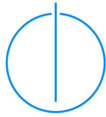
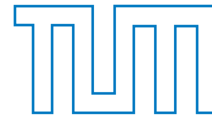


Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme



Topics in Meeting Complex Service Level Objectives for Mixed Database Workloads

Diplom-Informatiker Univ.
Stefan Krompaß

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Martin Bichler

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph. D.
2. Prof. Umeshwar Dayal, Ph. D.
   Institute of Science in Bangalore/Indien

Die Dissertation wurde am 22.12.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 24.03.2011 angenommen.

Abstract

In operational business intelligence (BI) database systems, the same database may support mixed workloads, e. g., short-running OLTP queries and batch jobs containing multitudes of queries with varying complexity. Different workloads may have different performance requirements, expressed in terms of service level objectives (SLOs). It is the task of a database administrator to manage the workload in order to fulfill the SLOs and, thus, to satisfy the objectives. This work investigates approaches and methods for using workload management policies to control complex database workloads.

We identify basic types of queries and describe the challenges they pose for SLO-aware workload management. We present a generic workload management framework that assists the administrator in managing the mixed workload. The framework implements a generic workload management architecture for a database system to meet the service level objectives. The architecture comprises three feedback control loops. The *query control loop* applies workload management policies to submitted and executing queries. The policies, e. g., when to process which queries or when to kill a particular query, are defined to ensure that the service level objectives are met. The *policy control loop* devises the policies based on the workload objectives and a characterization of the workload. It also adjusts the policies in response to expected changes in the workload. The *business control loop* negotiates service level agreements between the database service provider and the workload submitter (customer). For example, when the arrival rate of a workload is higher than specified, it may not be possible to fulfill the objectives and new service level agreements must be negotiated in order to resolve the situation. Based on this architecture, we study three different workload management scenarios.

First, we present a dynamic prioritization scheme for a workload where multiple users submit OLTP-style business transactions to a single database system. A common service level objective in such a transaction-processing business workload defines constraints on the percentile response time of the transactions started on behalf of a user. The objectives also define a deadline to avoid the starvation of transactions. The constraints on the percentile of the response time incur a penalty while the deadline enforces the execution of requests. The individual users may have different priorities, which are derived from the penalties defined for violating the constraints on the percentile response time. We devise an adaptive quality of service (QoS) management policy that is based on an economic model, which adaptively penalizes individual requests depending on the objectives of the respective service class and the current degree of service level conformance that the particular service class exhibits. We show that by using the dynamic prioritization, we can avoid

that high-priority user classes over-exceed their objectives at the expense of their lower-priority counterparts.

Second, mixed query workloads that run against very large data warehouses may contain queries whose execution times range, sometimes unpredictably, from seconds to hours. The presence of even a handful of long-running queries can significantly slow down a workload consisting of thousands of queries, creating havoc for queries that require a quick response. We present a systematic study of workload management policies in the query control loop, including many policies implemented by commercial database vendors. We build a taxonomy for long-running queries based on how they impact other queries. We describe an experimental framework that allows us to carry out an extensive set of experiments to evaluate different management policies and the relative and absolute thresholds that they may use. Based on the experiments, we make recommendations for which policies to use, when to apply them, and demonstrate how to set their thresholds.

Third, we consider database systems that run mixed workloads consisting of multiple service classes (sets of queries) where each class may have compound objectives, e. g., desired throughput and average response time. The database system provides control parameters for each service class where each parameter may be dynamically adjusted to affect the performance metrics and so achieve the class objectives. For example, the number of concurrently executing queries may be higher for a high priority class than for a lower priority class. We formulate the problem of setting the control parameters as a multi-dimensional search problem. We devise a model for the search space and explain why it cannot be searched exhaustively. We then present an algorithm that heuristically searches the space to find control parameter settings that satisfy all objectives for all classes or else indicates that the objectives are unsatisfiable. Our algorithm considers changes in the workload, e. g., when a new service class arrives or objectives of active service classes change. Our algorithm is contrasted with existing algorithms for mixed workloads where each service class has a single objective. We improve on that work by handling compound objectives per class. A performance analysis of our algorithm over different problem scenarios validates the practical utility of our approach.

Contents

1. Introduction **12**
- 1.1. Problem statement . . . 12
- 1.2. Design of experimental frameworks . . . 17
- 1.3. Contributions . . . 18
- 1.4. Outline . . . 20

2. Workload management in commercial database systems **21**
- 2.1. Query control loop . . . 21
- 2.2. Policy control loop . . . 25

3. Dynamic request prioritization for OLTP workloads **26**
- 3.1. Related work . . . 28
- 3.2. Service level objectives . . . 30
  - 3.2.1. Percentile objectives . . . 30
  - 3.2.2. Deadline objectives . . . 31
- 3.3. Architecture . . . 32
- 3.4. Penalty functions . . . 33
  - 3.4.1. Adaptive penalty function . . . 33
- 3.5. Request scheduling . . . 38
- 3.6. Experimental setup . . . 40
- 3.7. Experiments . . . 42
  - 3.7.1. Overhead . . . 42
  - 3.7.2. Evaluation of dynamic prioritization . . . 43
- 3.8. Conclusions and future work . . . 51

4. Query control for BI workloads **54**
- 4.1. Related work . . . 55
- 4.2. Long-running query taxonomy . . . 56
- 4.3. Experimental framework . . . 57
  - 4.3.1. Workflow . . . 58
  - 4.3.2. Simulator implementation . . . 59
  - 4.3.3. Experiment input and output . . . 59
  - 4.3.4. Validation against HP Neoview . . . 60
- 4.4. Experimental setup . . . 61
  - 4.4.1. Queries and query types . . . 61
  - 4.4.2. Workloads . . . 63
  - 4.4.3. Workload management policies . . . 63

4.4.4. Workload objective functions . . . 67
4.5. Results . . . 68
4.5.1. Admission control . . . 68
4.5.2. Scheduling . . . 70
4.5.3. Execution control: kill thresholds . . . 72
4.5.4. Execution control: different actions . . . 74
4.5.5. Execution control: overload situations . . . 76
4.6. Conclusions . . . 77

5. Policy control for mixed workloads 79
5.1. Related work . . . 80
5.1.1. Single control parameter . . . 81
5.1.2. Multiple control parameters, single performance objective . . . 81
5.1.3. Workload adaptation — maximize single objective . . . 81
5.2. Problem statement . . . 82
5.3. Finding points in the operating envelope . . . 87
5.3.1. Move along single dimension – single objective . . . 89
5.3.2. Extension for compound objectives . . . 91
5.4. Variance . . . 95
5.5. Operating envelope changes . . . 96
5.6. Experiments . . . 98
5.6.1. Extension of the workload adaptation approach . . . 98
5.6.2. Experimental setup . . . 99
5.6.3. Experimental framework . . . 100
5.6.4. Scenario 1: two service classes, no changes . . . 101
5.6.5. Scenario 2: change objectives, add new service class . . . 105
5.7. Dashboard . . . 112
5.7.1. Problem injection . . . 115
5.7.2. Attempt at manual correction . . . 116
5.7.3. Policy control feedback loop . . . 117
5.8. Conclusions and future work . . . 117

6. Conclusions 119

A. TPC-C 122

B. TPC-CH benchmark 125
B.1. DDL for TPC-CH . . . 125
B.2. OLAP query suite in the TPC-CH benchmark . . . 126

Bibliography 135

List of Figures

1.1. Visualization of the service mapping problem 14

1.2. The main components of the workload management architecture are the workload manager and the policy controller. The workload manager controls the execution of individual queries based on measurements it receives from the database system. The rectangles show the query control (dotted rectangle), policy control (dashed rectangle), and business control loop (dot-dashed rectangle). 15

3.1. Visualization of SLO objective d_1 31

3.2. Architecture overview 32

3.3. Penalty function that encodes a penalty p for starting the execution of a request after time t. 34

3.4. Visualization of the economic model 35

3.5. Opportunity costs in the interval $[0.9, 1.0]$. There are two functions shown: one for a high priority percentile objective (red solid line) and one for a lower priority percentile objective (blue dashed line). 36

3.6. Response time for transaction (taken from SLO), (estimated) execution time for requests, slack 38

3.7. Pseudo code of the Fisher-Krieger algorithm 39

3.8. Pseudo code of KAFKA 39

3.9. (FIFO, no deadlines) The graphs show the results after running a benchmark where the scheduler controls the parallelism in the database and puts the requests in a FIFO queue. 45

3.10. (Static, no deadlines) The graphs show the results after running a benchmark where the requests have static priorities and the scheduler always executes the highest priority request in the queue. 47

3.11. (Dynamic, MEFI, no deadlines) The graphs show the results after running a benchmark where the requests have dynamic priorities and the queue is managed by the MEFI scheduling discipline. 48

3.12. (Dynamic, KAFKA, no deadlines) The graphs show the results after running a benchmark where the requests have dynamic priorities and the queue is managed by the KAFKA scheduling discipline. 50

3.13. (Dynamic, KAFKA, with deadlines) The graphs show the results after running a benchmark where the requests have dynamic priorities and the queue is managed by the KAFKA scheduling discipline. The execution of requests is enforced after 15 seconds, 25 seconds, and 50 seconds for requests stemming from high, medium, and low priority terminals. 52

