Quality Contracts for Real-Time Enterprises

Alexandros Labrinidis, Huiming Qu, and Jie Xu

Advanced Data Management Technologies Laboratory
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
{labrinid,huiming,xujie}@cs.pitt.edu

Abstract. Real-time enterprises rely on user queries being answered in a timely fashion and using fresh data. This is relatively easy when systems are lightly loaded and both queries and updates can be finished quickly. However, this goal becomes fundamentally hard to achieve due to the high volume of queries and updates in real systems, especially in periods of flash crowds. In such cases, systems typically try to optimize for the average case, treating all users, queries, and data equally. In this paper, we argue that it is more beneficial for real-time enterprises to have the users specify how to balance such a tradeoff between Quality of Service (QoS) and Quality of Data (QoD), in other words, "instructing" the system on how to best allocate resources to maximize the overall user satisfaction. Specifically, we propose Quality Contracts (QC) which is a framework based on the micro-economic paradigm and provides an intuitive and easy to use, yet very powerful way for users to specify their preferences for QoS and QoD. Beyond presenting the QC framework, we present results of applying it in two different domains: scheduling in real-time web-databases and replica selection in distributed query processing.

1 Introduction

Globalization and the proliferation of the Web have forced most businesses to evolve into real-time enterprises; it is always daytime in some part of the world! Such real-time enterprises rely on vast amounts of collected data for business intelligence. Data is processed continuously and typically stored in data warehouses, for further analysis.

Given the real-time nature of businesses in our fast-changing world, getting answers in a timely fashion and using fresh data is of paramount importance. This is fairly easy to do in periods of light load, however, it becomes fundamentally hard to achieve in periods of high volumes of queries (e.g., multiple analysts working towards a deadline for end of the year reports) or updates (e.g., influx of sales data because of a 3-day special sale weekend)[1]. In cases of high load, systems will typically try to optimize for the average case, treating all user queries and quality metrics equally.

In this paper, we argue that it is more beneficial for real-time enterprises to have their users (business analysts in this case) supply their *preferences* on how the system should

[1] This scenario assumes a complete separation of operational and business analysis information systems; the situation is even worse if these are coupled together under a single system.

C. Bussler et al. (Eds.): BIRTE 2006, LNCS 4365, pp. 143–156, 2007.

balance the trade-off between Quality of Service (QoS) and Quality of Data (QoD), in other words, instruct the system on how to best allocate resources in order to maximize user satisfaction. We propose to do this by utilizing *Quality Contracts*, a framework for describing user preferences that is based on a micro-economic model. Quality Contracts (QCs) empower users to quantify QoS and QoD using their favorite metric(s) of interest and to specify their preferences (in an intuitive and integrated way) for how the system should allocate resources in periods of high load.

In order to compete successfully in today's highly dynamic environments, real-time enterprises are expected to rely on two types of querying capabilities. First, *ad hoc queries* are utilized by business analysts to explore previously collected data; such queries have been the staple of business intelligence units for decades. Secondly, *continuous queries* (CQs) are registered ahead of time and constantly monitor the incoming data feeds to detect patterns and other precursors of customer behavior. The goal in such cases is to provide actionable information as soon as possible, by continuously executing (i.e., re-evaluating) CQs with the arrival of new relevant data. Such CQs belong to a new data processing paradigm, that of Data Stream Management Systems (DSMSs) [7,17,6,4,20]. Clearly, both types of queries are crucial to improving the real-time enterprise's performance.

Although there exist multiple metrics for measuring QoS or QoD for ad-hoc and for continuous queries, they have two major shortcomings.

(1) Lack of a unified framework that can evaluate quality for both ad-hoc as well as for continuous queries: Currently, quality measures used for ad-hoc queries (in DBMSs) are different from those used for continuous queries (in DSMSs). It is not clear how these two types of quality measures relate to each other in a system that supports both kinds of queries. Many of these measures do not even have a bounded domain, which makes comparison impossible. The major problem this limitation creates is with regards to *provisioning of resources*: the system does not have a common framework to compare usage/utility of resources allocated to ad-hoc versus continuous queries. As such, the system is forced to allocate resources separately to the two types of queries with the danger of under-utilization and overloading, and all the consequences that these bring. Another problem is that of usability: users must "learn" two sets of quality metrics, one for traditional queries and one for continuous queries.

