MANAGING QUERY AND UPDATE TRANSACTIONS UNDER QUALITY CONTRACTS IN WEB-DATABASES

by

Huiming Qu

B.E, Northeastern University, P.R.China, 2000

Submitted to the Graduate Faculty of

Arts and Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2007

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Huiming Qu

It was defended on

August 31, 2007

and approved by

Dr. Alexandros Labrinidis, Department of Computer Science

Dr. Panos K. Chrysanthis, Department of Computer Science

Dr. Daniel Mossé, Department of Computer Science

Dr. Ming Xiong, Bell Laboratories

Dissertation Director: Dr. Alexandros Labrinidis, Department of Computer Science

ABSTRACT

MANAGING QUERY AND UPDATE TRANSACTIONS UNDER QUALITY CONTRACTS IN WEB-DATABASES

Huiming Qu, PhD

University of Pittsburgh, 2007

In modern Web-database systems, users typically perform read-only queries, whereas all write-only data updates are performed in the background, concurrently with queries. For most of these services to be successful and their users to be kept satisfied, two criteria need to be met: user requests must be answered in a timely fashion and must return fresh data. This is relatively easy when the system is lightly loaded and, as such, both queries and updates can be executed quickly. However, this goal becomes practically hard to achieve in real systems due to the high volumes of queries and updates, especially in periods of flash crowds. In this work, we argue it is beneficial to allow users to specify their preferences and let the system optimize towards satisfying user preferences, instead of simply improving the average case. We believe that this user-centric approach will empower the system to gracefully deal with a broader spectrum of workloads.

Towards user-centric web-databases, we propose a Quality Contracts framework to help users express their preferences over multiple quality specifications. Moreover, we propose a suite of algorithms to effectively perform load balancing and scheduling for both queries and updates according to user preferences. We evaluate the proposed framework and algorithms through a simulation with real traces from disk accesses and from a stock information website. Finally, to increase the applicability of Quality Contracts enhanced Web-database systems, we propose an algorithm to help users adapt to the Web-database system behavior and maximize their query success ratio.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS xi

1.0 INTRODUCTION 1

1.1 Architecture 1

1.2 Performance bottlenecks of data intensive web applications 2

1.3 Challenges: Balancing QoS and QoD 3

1.3.1 Application 1: stock information web sites 3

1.3.2 Application 2: travel reservation web sites 4

1.4 How can User Preferences help? 4

1.5 Contributions and Outline 5

2.0 RELATED WORK 8

2.1 User preferences 8

2.1.1 Distributed services 9

2.1.2 User profiles 9

2.1.3 Data stream management systems 9

2.1.4 Real-time systems 10

2.2 System optimization 11

2.2.1 Load management 11

2.2.2 Scheduling 12

3.0 QUALITY CONTRACTS FRAMEWORK 15

3.1 Web database system model 15

3.1.1 Data model 15

3.1.2 Workload 16

3.1.2.1 Queries . . . 16
3.1.2.2 Updates . . . 16
3.1.3 Performance metrics . . . 16
3.1.3.1 Quality of Service (QoS) . . . 16
3.1.3.2 Quality of Data (QoD) . . . 17
3.2 Quality Contracts (QCs) . . . 19
3.2.1 Existing schemes and motivation . . . 19
3.2.2 Quality Contract illustrations . . . 20
3.2.3 Quality Contract example . . . 22
3.2.4 Usability of Quality Contracts . . . 23
4.0 LOAD BALANCING (SERVER) . . . 25
4.1 Background and Definitions . . . 26
4.1.1 User Satisfaction Metric - simplified Quality Contract . . . 26
4.1.1.1 USM Range . . . 29
4.1.2 Queries and Updates . . . 30
4.2 UNIT load management . . . 30
4.2.1 System Overview . . . 30
4.2.1.1 Data flow . . . 31
4.2.1.2 Control flow . . . 31
4.2.2 Load Balancing Controller (LBC) . . . 32
4.2.3 Query Admission Control (AC) . . . 33
4.2.3.1 Query deadline check . . . 33
4.2.3.2 System `USM` check . . . 33
4.2.3.3 Tighten/Loosen Admission Control . . . 34
4.2.4 Update Frequency Modulation (UM) . . . 34
4.2.4.1 Degrading Updates . . . 34
4.2.4.2 Upgrading Updates . . . 36
4.3 Experiments . . . 37
4.3.1 Experimental Setup . . . 37
4.3.2 Update Frequency Modulation Evaluation . . . 39

4.3.3 Naive USM: Quantitative Evaluation . . . 41
4.3.4 Normal USM: Sensitivity Evaluation . . . 43
4.3.5 Insight into behavior of UNIT . . . 44
4.4 Summary . . . 45
5.0 SCHEDULING (SERVER) . . . 46
5.1 Motivation and Definitions . . . 47
5.1.1 Impact of CPU Scheduling . . . 47
5.1.2 Linear/Step Quality Contracts . . . 48
5.2 Baseline Algorithms . . . 50
5.2.1 Single Priority Queue . . . 50
5.2.2 Dual Priority Queue . . . 51
5.3 QUTS Scheduling . . . 52
5.3.1 Theoretical CPU Allocation . . . 53
5.3.1.1 Model the total profit with Query CPU allocation ρ . . . 53
5.3.1.2 The optimal ρ to maximize the total profit . . . 54
5.3.2 Implementation of CPU allocation ρ . . . 55
5.4 Experiments . . . 57
5.4.1 Experimental Setup . . . 57
5.4.1.1 Query and Update Traces . . . 57
5.4.1.2 System Parameters: τ and ω . . . 60
5.4.1.3 Performance Metric . . . 61
5.4.2 Performance Comparison . . . 62
5.4.2.1 Step QCs vs. Linear QCs . . . 62
5.4.2.2 Performance under different QCs . . . 65
5.4.3 Adaptability to User Preferences . . . 66
5.4.4 Sensitivity of QUTS to ω and τ . . . 68
5.5 Summary . . . 69
6.0 QUALITY CONTRACTS ADAPTATION (USER) . . . 70
6.1 System Architecture . . . 71
6.1.1 The Quality Contract (QC) Economy . . . 72

6.1.2 Server View 72
6.1.3 User View 73
6.1.4 Analysis of Baseline QC Adaptation Schemes 74
6.2 Adaptive Quality Contract Scheme (AQC) 76
6.2.1 Overbid Mode 77
6.2.2 Deposit Mode 80
6.2.3 How to switch between Deposit and Overbid mode 81
6.3 Experiments 82
6.3.1 Experimental Setup 82
6.3.2 Performance Comparison 82
6.3.2.1 Solo 83
6.3.2.2 Duet 87
6.3.2.3 Quartet 89
6.3.3 Population 91
6.3.4 Knowledge Scope 92
6.4 Summary 93
7.0 CONCLUSIONS AND FUTURE WORK 95
APPENDIX. QUIX SYSTEM DEMONSTRATION 97
BIBLIOGRAPHY 99

LIST OF TABLES

4.1 Table of Symbols for UNIT Load Balancing 29

4.2 Update Traces 38

4.3 `USM` weights with penalties < 1 for Figure 4.6(a) 43

4.4 `USM` weights with penalties > 1 for Figure 4.6(b) 43

5.1 Symbol Table of QUTS Scheduling 53

5.2 Pseudo-code of QUTS (Query-Update Time-Sharing) two-level Scheduling Algo. 56

5.3 Workload Information 57

5.4 Setup Indicators and Performance Metrics 61

5.5 Quality Contract Setup for Figure 5.7 62

5.6 Quality Contract Setup for Figure 5.9 64

6.1 Table of Symbols 76

6.2 **Relative Performance of AQC/RAN in Knowledge Scope.** With only 0.01% knowledge, AQC beats RAN by 30%. With 1% knowledge, AQC performs 85% better. 93

LIST OF FIGURES

1.1 Typical Web Site Architecture 2

1.2 Thesis Outline 6

3.1 General form of a Quality Contract 21

3.2 Quality Contract example 22

4.1 USM in the form of Quality Contracts with a discrete Admission function, a continuous QoS function, and a discrete QoD function. 27

4.2 UNIT Feedback Control System 31

4.3 Distribution of Queries over Data ID 40

4.4 Distribution of Updates over Data ID (Original vs. UNIT Degraded) 41

4.5 Performance Comparison when `USM` = Success Ratio 42

4.6 Non-zero Penalty Cost (med-unif trace) 44

4.7 Ratio Distribution 45

5.1 Impact of Scheduling on the Trade-off between Response Time and Staleness. 48

5.2 Quality Contract examples 49

5.3 QUTS Scheduling 55

5.4 **Trace A characteristics:** (a) query distribution has small changes over time despite one spike; (b) update distribution has downward trend over time with some fluctuations. 58

5.5 **Query vs. Update for each Stock:** log-log plot of the query and update frequency for each stock (depicted as circles). Stocks are concentrated below the diagonal because both traces have more updates than queries. 59

5.6 **Trace B characteristics:** (a) both query and update distributions are more stable than trace A; (b) the ratio between the number of queries and the number of updates is smaller than that of trace A (3.3 in B vs. 6 in B). . . . 60
5.7 Trace A: Profit Percentage with step and linear QC functions. 63
5.8 Trace B: Profit Percentage with step and linear QC functions. 63
5.9 Profit Percentage with Various QCs (FIFO and QUTS) 65
5.10 QoS/QoD Profit over Time 66
5.11 Total Profit and Query CPU Share ρ over Time 67
5.12 Sensitivity of QUTS over ω and τ 68
6.1 System Architecture 71
6.2 Performance of Baseline Algorithms 75
6.3 Effect of Overbid with FIX 75
6.4 Solo Environment: Success Ratio 83
6.5 Solo over Time (FIX and RAN) 85
6.6 Solo over Time (DYN and AQC) 86
6.7 Duet Environment: Success Ratio 87
6.8 Duet over Time 88
6.9 **Quartet Environment: 4 algorithms under different workload settings.** User view: the lighter the workload, the higher success ratio. Under high workload, AQC achieved 233% better performance than FIX, 155% than RAN, and 28% better than DYN. Server view: DYN and AQC utilized almost all their money, whereas FIX and RAN had a large portion wasted. 90
6.10 **Average performance for AQC and RAN with different populations.** As the percentage of AQC users increases, the performance of both algorithms decreases, since competition is more severe. 91
6.11 **Performance of AQC with different knowledge scope (Group Number shows how many groups exist).** AQC is stable; the performance improves (variance is eliminated) as the amount of sharing increases (i.e., the number of groups decreases). 92
A1 QuiX System Demonstration 98

ACKNOWLEDGEMENTS

Fore and most, I would like to thank my advisor, Alexandros Labrinidis. We joined the department in the same year, him as a professor, me as a student. I feel extremely lucky and thankful to find him as my advisor, without whose support and guidance all of these would not be possible. Alex has been a great researcher, a caring mentor and a sincere friend. As a researcher, he never neglects even the smallest detail in the work. This rigorous and thorough attitude towards research has benefited me deeply. As a mentor, he led me to database research, shaped my way of research thinking. His immense knowledge in databases has been a great source for my research. He always looks after people around him with great leadership and care. His cheerful personality has been my strongest support in this journey.

I would also like to thank my committee members, Panos K. Chrysanthis, Daniel Mossé, and Ming Xiong for spending their invaluable time reviewing the manuscript and providing insightful comments. In addition, thanks to Panos for his guidance in the early stage of my Ph.D. study, and to Daniel for his thoughtful input on the admission control work. Also, thanks to Kyoung-Don Kang who kindly shared his code to help me start the admission control project.

I am very grateful for being a member of the Advanced Data Management Lab where I had a wonderful time to learn and grow. Thanks to my lovely officemates, Qinglan Li and Jie Xu, with whom I shared so much laughter and tears through all these years. I would also like to thank my friends from University of Pittsburgh. The lunch in graduate lounge with Guanfeng Li, Huangyu Qiang, Shuyi Shao, Ruibin Xu, Haidong Xia, Min Zhao, and Shukang Zhou has been extremely entertaining and with all sorts of information which made my life so much easier.

I wish to thank all my friends in Pittsburgh, for being the surrogate family during the

five years I stayed there, and for their continued moral support there after. Special thanks to Yanna Shen, for her company as a best friend for 11 years.

Finally, I am forever indebted to my family. My mother Guizhen Li and my father Dianwen Qu have offered me endless encouragement and patience at all the times. My sister, Jinghua Qu, always gets me what I want most when I need those most. My husband, Jimeng Sun, who I owe my deepest gratitude to, has always believed in me, stood by me, and guided me through all the difficulties. With love, I dedicate this thesis to them.

1.0 INTRODUCTION

The wealth of information online has led to a myriad of data-intensive services and applications. Typical application of online information services include banking, shopping, booking travel, monitoring and trading of stock portfolios, aggregating blogs and news on specific topics, computer network monitoring (especially for intrusion detection), personalized weather forecasts, environmental monitoring (for example, USGS' National Water Information System Web Site), and many more. The Web has become an indispensable information portal.

Before we present our work, we introduce the architecture (in Section 1.1), the performance bottlenecks (in Section 1.2), and the challenges (in Section 1.3) of online information services through typical data-intensive web sites. Next, we advocate user preferences in solving the presented challenges in Section 1.4, then we present the contributions of this work along with the outline in Section 1.5.

1.1 ARCHITECTURE

A typical information portal has a front-end Web server and a back-end database (i.e., Web-database) as in Figure 1.1. Let us assume a stock information web-site, Quote.com, as an example. A user wants to check Dow Jones Industrial Average (INDU) and display it in real time. Once the user hits the "Go" button, the browser sends an HTTP request to the Web server containing the URL of the script and some parameters (INDU, real-time chart, etc.). The Web server then executes the script which involves (a) sending queries to the database server to create results from real-time feeds of INDU, and (b) assembling the results into an HTML page. Finally, the Web server sends the HTML page back to the client's browser.

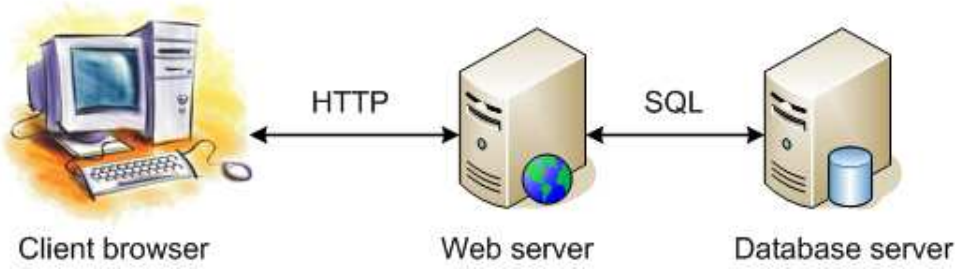


Figure 1.1: Typical Web Site Architecture

1.2 PERFORMANCE BOTTLENECKS OF DATA INTENSIVE WEB APPLICATIONS

In order to improve the user satisfaction, we have to identify and eliminate any performance bottlenecks. In this architecture, the performance bottleneck can occur anywhere: from the communication, to the Web server or the database server itself. The Web server and the database server may be connected with a high-speed LAN, whereas remote users may use a Dial-up internet which makes HTTP transmission a potential bottleneck, provided the server workload is low. In terms of the Web server and the database server, both the type of applications [9] and the mechanisms of implementing the Web server [23] affect where the longest delay happens. Applications that require complex business logic like an auction web site can easily stress the server front-end with a limited size of relational data, whereas data-intensive applications, such as benchmark TPC-W [1] for e-commerce applications, will give the back-end database server a much higher burden.

As the proliferation of networks and sensor devices is only going to make the volumes of collected data even more massive, resulting in what is being referred to as High-Fan-In systems [39], data-intensive web sites are expected to become even more data-intensive in the near future. As such, the corresponding back-end databases are going to face more intense challenges and are expected to become increasingly critical to the performance of online information services. In this work, we focus on the performance enhancement of web-databases so as to eliminate the possibility of over-stressing the back-end database server.

1.3 CHALLENGES: BALANCING QOS AND QOD

In web applications, Quality of Service (QoS) and Quality of Data (QoD) are of paramount importance to the end users. Quality of Service is essentially used to measure server throughput levels, such as response time. Quality of Data is used to measure the staleness of the requested data, such as the time elapsed since the last update. As we will see next, it is very hard to guarantee both qualities to the highest levels under the pressure of high loads.

Let us look at the challenge of balancing QoS and QoD as it materializes in two prevalent web applications that affect people's everyday lives.

1.3.1 Application 1: stock information web sites

Stock information web sites, such as E*Trade and Quote.com, receive stock ticks (i.e., updates on stock prices) on a regular basis, and users query current[1] stock prices (for example, a moving average of the stock price over the last half hour). Clearly, in such an environment, it is imperative to: (1) compute the results to a user's query based on fresh data and (2) give the user results as fast as possible. Results computed on old data can lead to misleading values (the market may have changed dramatically) and results that are delivered late can also lead to missed opportunities and financial losses. Although both high QoS and high QoD are desirable, these two performance metrics are always at a direct trade-off with each other. Answering queries without establishing the updates to the relevant stocks may speed up the response, whereas waiting until all updates are established will make answers fresher. When the server is lightly loaded, the response delay or freshness degradation may not be noticeable by the users. However, when there is a burst of updates, not only data gets stale very fast, but also queries wait longer to access fresh data. In this case, the server has to choose between maintaining high QoS (fast response) or high QoD (fresh data). In an unrealistic scenario, if the server could foresee when the queries and updates arrive as well as the read/write pattern on the stock information, balancing QoS and QoD could have been facilitated by delaying all the irrelevant updates. As a result, high QoD can be sustained

[1]For US stock market data, SEC regulations mandate giving only delayed data, by 15 to 20 minutes, to anonymous users. Registered users can receive up-to-update without restrictions.

without hurting much of QoS. However, as the workloads that Web servers are facing are unpredictable, balancing QoS and QoD is a real challenge with dynamic access patterns over data/time.

1.3.2 Application 2: travel reservation web sites

Online travel reservation web sites are utilized in order to get the best price and/or schedule from a vast amount of available options. Websites include official information portals for airline companies and hotel chains (such as United Airlines and Holiday Inn), as well as third party agents (such as Expedia and Hotwire). It is crucial that user queries on flight, car, and hotel information are answered in a timely fashion, using the most recent data possible. However, high QoS is hard to achieve because such dynamic information takes time to compile (e.g., pull updates from data sources and establish the changes in databases). The time it takes to return a ticket search result is especially long for those third party agents that try to obtain information from all kinds of sources. Users easily get frustrated by staring at the progress bar or the repeating patterns. At the same time, high QoD is also challenging to achieve because the data may get stale while being aggregated.

With the fast changing data at the sources and limited computing resources at the server, it is difficult to meet high demands on both QoS and QoD. As a result, the server may appear to have random performance degradation in terms of delays and stale information. Such uncharacteristic performance may seriously impair user satisfaction, thus reducing the website's popularity. The challenge of providing a consistent performance lies in balancing QoS and QoD according to the current system resources and user demands.

1.4 HOW CAN USER PREFERENCES HELP?

Commercial websites have started to realize the jeopardy of having unpredictable performance on both QoS and QoD. Some of them choose to consciously optimize either QoS or QoD by having fixed thresholds or guarantees on one metric and offer best effort on the

other metric. For example, Quote.com sets the QoD bar for unregistered general users by delivering stock trades that happened at least 15 minutes ago [83], whereas E*Trade offers guarantees on QoS by promising the user transactions to be completed within 2 seconds [36].

However, users may have different tolerance levels and perceive query qualities differently. For example, when checking on flight information, some users would prefer fast response times, 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).

Considering user preferences helps the server allocate the limited computing resources to enhance the most perceivable performance aspect to the users. Thus, the server is able to survive a large range of workloads without diminishing user satisfaction.

The importance of user preferences, although has received little attention in web-databases, has been recognized in many other areas, such as the utility functions in real-time systems [87, 16] and Service Level Agreements (SLAs) in web services [104] and Grid computing [20]. As prevailing as the user preferences in the above areas are, we believe that user preferences will improve the performance of web-databases and increase the effectiveness and the usability of web-databases even with increasing challenges from bursts of user requests or from the high volumes of information generated on the Web.

1.5 CONTRIBUTIONS AND OUTLINE

Towards embedding user preferences into web-databases, we propose the Quality Contracts (QC) framework, which enables users to express their preferences and to have the web-database performance optimization influenced by these individual user preferences. Specifically, the contributions of this dissertation are as follows:

- **Quality Contract framework** We introduce Quality Contracts (QCs), a unifying framework for expressing user preferences for QoS and QoD [59]. QCs are based on the micro-economic paradigm [38, 96], which allow users to specify their preferences among different

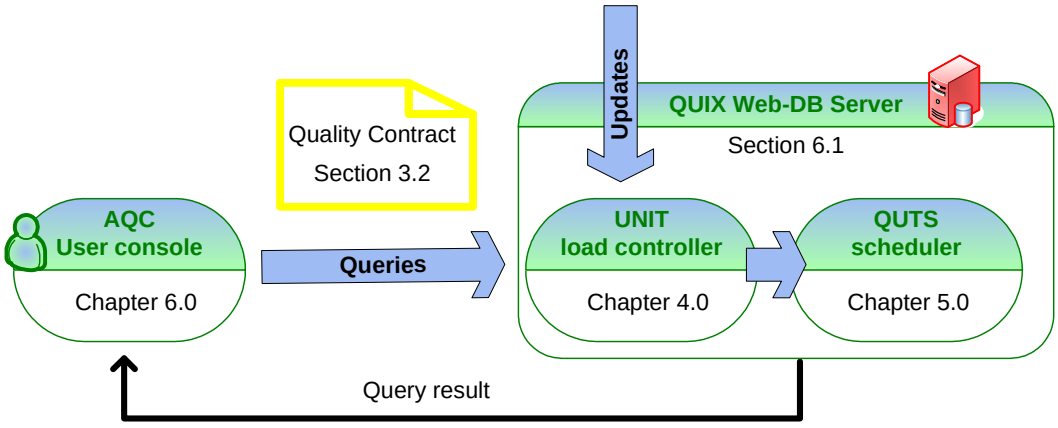


Figure 1.2: Thesis Outline

quality metrics by assigning the amount of "worth" for the corresponding performance expectation of each query.

- **UNIT load management** As a first step towards incorporating user preferences into the server optimization process, we first studied the load management of web-databases under a simplified QC framework. The load management layer can be used with any existing database system provided with sufficient log information. We proposed a suite of algorithms, UNIT [81], to maximize user satisfaction according to user specified QCs. UNIT guards the underlying web-databases so that the performance degradation due to the dynamic surges of high loads is minimized. Specifically, UNIT filters out the queries that are prone to hurt user satisfaction with limited system capacity, and discards the updates that are irrelevant to the users' interests. An experimental study using real disk traces showed that our algorithms outperform the current state of the art.

- **QUTS scheduling** To fully support the QCs framework, we further studied scheduling schemes within web-databases. We identified that the single-priority-queue scheduling is not feasible when both QoS and QoD preferences have to be considered. We proposed using a two-level scheduler, Query Update Time Share (QUTS), to separate the incomparable quality metrics of the QCs in a lower level and allocate CPU to queries and updates dynamically according to the comparable "worth" in QCs [80] in a higher

level. We compared QUTS with two other scheduling schemes with the same lower-level priority schemes and fixed CPU allocation on the higher level. Our extensive experimental study using real stock information web data show that QUTS performs better than baseline algorithms under the entire spectrum of QCs, QUTS adapts fast to changing workloads, and have little sensitivity of its own parameters.