4.1. Workflow of how we create and select from a pool of query objects, create workload input files, and specify parameters for our experiments. 58
4.2. Simulator validation: We compare the elapsed times of queries run on an HP Neoview database with their simulated elapsed times. A straight diagonal line of points would indicate a perfect correlation. Note that 5000 simulator time units corresponds to roughly 30 minutes of elapsed time on the real system. 61
4.3. Throughput (queries per hour, QPH) of the same workload when run on the Neoview four processor machine and on the simulator. For the simulator, we derived the QPH using the 30 minutes == 5000 simulator time units formula. 62
4.4. Comparison of estimated and actual CPU time for each query: the CPU time of the *surprise-heavy* queries is underestimated. The dashed vertical lines indicate our admission thresholds and the solid diagonal line shows our kill relative threshold. 64
4.5. Comparison of elapsed times of long-running queries in Expected-Heavy and Surprise-Hog workloads at MPL=4. All *surprise-hog* queries complete faster than their *expected-heavy* counterparts because they get a larger share of the resources. The dashed lines indicate our absolute kill thresholds. 65
4.6. Comparison of elapsed times of feathers and golf balls in Expected-Heavy and Surprise-Hog workloads at MPL=4. Queries above the diagonal run slower in the Surprise-Hog workload. The dashed lines indicate our absolute kill thresholds. The gray shaded area denotes *starving* queries. 66
4.7. Comparison of admission thresholds on different queries and workload types. The notations nA and nC above the bars indicate the number of admitted and completed long-running queries (out of the three submitted in each workload). Admission control is less effective when cost estimates are less accurate. Lower thresholds reject more "good" queries. 69
4.8. Admission thresholds are much less effective when the workload includes long-running queries that make heavy use of resources not measured by the admission threshold. In this case, the queries were disk-bound but admission control looked at CPU time estimates. . . 70

4.9. Comparison of absolute and relative *kill* thresholds in the Expected-Heavy workload: The lower absolute threshold kills many more queries unless their progress is checked. 72
4.10. Comparison of absolute and relative *kill* thresholds in the Disk-Heavy workload: the relative threshold compares actual and estimated CPU time and thus does not catch the *disk-heavy* queries. 73
4.11. Comparison of execution control policies with admission threshold *1.0m*. 75
4.12. Comparison of execution control policies using admission threshold *1.0m* at MPL=4 and MPL=10 (overload) for the Surprise-Heavy workload. All queries take longer at MPL=10, so policies with absolute thresholds kill more queries. 77

5.1. Control parameter-negative average response time hull for service class s_C when executed in parallel with service class s_H at different $MPL_1 - MPL_2$-settings. 84
5.2. Control parameter-negative average response time hull (from Figure 5.1) for service class s_C with lower and upper bounds. The resulting acceptable region is shown on the $MPL_1 - MPL_2$-plane. 85
5.3. Boundaries of the acceptable throughput- (solid) and average response time-regions (dashed) for classes s_1 (black) and s_2 (gray). The shaded area denotes the operating envelope. 86
5.4. The starting region for a workload with two service classes (and, thus, two control parameters) 87
5.5. Our algorithm moves along one dimension of the search space at a time. Points $X^{(1)}$ and $X^{(2)}$ denote the intermediate points, $X^{(3)}$ the final point. 88
5.6. Search steps made by the SingleDimSearch algorithm. Horizontal lines show three different scenarios (lower bounds). The arrows show the moves to the right (solid), binary search (dashed), and Fibonacci search (dotted). Binary search terminates when the minimum step width has been reached (e. g., last step in $scen_1$). Fibonacci search terminates when a measurement that exceeds the threshold ($scen_2$) or the maximum ($scen_3$) has been located. 89
5.7. Example to illustrate how to locate the minimum control parameter setting that satisfies the objectives of the two performance metrics using binary search. 93
5.8. Example for a change of the acceptable region when an objective for service class two is removed. The solid gray line denotes the acceptable region before, the dashed line the region after the change. The shaded area indicates the starting region for the search of the new operating envelope. 98
5.9. Search paths using different techniques to find the next probe along a search dimension. 103

5.10. Results of our MsCoSearch algorithm in scenario 1. The horizontal dotted lines for the performance graphs show the bounds for the respective service classes. The shaded areas indicate the time the algorithm is active. 104
5.11. The settings where none of the objectives are met (white), the throughput but not the average response time objective is met (dark pink), the average response time but not the throughput objective is met (orange), or both objectives (green) are met for service classes s_1 and s_2, respectively. The axes show the values for MPL_1 (x-axis) and MPL_2 (y-axis). 106
5.12. Results of the workload adaptation algorithm in scenario 1 with correct information about dominant objectives. 107
5.13. Results of the workload adaptation algorithm in scenario 1 with throughput as dominant objective for both workloads. With incorrect or incomplete knowledge about the dominance, the workload adaptation algorithm fails in finding the operating envelope. 108
5.14. Results of the workload adaptation algorithm in scenario 2. The vertical dashed lines indicate the times at which the objectives and the workload change. The horizontal dotted lines for the average response time and throughput graphs show the bounds for the respective service classes. 110
5.15. Results of the MsCoSearch algorithm in scenario 2. 111
5.16. Policy control screen 113
5.17. Performance objective screen after problem injection shows the OLTP service class not meeting objectives. 114
5.18. In the system resource utilization screen, the administrator can see that after the arrival of a long-running heavy-weight CEO query, resource contention interferes with the rest of the workload. 115
5.19. Performance objective screen after a run has completed. 116

B.1. Schema of the TPC-CH benchmark 126

List of Tables

2.1. Metrics and actions considered for admission control. Actions are executed when the threshold for the metric is exceeded (exception: check access permission returns true/false). 22
2.2. Metrics and actions considered for scheduling. If threshold for a metric is exceeded, queries are put in a waiting queue. 23
2.3. Metrics and actions considered for execution control. If threshold for a metric is exceeded, actions are triggered. 24

3.1. Percentile objectives for the TPC-C transactions 41
3.2. Average elapsed time in milliseconds for determining the "best" request in a queue with n requests. 42
3.3. Deadline objectives for the TPC-C transactions 49

4.1. Query taxonomy: We distinguish types of long-running queries based on whether (1) we expected the query to take a long time, (2) the query is making progress toward completion, and (3) the query is receiving an equal share of measured resources, such as CPU time or disk I/Os. 56
4.2. We created pools of candidate queries, categorized by the elapsed time needed to run each query on our 4-node Neoview database system. . 63
4.3. Time to complete a certain percentage of queries (in thousands of simulator time units) when trying to put expensive queries in a separate queue. 71
4.4. Time (in thousands of simulator time units) to complete a certain percentage of queries. 76

5.1. Decision table for next move 92
5.2. Decision table to tell whether to search in the left or in the right half of the binary search interval. 94
5.3. Bounds for scenario 1 101
5.4. Bounds for scenario 2 105

1. Introduction

Workload management enforces a contract between the submitter of a workload and the system owner. A workload manager is responsible for meeting the owner's objectives (e. g., reduce energy or maximize profit) while attempting to meet the performance objectives while monitoring the workload to ensure that the contract remains valid, e. g., the workload assumptions and characterizations are within an acceptable range so the objectives can be met.

1.1. Problem statement

In the database context, the term *workload* is widely used for two different concepts: (1) the set of queries that are submitted by a particular user or a particular application, or a set of users and/or applications and (2) the set of queries that are submitted by all users/applications. This work focuses on mixed workloads, i. e., different users and applications send queries with different characteristics and objectives to the same database. Therefore, we need to reason about finer-grained components of a workload.

A *submitted query* is a single unit of physical work for the database. It comprises an SQL string along with a user identifier, an application identifier, and a connection identifier. Note that although we refer to the unit of work as a query, the unit of work may also be an update operation. A *request* is a logical unit of work, which either represents a single submitted query or combines multiple queries in a single transaction. A *request stream* is a sequence of requests. Requests within a stream can either be submitted serially, i. e., the previous request must terminate before the next is submitted or as a batch, i. e., all requests are submitted at once. Note that the requests in a stream may be sent by a (human) user or by an application on behalf of a user. An example for a request stream is the requests that are sent on behalf of a single user over a single database connection.

A *user load* captures the sense of definition (1), above, i. e., it is a set of submitted queries that have a description and an associated service level objective. The description specifies the characteristics of the requests, such as arrival rate, arrival and termination time, resource usage, and variance of arrival time/arrival rate/resource usage. We also note that there may be dependencies between the requests, e. g., if the submitted requests belong to a multi-request database transaction. A transaction may comprise several individual requests. It is expected that the requests in a user load have similar characteristics, e. g., a stream of OLTP queries or a nightly batch of reports. However, there exist user loads where little is known or where there is high variance in resource usage, e. g., ad hoc analysis queries. For a user load, it

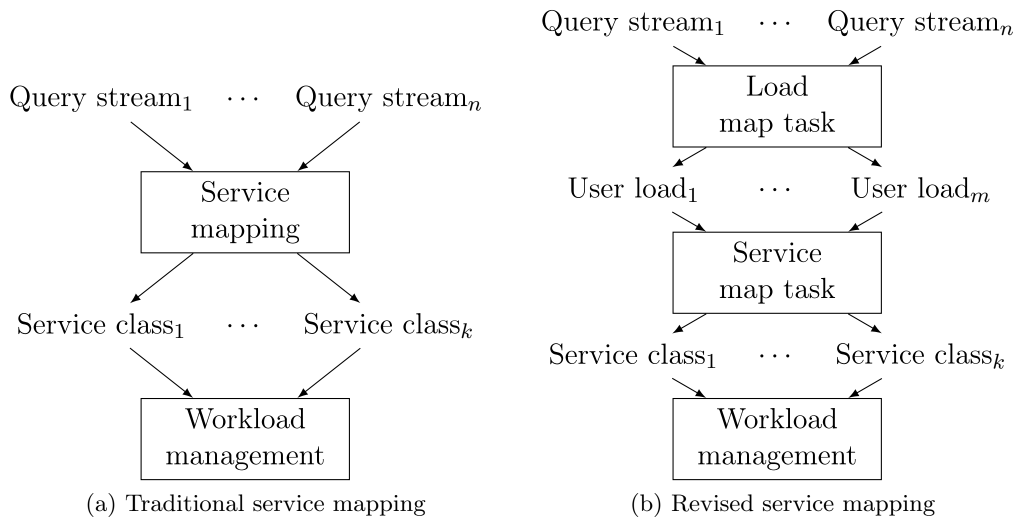
is not necessary that there is a one-to-one mapping between the submitted requests of a stream and a user load. We give more details on the mapping from submitted requests to user loads below.