(2) Limited consideration of user preferences in evaluating QoS/QoD: 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 [12,19,2] 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 [16,15,14], 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, as part of our preliminary work (presented in the previous section), we extended the work of [12]

to consider both QoS user requirements (i.e., deadlines) and QoD user requirements (i.e., freshness threshold).

In DSMSs, user preferences are indeed considered to some degree. Looking at the Borealis project [1] (which corresponds to the state of the art), we can see that a "Diagram Administrator" can provide QoS functions that could correspond to user preferences (in the same way as in the Aurora project [5]). However, the different components of the QoS (i.e., the Vector of Metrics) are aggregated into a single, global QoS score, using *universal* weights. In other words, the same QoS components are used for all queries and the same relative importance to each QoS component is assigned for all queries via system-wide weights. This system-based approach has another negative side-effect: the benefit of the overall system can often overweigh the benefit of the individual user or query (even by just a little), who/which can be "penalized" repeatedly for the benefit of the others, thus leading to starvation.

Desired Properties. Given the previously mentioned deficiencies, we believe that an effective framework for measuring QoS and QoD must have the following primary properties:

- handle ad-hoc queries and continuous queries at the same time,
- allow the user to choose from an array of QoS/QoD metrics in order to specify quality requirements/preferences,
- allow the user to combine multiple QoS/QoD metrics and indicate the relative importance of each individual metric (as a component of the overall Quality for the user),
- allow the user to specify the relative importance of different queries,
- do all of the above in a "democratic" way: it should not be that a user can always specify his/her queries to be more important than everybody else's, thus monopolizing system resources.

In the next section, we describe the proposed Quality Contracts Framework that addresses all of the above challenges.

2 Quality Contracts Framework

We propose a unified framework for specifying QoS/QoD requirements in systems that support both ad-hoc queries and continuous queries. Our proposed framework, *Quality Contracts*, is based on the micro-economic paradigm [22,21,8]. In our framework, users are allocated virtual money, which they spend in order to execute their queries. Servers, on the other hand, execute users' queries and get virtual money in return for their service. In order to execute a query however, both the user and the server must agree on a Quality Contract (QC). The QC essentially specifies how much money the server which executes the query will get. The amount of money allocated for the query is not fixed (as was the case in [21]). Instead, the amount of money the server receives depends on how well it executes the user's query. In fact, in the general case, QCs can even include refunds; a very poorly executed query can result in the user being reimbursed instead of paying for its execution (accumulated refunds can improve the odds of the user's query executing properly later).

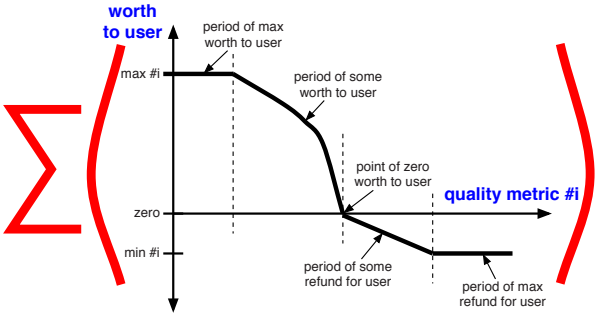


Fig. 1. General form of a Quality Contract

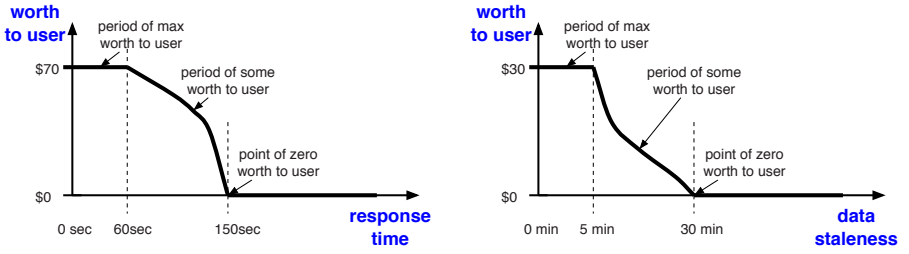
Under the proposed scheme, a user can specify how much money he/she thinks the server should get at various levels of quality for the posed query, whereas the server, if it accepts the query and the QC, essentially "commits" to execute the queries, or face the consequences. In this model, servers try to maximize their income, whereas users try to "stretch" their budget to run successfully as many queries as they can.