- **QuiX system** We implemented the QCs framework with a functional user interface on the front end, and with both admission control and scheduling components simulated on the back end. To enhance the applicability of the QCs framework, we also explored various user strategies with the freedom of specifying their preferences and influence the server behavior. We demonstrated the QuiX prototype system in [82].
- **AQC user adaptation** We identified two important issues in how users should adapt their QC selections to maximize the number of queries executed within their satisfaction: *payment expectation* and *savings ability*. 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 switches between two modes: *Overbid mode*, which maximizes the utility of the user's budget with the empirical expectation of QC expenditure; and *Deposit mode*, which gracefully builds up user savings. AQC consistenly outperforms compared algorithms by up to 233%.

2.0 RELATED WORK

There is a plethora of papers that focus on improving the performance of user requests to database-driven web sites, using caching [34, 64, 66] or materialization [60]. These approaches save the time to generate pages dynamically by returning the result that was served for similar queries. The challenge is how to keep the cache or materialized view up-to-date: some approaches only invalidate the cache entry and recompute it upon the request of the next query [51], while others constantly update the cached materialized view if the base data changes [27, 60, 74]. This previous work usually provides a best-effort solution in terms of either QoS or QoD [61, 62], but it typically ignores the various individual user preferences.

To the best of our knowledge, our work is the first to combine individual web-database users' preferences for both QoS and QoD. However, despite their "absence" from web-databases, user preferences have been considered in many other areas, which have greatly inspired our work.

In the following, we first introduce existing frameworks that incorporate user preferences (in areas other than web-databases) and then discuss how work in these areas inspired our work, along the different system components,

2.1 USER PREFERENCES

Beyond web-databases, user preferences have played an important role in many other areas in different forms.

2.1.1 Distributed services

Mariposa [93, 96] was one of the earliest distributed systems which explored an economic model for trading and managing resources by taking into account users' QoS requirements in a distributed environment. Later, the main concepts from Mariposa were repurposed for Service Level Agreements (SLAs) [32, 58] in Grid applications [7, 10, 18, 19, 20, 21, 101] and Web-services [8, 10, 31, 55, 73, 104] which consider users' QoS requirements as well as resource availability, capability and cost for effective resource management and application scheduling in a Grid computing environment. Clearly our work has different applications from the above systems.

2.1.2 User profiles

Another important class of user preference frameworks are the user profiles proposed in [25, 57]. By analyzing users' data requirements, profiles are generated. The profile-driven data management framework then provides services to thousands of clients by interpreting, consolidating and processing user profiles. It is especially useful to "compile" users' data needs in mobile data management applications and information services [56, 72]. Our Quality Contracts framework focuses on how best to execute user queries, instead of which data are of interest (and thus need to be made available to mobile users).

2.1.3 Data stream management systems

Data stream management systems (DSMS) [3, 24, 30, 42, 70] deal with continuous queries and give results based upon the never-ending flood of data. DSMS face challenges similar to web-database systems in that: (1) massive data streams come into the system at a very fast rate; (2) (near) real-time monitoring and analysis of incoming data streams is required. To maximize the utilization of system resources, some DSMS, such as in Aurora project [3, 14] and Borealis project [2], allow QoS functions and QoD functions, where performance is mapped to values of importance. 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. These system-wide weights have 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. In our work, we give emphasis to *individual* user preferences.

2.1.4 Real-time systems

In real-time systems user preferences can be expressed on the time criticality of tasks through deadlines. In many real-time systems, *utility functions* [16, 87] are used to map the task response time to the value of the performance. The system thus concentrates on how to manage resources or schedule tasks, so that the most critical tasks are accomplished first and the overall performance generates the most value according to the utility functions [4, 16, 49, 94]. A general model for satisfying requirements of multiple quality dimensions in a resource-constrained environment has been proposed in [85], and solutions have been given to the problem of allocating a single resource to meet multiple independent quality constraints [86]. However, the two quality constraints in web-databases are not independent since QoS and QoD are at a trade-off of each other. Similar trade-offs have also been seen in the power-aware real-time systems, where energy and QoS are also at a trade-off of each other. Reward-based scheduling in power-aware real-time systems has been studied extensively in [69, 79, 89, 95, 105]. Web-databases differ from those real-time systems essentially because the updates in web-databases are not associated with any user preferences (or utility functions), whereas each task in real-time systems is associated with a deadline or utility function. Hence, prior work in real-time systems can only be applied on managing user queries and not updates.

2.2 SYSTEM OPTIMIZATION

In order to maximize user satisfaction while having limited system resources, we utilize user preferences to guide the system optimization through (1) load management to prevent the system from overloading, and (2) query and update scheduling according to the criticality of different user queries over multiple quality metrics. Previous work, especially those from real-time databases, has greatly influenced our work as we describe shortly.

2.2.1 Load management

Data Stream Management Systems (like Web-databases) are usually challenged by irregular data feeds. Under transmission and storage constraints, active load shedding is necessary to prevent losing important data or to reduce the response time without sacrificing much of the result accuracy. Multiple load shedding techniques have been proposed to address bursty data streams. For example, [13, 33, 35] focus on the accuracy of the query answers, whereas [97] provides a mechanism to optimize on either latency, value-difference, or loss-tolerance. [99] proposed a control-based load shedding scheme to deal with the busty data input and variable unit processing cost. In the LoadStar system [26], statistical models are utilized to maximize the quality of stream mining results when load shedding has to be performed. The major task of load management in DSMS is to prevent the data from overflowing through the DSMS. However, web-databases are facing the challenges from both user queries and data updates. When necessary, admission control on both queries and updates has to be performed, which we address in this work.

Load management has also been applied in many real-time applications [65, 84]. In [17], authors proposed to treat periodic real-time tasks as springs, so the period (and also the workload) can be adjusted by changing the elastic coefficients to better conform to the actual load conditions. In [103], a deferrable schedule for update transactions is used to minimize update workload while maintaining freshness (temporal validity) of real-time data. QMF is proposed in [54] for real-time databases to main a global target on freshness and miss ratio with a feedback control loop. The work mentioned above either considers only individual

QoS in terms of the task deadline, or considers both QoS and QoD but with a global quality goal. Our work differs from their work in considering individual user preferences on both QoS and QoD.

2.2.2 Scheduling

Scheduling queries and updates under Quality Contracts in web-databases is closely related to transaction scheduling in real-rime databases systems due to the similarity between utility functions and the QoS part of Quality Contracts.

Real-time databases [87, 94] are where real-time systems meet databases. Traditional databases usually schedule transactions to minimize the average response time while ignoring the individual real-time constraint (deadline) for each transaction. Real-time systems have strong support for deadlines, but typically ignore the data consistency problems. Research in real-time databases is trying to address both real-time scheduling and data consistency.

Deadline-driven scheduling There are three kinds of real-time systems: *hard real-time systems*, where the deadlines have to be met for all transactions otherwise catastrophic effect entails; *firm real-time systems*, where transactions have no value to the system if their deadline are missed, thus those transactions are discarded when their deadline expires; and *soft real-time systems*, where transactions still have some value to the systems although they miss their deadlines, thus transactions are kept alive even after they miss their deadlines. Because of the unpredictable data access patterns, it is usually very hard to guarantee that all the deadlines are met. Thus, real-time databases normally adopt firm or soft deadlines. There is a lot of prior work that attempts to match the response time to a value function, called utility functions [102, 85, 106]. [88] provides a comprehensive survey on recent advances in time/utility function real-time scheduling and resource management. The basic idea of scheduling under utility functions is to consider both deadlines and values (or profit) of the tasks, such as assigning higher priorities to the transactions that have higher values and tighter deadlines [4]. Existing work on real-time transaction scheduling [4, 16, 46, 49, 76] can be applied in real-time databases provided that each transaction is associated with a utility function (or a deadline).

Concurrency control In conventional databases, there are two prevailing concurrency schemes: Two Phase Locking (2PL) and Optimistic Concurrency Control (OCC). People have extended the concurrency control schemes in both firm-deadline and soft-deadline real time database systems [4, 43, 44, 50, 67, 68, 90, 91]. Earlier studies [44] have shown that OCC schemes perform better than 2PL-based schemes in firm deadline systems, because in such systems tasks that miss their deadlines are discarded immediately, which is very beneficial to OCC's late conflict resolution. On the other hand, [43] showed that 2PL-HP [4] (Two Phase Locking - High Priority, which solves any conflict in favor of high priority tasks) outperforms others with finite resources in real-time database systems that keep tasks continue to run even after deadlines.

Update management Compared with real-time systems, Web-databases face one more challenge: managing the write-only updates in addition to read-only queries. Traditionally, real-time scheduling makes use of time critical value functions or simply deadlines to decide transaction priorities. However, only query transactions are associated with those weighted constraints in Web-databases which makes traditional scheduling useless for the update priority assignment. An effective and efficient update scheme has to be established to facilitate query scheduling.

There is prior work that deals with only the scheduling of updates [61], especially in the context of web crawling [28, 75]. Other prior work has focused on how to reduce the update workload [6, 17, 54, 103] as we introduced in Section 2.2.1, but very few have addressed scheduling updates and queries together. Adelberg has studied some basic techniques in [5] which showed that there are two promising schemes: update first (update always have higher priority than queries) and update on-demand (relative updates are executed only when queries find the needed data items are stale). These static schemes, although have predictable performance (both guarantee 100% query freshness), do not have the flexibility to consider user preferences and adapt to the changing system workload.

Other than real-time databases, query scheduling has also been studied extensively in DSMS. [12] concentrates on minimizing the inter-queue memory, whereas others (that are either operator based [22] or tuple based [11, 98]) aims at improving response time. Sharaf et al. have also developed scheduling policies [92] that are shown to optimize the average case

of different criteria (response time, stretch, etc) and also introduced policies that successfully strike a balance between the average and worst case. The focus of this dissertation is in Web-databases and in scheduling for ad-hoc queries instead of continuous queries as in DSMS.

3.0 QUALITY CONTRACTS FRAMEWORK

3.1 WEB DATABASE SYSTEM MODEL

For highly scalable information portals like stock information web sites or travel reservation web sites, real time response is critical. We believe that a main-memory database system [53] is the most suitable back-end support type for such web sites because:

1. main-memory database systems avoid having disk accesses in the critical path of serving user requests which is a well known bottleneck for database systems; and
2. although the source information (e.g., stock ticks) are vast and fast changing, the information services usually only need to provide a snapshot of the source instead of the full history of all source data and, as such, storage requirements are not as big.

3.1.1 Data model

We assume that data items are hash-based accessed [15] and updated periodically or aperiodically by external sources. External sources are obligated for maintaining the master copy and the whole history of updates on each data item. For example, in the case of stock information websites, the external sources are the service providers like NYSE and Nasdaq where the trades are executed. These external sources are responsible for maintaining the entire history of updates of each stock as well as pushing the updates (as stock ticks) to the registered information portals like Stock.com. We assume that data items are independent of each other and only the most recent updates need to be maintained in the web-database (i.e., data items are independently refreshed).

3.1.2 Workload

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.

3.1.2.1 Queries In general, a web page with dynamic content is created from the results of one or more queries [78]. These queries are either executed serially one at a time or concurrently and in groups after parsing the entire web page [77]. In this work, without loss of generality, we assume that the content of a web page is generated by a single query. In other words, we assume that each user request is fulfilled through one user query. Each query can be attached with its user preference in the form of a *quality contract*, which we describe in Section 3.2. The Web server needs to provide the users with the appropriate interface to set up the quality contracts or choose the QC adaptation schemes. Queries in our system can be selection, projection, join, or aggregation queries on multiple data items. Each user query can be associated with user preferences corresponding to the expected quality of the query.

3.1.2.2 Updates Updates in our system are assumed to be *blind*, since the update stream is coming directly from external sources. As a result, no examination of the data state is needed before performing the updates. We also assume that updates are idempotent and contain no order constraints. Each update refreshes one data item. Users are only interested in the most recent value, thus, we do not need to process all pending updates on the same data item. In other words, 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* which maps data items to the current pending updates. Invalidated updates are simply dropped from the system without violating data correctness.

3.1.3 Performance metrics

3.1.3.1 Quality of Service (QoS) QoS is used to collectively represent metrics that measure the system throughput of the web-database server, such as *response time* and *stretch*.

Response time is the delay from the time the query is issued to the web-database to the time the query result is returned[1] (i.e., time elapsed while the query is within the web-database). Response time has been widely accepted as the primary QoS metric in on-line scheduling systems. In systems with highly variable job sizes, stretch has also been adopted to relate the waiting times to user demands [71]. The stretch (also known as the slowdown) of a request is the ratio of the response time of the request to the processing time of the request. Intuitively, it reflects users' psychological expectation that in a system with highly variable job sizes, users are willing to wait longer for larger requests. In our system, we use response time to measure QoS of a user query.

3.1.3.2 Quality of Data (QoD) In our model, web-databases maintain data replicated from external sources, and, as such, it may get stale if the synchronization is not performed in time. The query result becomes stale when the query is computed based on stale data. QoD denotes the staleness/freshness of the query result. There are two types of measurements of staleness: (a) *binary*, where the staleness of a query result either 0 or 1 depending on whether it accessed stale data items; and (b) *staleness degree*, where the staleness of a query result is a real number depending on how many stale data items the query accessed and how stale each accessed data item was. We use the fine-grained staleness degree in this work. As we can see, query staleness depends on data staleness. Two questions need to be answered for measuring query staleness:

1. How to measure the staleness/freshness of the individual data items that were accessed by the query?
2. How to aggregate the data staleness over the multiple data items that are accessed to compute the query result?

In the following, we first discuss the computation of data staleness (to answer the first question), followed by the query freshness (to answer the second question). Last, we define the counter part of the staleness: data freshness and query freshness.

[1]We do not include the network component in the response time measurement. As we have discussed in the introduction, this dissertation is focused on the performance of the web-database server.

- **Data staleness:** Suppose query q_i is computed based on data set D_i. The staleness of data item d_j, where $d_j \in D_i$, can be computed in multiple ways. The computation basically falls into three distinct classes: *time-based*, *lag-based*, and *divergence-based* [62]. Time-based methods use the time elapsed from the previous update to quantify how stale a data item is. Such time duration includes both the communication time between the main databases (i.e., the external sources) and replication databases (i.e., the web-databases) and the processing time of the updates in the replication databases. Since web-databases have no control over the communication time, time-based methods appear to be inappropriate for measuring QoD in web-databases. Lag-based methods use the number of unapplied updates to quantify how stale a data item is. The assumption is that the more updates pending (or missing) for a data item, the more stale the current data item is. Finally, divergence-based methods compare the current version of a data item with the most up-to-date version and quantify the difference in values. The assumption is that the bigger the difference, the more stale the current data item is. Both lag-based and divergence-based approaches are applicable in measuring QoD in web-databases. The choice is application depended. We adopt the lag-based approach in this dissertation for its simplicity. Specifically, the staleness of a data item is measured as:

$$\text{staleness}(d_j) = UU_{d_j}, \tag{3.1}$$

  where UU_{d_j} is the number of updates that are dropped since the last successful update, i.e., the number of unapplied updates. This can be tracked in the update register table which is used for invalidating the pending update for each data item.

- **Query staleness:** Once we have the staleness of each data item computed, computing the query result staleness is up to the aggregation function which can be the *average*, the *minimum*, or the *weighted sum* of the individual data staleness values. The choice of the aggregation function is application-specific. In this work, we take a strict approach by using the *maximum* staleness over all the recorded staleness of the accessed data items. Formally,

$$\text{staleness}(q_j) = \max_j(UU_{d_j}), \quad \forall d_j \in D_i, \tag{3.2}$$

where UU_{d_j} is the number of unapplied updates for data item d_j, and D_i denotes the accessed data set of query q_i. Using the maximum provides a strong guarantee that all data items accessed in order to compute the query result have no greater staleness than the overall staleness for the query.

- **Data freshness:** Data freshness is basically the reverse of data staleness. Formally, the freshness of data item d_j is computed as:

$$\text{freshness}(d_j) = \frac{1}{1 + UU_{d_j}}. \tag{3.3}$$

The staleness of a data item is normalized to a real number within (0, 1]. When an enormous number of updates are missed as the data item is accessed, the data freshness value is near 0; whereas in a case of missing no updates, the data freshness reaches its maximum value of 1.

- **Query freshness:** To guarantee that the query freshness is at least as fresh as any data item it accesses, query freshness uses the minimum freshness of all accessed data items. Specifically,

$$\text{freshness}(q_i) = \min_j(\frac{1}{1 + UU_{d_j}}), \quad \forall d_j \in D_i. \tag{3.4}$$

In the following chapters, we use the notation freshness/staleness interchangeably due to the fact that both freshness and staleness are computed based on the number of unapplied updates.

3.2 QUALITY CONTRACTS (QCS)

3.2.1 Existing schemes and motivation

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 [62, 54]. 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 as in [2, 3], where the individual metrics are combined using a set of weights to embody the importance of different metrics. However, this approach usually follows a system-wide standard which does not consider the different preferences of individual users over the importance of different quality metrics (e.g., the importance of response time versus staleness).

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., prefer high 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*, or QCs for short. Quality Contracts are based on the microeconomic paradigm [96, 38, 29, 63] and can 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 along with their quality concerns and allocate the system resources to maximize the worth to the user, or, the "profit" to the system. With QCs, we can now cast the problem of system optimization according to user preferences into the problem of maximizing the total profit for the system.

3.2.2 Quality Contract illustrations

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

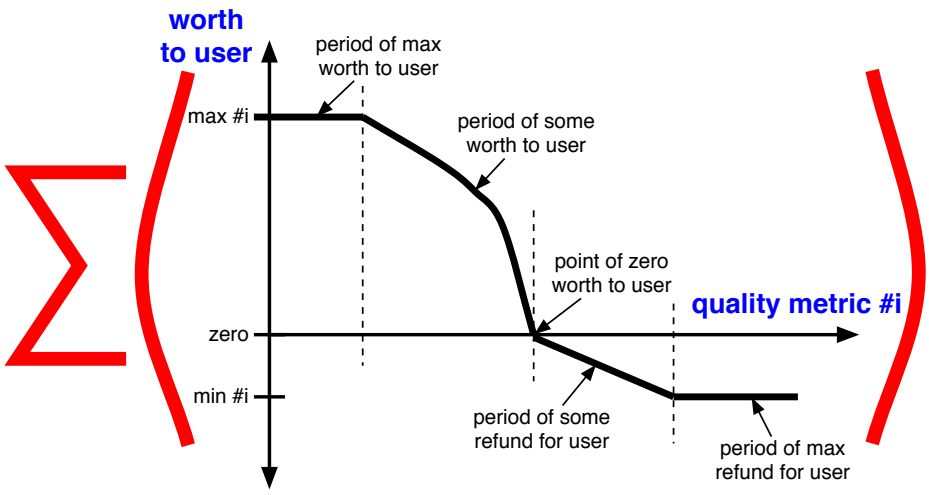


Figure 3.1: General form of a Quality Contract

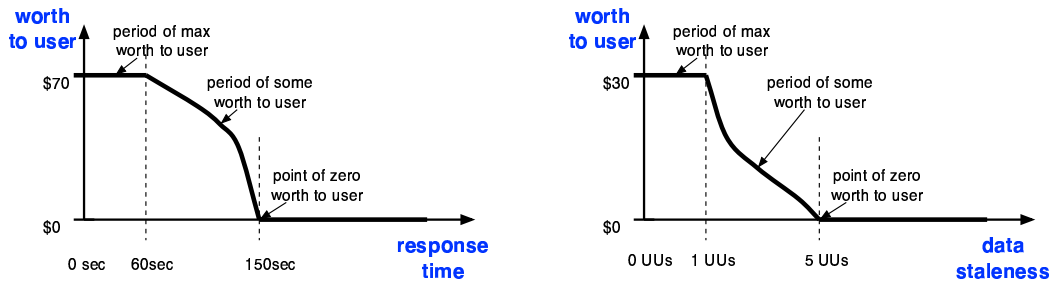
return for their service. In order to execute a query however, both the user and the server must agree on a Quality Contract (QC). The QC essentially specifies how much money the server that executes the query will get. The amount of money allocated for the query is not fixed. Instead, the amount of money the server receives depends on how well it executes the user's query. In the general case, QCs can also 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 being executed properly later).

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 3.1. Each graph represents one quality requirement from the user. The X-axis corresponds to an attribute that the user wants to use for different quality measurements, which could be response time, query staleness, or others of interest to the users. The value of the attribute

could either be continuous or discrete. 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, the user must provide additional virtual money to the server. The server optimization aims at obtaining the sum of all max amounts of the different QC graphs that a user submits along with a query. Nonetheless, if the query is not satisfactory completed, the server has to issue refunds based on the query-specific performance. 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.

3.2.3 Quality Contract example



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

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

Figure 3.2: Quality Contract example

Figure 3.2 is an example of Quality Contract (QC) for a query submitted by a user. This QC consists of two graphs: a QoS graph (Figure 3.2a) and a QoD graph (Figure 3.2b). 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. QoS is measured in response time (seconds), whereas QoD is measured in number of unapplied updates (UUs). Each graph (or function) has at least two critical values: max worth and zero worth quality constraint (i.e., $70, 150sec for QoS function and $30, 5UUs for QoD function). With a virtual money economy, users can now easily specify the relative importance of each component of the overall quality by allocating the query budget accordingly.

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**: QoD profit is added to the total profit of a query only if the QoS profit is more than zero (i.e., the query commits within the zero worth constraint). This appoints QoS as the prerequisite of considering QoD. The server adopting the QoS-Dependent scheme has no gain in keeping a query around if the query exceeds its maximal allowed response time defined in its QoS function.
- **QoS-Independent**: QoD profit is added regardless of the value of QoS profit, but the query still has to be completed by a *maximum lifetime* deadline to avoid keeping queries in the system forever. The server adopting the QoS-Independent scheme may not drop queries even if they no longer have QoS profit.

3.2.4 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). Using QCs in this way, service providers can better provision their systems, provide different classes of service, and allow end users to specify their preferences with minimal effort.

Although in this work we align QoS to response time and QoD to staleness, the Quality Contracts framework is general enough to allow for any quality metric including virtual attributes. Virtual attributes are computed over other attributes, possibly with statistics of the entire system. For example, user can specify QoS as the delay his/her queries received when compared to the average delay in the system. Such "comparative" QoS metrics some-

times is more intuitive: it is probably harder for 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.

Furthermore, we believe that the notion of Quality of Data can be extended in multiple ways. First, 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 [48], 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.

4.0 LOAD BALANCING (SERVER)

As mentioned earlier, Web servers are often characterized by their unpredictable access patterns over data/time, which typically translates to periods of peak request load. Web-database servers must be prepared to deal with such bursty accesses and balance the trade-off between query timeliness and data freshness.

In order to effectively utilize the resources of the Web-database server in times of peak load, we need to shed some of the load on the server. Since load on the server is due to both user queries and background updates, shedding load can be done in two ways: by dropping some of the user queries or by reducing the amount of the updates. Although performing all updates will guarantee the highest level of freshness for any user query, dropping some of the updates does not necessarily lead to decreased query freshness because data access pattern is usually not uniform. Revisiting the earlier stock monitoring example, if a stock receives hundreds of updates within a second and is only accessed once through a user query, we could easily ignore all updates until the last one before the access, without losing query freshness from the user's point of view.

