Quality of Service Enabled Database Applications

S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper

TU München, D-85748 Garching, Germany
{krompass,gmach,scholza,seltzsam,alfons.kemper}@in.tum.de

Abstract. In today's enterprise service oriented software architectures, database systems are a crucial component for the quality of service (QoS) management between customers and service providers. The database workload consists of requests stemming from many different service classes, each of which has a dedicated service level agreement (SLA). We present an adaptive QoS management that is based on an economic model which adaptively penalizes individual requests depending on the SLA and the current degree of SLA conformance that the particular service class exhibits. For deriving the adaptive penalty of individual requests, our model differentiates between *opportunity costs* for underachieving an SLA threshold and *marginal gains* for (re-)achieving an SLA threshold. Based on the penalties, we develop a database component which schedules requests depending on their deadline and their associated penalty. We report experiments of our operational system to demonstrate the effectiveness of the adaptive QoS management.

1 Introduction

Future business software systems will be designed as service oriented architectures. These services are accessed via the Internet by a variety of different users – as exemplified by providers and vendors of Web-based business software, including RightNow Technologies, Salesforce.com, hosted SAP, and Oracle. This Web-based software is characterized by a multitude of services which invoke other enterprise services and ultimately submit requests to databases. The Web-based business software is made accessible for a multitude of customers, where each customer may have individual quality of service (QoS) requirements. The more customers access the services, the more they compete for system resources. In an uncontrolled environment this may lead to unpredictable and unacceptable response times. To prevent the customers from suffering bad performance in terms of response times of their invoked services, service level agreements (SLAs) are negotiated.

An SLA is a formal agreement between the service provider and a customer. The establishment of an SLA imposes obligations on the service provider regarding the service level of the provided services. If the constraints formulated in the SLA are violated after a certain time window, the *evaluation period*, the service provider is fined. The penalty depends on the severity of the SLA violation and is negotiated in the SLA. SLAs are typically only defined for services directly invoked by customers. Thus, the goal is to establish an end-to-end control for the quality of service, which covers all layers of the Web service architecture.

The contribution of this paper is to enable QoS for the bottom layer of a service infrastructure, where almost all services access a shared database. This is a very common scenario in mission-critical enterprise services that rely on an integrated database. For this scenario, we assume that an SLA for every service submitting requests to the database has been negotiated. Due to the multitude of services which access the database, the workload of the database consists of requests stemming from many different customers with different service classes, each having a dedicated SLA.

The challenge is to schedule incoming database requests in order to meet the performance goals specified in the SLAs. Scheduling is based on *adaptive priorities* which are derived from the current level of conformance with the request's SLA, that is, the percentage of timely requests, and the economic importance of this SLA relative to other pending requests' SLAs.

Current solutions in database systems, e.g., the Query Patroller for DB2 [7] or the Oracle Resource Manager [13], assign groups of customers to performance classes with static priorities. Thus, each request is assigned its priority depending solely on the client by whom it has been submitted. This *static prioritization* is used to schedule the requests, so that high-priority clients should complete faster on average than their low-priority counterparts.

This approach is sufficient to fulfill the requirements of particularly valuable customers. However, it cannot adequately manage overall SLA enforcement. Consider an SLA which requires 90% of all service requests to be processed within a certain time window. With static prioritization, SLAs for high-priority customers are likely to be overfulfilled by processing almost all requests in time. However, during peak-load times, it is likely that they overachieve their SLAs at the expense of lower-priority users. From a business-oriented point of view, it is desirable to provide only the service level which has been negotiated in the SLA. If SLAs are not overfulfilled, the additional free resources are used for satisfying SLAs that are violated with the static prioritization.

For this purpose, we developed a QoS management concept based on an economic model which adaptively prioritizes individual requests depending on the SLA and the current degree of SLA conformance that the particular service class exhibits. The core of the QoS management consists of *penalty-carrying requests*, that is, database requests which carry the requirements needed to fulfill the SLA constraints from the submitting service to the database.

The rest of the paper is organized as follows: Section 2 describes the two cost components, *marginal gains* and *opportunity costs*, of our QoS model in detail and presents the adaptive QoS management with which penalty-carrying requests are derived. Section 3 describes the system architecture and the implementation of our QoS management. The scheduling of the requests is in the focus of Section 4, followed by the evaluation results of our prototypical implementation in Section 5. An overview of related work is presented in Section 6.