A Quality Contract (QC) is essentially a collection of graphs, like the one in Figure 1. Each graph represents a QoS/QoD requirement from the user. The X-axis corresponds to an attribute that the user wants to use in order to measure the quality of the results (e.g., response time or delay). The Y-axis corresponds to the virtual money the user is willing to pay to the server in order to execute his/her query. Notice that in order to specify more than one QC (i.e., to judge the quality of the results using more than one metric) the user must provide additional virtual money to the server. Put simply: the server can hope to receive the sum of all max amounts of the different QC graphs that a user submits along with a query. Of course, the level of money the server gets is differentiated according to the value of the quality metric for the results. There is also the possibility of the server having to issue "refunds" for queries that were not satisfactory completed. Next, we present examples of QC graphs in order to illustrate their features and advantages. For simplicity, we will use the dollar sign ($) to refer to virtual money for the remainder of this paper.

2.1 Quality Contracts Examples

Figure 2 is an example of Quality Contract (QC) for an ad-hoc query submitted by a user. This QC consists of two graphs: a QoS graph (Figure 2a) and a QoD graph (Figure 2b). 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.*

In the next example, we have QCs for two different continuous queries, Q_1 (Figures 3a & 3b) and Q_2 (Figures 3c & 3d), issued either by the same user or by two different users. In addition to highlighting different types of QC graphs (including more complicated



(a) QoS graph ($0 to $70) (b) QoD graph ($0 to $30)

Fig. 2. QC example for one ad-hoc query. The QoS metric is response time, whereas the QoD metric is data staleness. Data staleness is measured as the time between the last instant when the physical world has changed and the instant when the the local storage has been updated (i.e., time since the last update on a data item access by the query).

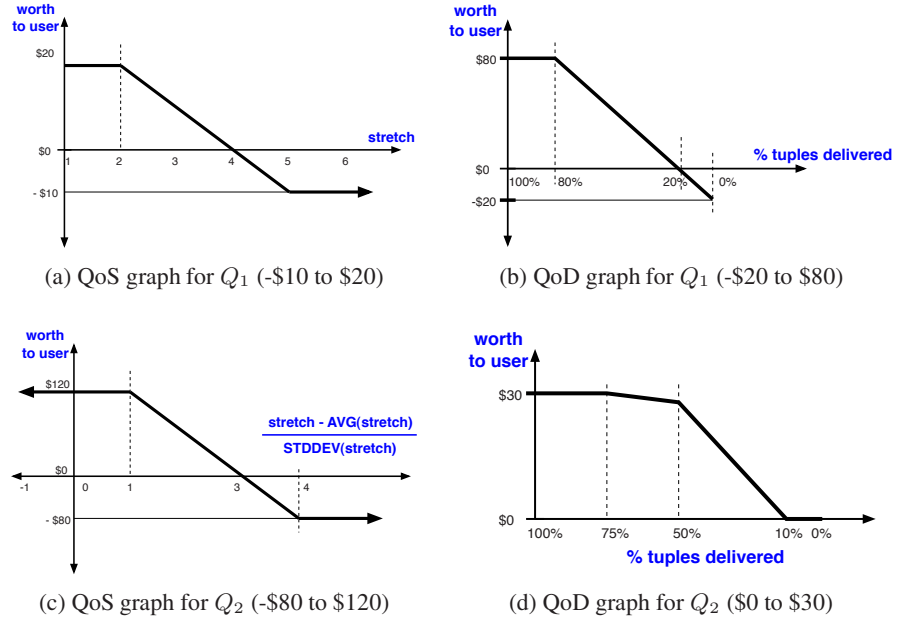
quality metrics such as those expressed by virtual attributes), this example also illustrates another important feature of the QC framework: *users can easily specify the relative importance of each query by allocating their budgets accordingly.* In our example, Q_1 has a total budget of $100 (with the most important quality metric being QoD, allocated $80 out of $100), and Q_2 has a total budget of $150 (with the most important quality metric being QoS, allocated $120 out of $150). Finally, this relative importance can also be evaluated over different types of queries altogether (e.g., the ad-hoc query of Figure 2 can be executed at the same time as the continuous queries of Figure 3).