In this chapter, we apply the idea of Quality Contracts to perform load shedding on queries and updates according to user preferences. Specifically,

- We use the User Satisfaction Metric (`USM`), a parameterized version of the general Quality Contracts, to represent user preferences over the different outcomes of user queries.
- We propose a suite of algorithms to maximize `USM` when system resources are not enough to guarantee a hundred percent of both freshness and timeliness. Specifically, we provide two algorithms: an *admission control* algorithm and an *update frequency modulation* algorithm. The admission control algorithm adjusts the user query workload by dropping

those transactions which threaten the system USM. The update frequency modulation algorithm adjusts the update workload by intelligently reducing the frequency of updates to data that have minimal harm to the overall user-perceived freshness. When and which workload to adjust depends on the decisions made by a general feedback control loop.

- We compare our proposed algorithms to two baseline algorithms and the current state of the art [54] with an extensive simulation study using workloads generated from real disk traces.

4.1 BACKGROUND AND DEFINITIONS

4.1.1 User Satisfaction Metric - simplified Quality Contract

We use the *User Satisfaction Metric* (USM) to study admission control. USM is a simplified version of Quality Contracts (introduced in Section 3.2.2). Under USM, users still specify their preferences for difference dimensions of quality (i.e., rejections, response time, and freshness). However, the preferences are parameterized using weights instead of being mapped through functions or graphs as in the full-fledged QCs. The reason of the simplification is to have an admission control that is general enough to work with database servers that do not support Quality Contract graphs. The admission control layer takes the responsibility of minimizing the disparity caused by the dynamic load challenges when the system cannot fully satisfy the user preferences indicated in USM. Web-databases only need to provide the log information and execution statistics for the admission control to take the appropriate and prompt reaction.

USM can been seen as generated from three Quality Contract functions: Admission function, QoS function, and QoD function, as shown in Figure 4.1. The QoS function depends on Admission function, meaning that only if a query is admitted can the system get any QoS profit from the query. Similarly, the QoD function depends on QoS function, meaning that only if a query generates positive QoS profit can the system get any QoD profit from the query. Since we are only using step functions, the dependency among functions actually

complicates the expression of user preferences. As summarized in the following, we find that a query can only have four outcomes under the circumstance. Thus, instead of having three functions with dependencies, we can just use a set of weights to express user preferences.

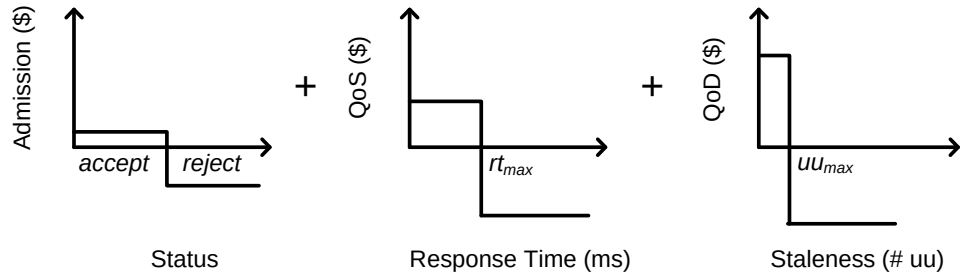


Figure 4.1: USM in the form of Quality Contracts with a discrete Admission function, a continuous QoS function, and a discrete QoD function.

Given the QoS and QoD constraints and function dependencies, we identify four possible outcomes for a user query:

- **Rejection** A query may be rejected from the system (i.e., the query did not pass the admission control phase). We refer to this case as a *rejection*.

- **Deadline-Missed Failure** A user query returns later than the maximal allowed response time, rt_{max}. We refer to this case as *Deadline-Missed Failure* (DMF).

- **Data-Stale Failure** Even if a query returns before rt_{max}, it will fail if it returns with staleness bigger than the maximal allowed staleness uu_{max}. We refer to this case as *Data-Stale Failure* (DSF).

- **Success** If a user query does not fail (for any of the above three reasons), it is considered *successful*.

Correspondingly, each outcome is associated with a weighting parameter. Thus, the *User*

Satisfaction of user query q_i, $\texttt{US}(q_i)$, could have four possible values as follows:

$$\texttt{US}(q_i) = \begin{cases} G_s^{(i)} & \text{if } q_i \text{ meets both } rt_{max} \text{and } uu_{max} \\ -C_r^{(i)} & \text{if } q_i \text{ is rejected} \\ -C_{fm}^{(i)} & \text{if } q_i \text{ fails to meet } rt_{max} \\ -C_{fs}^{(i)} & \text{if } q_i \text{ fails to meet } uu_{max} \end{cases} \tag{4.1}$$

where $G_s^{(i)}$ is the success gain of q_i, $C_r^{(i)}$ is the rejection penalty of q_i, $C_{fm}^{(i)}$ is the DMF (Deadline-Missed Failure) penalty of q_i, and $C_{fs}^{(i)}$ is the DSF (Data-Stale Failure) penalty of q_i. In this work, the penalties ($C_r^{(i)}$, $C_{fm}^{(i)}$, $C_{fs}^{(i)}$) are normalized to $G_s^{(i)}$. Thus, $G_s^{(i)}$ is 1.

We define the total *User Satisfaction Metric* of the system (over the set of all queries submitted by the users) as the sum of the *User Satisfaction* of each user query, $\texttt{US}(q_i)$:

$$\texttt{USM}_{total} = \sum_{q_i \in Q} (\texttt{US}(q_i)) \tag{4.2}$$

If we have N_s user queries that succeed, N_r that get rejected, N_{fm} that exhibit a deadline-missed failure, and N_{fs} that exhibit a data-stale failure, then by combining Equations 4.2 and 4.1 we have that:

$$\texttt{USM}_{total} = \sum_{k=1}^{N_s} G_s^{(k)} - \sum_{k=1}^{N_r} C_r^{(k)} - \sum_{k=1}^{N_{fm}} C_{fm}^{(k)} - \sum_{k=1}^{N_{fs}} C_{fs}^{(k)} \tag{4.3}$$

Now we have the four parts representing the gain and penalty according to the four outcomes of the transactions. If we divide the total `USM` by the total number of submitted user queries, we have the following average `USM`:

$$\texttt{USM} = S - R - F_m - F_s \tag{4.4}$$

which is the average success gain (S), deducted by average rejection cost (R), the average DMF cost (F_m), and the average DSF cost (F_s).

4.1.1.1 USM Range Higher values for the `USM` as defined in Equation 4.4, correspond to higher levels of user satisfaction. The maximum attainable value for `USM` is 1, for the case that all user queries are successful. The lowest possible `USM` value is $-\frac{\sum_i^N \max\ C_r^{(i)},\ C_{fm}^{(i)},\ C_{fs}^{(i)}}{N}$. In other words, the worst case scenario is when all the user queries fail and the type of failure matches what the users consider to be the most annoying (and have thus assigned to it the highest penalty).

Table 4.1: Table of Symbols for UNIT Load Balancing

Symbol	**Description**
rt_{max}	relative deadline of q_i, i.e., maximal response time allowed to prevent DMF
uu_{max}	freshness requirement of q_i, i.e., maximal staleness allowed to to prevent DSF
DMF	Deadline Missed Failure
DSF	Data Stale Failure
$\texttt{USM}_{total}$	total *user satisfaction*
`USM`	average *user satisfaction*
$\texttt{US}(q_i)$	*user satisfaction* of q_i
N_s	total number of successful transactions
N_r	number of rejected transactions
N_{fm}	number of DMFs
N_{fs}	number of DSFs
R_s	average success ratio
R_r	average rejection ratio
R_{fm}	average DMF ratio
R_{fs}	average DSF ratio
$G_s^{(i)}$	success gain for q_i, which is 1.
$C_r^{(i)}$	rejection cost for q_i, normalized to $G_s^{(i)}$
$C_{fm}^{(i)}$	DMF cost for q_i, normalized to $G_s^{(i)}$
$C_{fs}^{(i)}$	DSF cost for q_i, normalized to $G_s^{(i)}$

4.1.2 Queries and Updates

There are some additional facts about the characteristics of queries and updates in addition to what we have described in the system model (Section 3.1.2).

Each query is associated with three weighting parameters from `USM` (C_r, C_{fm}, C_{fs}) and two requirement parameters (the maximum allowed response time rt_{max} and maximum allowed staleness uu_{max}). Query q_i will be aborted if its running time exceeds rt_{max}.

We only consider periodic updates for the load management (Chapter 4), since most Web servers periodically pull updates or subscribe to update "feeds" being pushed from the source (e.g., NYSE, www.nyse.com). Notice that the work can be easily extended to sporadic updates because the staleness computation is lag-based. Essentially, increasing/decreasing updates allowed into the system is well aligned with decreasing/increasing data staleness. For example, to decrease the staleness of a data item, increasing the probability of admitting the updates of this data item (with sporadic updates) has similar effect with decreasing the update period of this data item (with periodic updates).

4.2 UNIT LOAD MANAGEMENT

Given the system `USM` as defined in the previous section, our goal is to maximize it by employing an adaptive load control scheme. This load control scheme should be able to sit upon general databases provided with query execution statistics. The idea is inspired by Kang's work [54] in which they use a feedback control loop to monitor the freshness and deadline miss ratio. We provide detailed comparison of our work to Kang's in Section 4.3.1.

4.2.1 System Overview

Figure 4.2 shows the overview of the feedback control system in UNIT, which is short for User-ceNtrIc Transaction management. There are two interrelated parts in this system: *data flow*, which corresponds to how queries and updates propagate through the system; and *control flow*, which corresponds to how the control process interacts with the data flow.

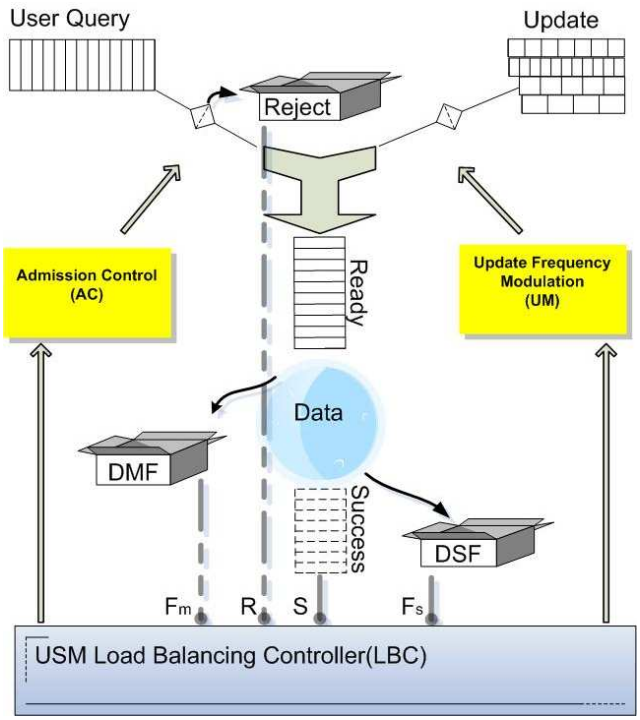


Figure 4.2: UNIT Feedback Control System

4.2.1.1 Data flow User queries and updates are submitted into the system and put into the *ready queue* if admitted. In general, the underlying databases can use any dispatching scheme. Without loss of generality, we assume the dispatching discipline used in the underlying database is a dual-priority queue: updates have higher priorities than queries, whereas within each group, preemptive EDF (Earliest Deadline First) is applied. When a query completes successfully (i.e., within deadline and freshness constraints), the query result becomes available in the *success queue*. The underlying database can also choose any concurrency control policy. We assume that Two-Phase Locking - High Priority (2PL-HP) [4] is adopted, since 2PL-HP is shown to outperform others with finite resources in real-time database systems that keep tasks continue to run even after deadlines [43].

4.2.1.2 Control flow The Load Balancing Controller (LBC) is responsible for regulating control flow. Specifically, it can tighten/loosen admission control by sending signals to the Query Admission Control module to allow less/more user queries into the system. LBC can

also increase/decrease the update frequency of updates by sending signals to the Update Frequency Modulation module to carry out more/less updates. The control flow is triggered periodically or when the USM shrinks by more than a certain threshold. The LBC also monitors queries for further load shedding. Since we assume firm deadlines, if a query deadline is missed while the query is in the ready queue or during its execution, the query has to be aborted. Similarly, query results with stale data items (DSF) can be discarded or returned to the user with a special notice.

4.2.2 Load Balancing Controller (LBC)

The Load Balancing Controller monitors the system statistics and initiates the Adaptive Allocation periodically or when there is a big drop of USM, that is, when ΔUSM is greater than a specified threshold; the threshold is usually 1% of the range of USM.

Algorithm 4.1: Adaptive Allocation Algorithm

Input : R_r R_{fm}, R_{fs}

Output: control signals

```
1  if ΔUSM > USM_t or Δtime >= Grace Period then
       //handle cases where all costs are zero
2      if C_r, C_fm,C_fs all equal 0 then
3          R = R_r; F_m = R_fm; F_s = R_fs
       //break ties randomly
4      switch max(R, F_m, F_s) do
           //rejection cost is highest
5          case R  Loosen Admission Control [Section 4.2.3]
           //DMF cost is highest
6          case F_m
7              Degrade Update [Section 4.2.4.1]
8              Tighten Admission Control [Section 4.2.3]
           //DSF cost is highest
9          case F_s  Upgrade Update[Section 4.2.4.2]
```

The policy executed by the LBC to decide on what to do (admit more/less user queries or improve/deteriorate the freshness through updates) is described in the Adaptive Allocation Algorithm (see Algorithm 4.1). The algorithm takes as input the rejection ratio R_r, the DMF ratio R_{fm} and the DSF ratio R_{fs}, and triggers a new control signal as the result. The main idea is to reduce the dominant penalty cost at the time a drop in the `USM` is detected. If C_r, C_{fm}, and C_{fs} are all 0, the system will focus only on reducing the failure with highest ratio to maintain a high success gain.

4.2.3 Query Admission Control (AC)

Query admission control filters out two types of user queries: those that have little chance to succeed by performing a *query deadline check*, and those that can significantly hurt the system performance by performing a *system `USM` check*.

4.2.3.1 Query deadline check We assume the average execution time for each query can be determined by the existing monitoring techniques that most database systems utilize for query optimization. These average execution times are used to check how promising a query is to finish on time before it is accepted. More specifically, for each query, the system keeps the Earliest-possible Start Time (EST). The system will check if $EST_i + e_i < rt_{max_i}$ before accepting the user query q_i, where e_i is the average execution time of q_i and rt_{max_i} is its relative deadline. Moreover, we also bring in a lag ratio C_{flex} to allow some flexibility to the scheduling; in other words, we check if $C_{flex} \cdot EST_i + qe_i < rt_{max}$. All the queries pass this test are called *promising queries*.

4.2.3.2 System `USM` check Usually, not all the *promising queries* are admitted, since they may overload the system. The system `USM` check considers the global impact of admitting a query, since the new query may delay the existing queries, which may lead to DMFs. We compute the consequence to the `USM` cost, by counting the total DMF cost of the endangered queries (i.e., the queries that might miss their deadlines due to the new incoming query). If the DMF of endangered queries is higher than the cost of rejecting the new query, the system rejects the incoming query.

The complexity of the admission control algorithm (query deadline check and system `USM` check) is $O(N_{rq})$ for each query, where N_{rq} is the length of the ready queue.

4.2.3.3 Tighten/Loosen Admission Control

The LBC will send TAC/LAC (Tighten / Loosen Admission Control) signals to adjust C_{flex} when needed. Notice that the larger the C_{flex} is, the tighter the Admission Control is. The initial value of C_{flex} is set to 1 and a TAC/LAC signal is to increase/decrease C_{flex} by 10%.

4.2.4 Update Frequency Modulation (UM)

We use Update Frequency Modulation (UM) to control the number of updates that are processed in the system by increasing or decreasing the frequency of updates. The reduction of updates is carried out on those data that have relatively little effect on query quality, upon receiving the *Degrade Update* control signal from LBC. Conversely, we increase the frequency of all degraded updates to help with the freshness, upon receiving the *Upgrade Update* control signal from LBC.

4.2.4.1 Degrading Updates

Next, we answer the following two questions:

(1) Which update to degrade?

(2) How aggressively to degrade the updates?

Which update to degrade? Intuitively, we want to degrade the updates for the data item that the system spends too much time updating, yet only few queries need to access. The probability a data item is chosen depends on its access pattern and update frequency.

We use Lottery Scheduling [100] to choose which data item to degrade, that is, whose update to make less frequent. Each data item is associated with a certain ticket value. The larger the ticket value a data item has, the higher the probability it will be chosen as the victim, and its update frequency will be decreased. The ticket value is decided as follows.

- ***Query effect on ticket values***: The ticket value of d_j is decreased every time there is a query access to d_j. The amount of decrease depends on the CPU utilization of the access query. Intuitively, we do not want to degrade those data items which are needed by the queries with high CPU utilization, because if the query's freshness requirement is not met, there will be less slack time for an additional update transaction to be issued to retrieve the fresh data item. Since the larger the ticket value is, the more chance it has to be degraded, for each query, the amount of decrease should be proportional to the CPU utilization. Formally, the amount by which the value will decrease for each query q_i accessing d_j is defined as:

$$DT_j = \frac{e_i}{rt_{max_i}} \tag{4.5}$$

  where e_i is the average execution time of q_i and rt_{max_i} is its relative deadline.
- ***Update effect on ticket values***: The system increases the ticket value of d_j whenever there is an update on d_j. The longer the execution time of the update, the larger the amount of increase on the ticket value. The idea is that we want to degrade those data items that have been updated relatively too often, and given two data items having the same number of updates, we want to obtain as much CPU time saving as possible by degrading the data item with longer update execution time. Among many choices to relate the ticket value to execution time, we use the *sigmoid function*. Using the information of average execution time among all the updates, the sigmod function smoothly converts the execution time to the (0, 1) range, and it nicely takes care of the effect of outliers. Formally, the amount by which the ticket value will increase for each update accessing d_j is defined as follows:

$$IT_j = \frac{1}{1 + e^{ue_{avg} - ue_j}} \tag{4.6}$$

  where ue_{avg} is the average update execution time of all updates in the system, and ue_j is the average execution time of update u_j. Note that the increase of the ticket value is the *sigmoid function* of the difference between execution time and average execution time.
- ***Forgetting***: In order to concentrate on current system status, we apply a forgetting factor C_{forget} to the computation of ticket values. With $C_{forget} = 1$, all historical accesses and updates are effective to the ticket values. The smaller C_{forget} is, the faster it forgets. We set $C_{forget} = 0.9$ in this paper, following the current practice in the literature [47].

- ***Overall ticket value computation***: The overall ticket value for data item d_j is computed as described below.

$$T_j = \begin{cases} T_j \cdot C_{forget} - DT_j & \text{from query } q_i \\ T_j \cdot C_{forget} + IT_j & \text{from update } u_j \end{cases} \tag{4.7}$$

In order to have non-negative ticket values for the Lottery Scheduling, we subtract the smallest ticket value, T_{min}, from all ticket values (i.e., $\forall j, T_j = T_j - T_{min}$). Then, we use Lottery Scheduling [100] that randomly picks a data item with probability proportional to the ticket value of the data item. The complexity of applying the Lottery Scheduling is $O(logN_d)$ [100], where N_d is the total number of data items.

How to degrade the update? Once d_j is chosen to be degraded, its current update period pc_j is increased with a certain percentage as specified in the following:

$$pc_j = pc_j \cdot (1 + C_{du}) \tag{4.8}$$

C_{du} is set to 0.1 in our experiments to gradually reduce the number of updates on d_j.

4.2.4.2 Upgrading Updates

Upgrading updates needs to be done when degrading updates affects the query freshness and the average DSF cost F_s becomes the leading cost in the `USM`. The periods of all degraded updates should be decreased as in Equation 4.9, until they are restored to the original period pi_j.

$$pc_j = \min(pi_j,\ pc_j - C_{uu} \cdot pi_j) \tag{4.9}$$

where $C_{uu} = 0.5$, in our experiments, to essentially cut the update period by half of original period and quickly restore the original update routines.

4.3 EXPERIMENTS

We evaluated UNIT by comparing it to different algorithms under various performance metrics and workloads (generated from real traces). Section 4.3.1 explains the experimental setup as well as the baseline algorithms. Section 4.3.2 evaluates UNIT's Update Frequency Modulation under different workloads. Section 4.3.3 quantifies the performance gain of UNIT to other algorithms. Section 4.3.4 evaluates how sensitive the algorithms are to the different weight settings. Finally, Section 4.3.5 provides further insight into the influence of cost factors over the different algorithms.

4.3.1 Experimental Setup

User Query Trace We generated user queries based on the HP disk *cello99a* [40] access trace, which captures typical computer system research disk workloads, collected at HP Labs in 1999. The trace lasts for 3,848,104 seconds and includes 110,035 reads. Each recorded entry in *cello99a* has its arrival time, response time, and location on the disk. We take the arrival time and response time of reads from the original trace and map their accessed logical block number (lbn) into our data set. The disk location was partitioned into 1024 consecutive regions, where each region represents a data item in our simulation. The deadline for each query was generated randomly and ranged from the average response time to 10 times of the maximal response time. We set freshness requirement for all user queries at 90%. Through the above process, we generate the user queries with each of them containing the arrival time, accessed data, estimated execution time, deadline and freshness requirements.

Update Trace Update workloads are classified into low, medium, and high workloads with 6144, 30000, and 61440 total updates, respectively, representing a 15%, 75%, and 150% CPU utilization. In general, we need to specify the spatial distribution (what data item to be accessed or updated) and temporal distribution (at what time it happens) of updates.

The spatial distribution essentially decides how many updates each data item gets, given

Table 4.2: Update Traces

Updates		**Traces**
Total Number	Distribution	
6144 (or 15% workload)	uniform	$low - unif$
	positive correlation	$low - pos$
	negative correlation	$low - neg$
30000 (or 75% workload)	uniform	$med - unif$
	positive correlation	$med - pos$
	negative correlation	$med - neg$
61440 (or 150% workload)	uniform	$high - unif$
	positive correlation	$high - pos$
	negative correlation	$high - neg$

the total number of updates on all data. The update workload could be uniformly spread out over all base data items, meaning that every data item has equal amount of updates. The update workload could also have some correlation with the user queries. We generate positively correlated updates to simulate a perfect correlation between the number of updates and the number of user queries on each data item, which means the total number of updates on each data item is proportional to the total number of accesses on that data item. Also, we generate negatively correlated updates to simulate the situation when more frequently queried data get relatively less updates. Since there is no direct way to generate the negative correlation between updates and queries, we assume the updates follow the power-law distribution [37]. Specifically, we rank the data items according to the ascending order of the numbers of queries on them. Then, based on this ranking, we generate the numbers of updates using the 80-20 rule [41] (i.e., 80% of the updates are generated for 20% of the data items). In short, we experimented with uniform, positive correlation and negative correlation on each update workload.

Once the number of updates for each data item are generated, we look at the temporal distribution of these updates to construct the update traces. Since we only consider periodic updates, the temporal distribution is uniform. Thus, the period for updating d_j is the ratio of the total number of updates for d_j over the whole simulation time.

With low, medium, high workloads and three types of spacial distributions, we generated nine update traces listed in Table 4.2. We generated estimated execution time for updates randomly in the range of the response time of writes in *cello99a*. Each entry contains an estimated execution time and an update period for a particular data item.

Baseline Algorithms We compared our scheme, UNIT, to two baseline algorithms (IMU and ODU) and the current state-of-the-art (QMF [54]).

