leads to better response time, thus higher QoS profit. With step QCs, more query CPU allocation leads to higher chances to finish within the maximum response time, thus more QoS profit as well. In other words, the higher the ρ, the more profit the system can gain from QOS_{max}. Thus, the total gained QoS profit QOS can be approximated as:

$$QOS = QOS_{max} \cdot \rho \quad (1)$$

Total QoD profit, similarly, relies on (1) the maximum QoD of each query, and (2) the staleness for each query. In general, higher update CPU allocation leads to lower data staleness, but queries also have to finish in time (before the *maximum query lifetime*) for the system to get any QoD profit. In other words, the possible QoD profit gains require a fair amount of update CPU share as well as the query CPU share. Thus, the total gained QoD profit QOD can be approximated as:

$$QOD = QOD_{max} \cdot \rho \cdot (1 - \rho) \quad (2)$$

Total profit is, thus, modeled as:

$$Q \approx QOS_{max} \cdot \rho + QOD_{max} \cdot (1-\rho) \cdot \rho, \quad 0 \leq \rho \leq 1. \quad (3)$$

The optimal ρ to maximize Q can be computed by solving the above quadratic function with linear constraints, which usually requires expensive quadratic programming to find the optimal solution. However, since there is only one variable ρ in the function, we can simplify it into a gradient descent problem. The optimal solution is:

$$\rho = min(\frac{QOS_{max}}{2 \cdot QOD_{max}} + 0.5,\ 1) \quad (4)$$

Notice that since both QOS_{max} and QOD_{max} 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 priority than updates under this model.

Adaptively adjusting ρ: With the workload and user preferences changing over time, ρ should be adjusted adaptively. QUTS tries to find the optimal ρ periodically. The *adaptation period* ω decides how often ρ is adjusted. The default value for ω is 1000 milliseconds. At the beginning of each ω, ρ is computed and smoothed with an aging scheme [8] which is similar to standard conjugate gradient optimization:

$$\rho_{new} = min(\frac{QOS_{max_{k-1}}}{2 \cdot QOD_{max_{k-1}}} + 0.5,\ 1) \quad (5)$$

$$\rho_k = (1 - \alpha) \cdot \rho_{k-1} + \alpha \cdot \rho_{new} \quad (6)$$

High Level	for each adaptation period ω **(Section 4.1)** Adjust ρ according to Equation 5, 6 for each atom time period τ **(Section 4.2)** (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.	
Low Level	query priority queue: VRD **(Section 3.2)**	update priority queue: FIFO

Table 2. Pseudo-code for the QUTS (Query-Update Time-Sharing) two-level Scheduling Algorithm

where $Q_{max_{k-1}}$ is the maximal sum of submitted QoS values during the previous adaptation period. Those QCs that change over time (e.g., linear QCs) will incur more overhead when $Q_{max_{k-1}}$ is recomputed. In general, α should be a small value, but the exact α does not matter much.

4.2 Implementation of CPU allocation ρ

Based on the previous discussion, the probability of a query running (or the query queue proceeding the update queue) should be ρ within the current ω. We pick for execution the head of the query queue with probability ρ and the head of the update queue with probability $1 - \rho$.

The problem is how often we select the next queue to execute. We can choose from as small a duration as one CPU cycle, or as large as ω. We do no want it to be too often, not only to avoid the overhead, but also because the data contention can be potentially increased with more and more unfinished transactions. On the other hand, we also cannot afford to wait too long, especially for the queries with stringent time constraints and high profit.

We define *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 priority 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 chooses from queries and updates for the next τ, as the example shown in Figure 4. A state change may happen every τ time, or if the picked queue is empty at any instant of time.

