Guiding Personal Choices in a Quality Contracts Driven Query Economy[*]

Huiming Qu
IBM Watson Research Center
hqu@us.ibm.com

Jie Xu
University of Pittsburgh
xujie@cs.pitt.edu

Alexandros Labrinidis
University of Pittsburgh
labrinid@cs.pitt.edu

ABSTRACT

The emergence of Web 2.0 has brought upon a plethora of database-driven web applications and services where both Quality of Service (QoS) and Quality of Data (QoD) are of paramount importance to end users. In our previous work, we have proposed Quality Contracts, a comprehensive framework for specifying multiple dimensions of QoS/QoD; we have also developed algorithms to maximize overall system performance under Quality Contracts. In this work, we turn our attention to the user side of the equation, on how to choose and adapt Quality Contracts to better serve users' needs in the presence of other users, who are competing for the same resources, in a virtual "economy" of Quality Contracts at the server. Towards this, we propose the Adaptive Quality Contract (AQC) scheme to maximize the success ratio of user queries. AQC switches between its Overbid (aggressive) mode and Deposit (conservative) mode, to allow users to survive through economic downturns and upturns. Extensive experiments with real traces show that our proposed scheme outperforms other competing schemes, under a variety of environments and a spectrum of workloads.

1. INTRODUCTION

How many times did you have to wait for your query to finish executing when visiting a travel reservation web site like Orbitz and Expedia? After getting the query results, how many times was the quoted price proved to be inaccurate when you clicked "buy this ticket"? This is just one example of a web-database system that illustrates the trade-off between Quality of Service (QoS) and Quality of Data (QoD). Clearly, some users would prefer fast response time, while tolerating slightly stale results (e.g., when they just want to find out about flight schedules). However, other users would instead prefer to get the most accurate query results, even if the response time was a bit higher (e.g., when they are ready to purchase a ticket). There are a number of issues that need to be addressed to implement a web-database system that is "receptive" to user preferences on QoS and QoD. We enumerate these next.

[*]Work supported in part by NSF Career award IIS-0746696 and NSF ITR award ANI-0325353.

Q1: How to describe user preferences? First and foremost, there needs to be a way for users to specify their preferences on QoS and QoD. One simple way would be to effectively assign users to equivalence classes (i.e., prefers QoS over QoD, or prefers QoD over QoS) and allow users to select which class they belong to. In our previous work, we have proposed (and advocated using) a more sophisticated framework, called *Quality Contracts* (QCs) [6] which is based on the micro-economic paradigm. The QC framework allows users to specify their preferences across a variety of quality dimensions. Similar proposals exist for other domains, such as the utility functions in real-time systems [15] and the service level agreements (SLAs) in Grid computing [3][1].

Q2: How are user preferences "implemented" to influence system decisions? Given a framework for users to specify their preferences over different quality dimensions, it is crucial to have a way to influence resource allocation decisions to maximize user satisfaction (i.e., compliance to user preferences). Towards this, we have developed admission control policies [13] and query & update scheduling algorithms [12] that maximize the overall system profit (to be gained by the server from satisfying QCs) and thus maximize the overall user satisfaction. The proposed algorithms are especially useful during periods of high server load, since they provide graceful service degradation.

In this work, we address another important problem that materializes after satisfactory solutions for questions Q1 and Q2 have been provided.

Q3: How should users adapt their Quality Contracts in the presence of competition? User preferences, if expressed through the Quality Contracts framework, include a constraint component (e.g., maximum acceptable data staleness) and a "worth" component (i.e., virtual money) for every quality dimension of interest to the user. In such an environment, always truthfully exposing the "worth" of the queries will not allow users to react to high competition (i.e., by "paying" a bit more than expected) nor to take advantage of reduced competition (i.e., by "paying" less than expected). In general, we consider the Quality Contracts submitted by the different users (along with their queries) as a *competitive economy*. As such, it is crucial for users to be able to adapt these QCs over time (while trying to achieve query quality that meets their preferences). In this paper, we propose user strategies to adapt QCs over time, during economic downturns (i.e., when the competition is less) and also during economic upturns (i.e., when the competition is higher).