- **IMU** (Immediate Update): All the updates are executed immediately; no admission control on queries. IMU achieves 100% freshness, but may suffer from low query success ratio for the high update load.
- **ODU** (On-demand Update): updates are executed only when a query finds that a needed data item is stale; no admission control on queries. ODU also achieves 100% freshness, but the additional update issued may also delay the query and lead to missed deadlines.
- **QMF**: [54] uses a feedback control loop to adjust admission control and adaptive update policy. With the CPU underutilized, QMF tries to update more often if the target freshness is not met, otherwise admits more queries. With the CPU overloaded, QMF updates less often if current freshness is higher than target freshness, otherwise drops incoming queries until the system recovers. The adaptive update policy controls how many updates to be dropped, and whose updates to be dropped (based on the ratio of number of accesses over number of updates on each data).

4.3.2 Update Frequency Modulation Evaluation

First, we want to verify if our Update Frequency Modulation can intelligently choose to drop updates that contribute little to the user query freshness. We show both the query access

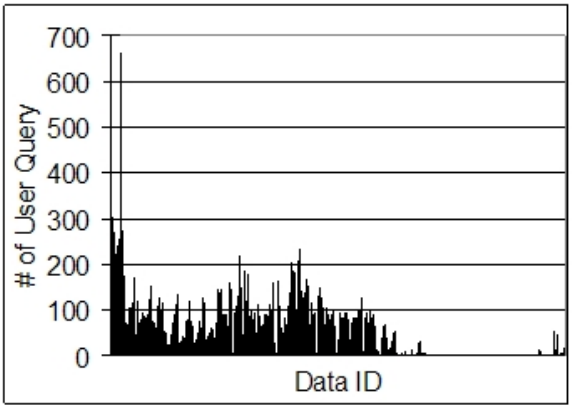


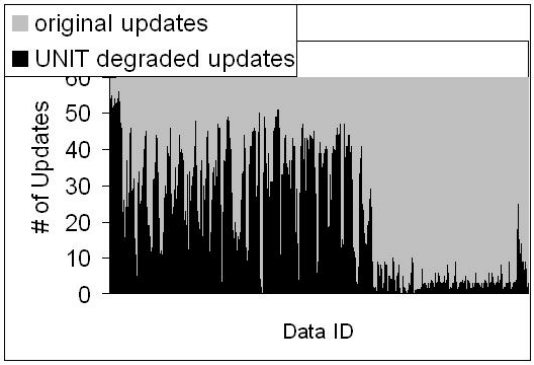
Figure 4.3: Distribution of Queries over Data ID

distribution over data and the update distribution over data. The results are from user query trace with update traces *med-unif* and *med-neg*.

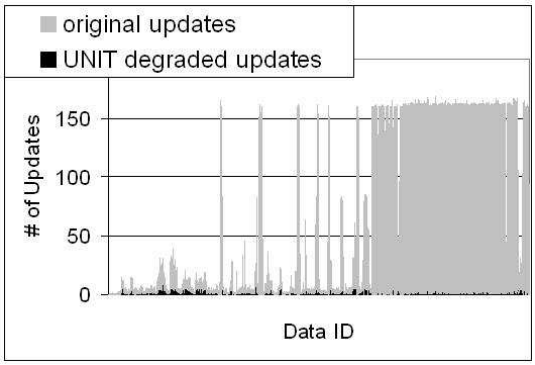
Figure 4.3 shows the number of queries per data item, illustrating a skewed distribution of requests over data items. Some data item has more than 650 query accesses, while some others almost get no query during the whole simulation time. Potentially, the web-database server can stop updating those data items that no query is interested in because they would not affect the user perceived QoD.

Case Study 1: med-unif Updates In this case study, the original update requests are distributed uniformly over all the data as indicated by the grey area in Figure 4.4(a). Intuitively, the system should cut down the updates on the less frequently queried items when necessary, which is exactly what UNIT does (see black lines in Figure 4.4(a)). By comparing black lines in Figure 4.3 and Figure 4.4(a), we can see that UNIT can adaptively follow the query distribution to select the important data items to update.

Case Study 2: med-neg Updates The grey area in Figure 4.4(b) (i.e., the update volume) shows that the number of updates over data is negatively correlated to the query distribution in Figure 4.3. This update trace has two prominent groups: hot updated and cold updated data. As shown in Figure 4.4(b) with the tiny black dots close to x-axis, more than 95% of the updates are dropped and the updates dropped concentrate on hot updated



(a) med-unif trace



(b)med-neg trace

Figure 4.4: Distribution of Updates over Data ID (Original vs. UNIT Degraded)

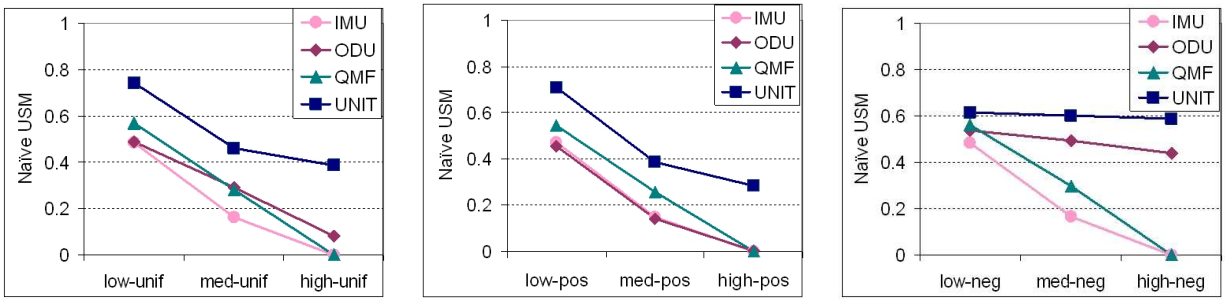
data which is also the data with less frequent accesses (smaller IDs). What we can also roughly see from the black dots in Figure 4.4(b) is that the hot accessed data has about the same number of updates than cold accessed data, instead of the big difference in Figure 4.4(a). The reason is that for the hot accessed data in Figure 4.4(b), originally the number of updates is very small, i.e., a relative small number of updates is enough to guarantee the freshness of the data.

We make the following observations: (1) It is not true that hot accessed data should always get more updates than other data. When the data are inherently stable, a small number of updates is enough. (2) Updates on cold accessed and hot updated data will be dropped more often than those on hot accessed and cold updated data.

4.3.3 Naive USM: Quantitative Evaluation

We now quantitatively compare UNIT to other algorithms. In order to be fair, we set all the weights C_r, C_{fm}, C_{fs} to 0, which means `USM` equals the traditional success ratio in this naive setting.

Figure 4.5 shows the naive `USM` over 3 different update distributions (unif/pos/neg) and



(a) Uniform (b) Positive correlation (c) Negative Correlation

Figure 4.5: Performance Comparison when `USM` = Success Ratio

three different update volumes (low/med/high). We can clearly see that UNIT has much higher `USM` (success ratio) over all of the different settings in Figure 4.5.

Specifically, UNIT outperforms other algorithms ranging from 17% to 39% improvement in Figure 4.5(a), from 13% to 28% improvement in Figure 4.5(b), and 6% to 58% improvement in Figure 4.5(c) in absolute values for `USM`. These differences translate to a 30%, 50% and 10% minimum relative improvement over the competitor algorithms and multiple orders of magnitude improvement in the best case (since some of the other algorithms produce near zero `USM`).

It is interesting to note that in Figure 4.5(a), QMF performs even worse than the simple on-demand update scheme (ODU), because QMF is trying to reduce the Miss Ratio (number of deadline misses over number of admitted queries), which makes QMF reject more aggressively to secure the admitted queries. As a result, QMF has fewer successful queries than ODU. In Figure 4.5(b), immediate updates (IMU) performs almost identical to ODU, because the query and update distributions are positively correlated. In Figure 4.5(c), ODU performs close to UNIT, because most of updates are "irrelevant" under the negative correlation. ODU by itself tries to minimize updates.

4.3.4 Normal USM: Sensitivity Evaluation

With UNIT, it is possible to assign different penalties for different type of failures (i.e., assign different values for C_r, C_{fm}, C_{fs}). In this set of experiments, we evaluate how sensitive UNIT is to the different cost functions. The main result is that UNIT is fairly stable in terms of `USM`, even when the cost functions change dramatically.

Table 4.3: `USM` weights with penalties < 1 for Figure 4.6(a)

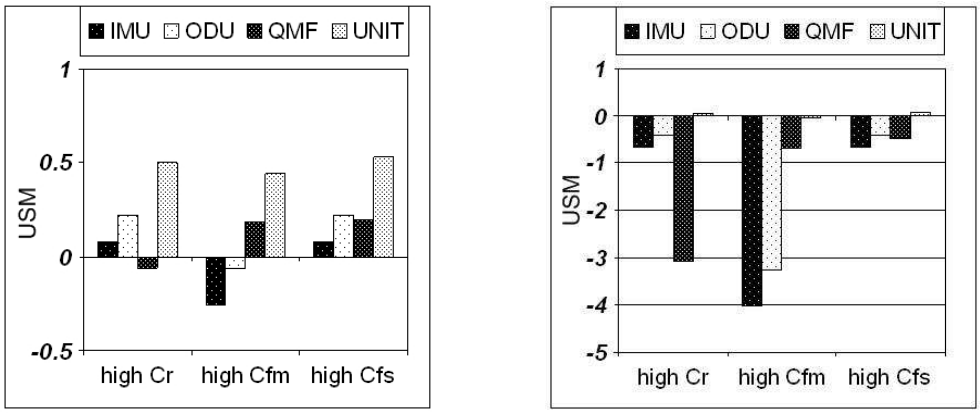
	C_s	C_r	C_{fm}	C_{fs}
high C_r	1	0.5	0.1	0.1
high C_{fm}	1	0.1	0.5	0.1
high C_{fs}	1	0.1	0.1	0.5

Table 4.4: `USM` weights with penalties > 1 for Figure 4.6(b)

	C_s	C_r	C_{fm}	C_{fs}
high C_r	1	5	1	1
high C_{fm}	1	1	5	1
high C_{fs}	1	1	1	5

Figure 4.6(a) and 4.6(b) show the performance (measured as `USM`) of different methods (under the query trace and update trace *med_unif*) with penalties less than 1 and greater than 1, respectively. The 3 values along the x-axis (high C_r, high C_{fm}, high C_{fs}) refer to the cases where the corresponding cost factor is higher than the other two costs. The exact weights for Figure 4.6(a) are shown in Table 4.3, and the weights setup for Figure 4.6(b) are shown in Table 4.4.

Figure 4.6 clearly shows that UNIT performs best in both cases: penalties < 1 and penalties > 1. Some interesting observations are: (1) QMF performs worse with high C_r, because it rejects many queries to favor the miss ratio; (2) IMU and ODU are worse with high C_{fm}, because they fail to finish many queries on time.



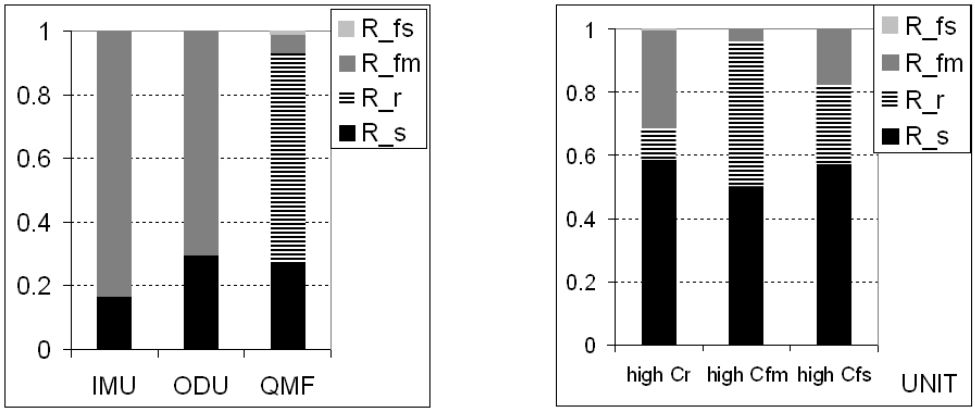
(a) Penalties < 1 (b) Penalties > 1

Figure 4.6: Non-zero Penalty Cost (med-unif trace)

4.3.5 Insight into behavior of UNIT

After showing the performance gain of *UNIT* in a variety of settings, we now elaborate on why UNIT gives a better and more stable `USM` than the other methods. User queries could have four outcomes: Success, Rejection, DMF, and DSF. We can collect how many queries fall into each group by having the number of Success/Rejection/DMF/DSF divided by the total number of queries, denoted as Success/Rejection/DMF/DSF ratio (i.e., $R_s/R_r/R_{fm}/R_{fs}$). By visualizing the ratio decompositions of the outcome of queries, we can have an insight about the results in the previous sections.

Figure 4.7(b) plots the four ratios for UNIT under the setup from Table 4.4. Figure 4.7(a) plots the four ratios for IMU, ODU, and QMF. Since these algorithms are insensitive to the weight variations, the four ratios are the same under all settings in Table 4.4. We observe the following: (1) Regardless of the penalty settings, UNIT gives a much higher success ratio than the others. (2) The ratio distribution of UNIT changes a lot with different cost setups. With the rejection cost smallest with high_Cr setup and DMF cost smallest in the high_Cfm setup, it can be explained why the `USM` for UNIT remains stable along different cost setups: UNIT effectively minimizes the portion that dominates the cost. (3) IMU, ODU and QMF



(a) Compared Algorithms (b) UNIT

Figure 4.7: Ratio Distribution

are not affected by the cost parameters and hold the same success ratio. Comparing the three algorithms, we find QMF's rejection ratio very high. The reason is that when there is a burst of requests, QMF is being conservative and drops many queries to guarantee the admitted queries to be successfully executed. Although within those admitted queries, the miss ratio is minimized, the overall success ratio is low too.

4.4 SUMMARY

Web-based database systems of today are challenged by unpredictable surges of request load. To optimize the system resource allocation towards users' most interests, we proposed a suite of load balancing algorithms, UNIT, to filter out the queries and the updates that may hurt the overall user satisfactions. UNIT uses a feedback control mechanism and relies on an intelligent admission control algorithm along with a new update frequency modulation scheme in order to maximize `USM` (a simplified version of Quality Contracts). Our evaluation showed that UNIT performs better than two baseline algorithms and the current state-of-the-art when tested using workloads generated from real traces.

5.0 SCHEDULING (SERVER)

The memory-residency of the types of systems we are interested in (as described in Section 3.1) eliminates the problems of complex buffer management and I/O scheduling that are crucial in traditional, disk-based database systems. Instead, in our system, CPU scheduling is the primary means of improving performance[1]. QoS and QoD have always been trade-off of each other. For example, if updates are scheduled before queries, users will perceive highest QoD, but with less QoS than the case if updates are scheduled after queries. When the server is under the challenge of the high load, the performance degradation on QoS/QoD will be exaggerated and easily perceived by the end users. In such cases, the order by which queries and updates are executed is expected to play a crucial role in affecting user satisfaction. We will further illustrate the impact of CPU scheduling in Section 5.1.1.

This chapter focuses on query and update scheduling under user preferences through Quality Contracts. 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. Specifically, we propose QUTS, a two-level scheduler that addresses the problem of prioritizing the scheduling of updates (associated with QoD) over queries (associated with both QoS and QoD) in the presence of user-specified Quality Contracts. QUTS supports both *linear QC functions* and *step QC functions* (explained in Section 5.1.2).

[1]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 is outside the scope of this work. In the experiment, we adopt Two Phase Locking - High Priority (2PL-HP) [4] 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 for each data item. Hence, there is no issue of deadlocks.

5.1.1 Impact of CPU Scheduling

To further illustrate the impact of query and update scheduling on response time and staleness and the ensuing trade-off, we ran a simple experiment with the following three naive scheduling policies[2].

(a) *First In First Out (FIFO)*: a non-preemptive single priority queue with both queries and updates, where queries and updates are executed according to their arrival times.

(b) *FIFO Update High (FIFO-UH)*: a dual priority queue (one for updates and one for queries) where the FIFO update queue has a higher priority than the FIFO query queue. The arrival of an update may preempt the execution of a query.

(c) *FIFO Query High (FIFO-QH)*: a dual priority queue (one for updates and one for queries) where the FIFO query queue has a higher priority than the FIFO update queue. The arrival of a query may preempt the execution of an update.

All three scheduling policies used the 2PL-HP (Two Phase Locking-High Priority) concurrency control scheme [4]. The plain FIFO could be seen as the most "fair" policy (to both queries and 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 5.1 shows the average staleness and average response time from running a simulation experiment using a real stock information web server trace. We measured average staleness as the number of *unapplied updates*, *uu*, 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 5.1, 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

[2]There was no load shedding on updates nor admission control on queries in this setup.

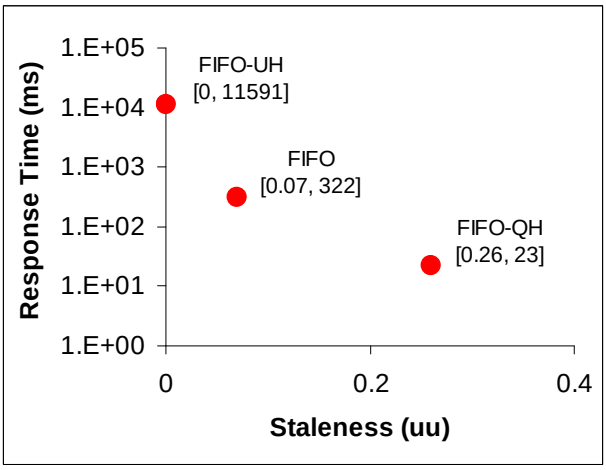


Figure 5.1: Impact of Scheduling on the Trade-off between Response Time and Staleness.

response time, but the worst staleness; the plain FIFO policy is somewhere in between the two extremes.

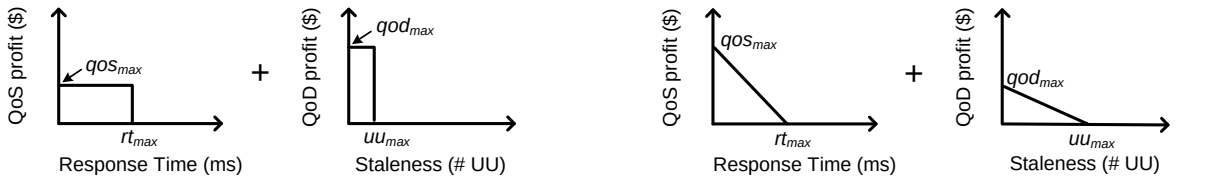
It is not clear which of these policies is better, since all three points are dominating points (i.e., for each point, no other point exists with smaller values on both dimensions). Without proper choice of scheduling, the server may easily go against the users' will and optimize towards opposite especially when user preferences changes dynamically.

5.1.2 Linear/Step Quality Contracts

In Chapter 3, we have proposed *Quality Contracts (QCs)* as a unifying framework for specifying QoS and QoD preferences. In the general case of Quality Contracts, users specify a number 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, in this work, we look into two types: (a) step functions and (b) linear functions.

Figure 5.2(a) has an example of QCs with step functions (or *step QCs* for short), with



(a) Step Quality Contracts

(qos_{max}= \$1, rt_{max}= 50ms, qod_{max}= \$2, uu_{max}= 1)

(b) Linear Quality Contracts

(qos_{max}= \$2, rt_{max}= 50ms, qod_{max}= \$1, uu_{max}= 2)

Figure 5.2: Quality Contract examples

only two functions specified: one for QoS, using response time, and one for QoD, using staleness. Notice that, the QoD function is discrete because we compute the query staleness with the maximum number of unapplied updates of all the data items the query accesses. 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 may 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 may have for the server to get any (QoD) profit from executing this query.

Figure 5.2(b) has an example of QCs with linear functions (or *linear QCs* for short), with the same setup as in Figure 5.2(a). In this work, we consider both step QCs and linear QCs and adopt QoS-Independent QCs where QoD profit is collected even if QoS profit is zero (see Section 3.2.3 for details).

5.2 BASELINE ALGORITHMS

As we have shown with the three naive algorithms (FIFO, FIFO-UH, and FIFO-QH in Section 5.1.1), there are two basic types of policies on scheduling queries and updates [5]: 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.

5.2.1 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 with rewards, which have been studied extensively in real time systems [16, 49, 76, 45]. The general guideline is to consider both dimensions (the time constraints and the 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[3]) is not comparable to response time. On the other hand, if we only consider the profit, which is commonly

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

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. Because of the random arrival and interleaving of queries and updates, FIFO may achieve better QoS/QoD than the policies with dual priority queues that favor either updates (such as FIFO-UH) or queries (such as FIFO-QH). Nonetheless, due to the same reason, FIFO's performance is not as predictable as the dual-priority-queue policies.

5.2.2 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) [45] 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 staleness and highest QoD regardless of the order of updates[4]. 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

[4]Update scheduling in UH can potentially affect query QoS in a subtle way. The reason is that the order of updates may cause different number of updates to be invalidated due to the newly arriving updates. As a result, the whole CPU time consumed by the updates is slightly different. Nonetheless, the benefit of using a more complex algorithm can barely cover the overhead.

performance than UH. Yet, its delayed execution of updates may accumulate too many unapplied updates for data items, and thus increase 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.

5.3 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, as well as 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?

Table 5.1: Symbol Table of QUTS Scheduling

Symbol	Meaning	Definition
N_q	total query set	
$z^{(i)}$	value of symbol z for the i^{th} query	
z_k	value of symbol z for the k^{th} adaptation period	
ω	adaptation period	
τ	atom time	
ρ	CPU percentage for queries, or query share; $0 \leq \rho \leq 1$	
QOS_{max}	maximal QoS profit	$\sum_{i \in N_q} qos_{max}^{(i)}$
QOD_{max}	maximal QoD profit	$\sum_{i \in N_q} qod_{max}^{(i)}$
QOS	gained QoS profit	$\sum_{i \in N_q} qos^{(i)}$
QOD	gained QoD profit	$\sum_{i \in N_q} qod^{(i)}$
Q_{total}	total gained profit from both QoS and QoD	$\sum_{i \in N_q} (qos^{(i)} + qod^{(i)})$

5.3.1 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. Furthermore, we want to determine the CPU allocation that maximizes the total profit. Please refer to Table 5.1 for the symbols used in this section.

5.3.1.1 Model the total profit with Query CPU allocation ρ 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 gained profit Q_{total} from all queries is maximized. Since Q_{total} comprises of two parts: QoS profit (denoted as QOS) and QoD profit (denoted as QOD), we will look at the relationship between ρ and these two parts respectively.

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 the total maximal profit of all queries in terms of QoS, QOS_{max}. Thus, the total gained QoS profit QOS can be approximated as:

$$QOS = QOS_{max} \cdot \rho \tag{5.1}$$

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 (from the total maximal profit of all queries in terms of QoD, QOD_{max}.) 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:

$$QOD = QOD_{max} \cdot \rho \cdot (1 - \rho) \tag{5.2}$$

Total profit is, thus, modeled as:

$$Q_{total} \approx QOS_{max} \cdot \rho + QOD_{max} \cdot (1 - \rho) \cdot \rho, \quad 0 \leq \rho \leq 1. \tag{5.3}$$

5.3.1.2 The optimal ρ to maximize the total profit The above quadratic function with linear constraints usually requires expensive quadratic programming to find the optimal solution. However, since there is only one variable ρ in Equation 5.3, we can simplify it into a gradient descent problem. The optimal solution is:

$$\rho = min(\frac{QOS_{max}}{2 \cdot QOD_{max}} + 0.5,\ 1) \tag{5.4}$$

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 [52] which is similar to standard conjugate gradient optimization:

$$\rho_{new} = min(\frac{QOS_{max_{k-1}}}{2 \cdot QOD_{max_{k-1}}} + 0.5,\ 1) \tag{5.5}$$

$$\rho_k = (1 - \alpha) \cdot \rho_{k-1} + \alpha \cdot \rho_{new} \qquad 0 < \alpha < 1 \tag{5.6}$$

where $QOS_{max_{k-1}}/QOD_{max_{k-1}}$ is the maximal sum of submitted QoS/QoD values during the previous adaptation period. Those QCs that change over time (e.g., linear QCs) will incur more overhead when QOS_{max}/QOD_{max} is recomputed. In general, α should be a small value, but the exact α does not matter much [52].