Each user load has an associated *service level objective* to specify the desired level of performance. Service level objectives are formulated based on performance metrics on a per-request level. The objectives can be specified per request or aggregated for all requests in the user load. Examples for per-request metrics are maximum response time for individual OLTP-transactions in a user load. Aggregated metrics include throughput, average response time, n^{th} percentile of the response time, or deadline for multiple requests of the user load. Objectives can either define thresholds for the performance metrics or require optimization of the performance values. An example for the latter case is to maximize the throughput. We refer to objectives based on multiple metrics as *compound objectives*. As an example, the TPC-C benchmark [59] defines an objective based on throughput, average response time, and the 90th percentile of the response times.

We define a *workload* to be the set of all user loads plus workload objectives. This corresponds to definition (2) above with the caveat that the set of all requests is partitioned into one or more user loads. The workload objectives differ from the service level objectives in that they specify how to balance competing service level objectives. For example, one user load may specify throughput requirements for OLTP requests and a second user load may specify response time requirements for OLAP queries. Examples for workload objectives require to maximize the profit for the system owner or to minimize the energy usage. Given the user loads and workload objectives, service level agreements (SLAs) for the workload and constituent user loads are negotiated between the customer and the service provider. This defines the "contract" mentioned earlier between the system owners and the workload submitters. An SLA may also specify a penalty for violating the performance objectives of a user load.

A *service class* defines the level of service to be allocated to *individual* queries belonging to this class, e. g., the share of the resources given to queries from this service class. The service class also defines a query's "priority". The priority of a query affects its importance in the workload manager (*workload manager priority*), e. g., in scheduling queues, and in the database engine (*database engine priority*), e. g., how often a query can access a resource other queries are also waiting for. Note that the workload manager priority and the the database engine priority are different concepts. For the remainder of this thesis, we focus on workload manager priorities. Consequently, we just use the term "priority" to denote the workload manager priority. Details on the workload manager are shown below.

It is important to note that service level objectives are defined for user loads and that the workload manager applies policies to queries in a service class. However, the workload manager has no information about users, SLAs, or user loads. As a consequence, a mapping from user loads to service classes is needed. For example, the individual queries in an OLTP-transaction must be mapped to a service class. We briefly describe the *service mapping problem* whose goal it is to infer the correct



(a) Traditional service mapping (b) Revised service mapping

Figure 1.1.: Visualization of the service mapping problem

service class for individual queries. Figure 1.1(a) shows the "traditional" approach to service mapping, which is commonly implemented in commercial workload management tools. A submitted query is mapped to a service class based on identifying attributes of the query, e. g., the user, application, or connection identifiers. However, a single user or application may want to send requests with different service level objectives. For example, in the TPC-C benchmark an individual user sends different business transactions where each business transaction is associated with its own service level objective. Using the concept of user loads allows us to formulate a revised service mapping problem, which is shown in Figure 1.1(b). User loads add a layer between the submitted query and the service class. Consequently, the service mapping problem is split into two mapping problems: how to map a submitted query to a user load (*load map task*) and how to map a user load to a service class (*service map task*). The service map task should be straightforward because there is a natural correspondence between user loads and service classes. The load map task is more challenging and corresponds to the "traditional" service mapping problem where submitted requests are mapped to service classes. However, we believe that the load map task is more tractable because the concept of user loads encapsulates knowledge about the characteristics of requests.

The workload manager is, in almost all commercial vendors, an add-on module that is external to the database system. A database system exposes some metrics, control parameters, and control actions to the workload manager. For example, the database system provides an interface to retrieve the costs from the query optimizer for an individual query. Examples for the control parameters are interfaces that provide methods to kill running queries or to change the priority of running queries

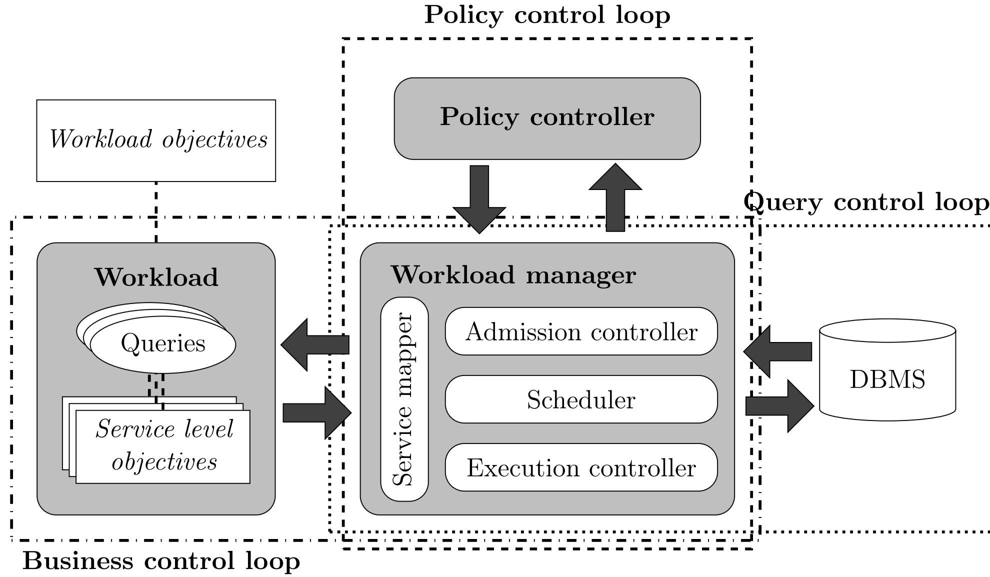


Figure 1.2.: The main components of the workload management architecture are the workload manager and the policy controller. The workload manager controls the execution of individual queries based on measurements it receives from the database system. The rectangles show the query control (dotted rectangle), policy control (dashed rectangle), and business control loop (dot-dashed rectangle).

to change the share of resources they get compared to other queries. A workload manager may define additional metrics and control parameters that are, however, independent of the underlying database system. For example, the workload manager may count the number of queries running in parallel (metric) to limit the multi-programming level (control parameter), i. e., the maximum number of requests that are executed in parallel. It is the metrics and control parameters that are available to the workload manager that are used for constructing policies, i. e., to decide based on the metrics which corrective action should be applied, i. e., which parameter should be changed.

Figure 1.2 shows the abstract of a database workload management system. The *workload manager* comprises multiple components: the *service mapper* maps submitted queries to service classes as described above, the *admission controller* decides whether or not to admit a newly arriving query into the system, i. e., either pass it to the scheduling component or reject it. The primary goal of the admission controller is to avoid accepting more queries than can be executed effectively with the available resources. It depends on the objectives defined in the SLA whether or not submitted queries can be rejected without penalties. Once the admission controller admits a query, it is passed to the *scheduler* which maintains the pending queries in one

or more queues. When there are enough system resources available, the scheduler dequeues a query and sends it to the database engine. It is important to note that admission control and scheduling apply to queries prior to their execution. These decisions are based on compile-time information, such as the optimizer's cost estimates for queries, plus information about the current operating environment, such as system load. However, at run-time, a query may behave significantly different from the estimates and/or the operating environment might have changed drastically. In order to compensate for the incomplete or imprecise knowledge about query resource usage and the system environment, the *execution controller* uses both compile-time and run-time information to apply corrective actions (e. g., kill or suspend&resume a query). The *query control loop* applies admission control, scheduling, and execution control policies to submitted and executed queries, where the policies are designed to meet the service level objectives.

The *policy controller* defines admission control, scheduling, and execution control policies, and adapts them to accommodate changes in the workload or the objectives. The policy controller can update the workload management policies based on time, e. g., if it is known that a certain user load arrives at a certain time. For example, we need to set new scheduling policies to prevent the system from becoming overloaded when a scheduled maintenance task starts. The workload manager and the policy controller implement the *policy control loop*. If the information about workload or objective changes is not known in advance, the workload manager may trigger the policy controller when it detects that the workload has changed. Examples for workload changes are the arrival of a new user load or when arrival rate of queries stemming from an active user load increases drastically. The workload manager also triggers policy changes when it detects that the objectives of the user loads cannot be met by applying the currently active workload management policies. If, for example, the number of admitted queries was too high, the policy controller could dial back the number of concurrently active queries to mitigate the overload situation.