Contributions The main contributions of this paper are:

- Given an environment where user preferences over different quality dimensions are expressed using Quality Contracts (QCs)

[1]We refer the reader to [6] for a detailed description of the QC framework and comparison to other approaches.

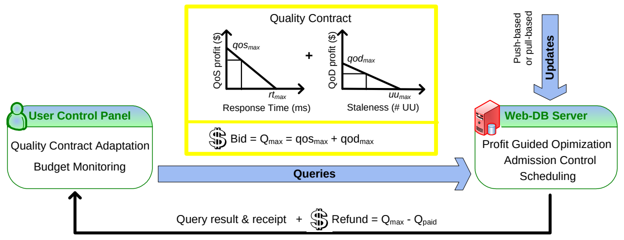


Figure 1: System Architecture

and are attached to queries, we look at the user viewpoint and, in particular, the framework for adapting QCs over time and the connection between a user's true preferences and his or her "exposed" QCs.

- We propose the *Adaptive Quality Contract (AQC)* strategy, which monitors a user's queries and the server's responses and automatically adapts the QCs of subsequent user-submitted queries. AQC[2] switches between two modes: *Overbid mode*, used at times of fierce competition among users; and *Deposit mode*, used at times of little competition.

We have demonstrated a QC-enabled web-database system during SIGMOD 2007 [14]. Our demo illustrated both the server view (for which the technical details were published in [12]) and a preliminary version of the user view, which is presented in this paper. In addition to introducing the AQC strategy (not in [14]), this paper explains its mathematical foundations, and presents a detailed experimental study using real traces.

2. SYSTEM ARCHITECTURE

We assume a web-database server architecture like the one in Figure 1. The system consists of two parts: the user module and the web-database server. Before describing these two parts, we discuss the basic concepts behind the QC economy.

2.1 The Quality Contract (QC) Economy

Economic mechanisms can be broadly classified into two types: commodity markets and auction markets [2]. Previous work has attempted to solve the system resource allocation problem under both paradigms. Under the commodity markets paradigm, commodities are exchanged in standardized contracts. Servers (acting as sellers) need to valuate their resources and assign prices for each unit. Users (acting as buyers) then decide from whom they buy the service to fulfill their queries. The shortcoming of commodity markets is the complexity and high cost for a server to valuate its resources, especially when the server workload fluctuates rapidly over time. To avoid this overhead at the server and make the valuation more precise, many systems follow the auction markets paradigm [2, 9, 16, 8, 19], where users need to give a price and bid on the resources or services provided by the server. Obviously, the uncertainty and burden of valuation never disappear; they are simply shifted to the user side. In this work, we adopt the auction markets paradigm. However, as we will elaborate in Section 3, we propose an adaptive bidding mechanism so that neither servers nor users have to worry about exact valuation.

In our system, users are allocated virtual money, which they spend in order to execute their queries according to their preferences; user preferences are described via QCs attached to each submitted query. Servers, on the other hand, execute users' queries and get virtual money in return for their service.

[2]AQC is pronounced AQuaC, which sounds like AFLAC; however, we do not have a fancy mascot.

The virtual money is "paid" upon submission of a query to the server as part of the bidding (i.e., Q_{max}); any refund is given back along with the query results (i.e., $Q_{max} - Q_{paid}$). In our work, we follow a hedonic price model [17]; goods (i.e., services in our case) are priced by the users' valuation of different characteristics (QoS and QoD in our framework) and their contribution to users' utility. Towards this, we adopt the Quality Contracts (QC) framework as shown in Figure 1 for service pricing (by the users). A QC consists of a *QoS function* (where response time is mapped to QoS profit for the server) and a *QoD function* (where staleness is mapped to QoD profit for the server). The QoS and QoD metrics are application-dependent and orthogonal to our work.

Although our framework allows for users to specify complicated functions for QoS and QoD, in reality we expect users to simply select from a set of predefined such functions, much like most of our other digital "products" with different levels of service (e.g., cell phone plans).

In the presence of QCs, users and servers have distinct objectives: servers try to maximize their income, whereas users try to "stretch" their budget to execute as many queries as they can.

2.2 Server View

The web-database server is responsible for processing both updates and queries in order to meet the service requirements specified in the QC of each query.