5.3.2 Implementation of CPU allocation ρ

Based on the previous discussion, the probability of a query running (or the query queue preceding 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 as $1 - \rho$.

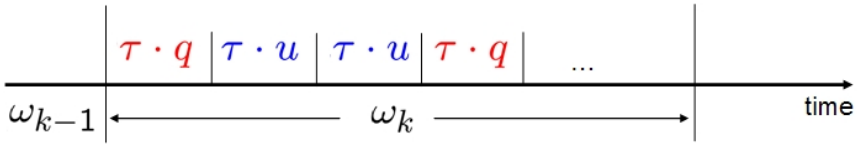


Figure 5.3: QUTS Scheduling

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 not want it to be too often, not only to avoid the overhead, but also because the data contention can be potentially increased

Table 5.2: Pseudo-code of QUTS (Query-Update Time-Sharing) two-level Scheduling Algo.

High Level	for each adaptation period ω **(Section 5.3.1)** Adjust ρ according to Equation 5.5, 5.6 for each atom time period τ **(Section 5.3.2)** (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 5.2)**	update priority queue: FIFO

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 5.3. A state change may happen every τ time, or if the picked queue is empty at any instant of time. The pseudo-code for the QUTS two-level scheduling algorithm is given in Table 5.2.

The complexity of QUTS in its uppper level is $O(N' + M)$ for each adaptation window, where N' is the number of queries appeared in an adaptation window, and M is the number of atom times in an adaptation window. In general, the lower level of QUTS can use any query scheduling scheme and update scheduling scheme. In particular, VRD has a complexity of $O(N \log N)$, where N is the total number of queries.

5.4 EXPERIMENTS

We have acquired access traces from a popular stock market information web site, Quote.com. 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. Our goal is to evaluate how well the proposed methods perform under the entire spectrum of Quality Contracts, and also gauge the adaptability and sensitivity of our proposed algorithm.

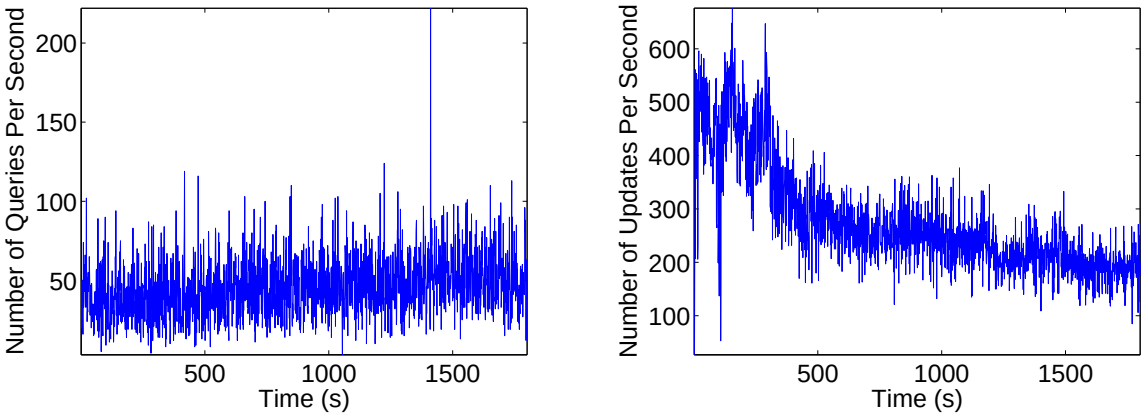
5.4.1 Experimental Setup

5.4.1.1 Query and Update 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.

Table 5.3: Workload Information

	Trace A (9:30am-10:00am)	Trace B (10:30am-11:00am)
number of queries	82,129	120,000
number of updates	496,892	396,291
number of stocks	4,608	4,108
query execution time	$5 \sim 9$ms	
update execution time	$1 \sim 5$ms	

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



(a) Query Distribution over Time

(b) Update Distribution over Time

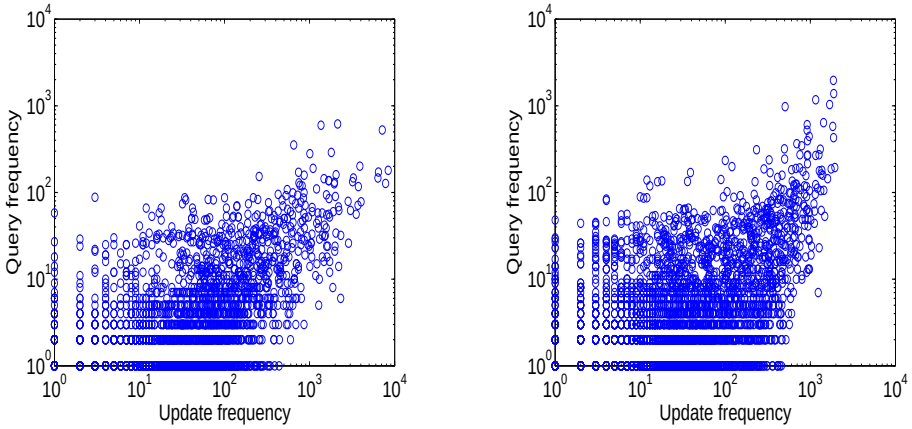
Figure 5.4: **Trace A characteristics:** (a) query distribution has small changes over time despite one spike; (b) update distribution has downward trend over time with some fluctuations.

presented in this chapter are based on two 30-minute intervals. In particular, trace A is from 9:30am to 10:00am with 82,129 queries, and trace two is from 10:00am to 10:30am with 120,000 queries. Both query traces access more than 4,000 different stocks.

To match our query workload, we extracted the actual trades on all securities listed on the NYSE during 9:30am-10:00am and during 10:30am-11:00am on April 24, 2000[5]. The update trace includes the stock ticker symbol, record date, trade time, and trade price per share. Update execution times range from 1 to 5 milliseconds. In particular, trace A has 496,892 updates, and trace B has 396,291 updates on different stocks which share the same indexing scheme with query traces, the stock ticker symbol. The workload information of the two traces is summarized in Table 5.3. Next, we illustrate the characteristics of the two traces.

Trace A (9:30am-10:00am): Figure 5.4(a) and (b) show the distributions over time for trace A's queries and updates respectively. The statistics are collected on each second. On

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

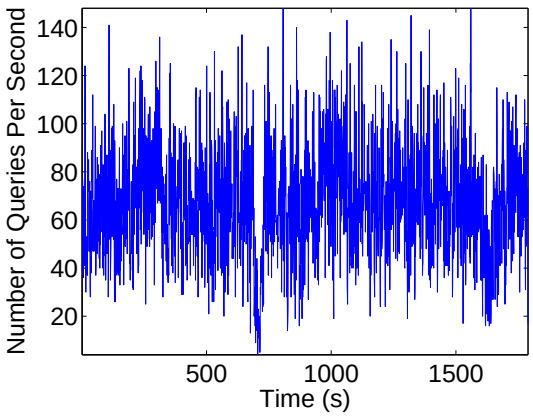


(a) Trace A (b) Trace B

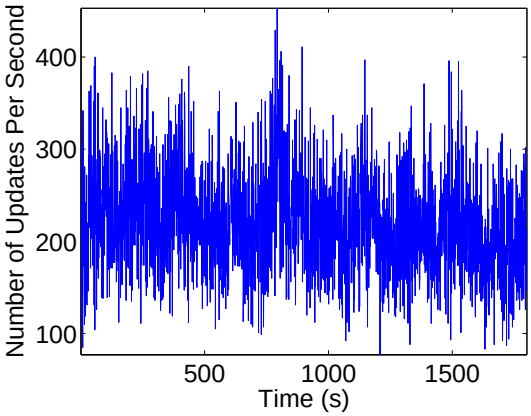
Figure 5.5: **Query vs. Update for each Stock:** log-log plot of the query and update frequency for each stock (depicted as circles). Stocks are concentrated below the diagonal because both traces have more updates than queries.

average, there are more updates than queries. Yet, the intensity of the updates reduce during the second half of the trace. Query distribution is rather stable over time despite one spike during the second half of the trace. Update distribution has downward trend over time with some fluctuations. Overall, the number of updates is almost 6 times of the number of the queries. Figure 5.5(a) 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 (i.e., most points are below the diagonal in Figure 5.5(a)). 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.

Trace B (10:30am-11:00am): Trace B is taken from the late morning and has less fluctuations than trace A. However, trace B concentrates on a heavier query workload, since the stock activities are generally increasing in the morning. We plot the query distribution over time in Figure 5.6(a). The distribution does not have any obvious spikes as in the earlier interval plotted in Figure 5.4(a) although the total number of queries is higher than the



(a) Query Distribution over Time



(b) Update Distribution over Time

Figure 5.6: **Trace B characteristics:** (a) both query and update distributions are more stable than trace A; (b) the ratio between the number of queries and the number of updates is smaller than that of trace A (3.3 in B vs. 6 in B).

earlier interval. The update distribution over time is plotted in Figure 5.6(b). Dislike the downward trend shown in trace A, updates in trace B is relatively more stable. Overall, the ratio between the number of queries and updates is around 3.3, which is smaller than the ratio in trace A (at around 6). We also plot the number of updates and queries over all the stocks in Figure 5.5(b). Similar to trace A, stocks are shown to be concentrated below the diagonal (i.e., most stocks have more updates than queries).

5.4.1.2 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.

As we will show in Section 5.4.4, the parameters can be chosen from a large range without having much influence on the performance.

Table 5.4: Setup Indicators and Performance Metrics

QOS_{max}	maximal QoS profit	$\sum_{i \in N_q} qos_{max}^{(i)}$
QOD_{max}	maximal QoD profit	$\sum_{i \in N_q} qod_{max}^{(i)}$
Q_{max}	total maximal profit from both QoS and QoD	$\sum_{i \in N_q} qos_{max}^{(i)} + qod_{max}^{(i)}$
$QOS_{max}\%$	maximal QoS gained profit percentage	$\frac{QOS_{max}}{Q_{max}}$
$QOD_{max}\%$	maximal QoD gained profit percentage	$\frac{QOD_{max}}{Q_{max}}$
QOS	gained QoS profit	$\sum_{i \in N_q} qos^{(i)}$
QOD	gained QoD profit	$\sum_{i \in N_q} qod^{(i)}$
Q_{total}	total gained profit from both QoS and QoD	$\sum_{i \in N_q} qos^{(i)} + qod^{(i)}$
$QOS\%$	QoS gained profit percentage	$\frac{QOS}{Q_{max}}$
$QOD\%$	QoD profit percentage	$\frac{QOD}{Q_{max}}$
$Q\%$	total profit percentage	$\frac{Q_{total}}{Q_{max}}$

where N_q is the total query set; $z^{(i)}$ is the value of symbol z for the i^{th} query.

5.4.1.3 Performance Metric The performance is measured by how much profit from all queries each algorithm gains. Depending on the goal of each set of experiment, we present either the *profit* or the *profit percentage* (i.e., the ratio of gained profit over the maximal profit). The profit can be denoted in terms of Q_{total}, QOS, and QOD. The profit percentage can be described using $Q\%$, $QOS\%$, and $QOD\%$. Please refer to Table 5.4 for the detail explaination and formal definition. For all metrics, the higher, the better.

We also define a couple of *setup indicators* ($QOS_{max}\%$ and $QOD_{max}\%$ in Table 5.4) to express the over all user preferences over QoS and QoD. $QOS_{max}\%$ and $QOD_{max}\%$ are defined as the maximal profit percentages that the system can gain from the users in terms of QoS and QoD. In the following experiments, we vary the user preferences by setting Quality Contracts with different ranges and achieving different $QOS_{max}\%$ and $QOD_{max}\%$.

5.4.2 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 with both trace A and trace B (Section 5.4.2.1), and also with changing qos_{max} and qod_{max} (Section 5.4.2.2). Please refer to Table 5.4 for the notations used in explaining the following experimental results.

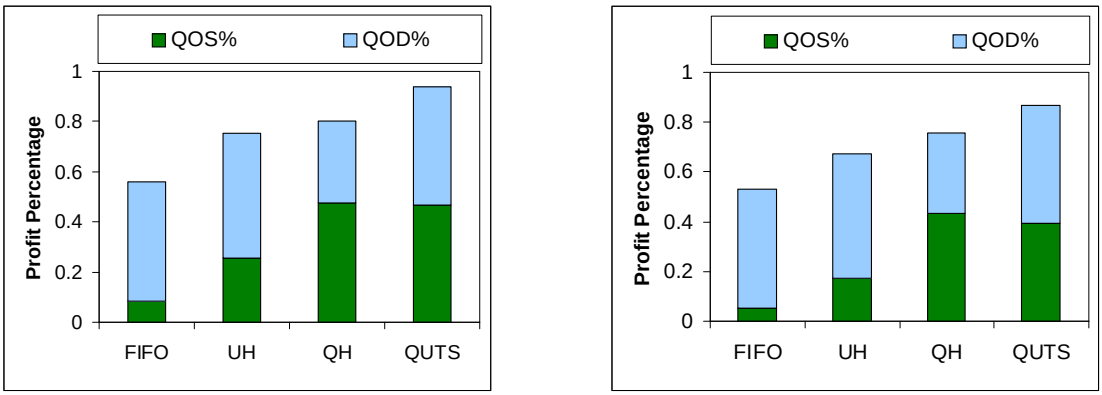
Table 5.5: Quality Contract Setup for Figure 5.7

$QOD_{max}\%$	0.5
$QOS_{max}\%$	0.5
qod_{max}	\$10 $\sim$ \$99
qos_{max}	\$10 $\sim$ \$99
rt_{max}	50ms $\sim$ 100ms
uu_{max}	1

5.4.2.1 Step QCs vs. Linear QCs

Experiment Design (Table 5.5): We have four traces in this experiment: trace A with step QCs, trace A with linear QCs, trace B with step QCs, and trace B with linear QCs. For all four traces, we use the same setup for QCs (as shown in Table 5.5): qos_{max} and qod_{max} are randomly chosen from a same range, \$10 $\sim$ \$99. Thus, user preferences are equally distributed over QoS and QoD, which means the maximal gained QoS profit over the the maximal total profit, $QOS_{max}\%$, equals to 0.5. Similarly, the maximal gained QoD profit percentage, $QOD_{max}\%$, equals to 0.5 too. 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).

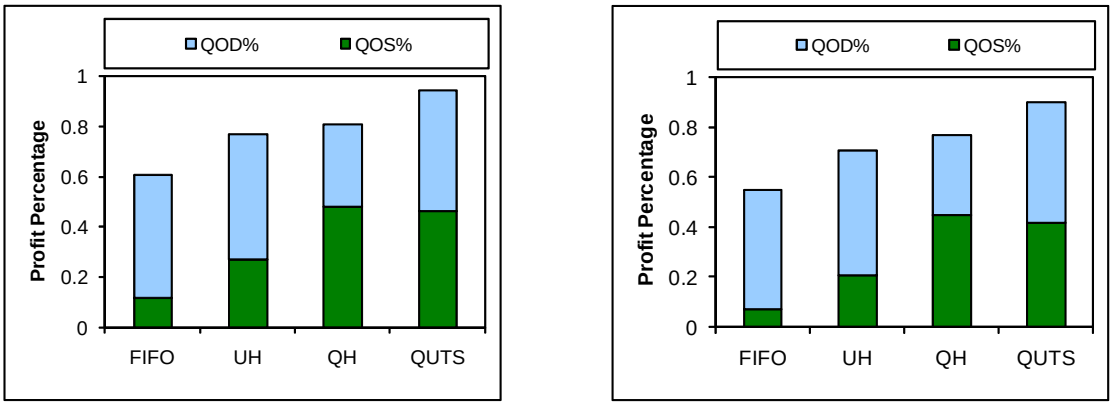
Results (Figure 5.7): Figure 5.7 shows the performance of trace A with step functions and with linear functions. For both plots, we show the gained QoS and QoD profit percentage. The total height of each bar is the total profit percentage gained (i.e., the sum of QoS profit percentage and QoD profit percentage). In Figure 5.7, the maximal QoS percentage is 0.5, since QoS and QoD share the same amount of profit in this setup.



(a) Step QCs with Trace A

(b) Linear QCs with Trace A

Figure 5.7: Trace A: Profit Percentage with step and linear QC functions.



(a) Step QCs with Trace B

(b) Linear QCs with Trace B

Figure 5.8: Trace B: Profit Percentage with step and linear QC functions.

Looking at the performance with step QCs in Figure 5.7(a), we see that QUTS gains the highest profit percentage with both QoS and QoD profit percentage close to the maximal.

Table 5.6: Quality Contract Setup for Figure 5.9

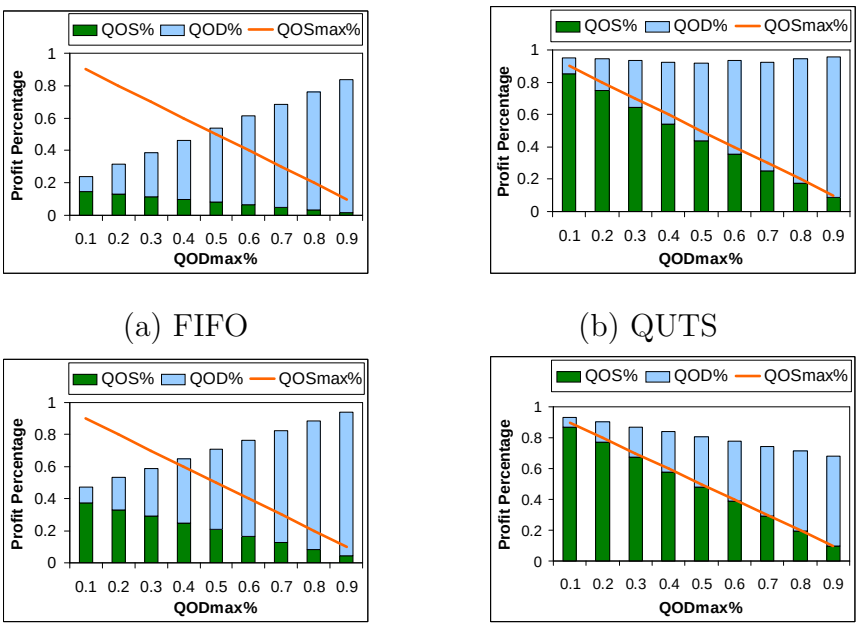
$QOD_{max}\%$	0.1	0.2	...	0.9
$QOS_{max}\%$	0.9	0.8	...	0.1
qod_{max}	\$10 $\sim$ \$19	\$20 $\sim$ \$29	...	\$90 $\sim$ \$99
qos_{max}	\$90 $\sim$ \$99	\$80 $\sim$ \$89	...	\$10 $\sim$ \$19
rt_{max}	50ms $\sim$ 100ms			
uu_{max}	1			

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 Figure 5.7(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.

Figure 5.8 shows the gained QoS and QoD profit percentage of trace B with step functions and with linear functions. As we can see, the performance of the four compared algorithms in trace B resembles that of trace A to a great extend. The biggest difference is that FIFO in trace B gains up to 4% better QoS% than FIFO in trace A. This performance enhancement may be caused by the relative stable query and update workload in trace B (e.g., no spike as in A's query trace). The variations of other algorithms are less than 2%.

Due to the similarity of the performance shown in both traces with step functions and linear functions, we only show the results of trace A with step functions in the rest of the experimental results.



(a) FIFO (b) QUTS

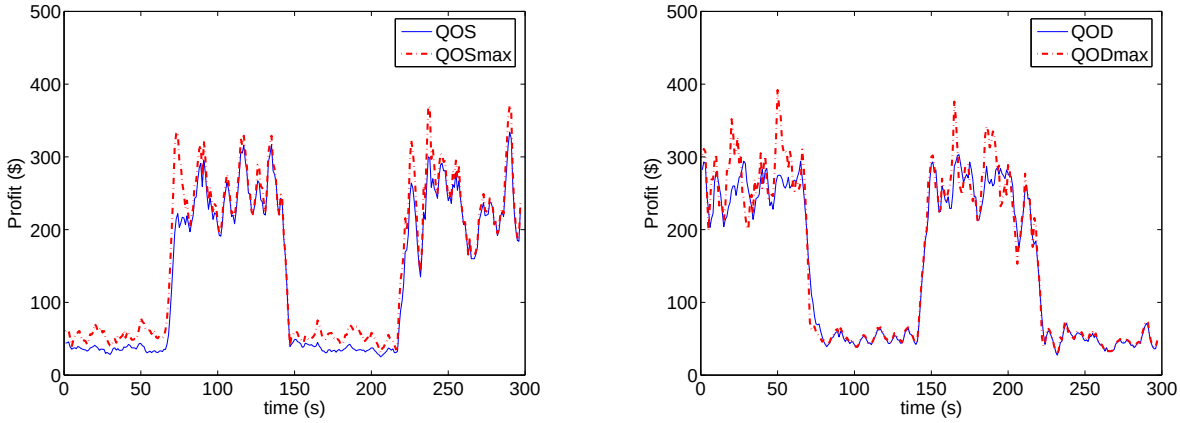
(c) Update High (UH) (d) Query High (QH)

Figure 5.9: Profit Percentage with Various QCs (FIFO and QUTS)

5.4.2.2 Performance under different QCs

Experiment Design (Table 5.6): In the above experiments, we used the same range to set qos_{max} and qod_{max} in QCs. This set of experiments is designed to show the performance of QUTS with various qos_{max} and qod_{max}. We prepared nine different QC sets, which we list in Table 5.6. Uniform distribution is adopted for all ranges listed in Table 5.6.

Results (Figure 5.9): Figure 5.9 shows the profit percentage from the FIFO, UH, QH, and QUTS policy over the different QC setups, respectively. 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 5.9(a) 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 5.9(c), the Update-High policy gains almost the maximal QoD profit percentage (the light colored bars), but performs poorly on QoS. In Figure 5.9(d), the



(a) QoS Profit

(b) QoD Profit

Figure 5.10: QoS/QoD Profit over Time

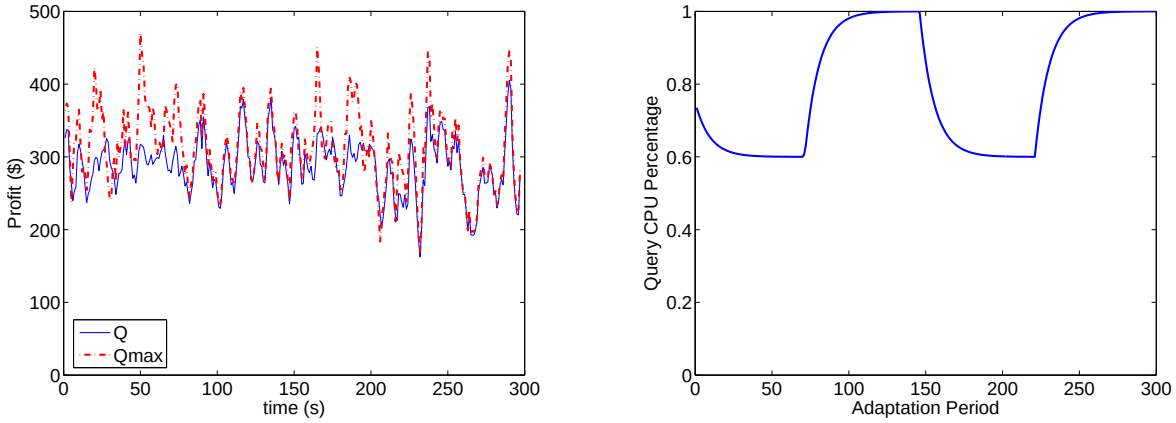
Query-High policy gains almost the maximal QoS profit percentage (the dark colored bars), but performs relative poorly on QoD. In Figure 5.9(b), QUTS gains almost the maximal QoS and QoD profit percentage with all QC sets. In fact, QUTS performs up to 101.3% better than UH and up to 40.1% better than 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.