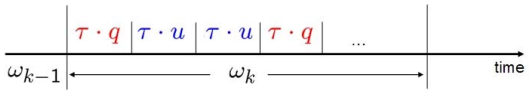
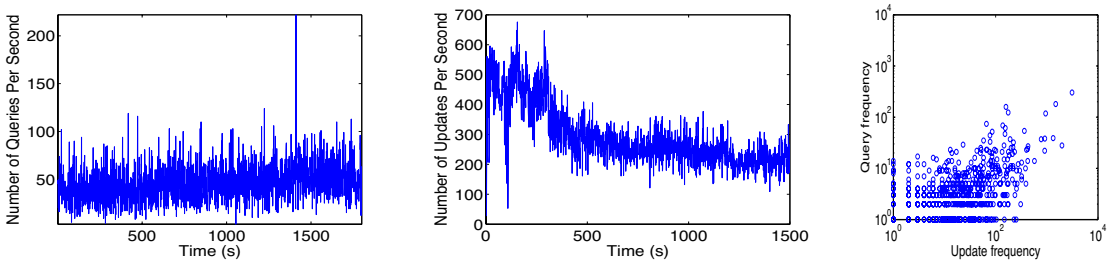


Figure 4. QUTS Scheduling

We give the pseudo-code for the QUTS two-level scheduling algorithm in Table 2.



(a) query distribution (b) update distribution (c) query vs update

Figure 5. Trace characteristics: (a) query distribution has small changes over time; (b) update distribution has downward trend over time; (c) stocks (points) are concentrated below the diagonal (i.e., most stocks have more updates than queries).

5 Experiments

We have acquired access traces from a popular stock market information web site which we will refer to as *Stock.com*[2]. We combined these access traces with the NYSE (New York Stock Exchange) update traces at the same time period, which enabled us to accurately generate both query and update workloads for our experiments, without having to resort to generating synthetic data.

Query Traces We used real trading queries from Stock.com for the date of April 24, 2000. Query types include, but are not limited to: (1) look-up, (2) computing moving average of stock prices, and (3) comparison among stocks. All queries are read-only. Each query has an arrival time and a stock symbol set to be accessed. Query execution time (CPU occupation) ranges from 5 to 9 milliseconds. Our source, Stock.com, is an online trading platform which provides various types of real-time queries and data analysis tools for stocks. The server is online 24×7, 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 flood of stock updates as well as the avalanche of 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. These results are representative of other intervals during the day.

Update Traces To match our query workload, we extracted the actual trades on all securities listed on the NYSE during 9:30am-10:00am on April 24, 2000[3]. The update trace includes the stock ticker symbol, record date, trade time, and trade price per share. Update execution times range 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. The statistics are collected on each second. On average, there are more updates than queries. Yet, the intensity of the updates reduces during the second half of the trace. 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 the stocks with very few queries (most points are below the diagonal in (c)). These updates could be reduced (or postponed) to save processing time without diminishing much of QoD, especially when QCs show that QoS is more important to users.

System Parameters: τ and ω We have two parameters in our system: (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.

query execution time	$5 \sim 9$ms	# queries	82129
update execution time	$1 \sim 5$ms	# updates	496892
default atom time (τ)	10ms	# stocks	4608
default adaptation period (ω)	1000ms		

Table 3. Workload Information and System Parameters

5.1 Performance Comparison

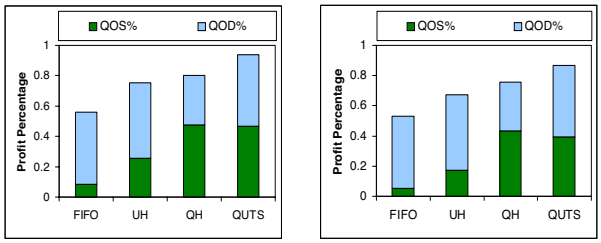
We compare QUTS with three baseline algorithms FIFO, UH, and QH under various quality contracts (QCs). We test QCs with step functions and linear functions, and for each type of functions, we vary one of the four characteristic parameters of QCs (qos_{max}, qod_{max}, rt_{max}, or uu_{max}) each time. Next, we present results from evaluating QUTS' performance under step QCs and linear QCs (Section 5.1.1), and also with changing qos_{max} and qod_{max} (Section 5.1.2).

5.1.1 Step Functions vs. Linear Functions for QCs

Experiment Design (Figure 6): For both step QCs and linear QCs we use the same four-parameter setup for QCs: qos_{max} and qod_{max} are randomly chosen from $\$10 \sim \$50$, thus $QOS_{max}\% = QOD_{max}\% = 0.5$, i.e., user preferences are equally distributed over QoS and QoD, rt_{max} is randomly chosen from 50ms $\sim$ 100ms, and uu_{max} is set to 1 (i.e., QoD profit is gained only when no update is missed).