Finally, in Section 7, we summarize the conclusions of our study and outline ongoing and future research on this subject.

2 Quality of Service Model

The central concept of our quality of service management is adaptive penalization of individual requests according to the current degree of *SLA conformance* c. The conformance is monitored per service class, that is, for each transaction type invoked by an individual customer and the associated SLA. We define c as

$$c = \frac{\text{Number of timely transaction invocations}}{\text{Total number of invocations of the transaction}}$$

In practice, so-called *step-wise SLAs* are commonly used to specify the QoS requirements of a service class. The SLAs consist of one or more *percentile constraints* and an optional *deadline constraint.* Percentile constraints require $n\%$ of all service requests to be processed within x seconds. If a percentile constraint is violated after the evaluation period, a penalty p for every m percentage points under fulfillment is due. Furthermore, p_{max} defines a maximum penalty for violating a percentile constraint. The deadline constraint – which does not incur any penalty – specifies an upper bound for the execution time of the service request. An example for a step-wise SLA with one percentile constraint d_1 and one deadline constraint d_2 is shown in the following:

d_1: 90% in less than $5s$; $p = \$900$ per 10 percentage points of underfulfillment, $p_{max} = \$1800$; evaluation period: 1 month

d_2: Deadline $15s$

The constraints above control the response time and the throughput of Web service transactions. In general, SLAs contain additional constraints such as sizing constraints which restrict the maximum number of transaction invocations per time period. We concentrate on fulfilling the percentile and deadline constraints, assuming any additional SLA constraints are obtained.

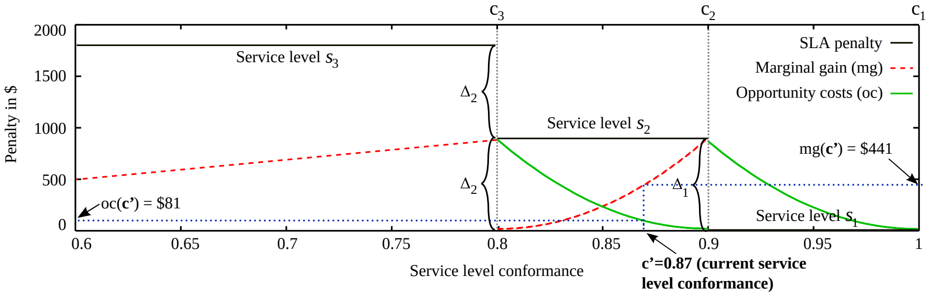

Fig. 1. Visualization of SLA constraint d_1

A percentile constraint in a fixed step-wise SLA implicitly defines an SLA penalty function with n steps. The penalty function for d_1 of our sample SLA is shown

as the step function in Figure 1 (black solid lines). With c_i, $1 \leq i \leq n+1$, we denote the boundaries of the steps of the SLA penalty function. For the example in Figure 1, we have $c_4 = 0$ (not in the figure), $c_3 = 0.8$, $c_2 = 0.9$, and $c_1 = 1$. Using the SLA penalty function, we define *service levels* as follows: For a penalty function with n steps, let $s_i, 1 \leq i \leq n$, denote the ith service level. This level is defined in the interval $[c_{i+1}, c_i[$, so that dropping to a lower service level corresponds to a higher penalty. Thereby, s_{i+1} denotes a lower service level than s_i, that is, the penalty incurred at s_{i+1} is higher than at s_i. We denote Δ_i as this cost difference between s_{i+1} and s_i.
As shown in Figure 1, our sample percentile constraint d_1 implicitly defines three service levels: Service level s_3 is defined in the interval $[0, 0.8[$, s_2 in $[0.8, 0.9[$, and s_1 in $[0, 1]$. The cost difference between service levels s_3 and s_2 is \$900 which is identical to the cost difference between s_2 and s_1.

2.1 Penalty-Carrying Requests

Penalty-carrying requests are queries with attached penalty information in a SQL-comment. For example, the penalty-carrying request for a `select`-Statement looks like this:

```
/* penalty ...
 * deadline ... */
select ... from ...
```

We use the SLA penalty function to compute these adaptive penalties for individual service requests. In the following section, we describe how to compute the adaptive penalty from the percentile constraint for an individual request. Then, we describe briefly the derivation of the deadline constraint for an individual query.