2.2 Quality Contracts Implementation

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. Examples of such QoS metrics include response time (esp. in connection with a soft or hard deadline), delay, stretch (average, maximum), etc. Examples of QoD metrics in the presence of ad-hoc updates include time-based, lag-based, and divergence-based definitions. Additionally, examples of QoD metrics for continuous queries include drop-based (like the example in Figure 3b), or value-based (i.e., assign worth to the user based on the values of the result, as in Aurora [5]).

Payment Stream. For continuous queries, QCs can be seen as a guarantee for a *payment stream*. In other words, the min/max virtual money values on the Y-axis correspond to a *rate of payment* rather than a one-time payment amount, which is the case for ad-hoc queries.

Virtual Attributes. One important aspect of QCs that we plan to explore further is the ability to specify arbitrary quality metrics, in the form of *virtual attributes* that are computed over other attributes, possibly including statistics of the entire system. We have already seen an example of this in Figure 3c where the user specified QoS as the delay his/her queries received when compared to the average delay in the system. We expect such "comparative" QoS metrics to be rather frequent: it is probably harder for



(a) QoS graph for Q_1 (-\$10 to \$20)

(b) QoD graph for Q_1 (-\$20 to \$80)

(c) QoS graph for Q_2 (-\$80 to \$120)

(d) QoD graph for Q_2 (\$0 to \$30)

Fig. 3. QC example for two continuous queries Q_1 and Q_2. The QoS metric for Q1 is stretch, or 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, where as the QoS metric for Q2 is a *virtual attribute*. Virtual attributes are computed over the entire system rather than just using the performance statistics of the individual query. In this example, the user simply wants to guarantee that his/her queries are not delayed more than average: the stretch observed for his/her queries needs to be at most one standard deviation away from the average stretch of the entire system for maximum payoff at the server (\$120). The QoD metric for both Q1 and Q2 is the percentage of tuples delivered.

a user to specify exact timing requirements, but it is easier to specify that he/she wants the submitted query to be executed within the top 20% of the fastest queries in the entire system.

Parameterized QCs. Making QCs easy to configure is fundamental to their acceptance by the user community. Towards this we plan on providing *parameterized versions of QC graphs* that the users can easily instantiate. For example, we can have a parameterized QoD function based on tuples dropped (similar to Figure 3b) with four parameters: maximum worth (e.g., \$80), maximum refund (e.g., -\$20), percentage point beyond which QoD drops below maximum (e.g., 80%), and percentage point after which user is entitled to a refund (e.g., 20%). We can assume a piece-wise linear curve and allow the user to specify more intermediate points. We can also assume a predetermined curve and allow the user to specify even less parameters (e.g., only the maximum worth). Finally, parameterized QCs could also reduce the overhead of evaluating the QCs in the system (by essentially "compiling" their definitions).

Contract Clauses. A simple form of a parameterized QC is that of a *"contract clause"*. This is the case when the user essentially promises a "bonus" to the server when a

certain quality metric is met (e.g., response time less than 30 minutes for a long analysis query), but no virtual money otherwise. The QC graph in this case is a simple step function, and the parameterized version needs two values: the maximum worth and the turnover threshold.

QC Classes. Another way to increase usability of QCs and also reduce the overhead of evaluating them is to introduce differentiated levels of service using different *"contract classes"*. In this way, users simply assign queries to a predefined class with specific characteristics (expressed by QCs) without having to specify a complicated QC. This approach is also more scalable, since it reduces the overhead of evaluating the QC for each query independently.