It is important to note that service level objectives may be unsatisfiable with the current system environment. It is the task of the *business control loop* to detect that the objectives cannot be met, i. e., the policy control loop cannot find policies to meet the objectives. In that case, the business control loop must identify the root cause of the problem and employ corrective actions, e. g., reconsider the sizing of the system, improve the policy control loop, or renegotiate the service level agreements. For example, consider a user load with an objective on average response time whose characteristics do not specify a maximum arrival rate. Also suppose the queries from this user load overload the database system and no admission control and scheduling policy can be devised to meet the objectives. In that case, an outcome of the business control loop may be a more precise characterization of the user load, i. e., the specification of a maximum arrival rate for which the objectives must be satisfied.

In this work, we concentrate on workload management for *mixed workloads* where the workload on the database system comprises different user loads. The trend towards mixed workloads is strengthened by the emergence of "operational data

stores" where – in contrast to traditional warehouses where OLTP and business intelligence queries were kept separate – the transactional and analytical queries are running on the same physical database instance at the same time.

1.2. Design of experimental frameworks

An important issue in workload management research is the design of an experimental framework. There are two core components that are necessary for workload management: the workload manager and the database. In order to experiment with workload management, there are two decisions that can be made: (1) Whether to use a commercial workload manager or implement a proprietary workload manager and (2) whether to use a "real" database system or implement a simulator.

Decision (1) above depends on the metrics and control parameters needed to explore the respective search space. For example, in order to explore the workload management policies for long-running queries in Chapter 4, we could not use one of the commercial workload managers because no single workload manager implements all the policies we wanted to explore. Additionally, if a new workload management policy has been devised that no workload manager implements, a workload manager that supports the policy must be built.

Similarly, the decision to use a "real" database system or a simulated database engine depends on what metrics the database system provides and what workload management actions can be applied. For example, it is not possible to investigate workload management policies for suspending queries at runtime using a commercial database system – suspend and resume has only been implemented in research prototypes so far. The options are to extend a database system for which the source code is available or to use a simulated database engine. Additionally, if the workload comprises long-running queries, an option is to build a simulated database engine, which abstracts from details of the query execution to speed up a single experiment. Results gathered using a simulated database engine may not be as accurate as those gathered from an actual database instance, and thus the simulated database engine must be validated thoroughly.

There are implications between the choice of the database system (real vs. simulated) and the choice of the workload manager (your own or one of the commercial ones). In almost all commercial vendors, workload managers are add-on modules that are only slightly integrated into the database engine. However, using a commercial workload manager implies that the database system for which the workload manager was designed must be used. The reason is that workload management is an after-thought to the design of database systems. In particular, database systems do not provide a standardized interface workload managers could use (e. g., metrics (from database engine to workload manager) or setting of control parameters (from workload manager to database)). As a consequence, it is not possible with commercial database systems to build a generic workload manager, i. e., one that works with an arbitrary database system. What would be needed is some equiva-

lent to ODBC/JDBC for workload management functions (as well as administrative functions).

In contrast, if a workload manager is built from scratch then either a "real" database system or a simulated database engine can be used. Usually, database systems do not expose too many database system-internals, yet the workload manager has to control the operation and efficiency of the database system. Consequently, the database may only provide a subset of the metrics that is needed to make workload management decisions. Even if the interface between the database system and the workload manager is sufficient for exploring the problem space, retrieving the metrics necessary for the workload management decisions (e. g., estimated costs for a query from the query optimizer) and applying the workload management actions (e. g., killing a query) causes overhead (inter-process communication, round-trip to the database).

Chapters 3 through 5 give more details on the experimental frameworks used for the experiments and why they were chosen.

1.3. Contributions

As most workload management research has focused on simple, static workloads with a single class, there is no terminology for specifying or comparing scenarios, mixed workload management mechanisms and solutions. In particular, there is no common understanding of how well the management capabilities of commercial vendors address the problems posed by mixed workloads: The unpredictable variance between queries in a mixed workload, e. g., OLTP and OLAP queries executed in parallel, results in unpredictable resource contention, which in turn results in unpredictable performance. Ultimately, the lack of predictability calls for adaptive workload management policies.

In principle, a mixed workload management scenario poses challenges similar to the "traditional" workload management – where only a user load that is mapped to a single service class runs against the database. One major difference is that the workload management mechanisms to control a single class cannot be applied to the multi-class case because the interaction of the queries that are mapped to different service classes must be considered. For example, in a scenario with two classes, increasing the number of concurrently active queries from one class not only has impact on the performance of that class but also on the performance of the other class because fewer resources are available to process queries from the second class.

The contributions of this thesis are as follows: We identify, examine, and present solutions for three workload management scenarios where adaptive mixed workload management is needed to enforce the objectives for all users in a mixed workload:

Dynamic request prioritization: The first scenario focuses on enabling quality of service (QoS) for the bottom layer of a (Web-)service infrastructure where transaction-processing business services access a shared database. This is a very common scenario in mission-critical enterprise services that rely on an integrated

database. The requests that run against the database are usually short-running and parameterized, and must be processed quickly in order to provide immediate feedback. Due to the multitude of services that access the database, the workload of the database consists of requests stemming from many different service classes each with an associated performance objective and possibly different priority. For this scenario, we assume that performance objectives for every service submitting requests to the database have been negotiated. A common objective in such a transaction-processing business workload defines constraints on the percentile response time for an individual transaction and defines a deadline to avoid the starvation of transactions. The constraints on the percentile of the response time incur a penalty while the deadline enforces the execution of requests. We devise a QoS management concept based on an economic model, which adaptively prioritizes individual requests depending on the performance objectives and the current degree of objective conformance that the particular service class exhibits. We also present algorithms to use the penalty specification to reorder the requests in order to minimize the incurred penalties for violating the percentile constraints.

Long-running query control: Our second scenario focuses on workload management policies to avoid "problem queries", e. g., queries that run far longer than expected due to poorly-written SQL, poorly-optimized execution plans, or unexpected resource contention. We build a taxonomy for long-running queries based on how they impact other queries and focus on how to identify and handle such long-running requests in three common scenarios: unreliable cost estimates, unobserved resource contention, and system overload. Furthermore, we describe an experimental framework that allows us to systematically evaluate the ability of existing workload management mechanisms to deal with these scenarios. We use the experimental framework to methodically explore the space of policy combinations and workloads. Finally, we make recommendations for which policies to use, when to apply them, and demonstrate how to set their thresholds.

Policy control for mixed workloads: In the third scenario, we present an approach to policy control. In particular, we investigate how to allocate resources when managing multiple service classes, each class with its own set of compound objectives. In order to control the performance of the service classes we may use the different types of control parameters mentioned earlier. For example, we may control one service class by the number of submitted requests scheduled to run simultaneously, while for another we might set an admission control threshold based on estimated resource costs. Our performance objectives may be defined by a value range, which defines a lower and an upper bound on the performance metric, instead of simply maximizing or minimizing a performance metric (service level agreements, SLAs). This last point means that we can essentially cast the problem as a search problem with the goal of locating any point within an operating envelope representing all control settings under which the workload can meet all performance objectives. We present an algorithm that automatically locates a point in the operating envelope, if it exists, or recognizes if no such point exists. We also define how the algorithm

adapts policies to deal with discrete changes in the workload and devise an extended experimental framework. Our experimental studies evaluate various algorithms for scheduling where we automatically set the number of concurrently active queries in a system to meet the service level objectives of all service classes.

1.4. Outline

The remainder of this thesis is organized as follows:

- Chapter 2 summarizes workload management approaches in commercial database systems. In particular, we summarize the workload management policies that are implemented in workload management tools provided by commercial database vendors.[1]
- Chapter 3 introduces our quality of service management concept based on an economic model for dynamically prioritizing requests based on the current degree of service level conformance. We describe experimental studies based on the TPC-C benchmark to validate the dynamic prioritization approach.[2]
- Chapter 4 presents the taxonomy for long-running queries. Based on the taxonomy, we present experiments for controlling the execution of long-running queries using our experimental framework. In addition, we present our lessons learned from the experiments.[3]
- Chapter 5 formulates the problem of allocating resources with workload management to meet the objectives in a mixed workload as a search problem. Based on the problem formulation we devise an algorithm to solve a restricted version of the search problem. The chapter also presents experimental studies that show how the algorithm finds solutions of the search problem, or that no solution exists at all. The experimental studies are carried out on an extended version of the experimental framework described in Chapter 4.[4]
- Chapter 6 concludes the thesis.

[1] Parts of the work presented in chapter 2 appeared in [34]
[2] Parts of the work presented in chapter 3 appeared in [24], [32], and [36]
[3] Parts of the work presented in chapter 4 appeared in [34]
[4] Parts of the work presented in chapter 5 appeared in [33] and [35]

2. Workload management in commercial database systems

Research in workload management has been mostly driven by commercial vendors. As there have been no other surveys of workload management capabilities of commercial database systems, this chapter provides an overview of the workload management tools of such systems. We note that the survey was done in 2009 and that features of systems are constantly changing. The purpose of this chapter was to understand the state of the art. The chapter describes the mechanisms for query control and policy control implemented in today's database systems. We distilled the information from the documentation of the different vendors, e. g., EMC Greenplum Database 4.0 [26], HP Neoview Workload Management Services (WMS) 2.3 [27], IBM Workload Manager (WLM) for DB2 9.5 [17] and DB2 Query Patroller [28], Microsoft SQL Server 2008 [45], Oracle Database [47, 51], and Teradata Dynamic Workload Manager (TDWM) 13.0.0.0 [57]. For the remainder of this chapter, we anonymize the products by referring to them as systems A-F.