2.2 Deriving the Penalty for Individual Requests

The penalty of an individual request is covering two different economic aspects. On the one hand, the *opportunity costs* model the danger of falling into the next lower service level. If the current SLA conformance $\mathbf{c}$ converges to the next lower service level, the penalty for processing the service too late increases, because delaying a further request increases the danger of an ultimate SLA violation. Then, the opportunity costs oc are piece-wise defined quadratic functions which are defined as follows:

$$oc(\mathbf{c}) := \begin{cases} \left(\frac{c_{n-1}-\mathbf{c}}{c_{n-1}-c_n}\right)^2 \cdot \Delta_{n-1}, & c_n \leq \mathbf{c} < c_{n-1} \\ \dots & \\ \left(\frac{c_1-\mathbf{c}}{c_1-c_2}\right)^2 \cdot \Delta_1, & c_2 \leq \mathbf{c} < c_1 \\ 0, & \text{otherwise} \end{cases}$$

The rationale for choosing squared terms is given below. For the opportunity costs, we derive the decreasing parts of the parabolas as in Figure 1.

On the other hand, with *marginal gains*, we model the chance that a service class re-achieves a higher service level, that is, reaches s_i from s_{i+1}. If this appears to be “within reach”, individual requests are penalized more and more to eventually achieve the higher level. The marginal gain mg is a piece-wise quadratic function:

$$mg(\mathbf{c}) := \begin{cases} \left(\frac{\mathbf{c}-c_{n+1}}{c_n-c_{n+1}}\right)^2 \cdot \mathit{\Delta}_{n-1}, & c_{n+1} \leq \mathbf{c} < c_n \\ \dots & \\ \left(\frac{\mathbf{c}-c_3}{c_2-c_3}\right)^2 \cdot \mathit{\Delta}_1, & c_3 \leq \mathbf{c} < c_2 \\ 0, & \text{otherwise} \end{cases}$$

Analogous to the opportunity costs, the rationale for choosing squared terms is given below. The marginal gain is depicted as increasing part of the parabolas in Figure 1.

If the SLA conformance of a request's service class is approaching the next lower service level, the chance for reaching the next higher service level is very small. Thus, the penalty of a request of this transaction is dominated by the opportunity costs. Similarly, the penalty is dominated by the marginal gain if the next higher service level is “within reach”. Therefore, we define the penalty as the maximum of the computed opportunity costs and the marginal gain of this service request.

To define opportunity costs and marginal gains, we use a squared term – resulting in the parabolas – to weight the distance from the borders of neighboring service levels. If linear terms are used, requests stemming from SLAs with high penalties are almost always be handled with top priority, because there is only a very small area in the middle of a service level where the calculated penalties are low. This leads to overfulfillment and therefore an inferior overall performance. In contrast to that, if the order of the functions is chosen too high, the request has high priority only for SLA conformances near the borders of the next higher and next lower service level, respectively. So, if the opportunity costs are defined by higher order polynomials, there are only very few requests with high priority. If all of these requests are delayed, e.g., by waiting for database locks, the SLA conformance falls onto the next lower service level. To justify this rationale, we conducted extensive experimental studies, which cannot be reported here for space limitations. These studies have shown that squared terms were better suited to model the opportunity costs and marginal gains than linear order higher order terms.

2.3 Deriving the Deadline Constraint for Individual Requests

The time constraint of a deadline constraint x_d specifies an upper bound for the processing time of a transaction. We therefore need to derive the deadlines for individual requests of that transaction. Requests which have passed their deadline are scheduled with maximum priority, that is, they are not delayed by other requests and most likely have a processing time that is less or equal to

the observed average processing time. Note that the deadline is no guarantee, as high priority requests can still be delayed if they access an object that is locked by a request with lower priority.
With enf_i, we denote the latest time at which a request r_i should be executed to be able to complete the respective transaction within the time constraint given by x_d. To compute enf_i, we monitor the execution times of requests already processed in the current transaction. In addition to that, we monitor previous invocations of the transaction and maintain the average processing time of each request. Thus, we derive the expected time to process the remaining requests by summing up the average response times of the requests. The time constraint enf_i for the current request is computed by subtracting the observed execution times and the expected time to process the remaining requests from x_d.