Overhead. We expect the overhead of evaluating different QCs to vary significantly. For example, evaluating the delay observed by tuples is fairly easy to compute (e.g., Figure 3b), whereas computing the average stretch and its standard deviation (e.g., Figure 3c) should be considerably more expensive. As such, *we propose that the cost of computing the QC is also included in the "price" that the user is supposed to pay to the server for successful execution of his/her query under the given QC*. This is a departure from current practices (where most quality metrics were very simple and therefore of similar cost), but is necessitated by the complexity of new, sophisticated quality metrics whose overhead would unfairly burden the system, but they would still be attractive to users. Given this setup, users still have a choice over a wide assortment of quality metrics for QCs, but essentially they have to pay a "commission" if they want to use a sophisticated metric.

2.3 Usability of Quality Contracts

The usability of the QCs must be address for the QC framework to be successful. 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* (as mentioned earlier) 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.

Although in this paper we align QoS to response time and QoD to data freshness, the Quality Contracts framework is general enough to allow for **any quality metric**. An example of this is the concept of *virtual attributes* that was introduced earlier, where a user-defined function is used as the quality metric. Furthermore, we believe that the notion of Quality of Data can be extended in multiple ways. First of all, it can be used to measure the level of *precision* of the result (i.e., similar to data freshness, but using the values to determine the amount of deviation from the ideal, instead of time since last update). Similarly, we can use approximate data to answer questions and this can be "penalized" accordingly by the user (while it also poses a clear trade-off between response time and accuracy of results). Secondly, it can be used in systems that

support *online aggregation*[10], where user queries can return results at various level of *confidence*. In such a case, QoD can be represented as a function over the confidence metric. Finally, QoD can be used to refer to *Quality of Information*, where, for example, a measure of trustworthiness of the provided information can be computed and users may express how much they are willing to "pay" for high-quality results.

2.4 Quality Contracts – Discussion

The proposed QC framework meets all the challenges set forth at the introduction. It is able to handle ad-hoc queries and continuous queries at the same time; by using virtual money as the underlying principle, different metrics can easily be compared. The proposed framework enables users to choose from a wide assortment of QoS/QoD metrics in order to specify quality requirements/preferences. QCs allow the user to combine multiple quality metrics for a single query and indicate their relative importance; the same applies for multiple queries. By employing a virtual money economy, users cannot monopolize resources (by falsely advertising their queries to be the most important), but at the same time users are safe from starvation (by accumulating virtual money when not "paying" for queries that executed below the acceptable quality level).

The proposed QC framework also introduces the following salient features. Individual users, not system administrators, are those specifying user preferences; the virtual money scheme is inherently intuitive and easy for users to grasp. To further increase usability, parameterized and class-based QCs are introduced. A wide assortment of QoS/QoD metrics (for both ad-hoc and continuous queries) is possible. The set of QoS/QoD metrics is enhanced by allowing for virtual attributes, which enable comparison of the performance to the individual query to system-wide measures. To counteract the evaluation cost of such sophisticated metrics, the overhead of computing them is include in the "price" of the query. The notion of refunds is introduced; this helps further towards eliminating starvation.

3 Transaction Scheduling Under Quality Contracts

In the first application of Quality Contracts, we considered a web-database server (for example, a stock quote information server) that answers user-submited ad hoc queries, while it processes updates in the background. Clearly, in this environment, high volumes of queries and/or updates can wreck havoc in the allocation of resources and result in many queries having unpredictable response time and/or returning stale data. In such an environment, the QC framework provides an intuitive way to express user preferences (in terms of response time and freshness requirements for queries) and thus enable the systtem to do a "better" job at allocating resources.

3.1 QUTS Scheduling Algorithm

We proposed the Query Update Time Share (QUTS) [18] scheduling algorithm to optimize the system profit in the presence of QCs. QUTS is a two-level scheme that can dynamically adjust the query and update share of the CPU, so as to maximize the overall

Table 1. Quality Contracts Used in Performance Comparison

Varying	ps	pd	rd	uu
ps ($)	{ 1, 2,. . . , 10}	50	50	0
pd ($)	5	{ 1, 2,. . . , 10}	50	0

system profit. At the high level, it dynamically allocates CPU to either the query queue or the update queue according to a profit. At the lower level, queries and updates have their own priority queues and potentially different scheduling policies. Specifically, we adopted Profit over Relative Deadline (PRD) [9] for queries and FIFO for updates. We used multiversion concurrency control to allow for maximal concurrency.