Workload management is mostly accomplished through the application of predefined policies to workloads in today's database systems; only a few vendors provide simple policy control mechanisms. The policies initiate control actions when specific conditions are reached. Thus far, commercial database vendors have led the state of the art in workload management, adding policies to respond to customer needs. The policies have not been studied systematically and their interactions are not well understood.

The goal of workload management is to satisfy the user's (customer's) workload objective. A simple objective is to complete all queries in the shortest time. A more complex objective is to provide fast response for short queries and to complete as many long queries as possible. The workload management system uses policies, tuned with parameter settings, to achieve these objectives.

2.1. Query control loop

This section summarizes the admission control, scheduling, and execution control mechanisms implemented in commercial workload management tools. For performing workload management, the workload managers support different service classes for which admission control, scheduling, and execution control policies are defined. Note that each vendor uses different terms to describe a "service class". As stated in Section 1.1, the workload management policies are applied to individual requests. The commercial workload management tools map submitted requests to service

	Num queries	Num connections	Expected costs	Check access permissions
System A	reject	—	—	—
System B	reject	—	—	—
System C	reject; warn	—	reject; warn	—
System D	—	reject	reject	—
System E	—	reject	reject	—
System F	reject; warn	reject; warn	reject; warn	reject; warn

Table 2.1.: Metrics and actions considered for admission control. Actions are executed when the threshold for the metric is exceeded (exception: check access permission returns true/false).

classes based on the identifying attributes, e. g., the name of the user, the application, or connection identifiers.

Admission control: Admission control policies in commercial database systems place different kinds of limits on the system, e. g., the number of requests running concurrently, the number of concurrent users, or the expected costs of the submitted requests. Typical admission control actions are: *warn*, which accepts the request but signals a warning and triggers the collection of additional data for an offline analysis to prevent this condition from happening again; and *reject*, which rejects the request. The remainder of this section discusses the admission control policies that are supported by commercial database vendors. Table 2.1 summarizes the policies. Common to all workload management tools is that the administrator must set the thresholds for triggering an action.

There are three major types of metrics for which an administrator can define thresholds to prevent overload on the system: One metric is to limit the number of queries (*num queries*). In this case, the workload manager either limits the number of active queries, i. e., the number of requests that are currently processed in the database, the number of queued queries in the scheduling queue, or both. A second commonly used metric is the number of connections to the database (*num connections*), e. g., via JDBC or ODBC. Third, the administrator can limit the maximum estimated costs of a query to be submitted (*expected costs*). The costs of a query are either defined on the total costs as estimated by the query optimizer, e. g., the estimated CPU or I/O costs, the estimated number of rows returned, or the estimated memory consumption. A different approach is to define admission control policies based on access permissions on database resources (e. g., database tables), which might be valid for a given time period. For example, it is possible to reject queries

	Num active queries	**Num connections**	**Expected costs**	**Resource usage**
System A	submitter	—	submitter	—
System B	—	—	—	submitter
System C	submitter; prop	submitter	submitter; prop	—
System D	submitter	—	—	—
System E	submitter	—	—	—
System F	submitter	submitter	submitter	—

Table 2.2.: Metrics and actions considered for scheduling. If threshold for a metric is exceeded, queries are put in a waiting queue.

from one user group on a table during the day. Note that the workload manager allows exceptions to be defined so that some queries can bypass admission control.

Scheduling: Table 2.2 summarizes the metrics and queue types supported by the schedulers in commercial products. The workload management systems provide one or more queues to queue the incoming queries. Each queue is managed first-in, first-out (FIFO). Database administrators can define queues on the submitter- or on the system-level. In the former case, queries are mapped to queues based on who submitted the query (*submitter*). For example, there could be different queues for requests from business analysts, reporting queries, and executive queries. Each of the queues can be assigned a different priority based on the priority of the submitters of the query so that more queries from high priority submitters are allowed to run concurrently. When queues are defined on the system-level, incoming queries from all submitters are put in queues based on either properties assigned with the query (*prop*). For example, some workload managers assign queues to requests based on the properties associated with the incoming user connection. For each queue in the workload manager, there is a separate threshold that controls how many queries from a queue can be processed in parallel.

Similar to the admission control component, the most commonly used metrics are the number of active queries (*num active queries*), the number of active connections (*num active connections*), and the costs as estimated by the query optimizer (*expected costs*). Similar to using the number of active queries allowed at a time, some workload management tools limit the number of worker threads that are available to process the incoming queries. If no more worker threads are available, the queries are queued. In addition, queries may be queued when the resource usage (e. g., CPU or memory usage) exceeds a threshold.

	Wait time	**Execution time**	**Resource time**	**Cardinality**
System A[1]	—	—	—	—
System B	kill	kill	—	—
System C	—	kill; warn; reprioritize	warn	kill; warn
System D	warn	—	—	—
System E	—	warn	—	stop
System F	—	kill; warn; reprioritize	kill; warn; reprioritize	kill; warn; reprioritize

Table 2.3.: Metrics and actions considered for execution control. If threshold for a metric is exceeded, actions are triggered.

Similar to the admission control, the workload managers grant a bypass privilege to requests based on estimated costs, based on who submitted the query, or based on the type of the request. For example, some of the workload managers can be configured so that SELECT statements whose costs as determined by the query optimizer are below a threshold defined by the administrator can bypass the scheduler.

Execution control: To compensate for estimation errors made at compile time, workload management tools implement policies to perform workload management actions during the execution time of the request. The major challenge is to detect that a query does not behave as expected. The most common metrics to perform workload management at runtime in commercial database systems are the time a request is queued in the scheduler (*wait time*), the time the database system has spent on processing the query (*execution time*), the resource time that is spent for processing the requests (*resource time*), and the number of result tuples produced so far (*cardinality*).

Workload management tools support three actions that can be applied to the requests at runtime: *reprioritize*, e. g., decrease the priority of a long-running request in order to make more resources available to other queries, *stop*, and *kill*. The latter action cancels the request execution, frees up the used resources, and all (intermediate) results generated on behalf of the request are lost. After killing a request, the request can be manually resubmitted to admission control. The actions can be executed either manually or automatically. In the former case no automatic execution is supported, the workload manager generates a log entry or a warning message which it sends to the database administrator, e. g., via dashboard or e-mail. It is

[1] The database administrator can manually kill queries

then the task of the administrator to examine the situation and apply the appropriate corrective action, or else to let the request run to completion. Similar to killing the execution, stopping the execution of a request cancels the execution and frees up resources. However, in contrast to killing the query, the results generated so far are returned to the client.

2.2. Policy control loop

Some database vendors implement simple policy control mechanisms on top of their query control loop. The tools support switching to a new set of workload management policies, i. e., a combination of admission control, scheduling, and execution control policies, triggered by time by an event, or both. An example for a set of workload management policies would reject queries based on the cost estimates (admission control), limit the number of concurrently active queries (scheduling), and kills queries that exceed a designated resource time (execution control). Time-triggered policy control applies different sets of policies for different times of the day if changes in the workload depend on the time. For example, there may be different policies for transactional loads during the business hours and analytical queries over night. For event-triggered policy changes, applications that connect to the database make an API call to notify the workload manager that an event occurred, e. g., the arrival of a new user load. As a consequence, the workload manager may load a new set of policies for admission control, scheduling, and execution control. However, note that the policies must be predefined by an administrator. In particular, the thresholds for the policies to trigger an action must be manually set.

3. Dynamic request prioritization for OLTP workloads

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 users, where each user may have individual quality of service (QoS) requirements. The more users 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 users from suffering bad performance in terms of response times of their invoked services, service level objectives (SLOs) are negotiated.

The establishment of an SLO expresses the performance expectations of the user and, thus, imposes obligations on the service provider regarding the service level of the provided services. If the constraints formulated in the SLO are violated after a certain time window, the *measurement interval*, the service provider is fined. The penalty depends on the severity of the SLO violation and is negotiated in the SLO. SLOs are typically only defined for services directly invoked by users, so performance requirements for "lower-levels" of the hierarchy must be inferred from the SLO. 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.

This chapter describes how 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. The invocation of a service results in the execution of a multi-request business transaction, i. e., every transaction may comprise several individual requests. The requests are short and parameterized and must be processed quickly in order to provide immediate feedback. For example, the invocation of an "order entry" service results in database queries that determine the tax for the order depending on where the purchase was made, that store which items have been ordered, and that update information about how many items are still available for future purchases. Each service may be repeatedly invoked on behalf of a user, where all users may have different performance expectations on a service. Using the terminology introduced in Section 1.1, all business transactions that stem from the invocation of a service on behalf of a user and the user's performance objectives form a user load. Our

particular interest is on how to meet performance objectives for "important" users during peak loads, i. e., where more requests arrive at the database system than can be processed.

A commonly used SLO in this OLTP-scenario requires a percentage of the transactions started on behalf of a user must complete within a given time, as exemplified by the objectives defined in the TPC-C benchmark [59]. For example, consider an SLO that requires 90% of all transactions to be processed within a certain time window (and another requirement to limit the response times for the remaining 10%). A penalty is due if fewer than 90% of the transactions complete within the given time window.