[2]We cannot disclose the true identity due to a confidentiality agreement.

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



(a) Step function (b) Linear function

Figure 6. Profit Percentage of four scheduling algorithms with step and linear QC functions. QUTS takes the "best" profit dimension of the other policies: high QoS from QH and high QoD from UH

Results (Figure 6): The performance with step functions is shown in Figure 6(a) and with linear functions in Figure 6(b). On each plot, we show the gained QoS and QoD profit percentage. The total height of each bar is the total profit percentage gained (i.e., the sum of QoS profit percentage and QoD profit percentage). In Figure 6, the maximal QoS percentage is 0.5, since QoS and QoD share the same amount of profit in this setup.

Looking at the performance with step QCs in (Figure 6a), we see that QUTS gains the highest profit percentage with both QoS and QoD profit percentage close to the maximal. As expected, QH has low QoD profit percentage, since it favors queries; UH has low QoS profit percentage, since it favors updates; FIFO has the lowest total profit percentage, with the worst QoS profit percentage among the four algorithms. Essentially QUTS is able to take the "best" profit dimension of the other policies: high QoS from QH and high QoD from UH.

Performance with linear QCs in (b) shows similar trends with step QCs despite a slightly lower total profit percentage. This is due to the fact that the maximal QoS profit in the linear function is actually unrealistic (no transaction can be returned in literally zero time), whereas there is no profit degradation with step functions. Overall, the performance difference among the compared algorithms is similar between step functions and linear functions. Due to limited space, we only show the results with step functions in the rest of the experimental results.

5.1.2 Performance under different QCs

Experiment Design (Figure 7, 8): This set of experiments is designed to show the performance of QUTS under QCs, by changing the qos_{max} and qod_{max}. We prepared nine different QC sets, which we list in Figure 4.

Results (Figure 7, 8): Figure 7 shows the profit percentage from the FIFO policy over the different QC setups. Figure 8 shows the profit percentages for the UH, QH, and QUTS policies. The actual QoS and QoD profit percentages are shown by bars in each plot, whereas the diagonal line corresponds to the maximum QoS profit percentage ($QOS_{max}\%$). We see in Figure 7 that FIFO gains the worst QoS profit percentage because it ignores the time constraints that users specified. Thus, although FIFO has a decent QoD profit, it still cannot avoid to have the worst total profit percentage. In Figure 8a, the Update-High policy gains almost the maximal QoD profit percentage (the light colored bars), but performs poorly on QoS. In Figure 8b, the Query-High policy gains almost the maximal QoS profit percentage (the dark colored bars), but performs relative poorly on QoD. In Figure 8c, QUTS gains almost the maximal QoS and QoD profit percentage with all QC sets. In fact, QUTS performs up to 101.3% better than UH and up to 40.1% better than QH, consistently performing better or as good as the best of the two policies. Clearly, the main weakness of both UH and QH is their fixed preferences over queries (QoS) or updates (QoD) which are detrimental in a mixed-preferences workload.

$QOD_{max}\%$	0.1	0.2	...	0.9
$QOS_{max}\%$	0.9	0.8	...	0.1
qod_{max}	\$10 ∼ \$19	\$20 ∼ \$29	...	\$90 ∼ \$99
qos_{max}	\$90 ∼ \$99	\$80 ∼ \$89	...	\$10 ∼ \$19
rt_{max}	50ms ∼ 100ms			
uu_{max}	1			

Table 4. Quality Contract Setup for Figure 7, 8

5.2 Adaptability to User Preferences

Experiment Design (Figure 9): We use the same traces, but instead of a static QC setup, we vary qos_{max} & qod_{max} over time. Specifically, we divide the experiment period evenly into 4 intervals, and have rt_{max} = 50ms ∼ 100ms, uu_{max} = 1, and vary the qos_{max} to qod_{max} ratio from 1:5 to 5:1 (i.e., qos_{max} = 5× qod_{max} or vice versa). We intentionally create 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 how quickly QUTS can react to the changes and adjust ρ accordingly.