3.2 QUTS Experimental Evaluation

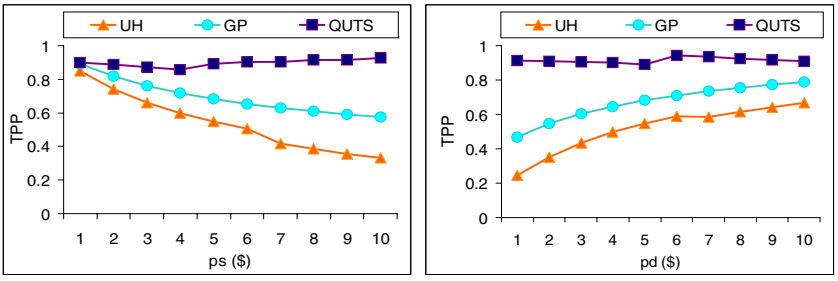
We compared QUTS with two baseline algorithms (Updated-High and Global-Priority), using both real and synthetic trace data. Our experiments showed that Quality Contracts are able to capture a wide spectrum of user preferences and that QUTS consistently outperforms existing methods, under the entire spectrum of quality contracts.

Baseline Algorithms:

- **Update High (UH).** UH has a dual priority queue where update queue has higher priority than query queue [3]. Priority schemes within each queue are same with QUTS. Two Phase Lock - High Priority (2PL-HP) is used for the concurrency control, where low priority transaction is aborted and hands the lock to high priority transactions. UH guarantees the highest data freshness, but it may waste a lot time updating data that have no contribution to the system profit.

- **Global Priority (GP).** 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 relative queries. Query priority scheme is HP (High Profit) which uses the sum of QoS maximal profit and QoD maximal profit. 2PL-HP is used for concurrency control. GP automatically pushes behind the updates which may not be contributing to the data quality of the queries, but it may still lead to query starvation when a surge of "good" updates arrives.

Experimental Setup. We used query traces from a stock market information web site and update traces from NYSE to drive our experiments. As part of the experimental setup, we attach a QC t o every query before it is submitted to our system. The QC is in the form of a positive, linear, monotonically decreasing function. Such QCs 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). Since there are four parameters in a quality contract, we vary one and fix the others to median values to see how the performance changes. Due to the space limitations, we only show



(a) vary QoS max (ps) (b) vary QoD max (pd)

Fig. 4. 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.

two cases here (Figure 1). We measure the performance by computing how much profit has the system gained, normalized by dividing the actual gained profit over the maximal possible profit, or what we call the *Total Profit Percentage* (TPP).

Experimental Results. Figure 4(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 4(b) plots TPP vs. pd. Again, QUTS outperforms all others.

4 Distributed Query Processing Using Quality Contracts

In the second application of Quality Contracts, we considered a distributed query processing environment, such as those that could be in place by a collaborative business intelligence application, where data is replicated across multiple nodes. On the one hand, data replication in this context is expected to improve reliability, expedite data discovery, and increase performance (i.e., Quality of Service, or QoS). On the other hand, however, it is also expected to have a negative impact to the Quality of the Data (QoD) that are being returned to the users. Getting results fast is crucial of course, but usually a limit to the degree of "staleness" is needed to make the results useful.

4.1 Replication-Aware Query Processing Scheme

We proposed *Replication-Aware Query Processing* (RAQP) [23], to address the problem of replica selection in the presence of user preferences for Quality of Service and Quality of Data (expressed as QCs).

Using the Quality Contracts framework as a natural and integrated way to guide the system towards efficient decisions, the RAQP scheme optimizes query execution

Table 2. Default System Parameters in Experiments

Simulation Parameter	Default Value
Core Node Number	100
Edge Node Number	1000
Unique Data Source	1000
Unique Data Number Per Data Source	U(10, 100)
Data Size	U(20, 200Mb)
# of Replicas Per Data	U(10, 30)
Bandwidth between each pair of Nodes	U(1, 50Mbps)

plans for distributed queries with Quality Contracts, in the presence of multiple replicas for each data source. Our scheme follows the classic two-step query optimization [21,11,13]: we start from a statically-optimized logical execution plan and then apply a greedy algorithm to select an execution site for each operator and also which replica to use. The overall optimization goal is expressed in terms of "profit" under the QC framework (i.e., the approach balances the trade-off between QoS and QoD).