We assume that the objective of the service provider is to minimize the penalties incurred by violating the percentile response time of all user loads. There are two measures that can be applied from a workload management point of view: First, to avoid overload on the database system during peak loads, the workload manager may define a scheduling policy that limits the number of requests that are executed in parallel. If too many requests are running in parallel, the execution time, i. e., the time that is spent for processing the request in the database grows too large due to resource contention and thrashing effects. So it may be beneficial to queue some requests so that fewer requests are running in parallel but the response time is lower and throughput is higher than when no requests are queued. Second, to minimize the overall penalty, the queued requests can be ordered according to the penalty that is defined in the objectives of the respective user load. Requests stemming from a user load that incurs a high penalty for violating the percentile response time should be executed before requests that stem from a user load with lower penalties. This *static prioritization* is used to schedule the requests, so that requests stemming from high priority user loads should complete faster on average than their low priority counterparts.

This approach is sufficient to fulfill the objectives of particularly valuable user loads. However, it cannot adequately manage overall SLO enforcement. Consider the example SLO above. With static prioritization, SLOs for high priority user loads are likely to be overfulfilled by processing almost all of the respective requests in time. However, during peak-load times, it is likely that they overachieve their SLOs at the expense of lower priority user loads. From a business-oriented point of view, it is desirable to provide only the service level that has been negotiated in the SLO. If SLOs are not overfulfilled, the additional free resources are used for satisfying SLOs that are violated with the static prioritization.

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

For this purpose, we developed a QoS management concept based on an economic model, which adaptively prioritizes individual requests depending on the SLO and the current degree of SLO conformance that the particular user load exhibits. The

core of the QoS management consists of *penalty-carrying requests*, i. e., database requests which carry the requirements needed to fulfill the SLO constraints from the submitting service to the database.

3.1. Related work

An approach for enabling end-to-end QoS for distributed multimedia databases is discussed in [60]. The presented idea is to generate a number of offline copies with QoS parameters for each media object in the database. These parameters, e. g., spatial and temporal resolution, and color depth are passed to the database by annotating them to the database requests. The copies are then distributed among the servers. During runtime, the query processor generates various execution plans for a request, depending on the information on data replication and runtime QoS adaptation options. The generated plans are then evaluated according to a predefined cost model.

Scheduling jobs with time-value functions has been studied in the area of real-time systems, e. g., Chen and Muhletaler [16] and Wang and Ravindran [61]. The system accrues some utility for completing individual jobs before their deadline. The utility decreases if the job is processed after its deadline. Although the approaches are feasible for arbitrary time-value functions, the time complexity of the presented algorithms is not feasible for scheduling queue lengths that we observed in our experiments.

Enabling QoS for Web service infrastructures is the focus of Braumandl *et al.* [6]. The work discusses distributed query processing systems on the Internet where the requests have different QoS demands. The authors present an extension to 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. Seltzsam *et al.* [55] 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.* [4] analyze service compositions at compile-time to gain further information on the service's behavior. Selecting services that are dynamically bound to composite services at runtime to satisfy user QoS requirements is presented by Maximilien and Singh [42], and Gibelin and Makpangou [23]. 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.* [30, 31] 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 changes, the priorities for the classes have to be recomputed.

An admission control and request scheduling system for e-commerce Web sites is presented by Elnikety *et al.* [20]. Their work focuses on achieving stable behavior during overload and improving response times. Analogous to our SLO based request management component they install a proxy between the Web service and the database. However, the optimization is not associated to the SLO conformance. As we will discuss in the following, considering the conformance is an integral part of an adaptive QoS management.

There has been some early work on meeting response time goals for multiclass workloads with per class response time demands on traditional database systems. The requirements are met by managing system resources or the access to system resources. A management of the disk(s) and the buffer pools for read-only workloads by priority-based algorithms is proposed in [10]. Other approaches are found in the domain of memory management, e. g., [7, 8, 9] where per-class response time goals in a multiclass workload are achieved by automatically adjusting memory allocation for each workload class. The adjustments are based on constant monitoring of the state of the system relative to the goals of each class. Nevertheless, these approaches are not sufficient when the response time demands are defined by an SLO for user loads with different economic value, which is an important part for enforcing quality of service.

As a testament to the importance of QoS enforcement, both IBM and Oracle provide commercial products, which can be integrated into their respective databases. Both IBM's DB2 Query Patroller [28] and Oracle's Database Resource Manager [47] let the administrator define user-groups to which a static priority and a share of system resources for each group is assigned. The higher the priority of a group, the more resources it is assigned. However, the static prioritization is not associated with response time requirements and the current SLO conformance, which our approach provides a vital part for enforcing QoS.

Recently, quality of service attracted more attention in the database community. A QoS based extension for the cost-based query optimization in federated systems is presented in [38]. The traditional query optimization that is based on database statistics, query statements, and the local and remote system configurations, is extended by considering the load on remote servers, the network latency between the nodes, and the availability of the remote sources. The work in [43] analyzes and proposes prioritization of workloads on traditional database systems. To meet the QoS requirements, the bottleneck resources for a DBMS are identified. These are then scheduled using statically assigned priorities in contrast to our dynamic prioritization.

Schroeder *et al.* [53] present a framework for providing QoS where the response time requirements are specified in an SLO. 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, unlike our approach their work

is not based on an economic model that optimizes the overall system performance across different classes.

3.2. Service level objectives

This section describes the type of objectives we consider for our dynamic prioritization approach. In practice, so-called *step-wise SLOs* are commonly used to specify the QoS requirements of a user load. The SLOs are defined on the percentile of the transaction response time (*percentile objective*) and give a deadline for each transaction (*deadline objective*).

3.2.1. Percentile objectives

Percentile objectives require $n\%$ of all transactions within a user load to have a response time $\leq x$ seconds. The response time of a transaction is defined as the time between sending the first byte of the first request of the transaction and receiving the last byte of the result of the last request. If a percentile objective is violated after a specified measurement interval, a penalty p for every m percentage points underfulfillment is due. Furthermore, p_{max} denotes a maximum penalty for violating a percentile objective. An example for a step-wise SLO with a percentile objective d_1 is shown in the following:

> d_1: Response time of 90% of all transactions in this user load must be less than 5 seconds each; $p = \$1000$ per 10 percentage points of underfulfillment, $p_{max} = \$2000$; measurement interval: 1 month (e. g., end of month)

The central concept of our quality of service management is adaptive prioritization of individual requests according to the current degree of *service level conformance c* of the transaction the request belongs to. The target value for the conformance can be derived from the percentile constraint. The conformance is monitored per user load. We define c as

$$c = \frac{\text{number of timely transaction invocations within the user load}}{\text{total number of invocations of the transaction within the user load}}$$

A percentile objective in a fixed step-wise SLO implicitly defines an SLO penalty function with n steps. The penalty function for d_1 of our sample SLO is shown as the step function in Figure 3.1. With c_i, $1 \leq i \leq n+1$, we denote the boundaries of the steps of the SLO penalty function. For the example in Figure 3.1, we have $c_4 = 0$ (not in the figure), $c_3 = 0.8$, $c_2 = 0.9$, and $c_1 = 1$.

Using the SLO penalty function, we define *service levels* as follows: For a penalty function with n steps, let $s_i = [c_{i+1}, c_i[$, $1 \leq i \leq n$, denote the ith service level, which is defined in the interval $[c_{i+1}, c_i[$. Dropping to a lower service level corresponds to a higher penalty, i. e., s_{i+1} denotes a lower service level than s_i. We denote Δ_i as the cost difference between s_{i+1} and s_i.

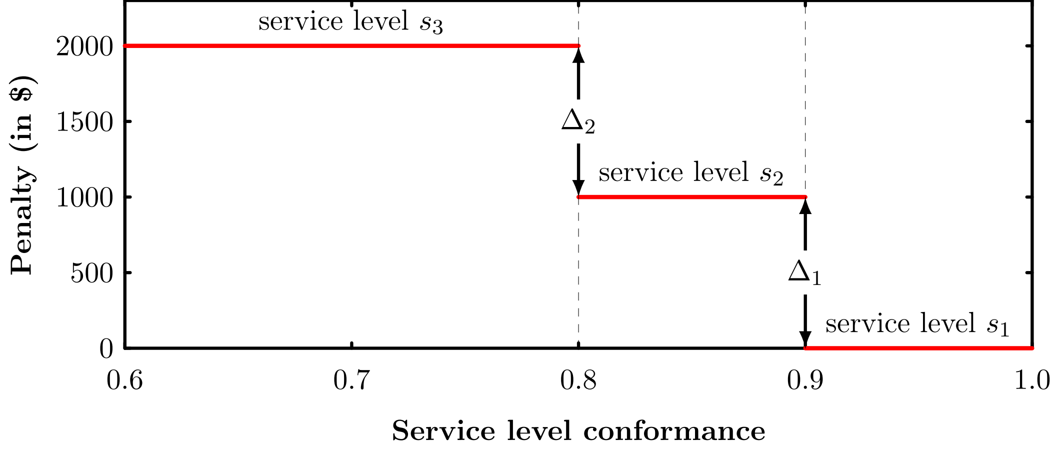


Figure 3.1.: Visualization of SLO objective d_1

As shown in Figure 3.1, our sample percentile objective 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 $[0.9, 1]$. The cost difference between service levels s_3 and s_2 is \$1000, which is identical to the cost difference between s_2 and s_1.