Server Objective: Maximize Profit. The server objective is to maximize its profit, gained from each QC, through admission control and scheduling.

Server Optimization Mechanism: There are two phases of server optimization schemes: (1) admission control upon arrival of a query or an update, and (2) transaction scheduling once admitted. In general, the higher the bid, the higher the chance that a query is admitted and completed with high quality. Due to the space limitation, please refer to [13, 12] for more details on these two phases. We adopted Two Phase Locking - High Priority (2PL-HP) [1] where the lower priority transaction releases the lock to the higher priority transaction at a conflict.

2.3 User View

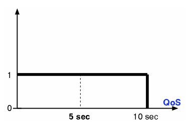
The user aspect of the system must include an interface for users to specify QCs and the ability to monitor the execution of QC-enabled queries, while keeping track of the current budget. Although the QC framework empowers users to influence resource allocation decisions at the server (to better meet their preferences), it also places the burden on the users to choose QCs (and adapt them over time). We expect that users will employ *user agents*, which will have explicit "instructions" from each user (on his/her true preferences and budget constraints) and a QC adaptation strategy.

Quality Contract / User Satisfaction: In this paper, we adopt QCs with linearly decreasing positive functions [12]. Intuitively, users can set the following four parameters to define a QC:

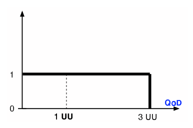
- qos_{max}, the maximum QoS profit,
- qod_{max}, the maximum QoD profit,
- rt_{max}, the maximum bearable response time, and
- uu_{max}, the maximum bearable staleness.

In this work, we simplify our model with an equivalent presentation, where the first two parameters (qos_{max} and qod_{max}) are replaced by:

- Q_{max}: qos_{max} $+qod_{max}$, the maximum payment for the query.
- γ: $\frac{qodmax}{qosmax}$, relative importance between QoS and QoD.



(a) QoS



(b) QoD

Figure 2: User Satisfaction Functions Example

As discussed in the introduction section (question Q3), in specifying QCs, users may not want to reveal the true worth of their queries (e.g., how much they can pay for a query result with a certain response time), although they would be willing to reveal their constraints (i.e., would prefer an answer within 10 seconds). In other words, the constraints in QCs are truthful, but the mapping to the worth dimention may not be. Figure 2 shows a simple example of what "truthful" *user satisfaction functions* might look like. Under this setting, queries are considered acceptable as long as the results meet the constraints on both QoS and QoD. Without loss of generality, we adopt such binary-step user satisfaction functions in this paper.

Following the example of Figure 2, we define two outcomes for a query for the general case:

- **Success:** A query succeeds if it is returned with valuable answers, meaning that the response time is shorter than the QoS constraint, rt_{max}, and the staleness is smaller than the QoD constraint, uu_{max}. Successful queries give to the server a nonzero payment, $Q_{paid} > 0$. The actual value of Q_{paid} depends on how well the server executes the query, given the QC. In terms of the user satisfaction functions, a successful query yields 1 from the product of all user satisfaction functions of the query (i.e., across all quality dimensions).
- **Failure:** If a query fails either the QoS or the QoD constraint, we call the query a *failure*, and Q_{paid} = 0. This allows the user to also infer the data freshness of the returned results.

User Objective: Success Ratio Maximization. The users' goal is to adapt Quality Contracts (e.g., by changing Q_{max}) to get as many as possible of his/her queries executed successfully, within the given total budget.

2.4 Analysis of Existing QC Adaptation Schemes

Assuming a user with N queries to submit and a total budget B, we consider three baseline strategies, which compute $Q^{(i)}_{max}$, the total bid for the QC of query i, as follows:

- **Fixed (FIX):** $Q^{(i)}_{max} = \frac{B}{N}$. FIX is a static policy, which assigns each query an equal share of the total budget.
- **Random (RAN):** $Q^{(i)}_{max}$ =*uniform*$[\frac{B}{N} - c,\ \frac{B}{N} + c]$, where c is a constant. This strategy uses $\frac{B}{N}$ as the mean, and $[\frac{B}{N} - c,\ \frac{B}{N} + c]$ as the range to pick Q_{max} uniformly.
- **Dynamic (DYN):** $Q^{(i)}_{max} = \frac{B_i}{N-i}$. This scheme monitors the current budget left B_i and the number of queries left $N - i$ before query i is issued.