4.2 RAQP Experimental Evaluation

We evaluated our proposed replication-aware query processing algorithm experimentally by performing an extensive simulation study using the following algorithms:

- **Exhaustive Search (ES):** Explore the whole search space exhaustively, thus guaranteeing to find the optimal allocation.
- **RAQP-G:** Greedy replication-aware initial allocation plus iterative improvement.
- **RAQP-L:** Bottleneck breakdown, local exhaustive search & iterative improvement.
- **Rand(k):** Random initial allocation plus k steps of iterative improvement (used as a "sample" of the search space).

Experimental results. One of the important features Quality Contracts hold is that users can easily specify the relative importance of each component of the overall quality by allocating the query budget accordingly. In order to observe the algorithm performance under different environments, we classify the users' quality requirements into 6 classes. We have three values for QoS and QoD: high (75), low (25), same (50) and two types of slope for the QC function: small and large, which produce 6 seperate classes. We report our results in Figure 5. Since our allocation initialization algorithm was aimed at response time improvement, QoS got more improvement than QoD in all the cases. Especially when QoS was assigned a higher budget, the effect on both QoS and total profit were obvious. When QoD was assigned a higher budget, the relative improvement of QoD also increased compared to the lower budget case. Our results clearly confirmed the functionality of Quality Contracts and our RAQP algorithm. Assigning higher "budget" to a quality dimension ends in that dimension achieving better performance by our optimization algorithm. The larger the budget difference the larger the difference in the resulting quality. This behavior is unique to our algorithm and allows the system to tailor its behavior according to the preferences of its users.

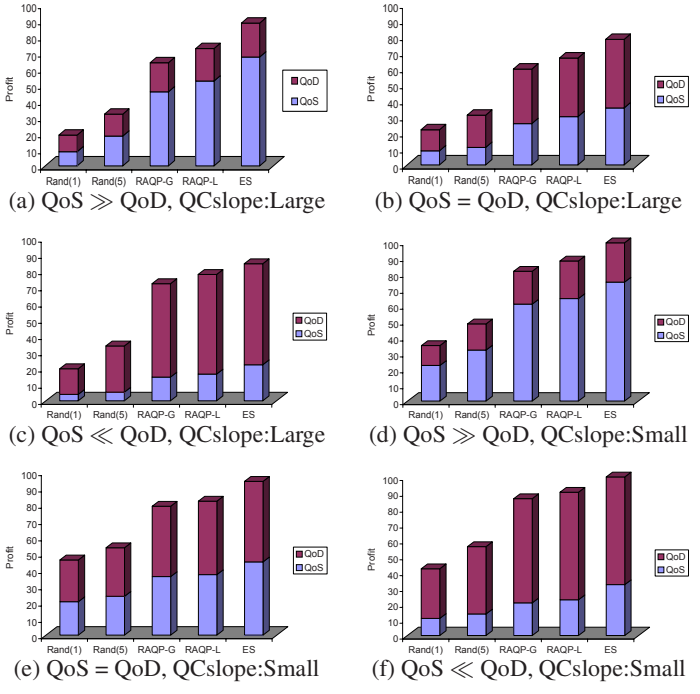


Fig. 5. Total profit of the algorithms under different classes

5 Conclusions and Future Work

In this work, we presented the Quality Contracts (QCs) framework that can be used to express user preferences for the QoS and QoD of submitted queries. QCs are based on the micro-economic paradigm and allow users to choose from a wide spectrum of QoS/QoD metrics, while, at the same time, indicating their relative importance. The QC framework is very intuitive, from a user perspective, and also provides a "clean" way for the system to quantify user preferences and allocate resources accordingly. It also integrates handling of both ad hoc and continuous queries that are crucial for real-time enterprises. Finally, we applied the QC framework in two different application domains: transaction scheduling in web databases and distributed query processing. For both cases, we introduced new algorithms that utilize the QC framework and also presented experimental results that illustrate the applicability of QCs and the high performance of our proposed algorithms.