3.2.2. Deadline objectives

Note that percentile objectives allow (100-n)% of the transactions stemming from a user load to be delayed forever. In order to avoid the starvation of transactions, a deadline objective enforces the execution of a transaction after a certain time. In our model, deadline objectives are strictly required to be met; it is not an option to violate them in exchange for incurring a penalty. As a consequence, requests that exceed that deadline will be executed, even if there are requests with higher penalties (derived from the percentile objective) stemming from other transactions.

There are multiple approaches to specifying deadline objectives. First, *static deadlines* define a fixed time as upper limit for the wait time of a transaction. Second, *dynamic deadlines* enforce the execution of transactions depending on the current level of service level conformance. An example for dynamic deadline objectives requires the average response time of a transaction to be less than $p\%$, $0 \leq p \leq 100$, of the response times monitored so far, i. e., the average response time must not exceed the pth percentile of all monitored response times of the transaction. Let $rt_{(1)}, \ldots, rt_{(n)}$ denote the monitored response times ordered from the smallest to the largest. Then, the pth percentile X_p and, thus, the dynamic deadline objective is defined as the average response time at position $k = \lfloor \frac{p}{100} \cdot n \rfloor$, i. e., $rt_{(k)}$.

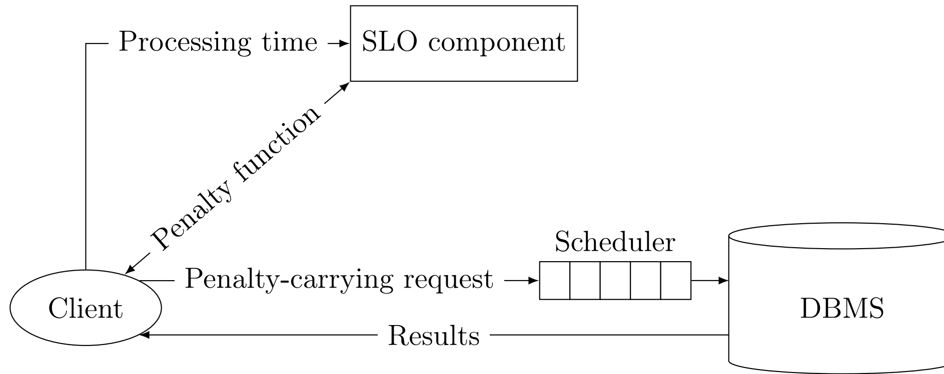


Figure 3.2.: Architecture overview

3.3. Architecture

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

A primary design goal for the implementation of our dynamic prioritization was to ease the future extension of the QoS management to entire Web service architectures. We therefore encapsulated all SLO-relevant functionality, including the monitoring of the SLO conformance and the generation of adaptive penalties, into a central entity, the *SLO component*. Figure 3.2 shows the resulting architecture. The SLO 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. Note that, although commercial workload managers do not allow to have an associated “penalty”. As a consequence, we implemented our own scheduler. Section 3.6 gives details on the implementation of the scheduler.

Upon completion of the database request, the SLO component is notified of the observed response time by the client and can thus update the current SLO conformance. The SLO component can be implemented either as a central component in the network so that it can be easily adapted to scheduling arbitrary service requests, besides the database requests exemplified here, or the SLO component can run as a dedicated component for each client to reduce the overhead of sending network messages. Each “private” SLO component observes the performance and computes the penalty function for the client it is dedicated to and does not need information from the other clients. The reason why the components can work independent from each other is that for computing the penalty for a particular client, no performance information from other clients is necessary.

The actual scheduling of the requests is based on the adaptive penalties and is

realized by the scheduler in the workload manager (for ease of presentation, Figure 3.2 just shows the scheduler and omits the other workload management components). The scheduler intercepts all arriving requests, puts them into a queue, and determines the appropriate order of the queued requests depending on the penalty functions of the requests and a scheduling algorithm (the next section gives more details on the scheduling disciplines). The scheduler releases the requests from the queue based on the number of currently active requests. As a consequence, requests belonging to a multi-request transaction are granted a bypass privilege, i. e., these requests are immediately executed. Using this approach, we help to avoid *lock convoys* [25]. 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. After dequeuing a request, the scheduler sends the request to the database system for execution. After completion, the database system returns the results of the request to the client.

3.4. Penalty functions

This section describes how we map the performance requirements that are defined on the transaction level to a penalty function for an individual database request. There are two components of the penalty information. First, we derive a penalty function that adaptively penalizes individual requests based on the current degree of service level conformance of the respective transaction. Second, we derive the time when to enforce the execution of the request from the deadline objective of the respective transaction.

3.4.1. Adaptive penalty function

In this section, we explain how we derive the penalty function for individual database requests. A penalty function encodes information about (1) the latest possible start time t for executing the request in order to complete the transaction in time and (2) the penalty p if the request is started late, i. e., after time t. Figure 3.3 shows an example for a penalty function.

Derive the penalty for an individual request

The penalty of a transaction covers two different economic aspects. *Opportunity costs* model the danger of falling from service level $s_i = [c_{i+1}, c_i[$ into the next lower service level s_{i+1}, thus causing an additional penalty Δ_i. If the current SLO conformance converges towards s_{i+1}, the penalty for processing the transaction too late increases, because delaying a further transaction increases the danger of an

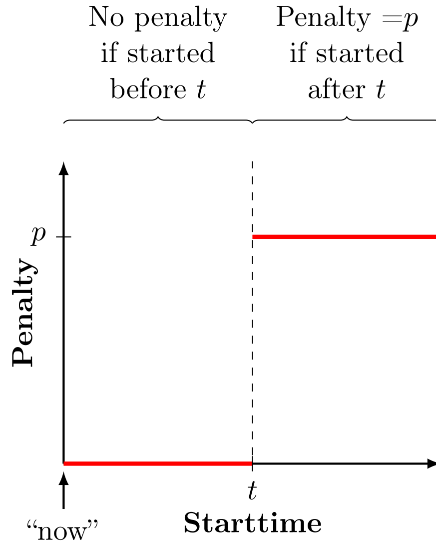


Figure 3.3.: Penalty function that encodes a penalty p for starting the execution of a request after time t.

ultimate SLO violation. Mathematically, function $oc(c)$ that models opportunity costs is a piece-wise monotonically decreasing function that has its maximum at (c_{i+1}, Δ_i) and its minimum at $(c_i, 0)$. Similarly, *marginal gains* model the chance to re-achieve a higher service level, i. e., to reach s_i from s_{i+1}, thus "saving" Δ_i. If this appears to be "within reach", transactions are penalized more and more to eventually achieve the higher level. Function $mg(c)$, which represents the marginal gains, is an increasing function between points $(c_{i+2}, 0)$ and (c_{i+1}, Δ_i).

If the SLO conformance c of a transaction'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 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 *pen* as the maximum of the computed opportunity costs and the marginal gain of this transaction: $pen(c) = \max\{oc(c), mg(c)\}$.

There are an infinite number of possible implementations for $oc(c)$. We evaluated two different families of functions: First, the opportunity costs can be implemented as a parabola with degree k that has its minimum at c_{i+1} and its maximum at c_i. The function that defines the opportunity costs for the entire SLO is defined as:

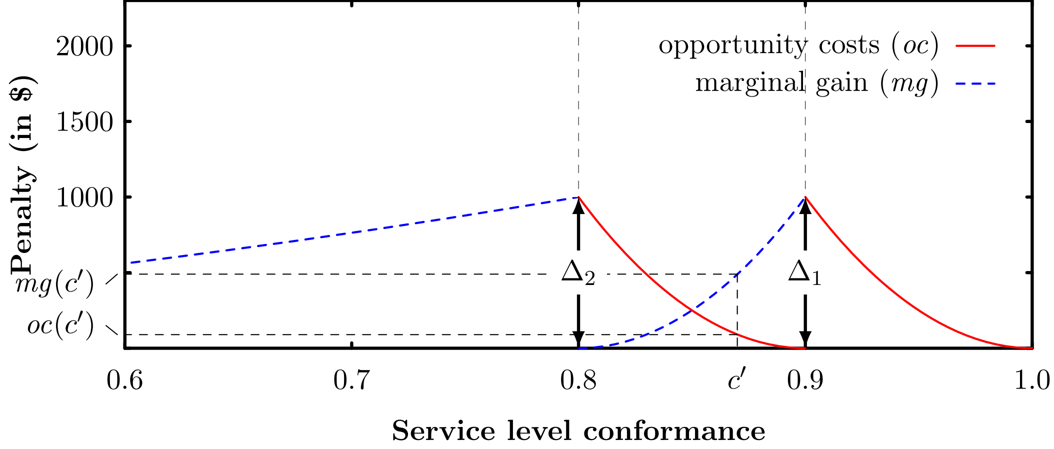


Figure 3.4.: Visualization of the economic model

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

Similarly, the marginal gain can be computed by a parabola of degree k:

$$mg(c) := \begin{cases} \left(\frac{c-c_{n+1}}{c_n-c_{n+1}}\right)^k \cdot \Delta_{n-1}, & c_{n+1} \leq c < c_n \\ \dots & \\ \left(\frac{c-c_3}{c_2-c_3}\right)^k \cdot \Delta_1, & c_3 \leq c < c_2 \\ 0, & \text{otherwise} \end{cases}$$

Figure 3.4 illustrates the computation of the penalty for objective d_1 using opportunity cost and marginal gain functions with degree $k = 2$. For this example, let the current conformance be $c' = 0.87$, i. e., the current service level is s_2. The additional costs for dropping to the next lower service level s_3 are $\Delta_2 = 1000$, resulting in opportunity costs of $oc(0.87) = 90$. The savings for reaching the next higher service level are $\Delta_1 = 1000$, yielding marginal gains $mg(0.87) = 490$. Thus, the penalty for the current transaction is $\max\{90, 490\} = 490$.