Results (Figure 9): We plot the actual and maximal profit of submitted queries over time in Figure 9a-c. As expected, the maximal line in (b) shows the QoS profit trend along

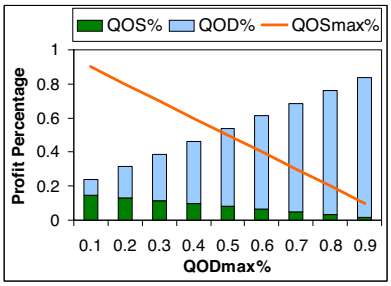
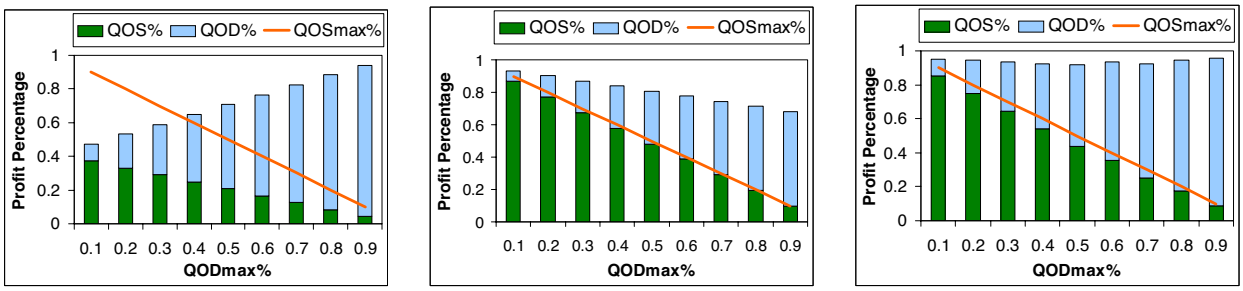
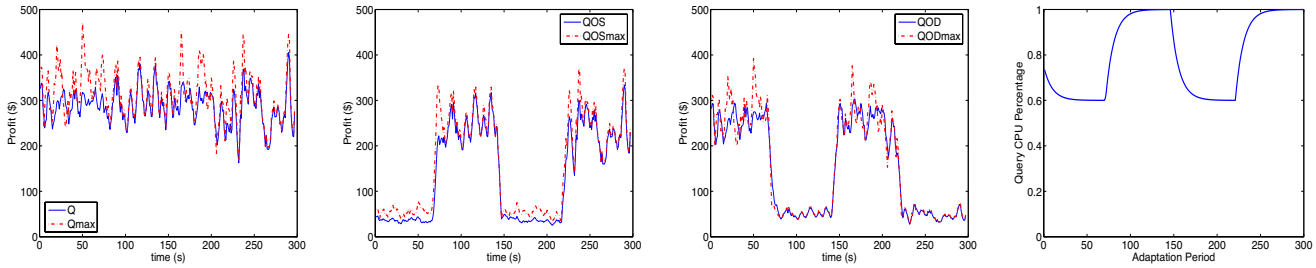


Figure 7. Profit percentage with various QCs for FIFO.



(a) Update High (UH) (b) Query High (QH) (c) QUTS

Figure 8. Profit percentages for QCs as in Figure 7. QUTS performs up to 101% better than UH and up to 40% better than QH.



(a) total profit (b) QoS profit (c) QoD profit (d) ρ over time

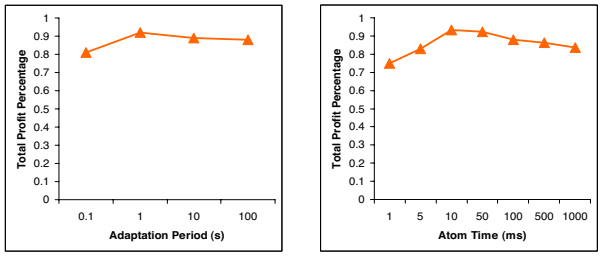
Figure 9. (a-c) QUTS performs very close to the ideal maximum, under changing QCs; (d) ρ quickly adapts to changing QCs.

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 is actual profit "gained" by QUTS, which is closely following the maximal line (sometimes higher due to the late completion of previously submitted queries). Note that the figure is plotted after applying a filter with the moving-window size of 5 seconds, to smoothen the data. Overall, QUTS performs very close to the ideal case.