5.4.3 Adaptability to User Preferences

Having shown that QUTS is much better and robust than the baseline algorithms in various QC setups, we now illustrate how quickly QUTS adapts to changing QCs.

Note that user preferences are reflected by quality contracts: if users indicate their preference more on the QoS, the system should try to answer the queries as fast as possible by increasing the CPU query share ρ; while if users care more on QoD, the system should provide as accurate answers as possible by reducing ρ (hence increasing the update share).

Experiment Design: We use the same traces, but instead of a static QC setup, we vary



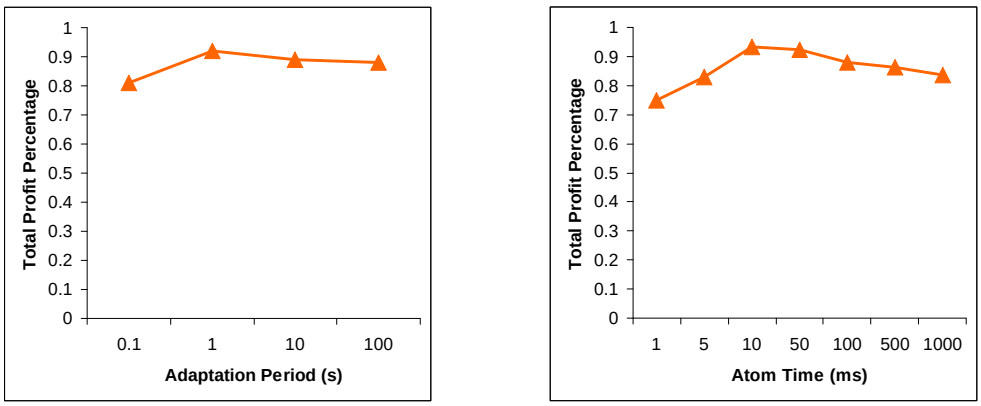
(a) Total Profit

(b) Query CPU Share ρ

Figure 5.11: Total Profit and Query CPU Share ρ over Time

qos_{max} and qod_{max} over time. Specifically, we divide the experiment period evenly into 4 intervals, and have $rt_{max} = 50\text{ms} \sim 100\text{ms}$ (uniformly distributed), $uu_{max} = 1$, and vary the qos_{max} to qod_{max} ratio from 1:5 to 5:1 (i.e., $qos_{max} = 5\times\ qod_{max}$ 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 5.10 and Figure 5.11): We plot the actual and maximal profit of submitted queries over time in Figure 5.10. As expected, the maximal line in Figure 5.10(a) shows the QoS profit trend along time: low-high-low-high, and the maximal line in Figure 5.10(b) shows the QoD profit trend along time: high-low-high-low. The maximal line in Figure 5.11(a) shows the total maximal profit which is the sum of the profits from Figure 5.10(a) and Figure 5.10(b). 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.



(a) Adaptation Time (ω)

(b) Atom Time (τ)

Figure 5.12: Sensitivity of QUTS over ω and τ

Figure 5.11(b) 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 Equation 5.4, ρ should be a number between 0.5 and 1, and should "track" the total maximal QoS profit. In Figure 5.11, 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.

5.4.4 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 Table 5.5.

Sensitivity of *Adaptation Period* ω (Figure 5.12(a)) 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 5.12(a) the overall performance varies little for a wide range of *adaptation periods*.

Sensitivity of *Atom Time* τ (Figure 5.12(b)) 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 5.12(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.

5.5 SUMMARY

In this chapter we addressed the problem of scheduling queries and updates in data-intensive web sites, in the presence of linear/step Quality Contracts (QCs). Specifically, 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 little sensitivity to its parameters.

6.0 QUALITY CONTRACTS ADAPTATION (USER)

In previous chapters, we have illustrated the trade-off between Quality of Service (QoS) and Quality of Data (QoD) in web-database systems and advocate using user preferences to guide system resource allocation. Specifically, we proposed Quality Contracts framework in Chapter 3 to empower users to indicate their preferences over multiple quality requirements and different queries. In our framework, users are allocated with virtual money. A QC essentially specifies how much virtual money the user is willing to pay to have his/her query executed. The actual money that the server will get in the end (i.e., the system profit) will depend on *how well* the query was executed. To take these user preferences into account, we have developed load balancing policy UNIT in Chapter 4 to filter out the transactions that are to the least interest to users according to a simplified version of QCs. To support the full-fledged QCs, we also developed a query/update scheduling algorithm QUTS in Chapter 5 which maximizes the overall system profit through a two-level scheduling mechanism.

In this chapter, we combine the load balancing policy and the scheduling scheme and focus on the applicability of the Quality Contract enhanced web-databases. Notice that when users get the freedom to specify their preferences using Quality Contracts, they also take the responsibility of assigning their virtual money in the framework. How to spend the virtual money could become complicated with the presence of other competing users. To eliminate the burden from users and take a step further in popularizing the Quality Contracts framework, we build a demo system and explored multiple mechanisms of user side adaptation on Quality Contract selection. Specifically,

- We combined our previous work on load balancing management and query/update scheduling and implement QuiX (QUality-aware Integrated admission Control and Scheduling)

system and evaluated different QC adaptation strategies in various settings with real data traces.

- We identify two important issues in QC selection schemes: *payment expectation* and *savings amount*.
- 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[1] switches between two modes: *Overbid mode*, which maximizes the utility of the user's budget with the empirical expectation of QC expenditure; and *Deposit mode*, which gracefully builds up user savings.
- We experimentally illustrate the effects of *user population distribution* and *knowledge scope* in adapting QCs.

6.1 SYSTEM ARCHITECTURE

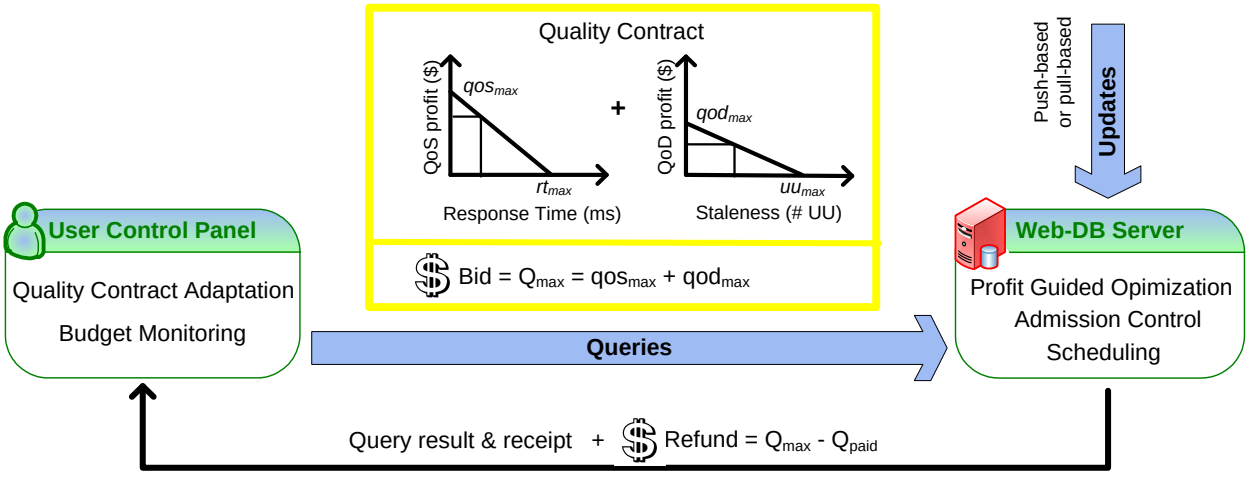


Figure 6.1: System Architecture

We assume a web-database server architecture like the one in Figure 6.1. 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.

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

6.1.1 The Quality Contract (QC) Economy

Users are allocated virtual money, which they spend in order to execute their queries according to their preferences, which 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.

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., $Q_{max} - Q_{paid}$). We adopt Quality Contracts as shown in Figure 6.1.

In the presence of Quality Contracts (QCs), the users and the server have distinct objectives: servers try to maximize their income, whereas users try to "stretch" their budget to run successfully as many queries as they can.

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

Admission Control Scheme: We employ a variant of our UNIT scheme as described in Chapter 4. UNIT eliminates those updates that have the least interests from queries and rejects those queries that threaten the overall user satisfaction. UNIT was originally designed to be used with non profit-driven scheduling schemes. To work with a profit-driven scheduling algorithm as QUTS, we expect admission control layer takes less responsibility. For example, load shedding on updates becomes redundant when unimportant updates can be automatically postponed (and possibly dropped due to invalidation) without hurting user satisfaction. In the QuiX system, we allow all updates entering the web-databases but keep the query admission control from UNIT. A query has high chance to be rejected if it is too demanding and/or has a small profit potential for the server, especially in periods of high load. Thus, a higher bid will increase the chance of a query being admitted.

Scheduling Scheme: We employ our QUTS scheduling scheme as described in Chapter 5 to maximize the server profit. QUTS is a two-level scheduling scheme: it keeps a separate priority queue for queries and another one for updates (each with its own scheduling mechanisms). QUTS dynamically allocates CPU resources between the two queues according to submitted QCs. The higher the bid, the higher the chance a query gets better QoS/QoD when executing.

Concurrency Control: We use Two Phase Locking - High Priority (2PL-HP) [4] where the lower priority transaction releases the lock to the higher priority transaction at a conflict.

6.1.3 User View

The user module must include an interface for specifying QCs and the ability to monitor the execution of QC-enabled queries, while keeping track of the current budget. Although the QC framework empowers the 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).

Quality Contract Adaptation Problem: In this work, we adopt QCs with linearly decreasing positive functions as in Figure 6.1 (although our proposed algorithm can be extended to other QC types). For each QC, users need to set four parameters: qos_{max}, the maximum QoS profit, rt_{max}, the maximum bearable response time, qod_{max}, the maximum QoD profit, and uu_{max}, the maximum bearable staleness. We assume that users know the quality constraints of their queries (i.e., rt_{max} and uu_{max}), and the relative importance between QoS and QoD (i.e., $\frac{qos_{max}}{qod_{max}}$). What users do not know is how to allocate and adapt their budget to maximize the queries with valuable results returned, or, in other words, how to choose Q_{max}, the budget for each query.

Query Outcomes: We define two outcomes for a query:

- **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.

- **Failure:** If a query fails either the QoS or the QoD constraint, we call the query a *failure*, and $Q_{paid} = 0$.

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.

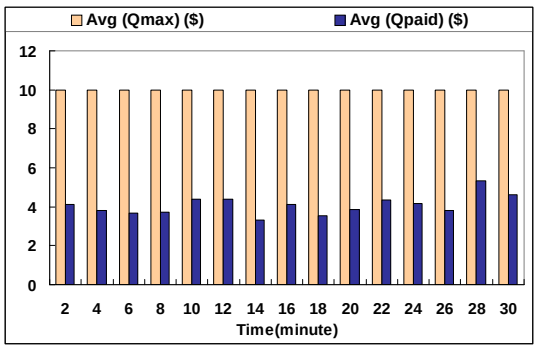
6.1.4 Analysis of Baseline QC Adaptation Schemes

Before introducing our proposed scheme, we look into three baseline strategies. Given N queries and a total budget B, the strategies compute $Q_{max}^{(i)}$, the total bid for the QC of query i, as follows:

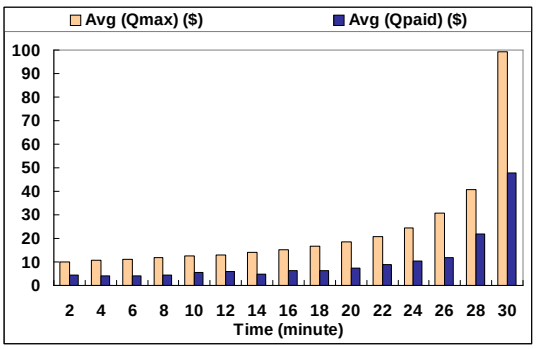
- **Fixed (FIX):** $Q_{max}^{(i)} = \frac{B}{N}$. FIX is a static policy, which assigns each query an equal share of the total budget.
- **Random (RAN):** $Q_{max}^{(i)} = uniform[\frac{B}{N} - c, \frac{B}{N} + c]$, where c is a constant. This strategy uses $\frac{B}{N}$ as the mean, and $[\frac{B}{N} - c, \frac{B}{N} + c]$ as the range to pick Q_{max} uniformly.
- **Dynamic (DYN):** $Q_{max}^{(i)} = \frac{B_i}{N-i}$. 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: We evaluate the above three schemes with a 30-minute trace containing 120,000 queries and 396,000 updates and set the initial budget per query to be \$10. Figure 6.2(a) shows the average Q_{max} and Q_{paid} of FIX users for each 2-minute time period. We can see that Q_{max} is always set to \$10. However, the actual payment is less than half of the budget. In other words, FIX does not make full use of the budget, because it ignores the refunds from the previous failed queries. The behavior of the RAN scheme is similarly problematic.

DYN addresses the issue of ignored refunds by dynamically updating the available budget. As shown in Figure 6.2(b), DYN users set Q_{max} to \$10 at the beginning and increase it over time, as refunds accumulate. As a result, DYN favors the queries issued later than earlier, thus creating an uneven distribution of the total budget.



(a) Expenditure with FIX



(b) Expenditure with DYN

Figure 6.2: Performance of Baseline Algorithms

If we know we will have a refund in the future, it makes sense to be aggressive and bid more than the current budget per query. Having this in mind, we ran another set of experiments with one DYN user and seven other users that used the FIX scheme, but with the addition of an *overbid factor*, as follows: $Q_{max} = \frac{B}{N} \times$ *overbid factor*. We ran the experiment with overbid factors of 0.6 (=under-bidding), 1 (=regular FIX), 1.4, 1.8, 2.2, 2.6, and 3. As Figure 6.3 shows, there is a peak in the success ratio when the overbid factor is 1.4. Also, both overbid factor 1.4 and 1.8 outperform DYN, because DYN is too conservative at the beginning.

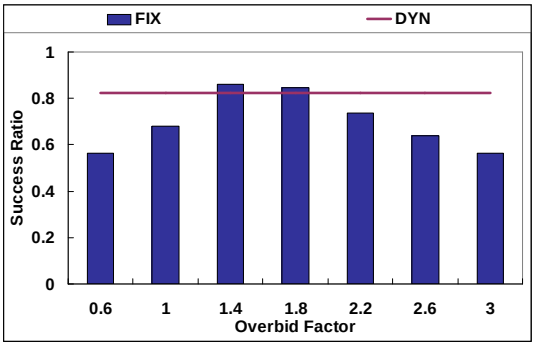


Figure 6.3: Effect of Overbid with FIX

Although the FIX policy with the overbid factor outperforms the DYN policy in some cases, clearly such a policy is not a viable option, because it relies heavily on the choice of

the overbid factor. However, this policy brings out an interesting idea: *overbidding can have a very positive effect on the success ratio of queries.* This is one of the ideas that we exploit in our proposed AQC scheme, which we describe next.

6.2 ADAPTIVE QUALITY CONTRACT SCHEME (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* (presented in Section 6.2.1) and *Deposit mode* (presented in Section 6.2.2); we discuss how AQC chooses between the two modes in Section 6.2.3. The symbols used for illustrating AQC is summarized in Table 6.1.

Table 6.1: Table of Symbols

Symbol	Explanation
B	total budget
B_i	current budget when the user issues the i^{th} query
N	total number of queries
qos_{max}, qod_{max}	max \$ promised in QC for QoS, QoD
qos_{paid}, qod_{paid}	\$ paid to server for QoS, QoD after query completion
rt_{max}	max response time specified in QC
uu_{max}	max # of unapplied updates in QC
Q_{max}	$qos_{max} + qod_{max}$
Q_{paid}	$qos_{paid} + qod_{paid}$
$z^{(i)}$	value of symbol z for the i^{th} query
$z^{(s)}$	value of z for last successful query

6.2.1 Overbid Mode

The Overbid mode of the AQC scheme addresses DYN's problem of unfairly "favoring" later queries by monotonically increasing Q_{max} as time progresses (and previously submitted queries have refunds). This behavior is roughly equivalent to last-minute spending by companies at the end of the fiscal year, since at that time, any of the remaining money in the current year's budget will effectively disappear unless spent immediately.

AQC solves 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, only to be recognized at the last minute).

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

$$\mathbf{E}_p[Q^{(i)}_{paid}(x,\, y)] = \text{budget per query} = \frac{B_i}{N-i} \tag{6.1}$$

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 6.1 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 QoS function and from QoD function respectively:

$$\mathbf{E}_p[Q^{(i)}_{paid}(x,\, y)] = \mathbf{E}_p[S(x)] + \mathbf{E}_p[D(y)] \tag{6.2}$$

If we combine Equation 6.1 with Equation 6.2, we have that:

$$\mathbf{E}_p[S(x)] + \mathbf{E}_p[D(y)] = \frac{B_i}{N-i} \tag{6.3}$$

In this work, we adopt linear segmented QCs where the QoS function can be represented as in Equation 6.4 and the QoD function can be represented as in Equation 6.5. If other formats of QC functions are adopted, Equation 6.4 and Equation 6.5 should be modified accordingly.

$$S(x) = \begin{cases} qos_{max}(1 - \frac{x}{rt_{max}}) & \text{if } x \in [0,\, rt_{max}] \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

$$D(y) = \begin{cases} qod_{max}(1 - \frac{y}{uu_{max}}) & \text{if } y \in [0,\, uu_{max}] \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

We compute the expectation of QoS profit using empirical expectation, as shown in Equation 6.6.

$$\begin{aligned} \mathbf{E}_p[S(x)] &= \int S(x)p(x)\,dx \\ &= qos_{max}\int_0^{rt_{max}} p(x)\,dx - \frac{qos_{max}}{rt_{max}}\int_0^{rt_{max}} xp(x)\,dx \\ &\approx qos_{max}(\mathrm{P}(x < rt_{max}) - \frac{\bar{x}}{rt_{max}}) \end{aligned} \tag{6.6}$$

where $\mathrm{P}(x < rt_{max})$ 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 $\mathrm{P}(x < rt_{max})$ 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:

$$\begin{aligned} \mathbf{E}_p[S(x)] &\approx qos_{max} \cdot \alpha \\ \alpha &= \mathrm{P}(x < rt_{max}) - \frac{\bar{x}}{rt_{max}} \end{aligned} \tag{6.7}$$

Similarly, we compute the expectation of QoD profit:

$$\begin{aligned} \mathbf{E}_p[D(y)] &= \int S(y)p(y)\,dy \\ &= qod_{max}\int_0^{uumax} p(y)\,dy - \frac{qod_{max}}{uu_{max}}\int_0^{uumax} xp(y)\,dy \\ &\approx qod_{max}(\mathrm{P}(y < uu_{max}) - \frac{\bar{y}}{uu_{max}}) \end{aligned} \tag{6.8}$$

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

$$\begin{aligned} \mathbf{E}_p[D(y)] &\approx qod_{max} \cdot \beta \\ \beta &= \mathrm{P}(y < uu_{max}) - \frac{\bar{y}}{uu_{max}} \end{aligned} \tag{6.9}$$

As described in Equation 6.3, the total expected profit from both QoS (Equation 6.7) and QoD (Equation 6.9) should be set to the current budget per query $\frac{B_i}{N-i}$:

$$qos_{max} \cdot \alpha + qod_{max} \cdot \beta = \frac{B_i}{N-i} \tag{6.10}$$

where α and β are computed based on query execution history (as shown in Equation 6.7 and Equation 6.9). We assume that the relative importance between QoS and QoD for the same user and query is known by the user. Let the ratio between QoS and QoD be γ, we have:

$$\begin{aligned} Q_{max} &= qos_{max} + qod_{max} \\ qod_{max} &= \gamma \cdot qos_{max} \end{aligned} \tag{6.11}$$

We solve Equation 6.11 and Equation 6.10 to get the final solution of Q_{max}:

$$Q_{max}^{(i)} = \frac{B_i}{N-i} \cdot \frac{1}{\alpha + \gamma \cdot \beta} \tag{6.12}$$

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

6.2.2 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 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 less than usual budget 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 savings 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 user will need to reduce Q_{max} when he/she recognizes a "string" of consecutive successful query executions. Towards this, we record the number of failures f within a window size w. If $Q_{max}^{(s)}$, $Q_{paid}^{(s)}$ are the budget and the payment for the most recent successful query, then we can set the new $Q_{max}^{(i)}$ in Deposit mode as follows:

$$Q_{max}^{(i)} = Q_{max}^{(s)} \cdot (1 - \frac{Q_{paid}^{(s)}}{Q_{max}^{(s)}}) \tag{6.13}$$

The idea is that the closer $Q_{paid}^{(s)}$ is to $Q_{max}^{(s)}$, the further the actual response time is to the response time constraint (and similarly for the staleness constraint). In other words, the server can still satisfy the user requirements even if it gave slightly lower priority to that query, and thus the user might have paid too much. Therefore, the decrease of Q_{max} should be positively correlated to the ratio of $Q_{paid}^{(s)}$ and $Q_{max}^{(s)}$.

6.2.3 How to switch between Deposit and Overbid mode

At the beginning, the system is set to default mode, which is the overbid mode. 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 in line 1 of Algorithm 6.1), the system is set to deposit mode. This is because a consecutive successful 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, $successQ.size$ can decrease due to two reasons: (1) there are no more queries to be completed (i.e., neither query success nor query failure), thus $successQ.size$ 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 high priorities.

Algorithm 6.1: Adaptive Quality Contract Mode Selection

Input: successQ.size - number of consecutive query successes within time window w

1 **if** $successQ.size > c$ **then**
2 | *Deposit Mode* [See Equation 6.12]
3 **else**
4 | *Overbid Mode* [See Equation 6.13]

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.

6.3 EXPERIMENTS

6.3.1 Experimental Setup

As we have illustrated in Section 5.4.1.1, we acquired query traces from a popular stock information web site, Quote.com and combined them with the NYSE (New York Stock Exchange) update traces for the same time period. Section 5.4 shows that QUTS gives consistent high performance under both trace A (9:30am-10:00am) and trace B (10:30am-11:00am). To study how user strategies affect their query success ratio, in this experiment, we use trace B because it has a higher query concentration than trace A.

Comparison Algorithms: To evaluate our proposed QC adaptation strategy, we performed an extensive simulation study using the FIX (without overbid), RAN, DYN schemes (Section 6.1.4) and our proposed AQC strategy (Section 6.2).

Experiment Design: We designed three sets of experiments:

1. performance comparison of different algorithms under various workloads settings, and
2. evaluation of how different populations of users using various algorithms will affect the algorithm performance, and
3. evaluation of how the amount of knowledge the users can use will affect the algorithm performance.

We attach user information with each query, which includes the user preference on the QC and the adopted QC adaptation strategy. Detailed setup information is provided at the beginning of each set of experiments.