3 System Architecture and Implementation

To provide end-to-end quality of service for Web services, it is essential to incorporate all components of a Web service architecture, that is, the invoked service itself, all called sub-services and the databases at the bottom layer.

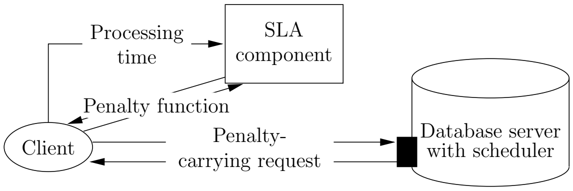

Fig. 2. Architecture Overview

A primary design goal for the implementation of the described concepts was to ease the future extension of the QoS management to entire Web service architectures. We therefore encapsulated all SLA-relevant functionality, including the monitoring of the SLA conformance and the generation of adaptive penalties, into a central entity, the *SLA component*. Figure 2 shows the resulting architecture. The SLA component can easily be extended to monitor the overall execution of Web service requests and not only derive adaptive penalties for the database layer, but also for all sub requests on the Web service layer. The adaptive penalties are piggybacked onto the corresponding requests and transported as penalty-carrying requests to the database. Upon completion of the database request, the SLA component is notified of the observed response time by the client and can thus update the current SLA conformance ratio.
The actual scheduling of requests is based on the adaptive penalties and is realized by a *scheduler*. The scheduler intercepts all arriving requests and carries out the admission control and the reordering of individual requests. The scheduler is architected as an external component so that it can be easily adapted to scheduling arbitrary service requests, besides the database requests exemplified here.

4 Request Scheduling

At the database server, the processing of a newly arriving penalty-carrying request works as follows. To prevent the database from being overloaded, the *admission control* limits the number of simultaneously executing requests. If the request it not immediately executed, it is *queued*. Prior to dequeueing a request, all queued requests are *scheduled*, that is, they are ordered by their priority. If there are sufficient system resources, requests are dequeued by the admission control.

In most current database systems, processes are assigned the same amount of resources, irrespective of the priority of the respective request. This implies that the available resources of the database are assigned in a round-robin manner to all active requests. In other words, all requests are equally important. To limit the database load it is therefore sufficient to restrict the number of concurrent queries, irrespective of their individual complexity [15].

As an alternative, we experimented with an admission control that is based on the optimizer costs of the requests that are currently being processed. However, our empirical studies, which cannot be shown here due to space restrictions, revealed that the query-complexity based admission control performed worse than simply controlling the multi-programming level by restricting the maximum number of concurrently processed requests.

Requests which are held back are put in one of two queues, as shown in Figure 3. Queue A holds requests which belong to running transactions, requests of transactions not yet started are maintained in queue B. Statements to be processed are chosen from queue A. Only if this queue is empty, new transactions are started by picking statements from queue B, so that running transactions are not unnecessarily delayed. Using this approach, we avoid the problem of *lock convoys* [6]. Lock convoys can arise if a transaction T_L which submits various requests to the database, exclusively locks a database object and there are pending requests of other transactions which intend to lock the same object. The queue of waiting objects does not shrink as long as the locking transaction is not finished. Before T_L releases the blocking lock, all of its requests need to be processed. Thus, intuitively, requests from active transactions are prioritized over requests from pending transactions.

Our goal is, prior to dequeuing a request, to create a schedule of the pending requests, such that the overall sum of incurred penalties is minimized. Thus, the requests are ordered in both queues according to their adaptive penalties. So, a request is inserted and removed, respectively, in $O(\log n)$ time by using a priority queue implementation, that is, the overhead for scheduling a request is negligible. For queue lengths of 150, which we observed in our benchmarks, the scheduling of a single request took about 0.28 milliseconds.