Problems with existing schemes: FIX does not make full use of the budget, because it ignores the refunds from the previous failed queries. The RAN scheme is similarly problematic. DYN addresses the issue of ignored refunds by dynamically updating the available budget. However, DYN favors the queries issued later than earlier and creates an unfair allocation of the total budget.

3. ADAPTIVE QUALITY CONTRACT (AQC)

In this section, we present our proposed *Adaptive Quality Contract Scheme (AQC)*, which addresses the problems and limitations of the baseline algorithms that were presented in the previous section. Our AQC scheme switches between two modes: *Overbid mode* (Section 3.1) and *Deposit mode* (Section 3.2); we discuss how AQC chooses between the two modes in Section 3.3.

3.1 Overbid Mode

As we have shown, DYN unfairly "favors" later queries by monotonically increasing Q_{max} as time progresses using the cumulative refunds from previously finished queries. This behavior is roughly equivalent to last-minute spending by companies at the end of a fiscal year, since at that time, any of the remaining money in the current year's budget will effectively disappear unless spent immediately.

The Overbid mode of AQC addresses this problem by setting the budget of the submitted quality contracts for each query to be such that the *expected payments sum up to the overall budget*. In contrast, the DYN scheme sets the bid per query to be such that the individual bids sum up to the total budget (which clearly results in under-utilization of the budget, until the last minute).

In order to make the expected payments sum up to the overall budget, we need to make sure that the expected payments for the i^{th} query sum up to its fair share of the budget:

$$\mathbf{E}_p[Q^{(i)}_{paid}(x,\ y)] = \text{budget per query} = \frac{B_i}{N-i} \tag{1}$$

Then, in order to find how to set the QC for the query, we have to essentially express Q_{paid} in terms of Q_{max}, and solve Equation 1 for Q_{max}. Q_{paid} depends on the QoS function S, QoD function D, and how well the server returns the query (response time x and staleness y). Thus, as we show next, the expectation of Q_{paid} over the probability distribution of response time (x) and staleness (y) can be expanded as the sum of expected expenditure from the QoS function and from the QoD function respectively:

$$\mathbf{E}_p[Q^{(i)}_{paid}(x,\ y)] = \mathbf{E}_p[S(x)] + \mathbf{E}_p[D(y)] \tag{2}$$

If we combine Equation 1 with Equation 2, we have that:

$$\mathbf{E}_p[S(x)] + \mathbf{E}_p[D(y)] = \frac{B_i}{N-i} \tag{3}$$

In this work, we adopt linear segmented QCs where the QoS function can be represented as in Equation 4 and the QoD function can be represented as in Equation 5. If other formats of QC functions are adopted, Equation 4 and Equation 5 should be modified accordingly.

$$S(x) = \begin{cases} qos_{max}(1 - \frac{x}{rtmax}) & \text{if } x \in [0,\ rt_{max}] \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$D(y) = \begin{cases} qod_{max}(1 - \frac{y}{uumax}) & \text{if } y \in [0,\ uu_{max}] \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

We compute the expectation of QoS profit using empirical expectation, as shown in Equation 6.

$$\begin{aligned} \mathbf{E}_p[S(x)] &= \int S(x)p(x)\,dx \\ &= qos_{max}\int_0^{rtmax} p(x)\,dx - \frac{qos_{max}}{rt_{max}}\int_0^{rtmax} xp(x)\,dx \\ &\approx qos_{max}(\mathrm{P}(x < rt_{max}) - \frac{\bar{x}}{rt_{max}}) \end{aligned} \tag{6}$$

where P$(x < rt_{max})$ is the percentage of cases that the response time of the user query is smaller than its response time constraint rt_{max}, and $\bar{x}$ is the average response time. Both P$(x < rt_{max})$ and $\bar{x}$ can be computed based on the query execution history. We introduce α to denote this part of computation and summarize the expected QoS profit as follows:

$$\mathbf{E}_p[S(x)] \approx qos_{max} \cdot \alpha \tag{7}$$
$$\alpha = \mathrm{P}(x < rt_{max}) - \frac{\bar{x}}{rt_{max}}$$