Acknowledgments

This work was funded in part by NSF ITR Award ANI-0325353 and by NSF Award IIS-0534531. The authors also thank the anonymous referees for their helpful comments.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA (January 2005)
2. Abbott, R.K., Garcia-Molina, H.: Scheduling real-time transactions: a performance evaluation. ACM Transactions on Database Systems 17(3), 513–560 (1992)
3. Adelberg, B., Garcia-Molina, H., Kao, B.: Applying update streams in a soft real-time database system. In: Proc. of the 1995 SIGMOD conference, pp. 245–256, San Jose, California, United States (1995)
4. Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., Zdonik, S.: Retrospective on aurora. The. VLDB Journal 13(4), 370–383 (2004)
5. Carney, D., Getintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams: A new class of data management applications. In: Proc. of the 28th VLDB conference, pp. 215–226 (2002)
6. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, V.R.S., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, CA (January 2003)
7. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: a scalable continuous query system for internet databases. In: Proc. of the 2000 ACM SIGMOD Conference, pp. 379–390, Dallas, Texas, United States (2000)
8. Ferguson, D.F., Nikolaou, C., Sairamesh, J., Yemini, Y.: Economic models for allocating resources in computer systems. In: Market-based control: a paradigm for distributed resource allocation, pp. 156–183. World Scientific Publishing Co. Inc., River Edge, NJ, USA (1996)
9. Haritsa, J.R., Carey, M.J., Livny, M.: Value-based scheduling in real-time database systems. The. VLDB Journal 2(2), 117–152 (1993)
10. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: Proc. of the 1977 ACM SIGMOD Conference, pp. 171–182, Tuscon, Arizona, United States (1997)
11. Hong, W., Stonebraker, M.: Optimization of parallel query execution plans in xprs. In: Proc. of PDIS, pp. 218–225. IEEE Computer Society Press, Los Alamitos (1991)
12. Kang, K.-D., Son, S.H., Stankovic, J.A.: Managing deadline miss ratio and sensor data freshness in real-time databases. IEEE Transactions on Knowledge and Data. Engineering (TKDE) 16(10), 1200–1216 (2004)
13. Kossmann, D.: The state of the art in distributed query processing. ACM Computing Surveys (CSUR) 32(4), 422–469 (2000)
14. Labrinidis, A., Roussopoulos, N.: Webview materialization. In: Proc. of the 2000 ACM SIGMOD Conference, pp. 367–378, Dallas, Texas, United States (2000)
15. Labrinidis, A., Roussopoulos, N.: Balancing performance and data freshness in web database servers. In: Proc. of the 29th VLDB Conference, pp. 393–404 (September 2003)
16. Labrinidis, A., Roussopoulos, N.: Exploring the tradeoff between performance and data freshness in database-driven web servers. The. VLDB Journal 13(3), 240–255 (2004)
17. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation in a data stream management system. In: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, CA (January 2003)
18. Qu, H., Labrinidis, A.: Preference-aware query and update scheduling in web-databases. In: Proceedings of the, International Conference on Data Engineering (2007)

19. Ramamritham, K., Stankovic, J.: Scheduling algorithms and operating systems support for real-time systems. In: Proceedings of the IEEE, vol. 82(1), pp. 55–67 (1994)
20. Sharaf, M., Chrysanthis, P.K., Labrinidis, A., Pruhs, K.: Efficient scheduling of heterogeneous continuous queries. In: Proc. of 32nd VLDB Conference, Seoul, Korea (2006)
21. Stonebraker, M., Aoki, P.M., Litwin, W., Pfeffer, A., Sah, A., Sidell, J., Staelin, C., Yu, A.: Mariposa: a wide-area distributed database system. The. VLDB Journal 5(1), 48–63 (1996)
22. Sutherland, I.E.: A futures market in computer time. Communications of the ACM 11(6), 449–451 (1968)
23. Xu, J., Labrinidis, A.: Replication-aware query processing in large-scale distributed information systems. In: Proc. of the Nineth International ACM Workshop on the Web and Databases (WebDB'06), Chicago, IL, United States (2006)