5 Performance Evaluation

We performed comprehensive benchmarks using our prototype implementation to assess the effectiveness of the adaptive request-penalization. For the perfor-

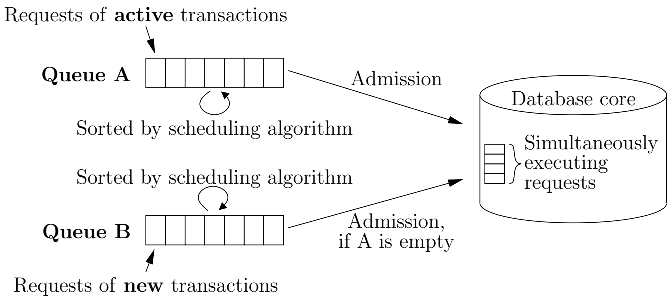
Requests of **active** transactions
Queue A
Admission
Database core
Sorted by scheduling algorithm
Simultaneously executing requests
Sorted by scheduling algorithm
Queue B
Admission, if A is empty
Requests of **new** transactions

Fig. 3. Dual Queue Scheduling

mance evaluation, we chose the TPC-C benchmark as a representative Online Transaction Processing (OLTP) workload.

5.1 Description of the Benchmarks

The TPC-C-benchmark models a company which is a wholesale supplier operating several warehouses which serve customers in geographically distributed sales districts. The database workload of the benchmark is centered around five principal business transactions of an order-entry environment. The transactions are invoked by *emulated users* whose behavior is controlled by *think times* and *keying times*. The detailed specification of the TPC-C benchmark can be found in [16].

The SLA for a transaction is based on the corresponding response time goal. For our experiments, we specified the SLAs using XML, similar to WS-Agreement [10], which is becoming a standard for establishing a service agreement between a service provider and a client. Our experiments are conducted with the step-wise SLAs introduced in Section 2. For each transaction, we define an SLA with a percentile and an deadline constraint. The percentile constraint requires 90% of the invocations to be processed in less than the corresponding response time requirement which is specified for each transaction in [16]. A violation of this constraint is fined with a penalty which depends on the terminal representing the client from which the transaction is invoked, that is, the SLA applies for the terminal and all transactions that are invoked from this terminal. In our test scenario, we chose a customer-mix where 15% of the terminals incur high ($1000), 35% incur medium ($200), and the remaining terminals incur low penalties ($40) if the corresponding SLA is violated. This customer mix models a service provider with a high number of regular customers that must be preferably processed compared to “normal” users. In order to avoid starvation of queued requests, we define an upper bound for the execution time of a transaction in our benchmark SLA for high and medium priority customers. The deadline is set to three times the response time requirement for transactions stemming from high-priority customers and four times for medium-priority transactions. There are no deadline constraints for low-priority customers.

For our experiments, we implemented our own version of the TPC-C benchmark based on MaxDB Version 7.5 [11]. The number of warehouses is held constant at 20, thus, the size of the database is about 2GB. As specified by the TPC-C, the number of terminals is ten times the number of warehouses, thus yielding a total number of 200 terminals during the benchmark.

For the benchmarks, we dimensioned the 100%-workload such that the required response times of the specification are met without any scheduling and admission control at all. Furthermore, we define a productive workload of 80%, as databases should not be operated at its limit due to possible load peaks. We control the workload by multiplying the keying and think times with a scaling factor.

A single benchmark consists of several phases. First, there is a "warmup" phase where the database is operated at 80% load for 15 minutes. Subsequently, 8 minute-periods of peak load (200% workload) alternate with "rest periods" (80% workload) which again last for 15 minutes. The benchmark terminated after an evaluation period of 65 minutes. After this time, the requests that have been accumulated in the second load peak, have been processed, so that the number of queued requests is reduced to the normal level again. The scheduler with admission control is configured such that the throughput is identical to the throughput of a benchmark with terminals directly connected to the database.

Our experiments are performed running the QoS-enabled database on a server with 1GB RAM and an Intel Xeon processor with 2.8GHz. The operating system is SUSE Linux Enterprise 9 based on kernel 2.6. All terminals run on another server with identical hardware and connect to the database via Gigabit-Ethernet using the MaxDB JDBC-driver.