Similarly, we compute the expectation of QoD profit:

$$\mathbf{E}_p[D(y)] \approx qod_{max} \cdot \beta \tag{8}$$
$$\beta = \mathrm{P}(y < uu_{max}) - \frac{\bar{y}}{uu_{max}}$$

As described in Equation 3, the total expected profit from both QoS (Equation 7) and QoD (Equation 8) should be set to the current budget per query $\frac{B_i}{N-i}$:

$$qos_{max} \cdot \alpha + qod_{max} \cdot \beta = \frac{B_i}{N-i} \tag{9}$$

where α and β are computed based on query execution history (as shown in Equation 7 and Equation 8). Since the ratio between qos_{max} and qod_{max} is known as γ, we have:

$$Q_{max} = qos_{max} + qod_{max}$$
$$qod_{max} = \gamma \cdot qos_{max} \tag{10}$$

We solve Equations 9 and 10 to get the final solution of Q_{max}:

$$Q^{(i)}_{max} = \frac{B_i}{N-i} \cdot \frac{1}{\alpha + \gamma \cdot \beta} \tag{11}$$

In the above solution, $\frac{1}{\alpha+\gamma\cdot\beta}$ is essentially the overbid factor.

3.2 Deposit Mode

Although Overbid mode successfully utilizes as much of the budget as possible (and in a fair manner across all queries), it will not detect the cases of "overpayment" because of the server having a light load. In such cases, there is not much "competition" from other users, and as such the user could have paid less than what Overbid mode would suggest.

The benefit of detecting these cases comes from the inherent dynamic nature of typical web-database servers. The load at such servers can fluctuate from *very high* (e.g., in periods of flash crowds), where queries would require a high budget or they will not be able to execute, to relatively *low*, where queries would require a much less budget than usual to execute.

In order to make sure that the AQC scheme can successfully react to the inherent dynamic nature of web-database servers, we introduce a budget saving scheme which we call *Deposit mode*. The main idea behind Deposit mode is to recognize cases when users can spend less of their budget (because of a less competitive situation), so that they are ready to spend more when facing stronger competition from other users.

To implement Deposit mode, the Q_{max} is reduced when there is a row of consecutive successful query executions. Let $Q^{(s)}_{max}$, $Q^{(s)}_{paid}$ be the budget and the payment for the most recent successful query. We set the new $Q^{(i)}_{max}$ in Deposit mode as follows:

$$Q^{(i)}_{max} = Q^{(s)}_{max} \cdot \left(1 - \frac{Q^{(s)}_{paid}}{Q^{(s)}_{max}}\right) \tag{12}$$

We call the ratio of $\frac{Q^{(s)}_{paid}}{Q^{(s)}_{max}}$ as the *deposit factor*. Notice that the closer $Q^{(s)}_{paid}$ is to $Q^{(s)}_{max}$, the bigger the deposit factor is. The intuition is that we could deposit more and bid less when historical success comes with very good performance (a high $Q^{(s)}_{paid}$ usually corresponds to high QoS and high QoD). A high deposit factor thus may indicate that the system is currently lightly loaded. Although a lower bid will decrease the priority of the query, hopefully in a lightly loaded server, the query can still be answered within constraints. On the contrary, if the last successful query barely meets the QoS and QoD constraints, the deposit factor will be close to zero and the query will be kept with a competitive bid.

3.3 Switching between Deposit and Overbid

At the beginning, the system is set to the overbid mode by default. AQC keeps track of the number of consecutive query successes ($successQ.size$) and uses it to decide the current system mode.

If the number of successes is significantly large (i.e., larger than a threshold c), the system is set to deposit mode. This is because a consecutive successful query history indicates a less competitive environment or a lightly loaded web-database server, thus the bid can potentially be decreased without hurting the success ratio.