6.3.2 Performance Comparison

To create environments for a fair comparison, we generated three sets of traces: solo, duet, and quartet. In solo, each trace contains only one class of users (i.e., one algorithm), simulating a naive environment where no competitors exist. In duet, each trace contains two

classes of users (i.e., two algorithms), creating an one-on-one confrontation to show directly which algorithm performs better. In quartet, we put all four algorithms into the trace, where they all interact and compete with each other.

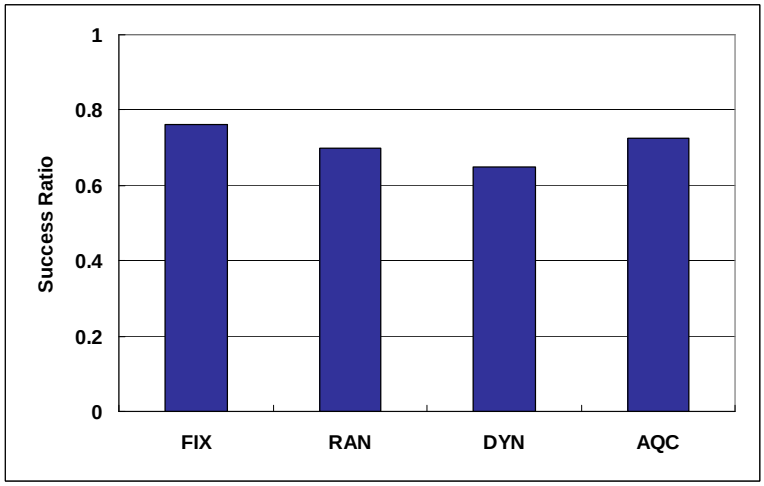


Figure 6.4: Solo Environment: Success Ratio

6.3.2.1 Solo

Experiment Design: We created four traces for the four algorithms respectively. Each class of users has 120,000 queries (i.e., the whole trace) with initial budget per query to be $10. The total budget for each user is the budget per query times the number of queries he/she has.

Results (Figure 6.4) We measure the query success ratio for each adaptation strategy in Figure 6.4. As we expected, the performance difference from each algorithm is small, because no other competitors exist in the system. We expect the performance of a single strategy highly depend on the server capacity, arrival patterns of the queries/updates, and the read/write conflicts. In this set of experiments, FIX performs the best among all because FIX gives constant bids, thus incurs least preemptions among its own queries. DYN performs the worst since it naturally increases the bid by taking refunds from previous queries. Such

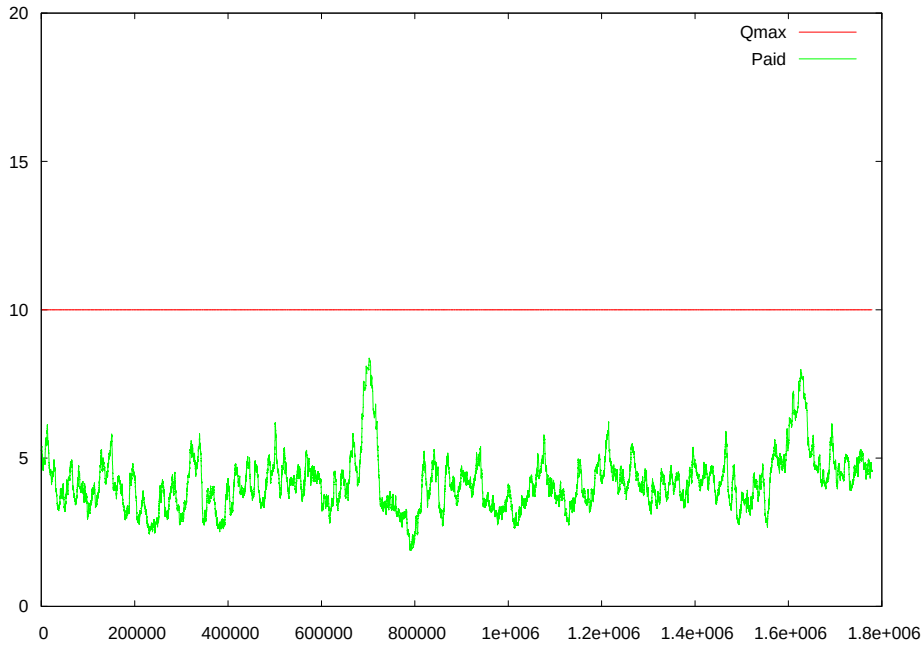
monotonic increasing generates the most preemptions which result in the most restarts and waste of system resources. Our proposed AQC algorithm performs very close to the best algorithm in this naive environment.

Solo Over time Results (Figure 6.5 and Figure 6.6) To show how different adaptation strategies perform over time, we record the bid Q_{max} and the money actually paid Q_{paid} for each query. We plot their moving averages over time with a window size of 10 seconds. The upper line is Q_{max} and lower line is the money actually paid Q_{paid}.

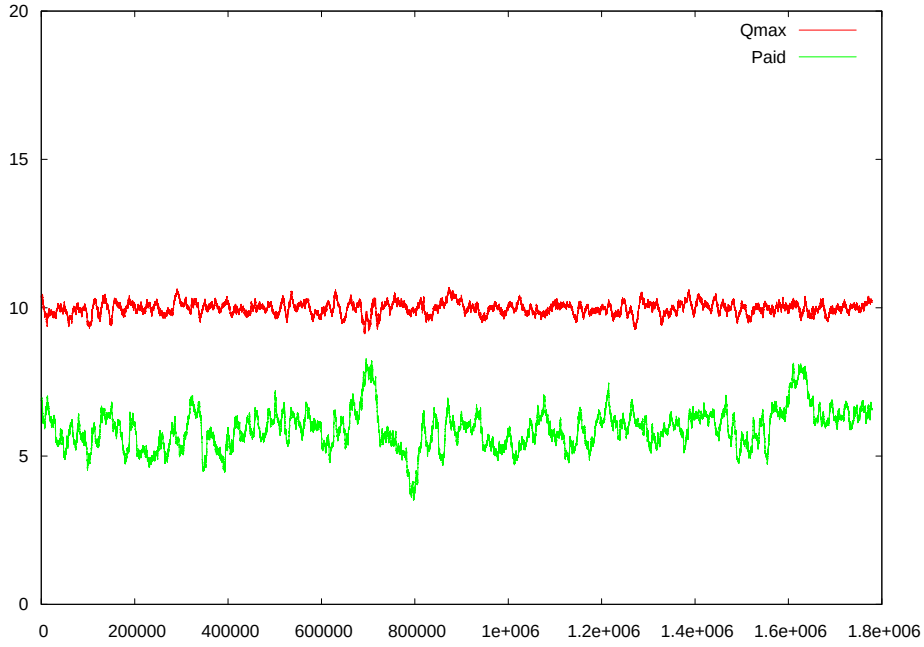
It is obvious that FIX gives the constant bid ($10, the budget per query) for each query as shown in Figure 6.5(a). Due to the linear shape of the QC functions, usually Q_{max} will not be fully spent because of unavoidable CPU time and unpredictable queuing time. Thus, we see a much lower actual expenditure (around $5 on average) than Q_{max}. RAN has Q_{max} varying from $9 to $11 since the budget per query is $10 as shown in Figure 6.5(b). Similar to FIX, RAN's actual money expenditure Q_{paid} is also much smaller than Q_{max}.

DYN, as shown in Figure 6.6(a), adaptively sets Q_{max} as the future budget per query, which picks up all previous savings since the actual expenditure is usually much smaller. However, the saving also causes Q_{max} booming at an increasing speed and finally skyrockets in the end. DYN is still being conservative on early issued queries, which not only jeopardizes the fairness of queries coming at different times, but also hurts its overall success ratio because of the preemptions and restarts generated by the monotonic increase of bids.

AQC, as in Figure 6.6(b), shows its characteristics with its overbid and deposit mode, although there is no other competitors in this solo environment. We can see that the actual money paid Q_{paid} is on average $10 (the budget per query) which confirms that AQC realizes its goal to raise the bids as much as possible but still not overspending the budget. The over time plot also shows that AQC occasionally decreases Q_{max} when AQC tries to save money after a sequence of successful records, so that it can bid higher to survive a tough situation with a much higher chance.

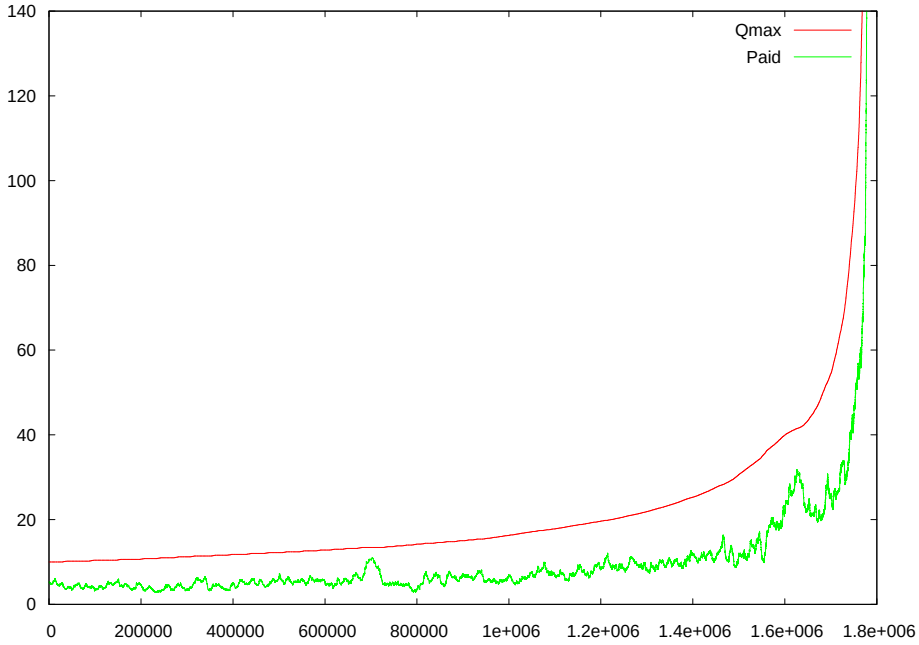


(a) FIX

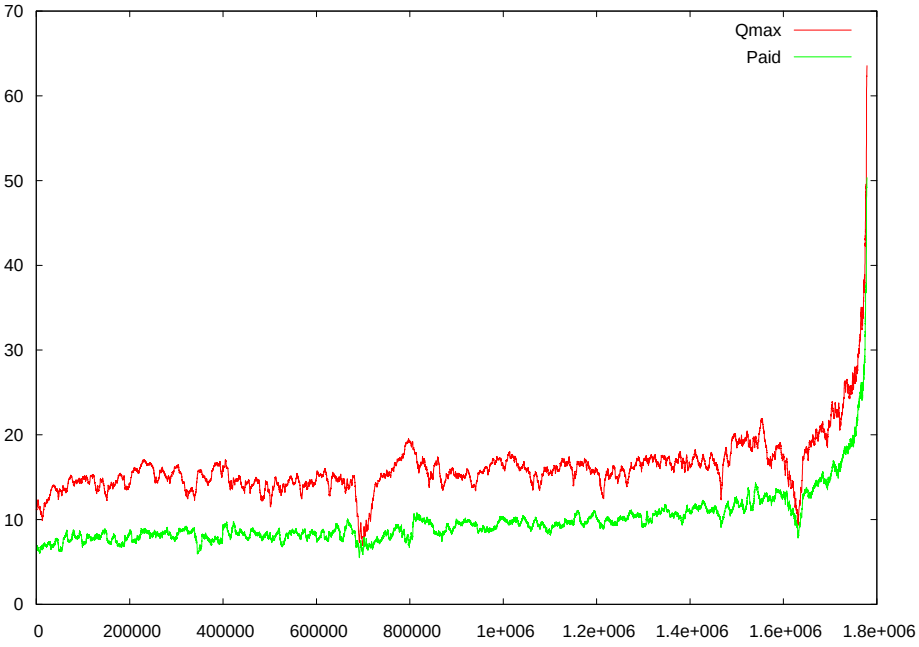


(b) RAN

Figure 6.5: Solo over Time (FIX and RAN)



(a) DYN



(b) AQC

Figure 6.6: Solo over Time (DYN and AQC)

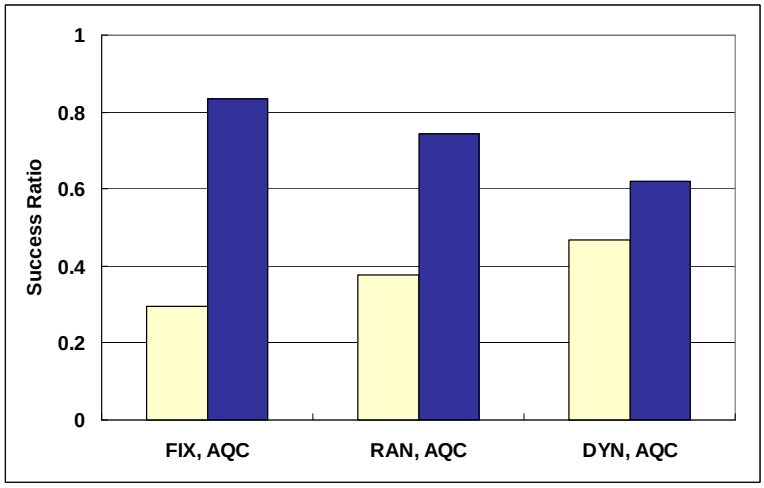


Figure 6.7: Duet Environment: Success Ratio

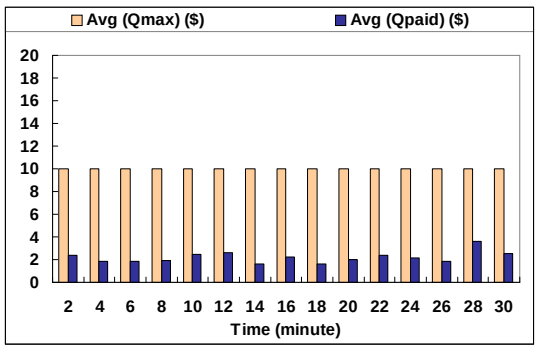
6.3.2.2 Duet

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

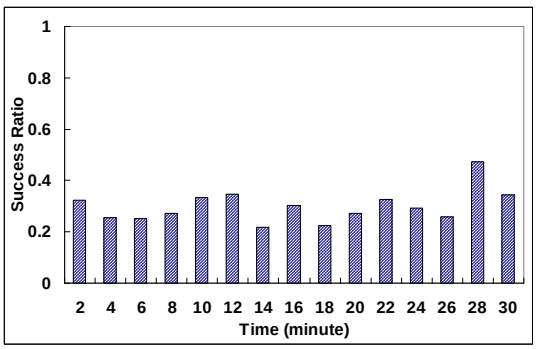
Results (Figure 6.7) We run each trace 20 times and report the average query success ratio for the three comparisons in Figure 6.7. The performance difference is much more obvious compared to the solo environment. AQC users perform 183% better than FIX users, almost 100% better than RAN users, and more than 30% better than DYN users. Another expected observation is that AQC's performance is better when competing with weaker competitors such as FIX.

Duet Over time Results (Figure 6.8) With the same set of experiment, we record the bid Q_{max} and the money paid Q_{paid} for each query as well as the query success ratio for each 2-minute window.

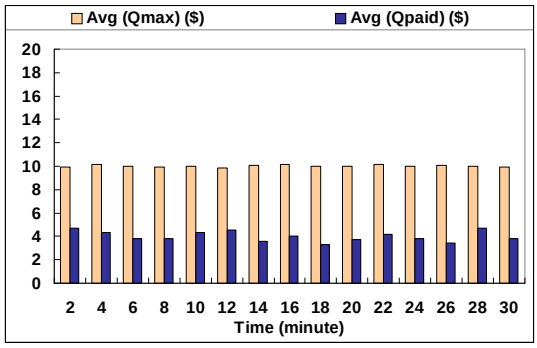
In Figure 6.8(a), FIX gives a constant bid ($10) for each query. When running against



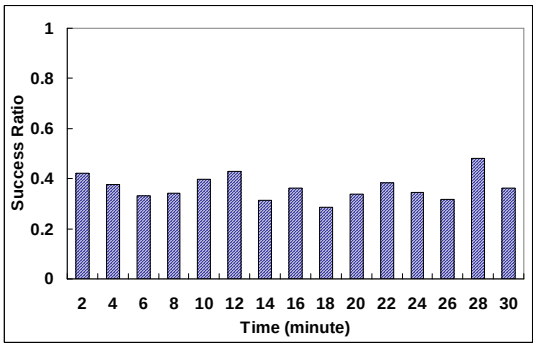
(a) FIX: Money Expenditure



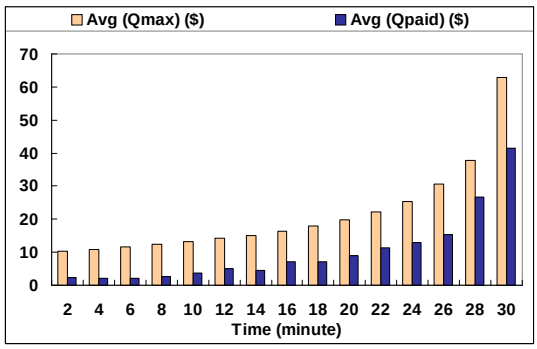
(b) FIX: Success Ratio



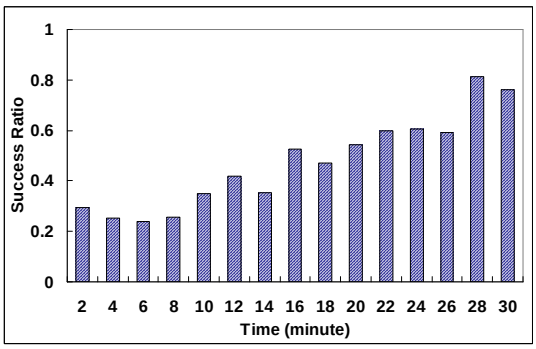
(c) RAN: Money Expenditure



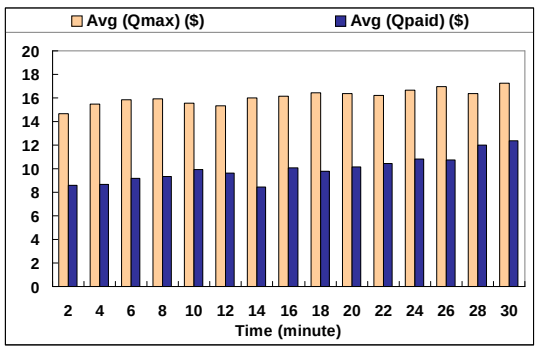
(d) RAN: Success Ratio



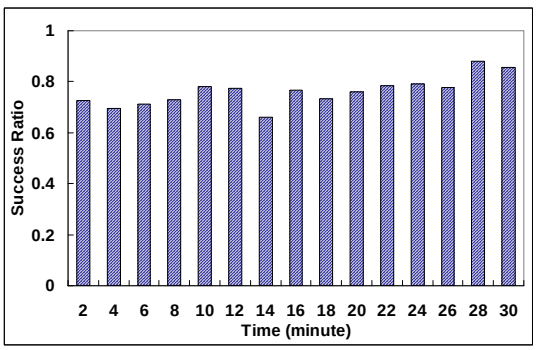
(e) DYN: Money Expenditure



(f) DYN: Success Ratio



(g) AQC: Money Expenditure



(h) AQC: Success Ratio

Figure 6.8: Duet over Time

AQC in the duet environment, FIX's success ratio is less than 1/3 of FIX running solo. As a result, FIX's real expenditure in duet is also much smaller than FIX in solo. FIX also has the smallest expenditure among all algorithms, leaving FIX with smallest success ratio shown in Figure 6.8(b). RAN, shown in Figure 6.8(c) and (d), has not much improvement over FIX. Q_{max} varies around \$10 and Q_{paid} is around \$4 on average. Comparing with FIX, RAN only gains less than 50% better successes with more than 100% expenditure than FIX.

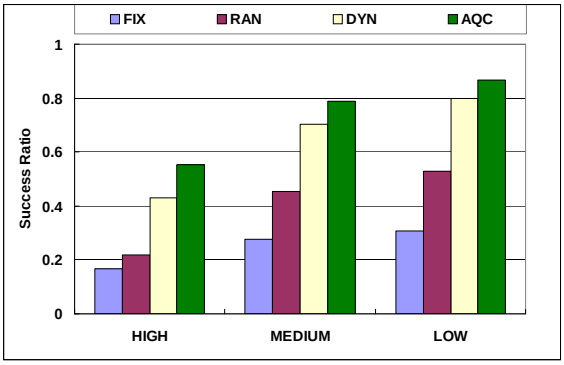
In Figure 6.8(e), we see DYN dynamically adjust the current available budget and increase Q_{max} over time. As a result, DYN's success ratio increases along time too, as shown in Figure 6.8(d). However, similar to DYN's solo over time, it is still too conservative on early issued queries.

Finally, Figure 6.8(g) and (h) show AQC's improvement from two sides. First, 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 from its overbid mode. Second, AQC tries to save money after consecutive successes from its deposit mode, so that the user can bid higher to survive a more competitive situation later. This is why we see a few decreasing bids on Figure 6.8(g). Both the expenditure expectation and saving with deposit mode help AQC achieve significantly better results when it competes with other algorithms.

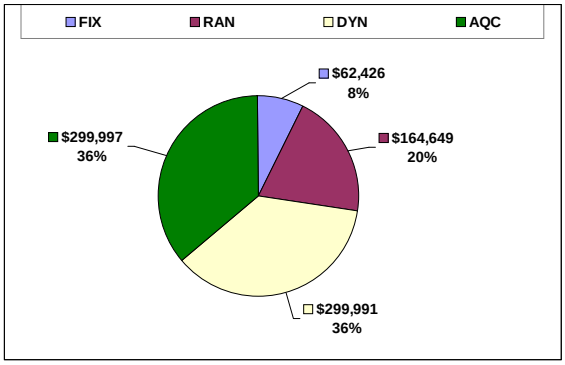
6.3.2.3 Quartet

Experiment Design: Having observed different algorithms' behavior respectively (i.e., solo) and pairwise (i.e., duet), we mix the four user algorithms in one trace; 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 6.9(a)), and (2) showing both the user view and the server view (Figure 6.9(b)).

Results (Figure 6.9(a)) We change the user constraints on QoS to be tight, medium, and loose to generate three traces with High, Medium, and Low workload in Figure 6.9(a). As expected, for all algorithms, the success ratio is higher with Low system workload (which has loose quality constraints). Compared to the other algorithms, AQC performs the best



(a) User View: Success Ratio



(b) Server View: Profit

Figure 6.9: **Quartet Environment: 4 algorithms under different workload settings.** User view: the lighter the workload, the higher success ratio. Under high workload, AQC achieved 233% better performance than FIX, 155% than RAN, and 28% better than DYN. Server view: DYN and AQC utilized almost all their money, whereas FIX and RAN had a large portion wasted.

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 FIFO by 233%. AQC also achieves 155% better performance than RAN and 28% better than DYN.

Results (Figure 6.9(b)) After watching different algorithms from the users' point of view, we show the server profit gains from each user algorithm under medium workload. Similar trends can be found with both high and low workload. Figure 6.9(b) shows that the server gains much more profit from DYN and AQC, thus tends to serve them better than FIX and RAN. Making full use 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.