5.2 Results

First, we present the analysis of the SLA conformance using static prioritization, that is, the priority of a customer remains constant throughout the entire benchmark. Figure 4 shows the SLA conformance for the NewOrder transaction which is the central transaction of the TPC-C benchmark. The values shown are the conformances at the end of the evaluation period for each of the terminals involved. The SLA conformances are ordered decreasingly, grouped by the priority of the terminals. With static prioritization, all SLAs for transactions stemming from high-priority terminals are overfulfilled, reaching an SLA conformance near 1. All low-priority terminals underachieve their SLAs. For 46% of the terminals having medium priority, the SLAs are obtained, while for 54%, the corresponding SLA is violated. This inequity between terminals having equal priority arises if transactions from one terminal compete with more high-priority transactions than the other. Due to the lack of SLA awareness, the static prioritization cannot differentiate between a transaction stemming from a customer whose SLA is currently vastly overfulfilled and a transaction where the next higher service level is within reach.

In contrast to this, the SLA conformance using adaptive prioritization is far more balanced within a group. Figure 5 shows the SLA compliance of all terminals using our novel adaptive penalization. Again, all high-priority terminals

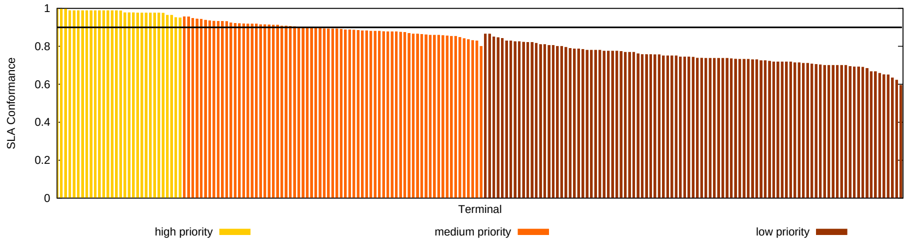

Fig. 4. SLA Conformance for all Terminals Using Static Prioritization

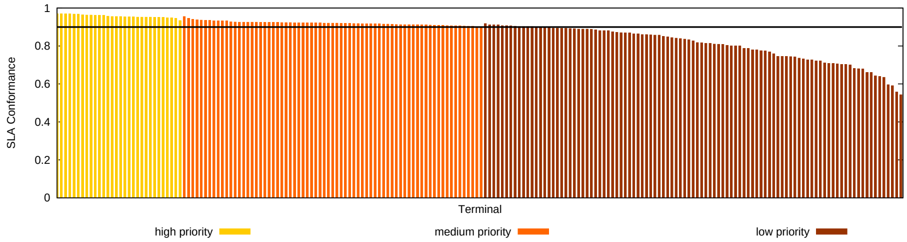

Fig. 5. SLA Conformance for all Terminals Using Adaptive Penalization

satisfy their SLAs. But the SLAs are not overfulfilled to the extent as with static prioritization, that is, the SLA conformance with static prioritization is between 95.7% and 100% and with our adaptive prioritization between 93.4% and 97%. This adaptive "down-grading" of requests stemming from high-priority terminals is used to free resources for requests from low- and medium-priority terminals. Furthermore, as requests stemming from low-priority terminals do not have deadlines, these requests are delayed as long as possible to allow the prioritized execution of higher priority requests.

If the pending requests are statically prioritized, the reduction of costs induced by violating the SLAs of the terminals is due to favoring requests stemming from high-priority terminals to lower-priority requests. For our example configuration, the costs decreased by 55%, from $\$28,080$ using the static prioritization to $\$12,600$ with the adaptive penalization.

6 Related work

Enabling QoS for Web service infrastructures is in the focus of our research group. Braumandl et al. [2] discuss distributed query processing systems on the Internet where the queries have different QoS demands. The paper presents an extension to the distributed query processing to support user QoS constraints. The query processor generates plans in such a way that its quality estimates are compliant with the user-defined quality constraints. Gmach et al. [5] present a fuzzy controller module which supervises services in a service oriented architecture. The controller executes appropriate actions to remedy overload, failure, and idle situations in the service architecture.