Notice that $successQ.size$ only includes those queries that are completed within the time window w. Thus, $successQ.size$ may decrease due to two reasons: (1) there are no more queries to be completed (i.e., neither query success nor query failure), as a result, $successQ.size$ decreases as the moving window w moves on. If $successQ.size$ drops below c, the system mode will be set to overbid because of the lack of successful feedbacks; (2) there is a query failure, which will reset $successQ.size$ to zero immediately. In both cases, system mode is set to overbid promptly to utilize the user's budget as much as possible so that the server is motivated to execute the users queries with higher priorities.

By switching between the overbid and deposit modes according to the query success/failure, the AQC scheme naturally follows the law of supply and demand. We expect the overhead of adaptation to be linear to the total number of queries processed.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

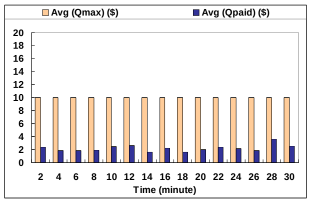
We have acquired access traces from a popular stock market information web site, Quote.com, and combined them with the NYSE (New York Stock Exchange) update traces for the same time period, which enabled us to accurately generate both query and update workloads for our experiments.

Query Traces We use real queries from Quote.com for April 24, 2000. All queries are read-only. We concentrated on a "heavy" workload for the server, a 30-minute (10:30am-11:00am) interval with over 120,000 queries on 4,107 different stock symbols.

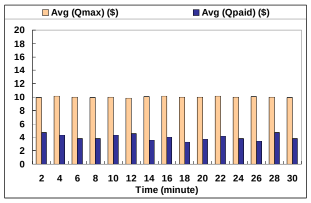
Update Traces We extracted the actual trades on all securities listed on NYSE during the same time interval as our query trace (10:30am-11:00am on April 24, 2000). The update trace shares the same indexing scheme with the query trace. The update trace fragment we used has over 396,000 entries.

Comparison Algorithms To evaluate our proposed QC adaptation strategy, we performed an extensive simulation study using the FIX, RAN, DYN schemes (Section 2.4) and our proposed proposed AQC strategy (Section 3).

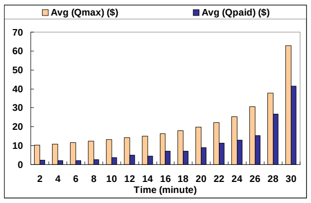
Each query is submitted to the system along with a user-specified QC; each user also has an initial budget, which for simplicity is



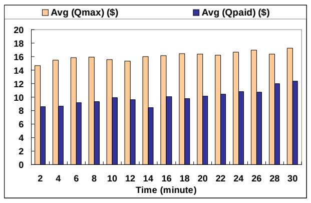
(a) FIX: Money Expenditure



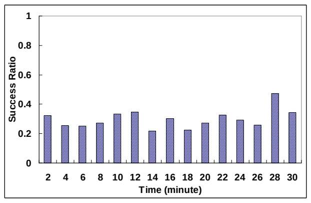
(c) RAN: Money Expenditure



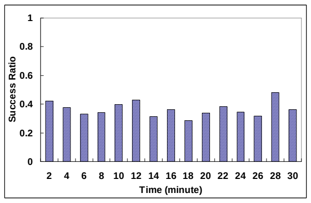
(e) DYN: Money Expenditure



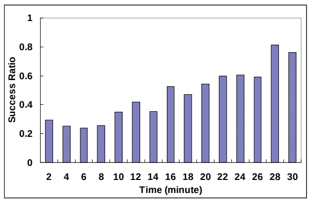
(g) AQC: Money Expenditure



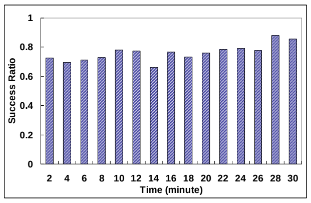
(b) FIX: Success Ratio



(d) RAN: Success Ratio



(f) DYN: Success Ratio



(h) AQC: Success Ratio

Figure 3: Duet Over time

equal among all users.

4.2 Performance Comparison

For a fair comparison, we present results from two different execution runs: duet and quartet. In duet, each run contains two classes of users (i.e. two algorithms), creating a one-to-one confrontation to show directly which algorithm performs better. In quartet, we put all four algorithms into the run, where they all interact and compete with each other.