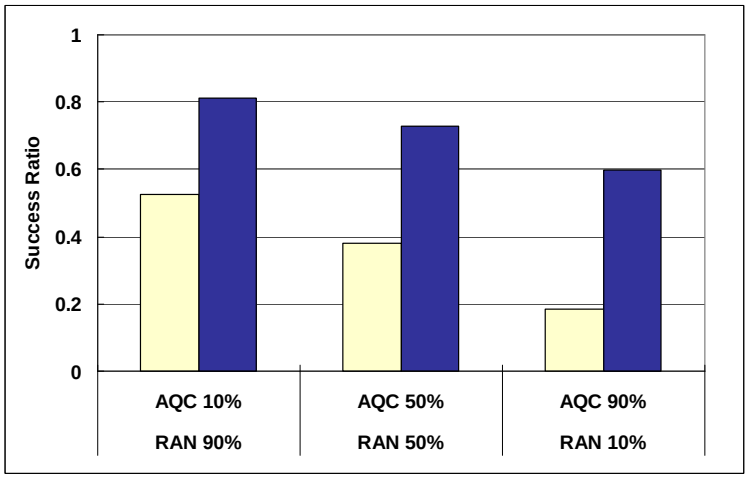


Figure 6.10: **Average performance for AQC and RAN with different populations.** As the percentage of AQC users increases, the performance of both algorithms decreases, since competition is more severe.

6.3.3 Population

Experiment Design: Previous experiments assigned equal proportions of users to the different algorithms when compared in a single experiment. In this set of experiments, we pick AQC and RAN to evaluate the impact of different population distributions. We generate three traces where the population of AQC and RAN varies from (90%, 10%), (50%, 50%) to (10%, 90%) in each trace.

Results (Figure 6.10) From the average query success ratio for each user class in Figure 6.10, we can see that no matter how small our algorithm's user population is (e.g., 10%), we can easily beat RAN by up to 227%, which well supports our hypothesis, that changing population will not affect the performance of our algorithm. Another observation is that as the population of AQC increases, the average query success ratio of both AQC and RAN decreases and the performance difference increases. The reason is that as the population of smart/aggresive users increase, the environment becomes more competitive. All users' performance get affected and decreased, and RAN suffers more than AQC.

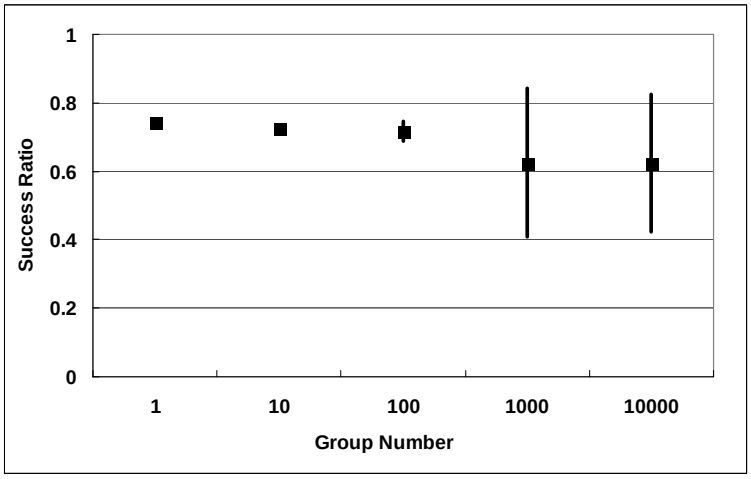


Figure 6.11: **Performance of AQC with different knowledge scope (Group Number shows how many groups exist).** AQC is stable; the performance improves (variance is eliminated) as the amount of sharing increases (i.e., the number of groups decreases).

6.3.4 Knowledge Scope

Experiment Design: The advantage of our algorithm is that we use historical information, such as previous query bid and if they succeeded, to adapt the new query bids based on current system status. One of our hypotheses is that the more information we have, the more precise our prediction will be, and the better our algorithm will perform. On the other hand, we hope that our algorithm is stable enough so that the performance will not be compromised even if we do not have that much information, which is important to ad-hoc users with sporadic queries. In order to verify our hypothesis, we tested how our algorithm performs under a varying knowledge scope.

We inject two classes of users, AQC and RAN, into each trace. Each class has 60,000 queries with a mean Q_{max} of \$10. We divide AQC users into different number of groups, 1, 10, 100, 1000 and 10000, thus generating 5 testing traces. Within each group, the AQC users have **all other users'** historical information. As such, the smaller the number of groups is, the larger the size of each group is, and the more knowledge those AQC users have.

Table 6.2: **Relative Performance of AQC/RAN in Knowledge Scope.** With only 0.01% knowledge, AQC beats RAN by 30%. With 1% knowledge, AQC performs 85% better.

Group Number	1	10	100	1,000	10,000
$\frac{\text{success ratio}(AQC)}{\text{success ratio}(RAN)}$	1.96	1.89	1.85	1.30	1.29

Results (Figure 6.11 and Table 6.2) We report the average query success ratio and their 95% confidence intervals for AQC users with different knowledge scopes in Figure 6.11. We present the relative performance (AQC's divided by RAN's success ratio) in Table 6.2.

Figure 6.11 clearly demonstrates that the more information we have, the larger success ratio our algorithm can achieve and the less variance among results of different groups. When we have 1,000 and 10,000 groups (or 0.1% and 0.01% knowledge), the results have large variance. At a 95% confidence interval, the query success ratio will fall in [40%, 84%]. However, when we have less than 100 groups (> 1% knowledge), the results are almost deterministic. Another observation is that as the knowledge scope becomes larger, the performance does not increase linearly. After our information reaches 1%, the performance encounters the plateau, which means that 1% of the global information available is good enough to support our algorithm running at peak performance.

Our hypothesis is also confirmed by Table 6.2 which shows the success ratio of AQC divided by the success ratio of RAN. With only 0.01% and 0.1% knowledge, our algorithm outperforms RAN by around 30%. With more than 1% information, our query success ratio is 85% larger than RAN users.

6.4 SUMMARY

In this chapter, we turn our attention to the user side of the equation and propose *user strategies to adapt Quality Contracts over time*. Towards this, we identified two important issues in QC adaptation schemes: *payment expectation* and *savings ability*. Specifically, we

proposed the Adaptive Quality Contracts (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%.

7.0 CONCLUSIONS AND FUTURE WORK

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 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.

Given the Quality Contracts framework, we developed load management and scheduling techniques to optimize the overall system performance. As the first step, we proposed a load management scheme, UNIT, which optimizes USM (a simplified version of Quality Contracts) to lower the requirement of in-database scheduling so that UNIT load management can be applied on top of most general web-databases. UNIT uses a feedback control mechanism and relies on an intelligent admission control algorithm along with a new update frequency modulation scheme in order to maximize the overall user satisfaction. Our evaluation shows that UNIT performs better than two baseline algorithms and the current state-of-the-art when tested using workloads generated from real traces.

To fully support the Quality Contracts framework, we proposed a scheduling scheme, QUTS, to schedule updates along with queries (with Quality Contracts attached) in Web-databases. QUTS is a two-level scheduling algorithm that adaptively 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 Quote.com, 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 two parameters.

Furthermore, we combine the load management and scheduling work in our demo, QuiX (QUality-aware Integrated admission Control and Scheduling). Admission control in QuiX focuses exclusively on queries, to filter out infeasible queries and prevent the possible penalty from failing an admitted transaction. QuiX does not need to perform load shedding on updates, because useless updates will be postponed by QUTS scheduling and have few chances to hurt the quality of queries.

To advocate the use of Quality Contracts, we also turned our attention to the user side of the equation. Given competition from other users, a user will need to adapt his/her QCs over time to maximize the number of queries executed within his/her satisfaction given a certain amount of budget. Towards this, we identified two important issues in QC adaptation schemes: *payment expectation* and *savings ability*. We proposed the Adaptive Quality Contracts (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. AQC switches between Overbid mode (according to the current payment expectation) and Deposit mode (to sustain savings ability). We performed an extensive simulation study with real traces, which showed that AQC consistently outperforms baseline algorithms by up to 233%.

In this work, we proposed Quality Contracts for users to specify their preferences over multiple quality metrics. Towards "implementing" those preferences in system optimization, we focused on quality metrics including response time and staleness. As part of our future work, we plan to explore more dimensions of quality metrics, such as the virtual attributes as we have discussed in Section 3.2.4. Virtual attributes can be defined over other attributes, possibly including statistics of the entire system. Although it poses quite a challenge with comparison of the individual query's performance to system-wide measures, it might be very appealing from users' point of view: it is probably harder for 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.

APPENDIX

QUIX SYSTEM DEMONSTRATION

We named our system *QuiX*, which is short for QUality-aware Integrated admission Control and Scheduling. The QuiX system consists of two parts: the user console and the web-database server simulation. In the presence of Quality Contracts (QCs), the users and the server have distinct objectives. Users want to have most of their queries executed within their satisfactions, whereas the server wants to maximize its profit earned by completing user queries.

Figure A1 shows a screen shot of one user console. On the top is the user input panel, where a user can specify his/her favorite QC adaptation algorithm, the number of queries, and the amount of budget he/she wants to spend in the next batch. Supported QC adaptation algorithms include Fixed, Random, and Dynamic as described in Section 6.1.4 and AQC as decribed in Section 6.2. To the right hand side of the user input panel shows the Quality Contracts, where the users can specify the quality constraints of their queries as well as the relative importance between QoS and QoD, through dragging the lines between the axes. After these settings, the user clicks the "Start" button to send the information to the server, where his/her queries will be concurrently running with other users' queries as well as the background updates.

After all his/her queries are done, the performance statistics will be sent back to the user console and shown on the output panel at the bottom in Figure A1. The user will be informed with how much money has been spent and how many queries have been successful. There will also be over-time plot showing the money expenditure for each query. By dragging

the vertical line on the over-time plot, the user can check out the performance of a individual query shown on the right.

Our demonstration shows the power and flexibility of QuiX system. QuiX allows users to interactively set their Quality Contracts, send queries, observe the performance, and then adaptively modify their strategies to fit the environmental changes in the system so that users can get the best service.

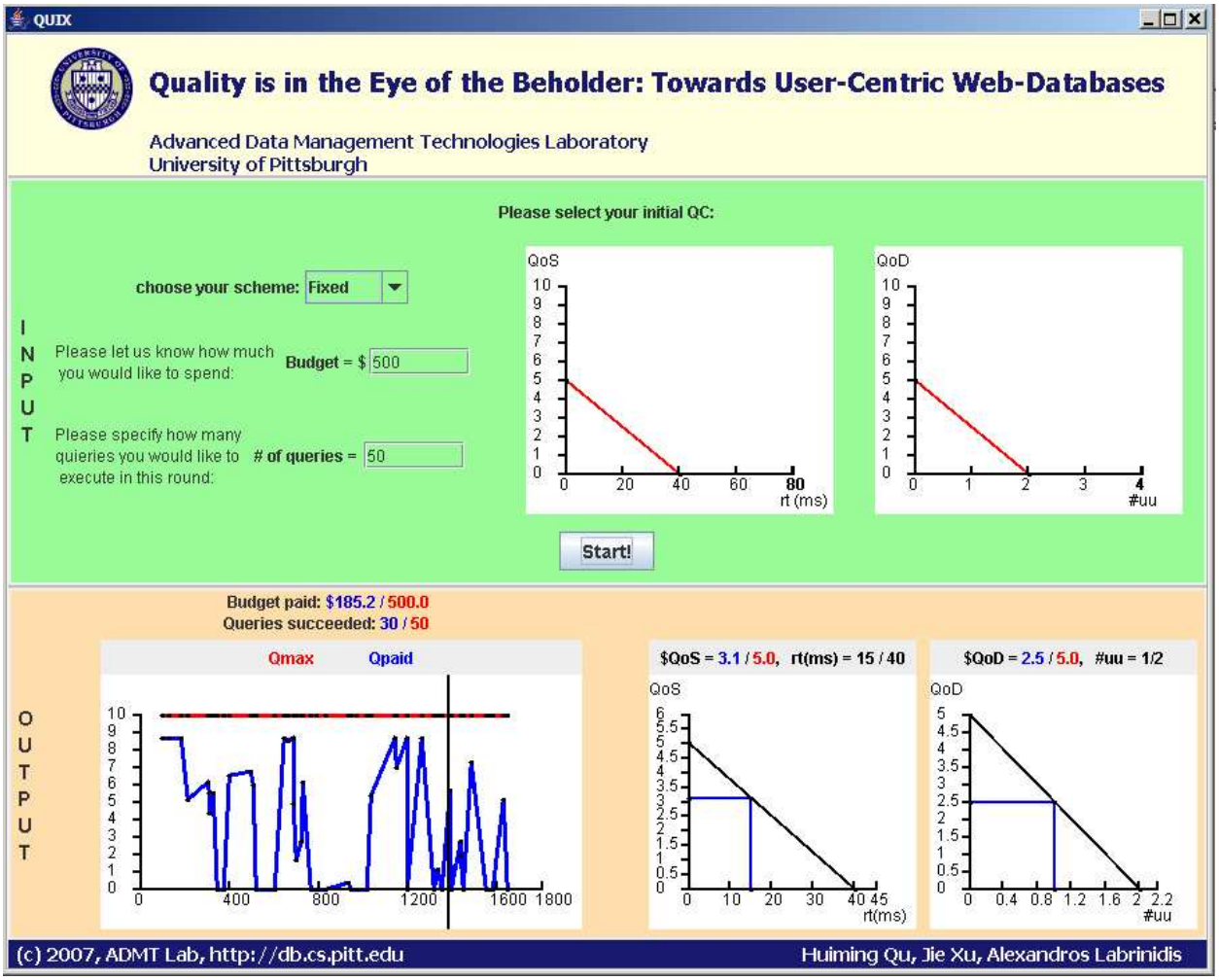


Figure A1: QuiX System Demonstration

BIBLIOGRAPHY

[1] Transaction processing performance council. http://www.tpc.org/.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[3] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

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

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

[6] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, pages 223–240, London, UK, 1996. Springer-Verlag.

[7] V. Agarwal, G. Dasgupta, K. Dasgupta, A. Purohit, and B. Viswanathan. Deco: Data replication and execution co-scheduling for utility grids. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC '06: Proceedings of the 4th International Conference on Service Oriented Computing*, volume 4294, pages 52–65. Springer, 2006.

[8] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, November 2003.

[9] C. Amza, E. Cecchet, A. Chanda, S. Elnikety, A. Cox, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck characterization of dynamic web site benchmarks. Technical Report TR02-388, Rice University, 2002.

[10] A. AuYoung, L. Grit, J. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *HPDC '06: Proceedings of 15th IEEE International Symposium on High Performance Distributed Computing*, pages 119– 131, June 2006.

[11] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000.

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, 2004.

[13] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, Washington, DC, USA, 2004. IEEE Computer Society.

[14] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[15] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 163–174, New York, NY, USA, 2001. ACM Press.

[16] 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.

[17] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[18] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid, 2000.

[19] R. Buyya, D. Abramson, and J. Giddy. A case for economy grid architecture for service oriented grid computing. In *IPDPS '01: Proceedings of the 10th Heterogeneous Computing Workshop*, Washington, DC, USA, 2001. IEEE Computer Society.

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

[21] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, 2002.

[22] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB '03: Proceedings of the 29th International Conference on Very Large Data Bases*, Sept. 2003.

[23] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[24] J. Chen, D. J. DeWitt, F. Tian, and Y .Wang. NiagaraCQ: A scalable continuous query system for internet databases. volume 29, pages 379–390, New York, NY, USA, 2000. ACM Press.

[25] M. Cherniack, E. F. Galvez, M. J. Franklin, and S. B. Zdonik. Profile-driven cache management. In *ICDE '03: Proceedings of the 19th International Conference on Data Engineering*, pages 645–656, 2003.

[26] Y. Chi, H. Wang, and P. S. Yu. Loadstar: load shedding in data stream mining. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1302–1305. VLDB Endowment, 2005.

[27] J. Cho and H. García-Molina. Synchronizing a database to improve freshness. In *Proceedings of SIGMOD '00*, pages 117–128, New York, NY, USA, 2000. ACM Press.

[28] J. Cho, H. García-Molina, and L. Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1–7):161–172, 1998.

[29] B. Chun, Y. Fu, and A. Vahdat. Bootstrapping a distributed computational economy. In *Proceedings of 1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[30] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[31] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The next step in web services. *Communications of the ACM*, 46(10):29–34, 2003.

[32] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM Systems Journal*, 43(1):136–158, 2004.

[33] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[34] A. Datta, K. Dutta, H. Thomas, D. E. VanderMeer, Suresha, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2002.

[35] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 98–107, New York, NY, USA, 2004. ACM Press.

[36] E*Trade. http://www.etrade.com/2secondgurantee/.

[37] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 251–262, New York, NY, USA, 1999. ACM.

[38] D. F. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. *Economic models for allocating resources in computer systems*, pages 156–183. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

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

[40] M. E. Gómez and V. Santonja. A new approach in the modeling and generation of synthetic disk workload. In *MASCOTS '00: Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 199, Washington, DC, USA, 2000. IEEE Computer Society.

[41] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 243–252, New York, NY, USA, 1994. ACM.

[42] M. A. Hammad, M. F. Mokbel, M. H. Ali, Walid G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, 2004.

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

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

[45] 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.

[46] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *RTSS '91: Prooceedings of the 12th Real-Time Systems Symposium*, pages 232–243, 1991.

[47] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, 1992.

[48] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD '77: Proceedings of the 1977 ACM SIGMOD international conference on management of data*.

[49] D. Hong, T. Johnson, and S. Chakravarthy. Real-time transaction scheduling: a cost conscious approach. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on management of data*, pages 197–206, New York, NY, USA, 1993. ACM Press.

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

[51] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, Berkeley, CA, USA, 1997. USENIX Association.

[52] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM Press.

[53] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 48–59, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[54] K. D. Kang, S. H. Son, and J. A. Stankovic. 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.

[55] A. Keller, G. Kar, H. Ludwig, A. Dan, and J. Hellerstein. Managing dynamic services: A contract-based approach to a conceptual architecture. In *NOMS '02: Proceedings of the 8th Network Operations and Management Symposium*, April 2002.

[56] P. Konana, A. Gupta, and A. B. Whinston. Integrating user preferences and real-time workload in information services. *Information Systems Research*, 11(2):177–196, 2000.

[57] G. Koutrika and Y. Ioannidis. Personalization of queries in database systems. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 597, Washington, DC, USA, 2004. IEEE Computer Society.

[58] S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper. Qed: Quality of service enabled databases. In *ICSOC '06: Proceedings of the 4th International Conference on Service Oriented Computing*. Springer, 2006.

[59] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *BIRTE '06: Proceedings of the First International Workshop on Business Intelligence for the Real Time Enterprise*, September 2006.

[60] A. Labrinidis and N. Roussopoulos. Webview materialization. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 367–378, New York, NY, USA, 2000. ACM Press.

[61] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *VLDB '03: Proceedings of the 29th International Conference on Very Large Data Bases*, pages 393 – 404, September 2003.

[62] 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, September 2004.

[63] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Proceedings of 1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[64] P. Larson, J. Goldstein, and J. Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, pages 177–189, Washington, DC, USA, 2004. IEEE Computer Society.

[65] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Systems*, 23(1-2):85–126, 2002.

[66] 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 '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[67] A. Merchant, K. Wu, P. S. Yu, and M. Chen. Performance analysis of dynamic finite versioning for concurrent transaction and query processing. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 103–114, New York, NY, USA, 1992. ACM Press.

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

[69] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)*, Philadelphia, PA, USA, 2000.

[70] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[71] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. Gehrke. Online scheduling to minimize average stretch. In *IEEE Symposium on Foundations of Computer Science*, pages 433–442, 1999.

[72] F. Naumann. *Quality-driven query answering for integrated information systems.* Springer-Verlag New York, Inc., 2002.

[73] X. T. Nguyen, R. Kowalczyk, and J. Han. Using dynamic asynchronous aggregate search for quality guarantees of multiple web services compositions. In *ICSOC '06: Proceedings of the 4th International Conference on Service Oriented Computing*. Springer, 2006.

[74] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 73–84, New York, NY, USA, 2002. ACM Press.

[75] S. Pandey and C. Olston. User-centric web crawling. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 401–411, New York, NY, USA, 2005. ACM Press.

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

[77] S. Papastavrou, G. Samaras, P. Evripidou, and P. K. Chrysanthis. Fine-grained parallelism in dynamic web content generation: The parse & dispatch approach. In *Proceedings of IFCIS Cooperative Information Systems*, pages 573–588, November 2003.

[78] S. Papastavrou, G. Samaras, P. Evripidou, and P. K. Chrysanthis. A decade of dynamic web content: A structured survey on past and present practices and future trend. In *IEEE Communications Surveys and Tutorials*, pages 8(2):52–60, June 2006.

[79] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, New York, NY, USA, 2001. ACM Press.

[80] H. Qu and A. Labrinidis. Preference-aware query and update scheduling in web-databases. In *ICDE '07: Proceedings of the 23nd International Conference on Data Engineering (ICDE'07)*, pages 356–365, Washington, DC, USA, April 2007. IEEE Computer Society.

[81] H. Qu, A. Labrinidis, and D. Mossé. Unit: User-centric transaction management in web-database systems. In *ICDE '06: Proceedings of the 22nd International Conference*

on Data Engineering (ICDE'06), Washington, DC, USA, April 2006. IEEE Computer Society.

[82] H. Qu, J. Xu, and A. Labrinidis. Quality is in the eye of the beholder: towards user-centric web-databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1106–1108, New York, NY, USA, 2007. ACM Press.

[83] Quote.com. http://new.quote.com/.

[84] A. Raha, S. Kamat, and W. Zhao. Admission control for hard real-time connections in ATM LANs. In *INFOCOM (1)*, pages 180–188, 1996.

[85] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium*, page 298, Washington, DC, USA, 1997. IEEE Computer Society.

[86] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical solutions for QoS-based resource allocation. In *RTSS '98: Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 296–306, 1998.

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

[88] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 55–60, Washington, DC, USA, 2005. IEEE Computer Society.

[89] C. Rusu, R. Melhem, and D. Mossé. Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing*, 1(2):271–283, 2005.

[90] L. Sha, R. Rajkumar, and J. P. Lehooczky. Concurrency control for distributed real-time databases. *SIGMOD Record*, 17(1):82–98, 1988.

[91] L. Sha, S. H. Son, R. Rajkumar, and C. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, 1991.

[92] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 511–522. VLDB Endowment, 2006.

[93] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in mariposa. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 485–494, Washington, DC, USA, 1996. IEEE Computer Society.

[94] R. M. Sivasankaran, J. A. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham. Priority assignment in real-time active databases. *The VLDB Journal*, 5(1):019–034, 1996.

[95] J. A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763, 1996.

[96] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5(1), 1996.

[97] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB '03: Proceedings of the 29th International Conference on Very Large Data BasesVLDB*, 2003.

[98] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB '03: Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.

[99] Y. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 787–798. VLDB Endowment, 2006.

[100] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, 1995.

[101] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational Grid. *The International Journal of High Performance Computing Applications*, 15(3):258–281, Fall 2001.

[102] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Time/utility function decomposition techniques for utility accrual scheduling algorithms in real-time distributed systems. *IEEE Transactions on Computers*, 54(9):1138–1153, 2005.

[103] M. Xiong, S. Han, and K. Lam. A deferrable scheduling algorithm for real-time transactions maintaining data freshness. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 27–37, Washington, DC, USA, 2005. IEEE Computer Society.

[104] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 411–421, New York, NY, USA, 2003. ACM Press.

[105] D. Zhu, D. Mossé, and R. Melhem. Power-aware scheduling for and/or graphs in real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):849–864, 2004.

[106] S. Zilberstein and S. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213, 1996.