Quality of Service is an important issue for e-commerce and other e-services. Beeri et al. [1] analyze service compositions at compile-time stage to gain further information on the service's behavior. Selecting services which are dynamically bound to composite services at runtime to satisfy user QoS requirements is presented by Maximilien and Singh [12], and Gibelin and Makpangou [4]. However, these approaches are only applicable if there are several concrete services which implement the same interface. This is not necessarily true for enterprise services. Kraiss et al. [8, 9] describe an analytical model for the HEART tuning tool for message oriented middleware. The tool assigns static priorities to different workload classes. The messages of the different classes are then processed by a priority based scheduling algorithm in the middleware. The approach differs from our work in three points. First, there is a fixed number of workload classes. Second, for each class, the workload parameters have to manually be specified by an administrator. Third, if the workload change, the priorities for the classes have to be recomputed.

An admission control and request scheduling for e-commerce Web sites is presented by Elnikety et al. [3]. Their work focuses on achieving stable behavior during overload and improved response times. Analog to our SLA based request management component they install a proxy between the Web service and the database. However, the optimization is not associated to the SLA conformance. As we have discussed in this paper, considering the conformance is an integral part of an adaptive QoS management.

Schroeder et al. [14] present a framework for providing QoS where the response time requirements are specified in an SLA. To meet the multiclass response time goals, the number of concurrently executing requests is dynamically adjusted using a feedback control loop which considers the available hardware resources and concurrently executing queries in the database. However, other than our approach their work is not based on an economic model that optimizes the overall system performance across different classes.

7 Conclusion and Future Work

In this paper, we presented and evaluated an adaptive QoS management that is based on an economic model which adaptively penalizes individual requests depending on the SLA and the current degree of SLA conformance. Our economic model differentiates between opportunity costs and marginal gains. Using this economic model, we compute adaptive penalties and annotate them to individual requests, thus creating penalty-carrying queries. Second, we described the architecture and the implementation of our QoS management. Third, we presented the scheduling of the requests which is based on an admission control. We integrated our research prototype of a QoS-enabled database into MaxDB. Using our prototype, we demonstrated the effectiveness of our proposed approach by performing comprehensive real-world studies using the TPC-C benchmark as OLTP workload.

Having shown the effectiveness of our approach for databases, we now move towards scheduling in multi-level service infrastructures.

References

1. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes with BP-QL. In *Proceedings of the 31st International Conference on Very Large Data Bases*, Trondheim, Norway, September 2005.
2. R. Braumandl, A. Kemper, and D. Kossmann. Quality of Service in an Information Economy. *TOIT*, 3(4):291–333, 2003.
3. S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the 13th International Conference on WWW*, pages 276–286, New York, NY, USA, 2004. ACM Press.
4. N. Gibelin and M. Makpangou. Efficient and Transparent Web-Services Selection. In *Proceedings of the 3rd International Conference on Service Oriented Computing*, Lecture Notes in Computer Science (LNCS), Vol. 3826, pages 527–532, 2005.
5. D. Gmach, S. Krompass, S. Seltzsam, and A. Kemper. AutoGlobe: An Automatic Administration Concept for Service-Oriented Database Applications. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2006.
6. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
7. IBM DB2 Query Patroller. `http://www-306.ibm.com/software/data/db2/querypatroller/`.
8. A. Kraiss, F. Schön, G. Weikum, and U. Deppisch. Towards Response Time Guarantees for E-Service Middleware. *IEEE Data Engineering Bulletin*, 24(1):58–63, 2001.
9. A. Kraiss, F. Schön, G. Weikum, and U. Deppisch. With HEART Towards Response Time Guarantees for Message-Based E-Services. In *Proceedings of the 8th International Conference on Extending Database Technology*, pages 732–735, London, UK, 2002. Springer.
10. H. Ludwig and Toshiyuki. WS-Agreement Concepts, Use, and Implementation. In *Tutorial at the ICSOC*, 2005.
11. MaxDB. `http://www.mysql.com/products/maxdb/`.
12. M. Maximilien and M. P. Singh. Toward Autonomic Web Services Trust and Selection. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 212–221, New York, NY, USA, 2004. ACM Press.
13. Oracle Database Resource Manager. `http://www.oracle.com/technology/deploy/availability/htdocs/rm_overview.html`.
14. B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum. Achieving Class-Based QoS for Transactional Workloads. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2006.
15. B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum. How to Determine a Good Multi-Programming Level for External Scheduling. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2006.
16. TPC Benchmark C, Standard Specification Version 5.4. `http://www.tpc.org/tpcc/`, April 2004.