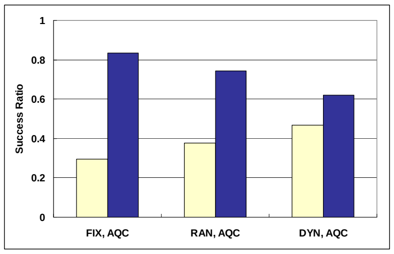


Figure 4: Duet Environment: AQC vs. Baseline Algorithms. AQC outperforms baseline algorithms by up to 183%.

4.2.1 Duet

Experiment Design (Figure 4) We compare AQC with each baseline algorithm individually to eliminate unnecessary interactions from multiple algorithms. We performed three runs: (FIX, AQC), (RAN, AQC), and (DYN, AQC). Each query from the trace is randomly associated with one of the two algorithms in the experiment.

Results (Figure 4) We run each trace 20 times and report the average query success ratio for the three comparisons in Figure 4. The performance difference is quite obvious: AQC users perform 183% better than FIX, almost 100% better than RAN, and more than 30% better than DYN.

Results Over Time(Figure 3) Given that the four algorithms have different behavior over time, we also plot the bid Q_{max} and the money paid Q_{paid} for each query as well as the query success ratio with a 2-minute window over time.

In Figure 3(a), FIX gives a constant bid ($10) for each query. Due to the unavoidable CPU time and unpredictable queuing time, the real expenditure is only around $2 on average. Failing to reuse the refund leaves FIX with a small success ratio shown in Figure 3(b). RAN has similar results as shown in Figure 3(c) and (d). Q_{max} varies around $10 and Q_{paid} is around $4 on average. With more than half the budget wasted, RAN gains less than 50% success ratio over time.

In Figure 3(e), we see that DYN dynamically adjusts the current available budget and increases Q_{max} over time. As a result, DYN's success ratio increases over time too, as shown in Figure 3(f). However, DYN is still conservative on early issued queries, which not only jeopardizes the fairness of queries coming at different times, but also hurts its overall success ratio.

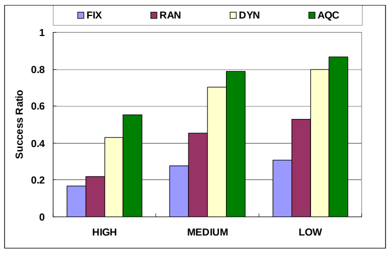
Finally, Figure 3(g) and (h) show AQC's improvement from two sides. First, the average Q_{paid} is around $10, thus the budget is fully used to increase the quality of query results. AQC is able to set Q_{max} higher than $10 because of foreseeing the expected expenditure. Second, AQC tries to save money after consecutive successes, so that it can bid higher to survive a tougher situation later. This is why we see a few downward trends in Figure 3(g). Both aspects help AQC achieve significantly better results when it competes with other algorithms.

4.2.2 Quartet

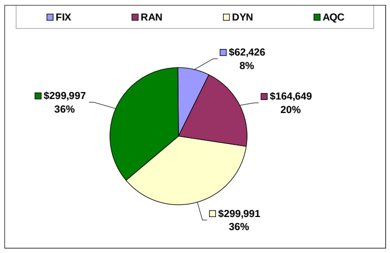
Experiment Design (Figure 5) Having compared the different baseline algorithms (FIX, RAN, DYN) to our proposed algorithm (AQC), we mix the four user algorithms in the same execution run; each class of users has 30,000 queries with a mean Q_{max} of $10. In this set of experiments, we focus on (1) varying quality constraints (Figure 5(a)), and (2) showing both the user view and the server view (Figure 5(b)).

Results (Figure 5(a)) We change the user constraints on QoS to be tight, medium, and loose to generate three traces, High, Medium, and Low workload respectively. As expected, for all algorithms, the success ratio is higher with Low system workload (which has loose quality constraints). In comparison to other algorithms, AQC performs the best under the whole spectrum of workloads. Another observation is that a high system workload also exaggerates the performance differences among the algorithms. Under high workload, AQC outperforms FIX by 233%. AQC also achieves 155% better performance than RAN and 28% better than DYN.