Figure 9d shows the ρ over time. ρ is the system's probability of queries having higher priority than updates. According to the solution that optimizes the total actual profit given by Figure 4, ρ should be a number between 0.5 and 1, and should "track" the total maximal QoS profit. In Figure 9d, it is very easy to observe four regions where the ρ follows the QoS profit trend: low-high-low-high; it ranges from around 0.6 to around 1. With $\rho = 1$, updates are still executing, but only when no queries are waiting. This automatic adaptation behavior agrees with the actual scenarios.

5.3 Sensitivity of QUTS to ω and τ

In this section, we evaluate the impact of two system parameters of QUTS: *atom time* τ and *adaptation period* ω. We use the same setup with that of Section 5.2.



(a) Adaptation Time (ω) (b) Atom Time (τ)

Figure 10. Sensitivity of QUTS over ω and τ

Sensitivity of *Adaptation Period* ω (Figure 10a) The adaptation period determines how often the top-level rescheduling of QUTS occurs. If the adaptation period is too small, QUTS may make wrong decisions, if it is too large, the performance may suffer. However, as we see in Figure 10a the overall performance varies very little for a wide range of *adaptation periods*.

Sensitivity of *Atom Time* τ (Figure 10b) Atom time is the minimal time unit before the system can switch between the query queue and the update queue. Small τ values can potentially lead to more conflicts; bigger τ values may also hurt performance. In this set of experiments, we fix the adaptation time to 1000ms and vary τ from 1ms to 1000ms.

Figure 10(b) shows the total profit percentage gained by QUTS with different τ. The best performance is gained at around 10ms, which is close to the maximum execution time of our queries (5ms $\sim$ 9ms). As such, a simple rule of thumb for setting τ is to set it above the maximum execution time of most of the queries in the system.

6 Related Work

We described the QC framework in more detail in [10]. To our knowledge, there is no work beside ours that combines individual users' preferences for both QoS and QoD in database systems. However, there is related work in capturing user preferences and query/update scheduling.

User Preferences There exist previous approaches that combined user preferences on different metrics into a single metric. For example, Borealis [1] used a utility-function-based QoS model, assigning different weights on the multiple metrics to form an aggregate metric. Our previous work on user-centric transaction management [14] used a single aggregate metric, *User Satisfaction Metric (USM)*, to guide admission control for incoming query and update transactions. However, in both works, individual user preferences over multiple metrics are not differentiated, rather, a global weighting scheme was used.

Query Scheduling There is a lot of research in real-time database systems that deals with transaction scheduling in the presence of deadlines or rewards [4, 7, 13, 6], which is similar to our query prioritization. All these schemes can be used for our lower level query queue scheduling and thus are orthogonal to this work.

Update Scheduling [3] discussed basic techniques (e.g., query first or update first) on how to apply updates in real-time database systems, but it has not proposed a unified metric to optimize. Our baseline algorithms are based on similar schemes, whose fundamental flaw is to have fixed priorities between queries and updates. In [11], we also proposed a scheduling scheme for updates, however we did not consider scheduling of queries.

We applied the QC framework for replica selection in distributed systems in [17]. There is a lot of related work on data replication; it is beyond the scope of this paper.

7 Conclusions

In this work we addressed the problem of scheduling queries and updates in data-intensive web sites. We presented the concept of Quality Contracts (QCs), 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 to maximize the overall system profit (and, as such, the overall user-satisfaction). We compared QUTS to three baseline algorithms, 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 QCs, adapts very well under changing workloads, and has very little sensitivity to its parameters.

References

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

[2] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, 1992.

[3] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *SIGMOD*, 1995.

[4] A. Burns, D. Prasad, A. Bondavalli, et al. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46(4), 2000.

[5] D. F. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. *Economic models for allocating resources in computer systems*. World Scientific Publishing, 1996.

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

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

[8] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM*, Aug. 1988.

[9] 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.

[10] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *Proc. of Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)*, 2006.

[11] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *VLDB*, 2001.

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

[13] H. Pang, M. J. Carey, and M. Livny. Multiclass query scheduling in real-time database systems. *TKDE*, 7(4), 1995.

[14] H. Qu, A. Labrinidis, and D. Mosse. UNIT: User-centric Transaction Management in Web-Database Systems. In *ICDE*, 2006.

[15] M. Stonebraker, P. M. Aoki, et al. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5(1), 1996.

[16] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[17] J. Xu and A. Labrinidis. Replication-aware query processing in large-scale distributed information systems. In *WebDB*, 2006.