Results (Figure 5(b)) In addition to the users' view of these algorithms, we also show the server profit gains from each user algorithm under the medium workload. We observed similar trends with both the high and the low workload. Figure 5(b) shows that the server stands to gain much more profit from DYN and AQC, thus tends to serve them better than FIX and RAN. Making full use



(a) User View: Success Ratio



(b) Server View: Profit

Figure 5: Quartet Environment: 4 algorithms under different workload settings.

of the budget is a win-win strategy from both users' and server's point of view.

To summarize, AQC not only gives the best success ratio under various workloads, but also makes the users most popular from a system's point of view, as the system can make much more profit from users utilizing AQC.

5. RELATED WORK

Web-databases There is a plethora of papers that focus on improving the performance of user requests to database-driven web sites, using caching [4, 11] or materialization [7]. These approaches usually provide a best-effort solution in terms of either QoS or QoD. In our recent work [6], we introduced the Quality Contract framework to combine individual users' preferences for both QoS and QoD. We demonstrated the QC framework in [14], in combination with our policies for admission control [13], and query & update scheduling [12]. Our demo also highlighted the benefits of user-side adaptation of QCs (although we did not provide the AQC scheme, as we do in this paper).

Grids and Web Services Service Level Agreements (SLAs) in Grid applications [3] is another related area. In SLAs, resource availability, capability, and cost are considered for effective resource management. SLAs also exist for Web-services [10, 18, 20]. Although sharing the general goal of resource regulation and cost controlling, our work focuses on one specific resources: web-databases.

6. CONCLUSIONS

In previous work we have proposed the Quality Contracts (QCs) framework, and introduced the supporting system optimizations. In this work, we turn our attention to the user side of the equation and consider *user strategies to adapt Quality Contracts over time*. Towards this, we proposed the Adaptive Quality Contract (AQC) strategy, which monitors a user's queries and the server's responses in order to automatically adapt the QCs of subsequent user-submitted queries. We performed an extensive simulation study with real traces, which showed that AQC consistently outperforms baseline algorithms (by up to 233%).

Currently, we are exploring strategy-proof mechanisms for the bidding process and are considering how other schedulers (e.g., [5]) could be adapted to "understand" Quality Contracts.

7. REFERENCES

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

[2] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the Ondemand IT InfraStructure*, Oct 2004.

[3] R. Buyya, D. Abramson, and S. Venugopal. The Grid Economy. *Proceedings of the IEEE*, 93(3):698–714, 2005.

[4] A. Datta et al. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *SIGMOD Conference*, 2002.

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

[6] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *BIRTE Workshop*, 2006.

[7] A. Labrinidis and N. Roussopoulos. Webview materialization. In *SIGMOD Conference*, 2000.

[8] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, 2005.

[9] H. C. Lau, S. F. Cheng, T. Y. Leong, J. H. Park, and Z. Zhao. Multi-period combinatorial auction mechanism for distributed resource allocation and scheduling. In *IAT*, 2007.

[10] Y. Liu, A. H. Ngu, and L. Z. Zeng. Qos computation and policing in dynamic web service selection. In *WWW Alt.*, pages 66–73, 2004.

[11] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier Database Caching for e-Business. In *SIGMOD Conference*, 2002.

[12] H. Qu and A. Labrinidis. Preference-aware query and update scheduling in web-databases. In *ICDE Conference*, 2007.

[13] H. Qu, A. Labrinidis, and D. Mosse. Unit: User-centric transaction management in web-database systems. In *ICDE Conference*, 2006.

[14] H. Qu, J. Xu, and A. Labrinidis. Demo: Quality is in the eye of the beholder: Towards user-centric web-databases. In *SIGMOD Conference*, 2007.

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

[16] O. Regev and N. Nisan. The popcorn market—an online market for computational resources. In *Proc. of the first international conference on Information and computation economies*, 1998.

[17] S. Rosen. Hedonic prices and implicit markets: Product differentiation in pure competition. *The Journal of Political Economy*, 82(1):34–55, Janurary - February 1974.

[18] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, and F. Casati. Automated sla monitoring for web services. In *IEEE/IFIP DSOM*, 2002.

[19] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Trans. on Software Engineering*, 18(2):103–117, February 1992.

[20] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *WWW Conference*, 2003.