To define opportunity costs and marginal gains, we can choose different values for k to weight the distance from the borders of neighboring service levels. For small values for k (e. g., $k = 1$), transactions stemming from SLOs with high penalties will almost always be preferred compared to their lower priority counterparts. Figure 3.5(a) illustrates an example, with two users u_l and u_h that have a 90% objective. The

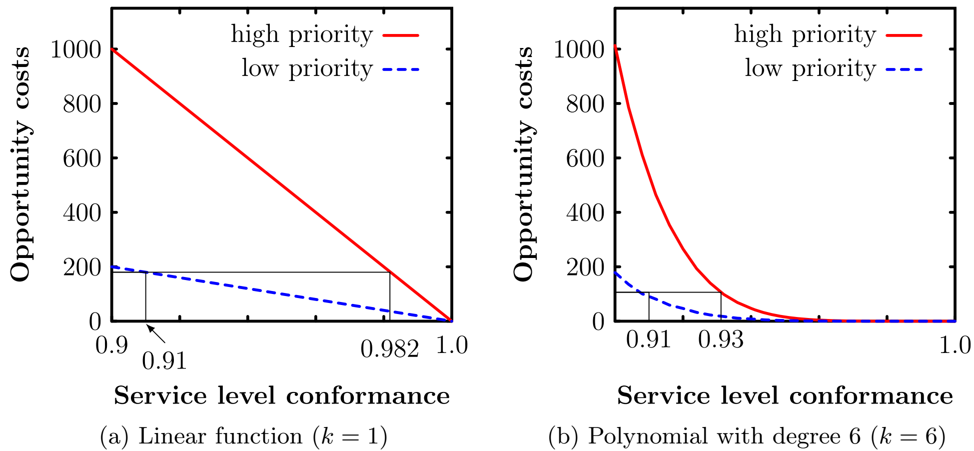


Figure 3.5.: Opportunity costs in the interval $[0.9, 1.0]$. There are two functions shown: one for a high priority percentile objective (red solid line) and one for a lower priority percentile objective (blue dashed line).

costs for dropping to the next lower service level is 1000 for the "high priority" user u_h and 200 for the "low priority" user u_l. Let the current conformance of u_l be 0.91, i. e., very close to the 90% objective, resulting in an opportunity cost value of 180. However, the opportunity cost values of the u_h is greater than or equal to 180 for conformance values ≤ 0.982. Although there is a high risk that u_l drops to the next lower service level, transactions from u_h will almost always be preferred even though the distance to the next lower service level may by considerably big. Figure 3.5(b) illustrates the marginal gains defined by a polynomial of degree 6. Again, we assume that the conformance of u_l is 0.91, resulting in a penalty value of about 106. With the higher degree polynomial, u_h has a higher opportunity cost value if the conformance drops below 93.1%. As a consequence, the higher degree polynomial gives a higher chance to lower priority users if their conformance is close to a service class border. As a consequence, the transactions from all users have high priority only for SLO conformances near the borders of the next higher and next lower service level, respectively. So, if marginal gains and opportunity costs are defined by higher order polynomials, there are only very few transactions with high priority. If all of these transactions are delayed, e. g., by waiting for locks, the SLO conformance falls onto the next lower service level.

Estimation of execution times

The computation of the time constraints derived from the percentile and deadline objectives requires information about the estimated execution times of queries. Therefore, we briefly discuss our approach to estimate the execution time of requests.

For the computation of the time objective of an individual request, information

about the number of requests belonging to a single transaction must be known. If, e. g., requests are executed in a loop, the number of iterations must be available prior to the execution of the first request. For transactions with unknown characteristics, techniques like code inspection or machine learning approaches could be used to derive for a given set of parameters the number of requests that will be sent in this invocation of the transaction. However, the exploration of such techniques for handling transactions with unknown characteristics is not in the scope of this thesis.

For estimating the execution time for a request, we assume a linear relationship between the number of requests that are processed in parallel and the response time of an individual request. We can observe this linear relationship if two conditions hold: (1) The database systems assign the system resources in a round-robin manner, i. e., all pending requests get an equal share of the resources, e. g., time-slices on the CPU with equal length. (2) The database system is not in overload, e. g., switching between different requests causes significant overhead. Since the priorities we compute do not affect the execution of requests in the database, all requests have the same priority in the database system. As a consequence, in the database systems we used we assume that condition (1) holds. To meet condition (2), i. e., to avoid overload on the database system, we limit the maximum number of requests that are processed in parallel.

One important aspect is that the influence of the server load on the execution time is dependent on the type of the request. The impact of the database load on very simple selects on a primary key is different than the impact on more complex requests. As a consequence, we monitor the execution times per request type in a user load. For every occurrence of a request, we maintain the execution time and the number of requests that were executed in parallel when the request was started. We note that the number of requests in the database system may change during the execution of a request. However, since the execution times of the requests in our scenario are short, we assume that the variance of the number of concurrent requests is negligible. Based on the monitoring data, we apply linear regression [29]. We note that for different workloads, different techniques for estimating the execution times of queries may be necessary. However, further estimation techniques are not in the focus of this thesis.

Derivation of the time objective for an individual request

With $et_1^{est}, \dots, et_n^{est}$, we denote the estimated execution time of the requests in the transaction. We explain below how we estimate the execution time of the requests of the transaction. Since requests arrive one after the other within a transaction, $\sum_{k=1}^{n} et_k^{est}$ denotes the sum of execution times of the requests that are still to be executed in the transaction, i. e., the time needed to complete the transaction without delaying any of the requests $r_1, \dots, r_n$. The *slack*, the difference between the remaining time before the transaction must be complete and the sum of the execution times of the requests, defines the maximum amount of time the execution of the first request can be delayed. Figure 3.6 summarizes the relationship between response

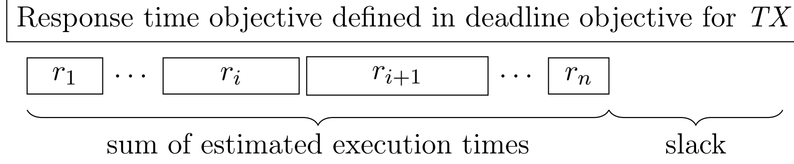


Figure 3.6.: Response time for transaction (taken from SLO), (estimated) execution time for requests, slack

time objective, estimated execution time, and slack.

As mentioned in Section 3.3, only the first request r_1 of a transaction TX is queued. All other requests $r_2, \ldots, r_n$ in this transaction bypass the scheduler. Let x_p denote the response time defined in the percentile objective of the transaction. As a consequence, we can assign the entire slack to the first request in the transaction, i. e., the time objective tc_1 for the first request is $tc_1 = now + x_p - \sum_{k=1}^{n} et_k^{est}$ where now is the current time.

Derive the deadline objective for individual requests

The time objective of a deadline objective x_d specifies an upper bound for the response time of a transaction. Similar to the computation of the time constraint from the percentile objective, we need to derive the deadlines for the first request of that transaction. With enf_1, we denote the latest time at which request r_1 should be executed to be able to complete the respective transaction within the time objective given by x_d. To compute enf_1, we can reuse the formula above: $enf_1 = now + x_d - \sum_{k=1}^{n} et_k^{est}$.

3.5. Request scheduling

Our goal is to dequeue requests based on their penalty functions such that the overall sum of incurred penalties is minimized. In general, finding an optimal solution to this minimization problem in NP-hard [37] and hence the study of heuristics is of interest. We concentrate on three heuristic algorithms, the Most Expensive First (MEFI), the Fisher-Krieger algorithm [21] and the Keep Approximation of the Fisher-Krieger Algorithm (KAFKA), a simplified version of the Fisher-Krieger algorithm.

Using the MEFI algorithm, requests are scheduled based on the height of the next step in the corresponding penalty function. Let pf_r denote the penalty function attached to a request r. Furthermore, let p_r denote the penalty at the current time t, i. e., $p_r = pf_r(t)$, and let $next_p_r$ denote the penalty at the "next step" of the penalty function. If there is no such next step, $next_p_r = p_r$. The MEFI algorithm maintains a queue where the requests are ordered by decreasing values of $next_p_r - p_r$. Using a priority queue implementation, requests are inserted and removed, respectively, in $O(\log n)$ time, where n is the number of requests in the queue. A request must be

```
FisherKrieger(R)
 1  t ← 0
 2  N ← R                         ▷ Requests yet to be processed
 3  T ← Σ_{r∈R} et_r^est          ▷ Sum of estimated execution times et_r^est
                                  ▷ of the queries r ∈ R
 4  for i = 0 to |R| − 1
 5        do
 6           ▷ pf_r: penalty function associated to r
 7           Choose r ∈ N such that (pf_r(T) − pf_r(t)) / et_r^est = max_{r̂∈N} ({ (pf_r̂(T) − pf_r̂(t)) / et_r̂ })
 8           Schedule r at position i
 9           t ← t + et_r^est
10           N ← N \ {r}
```

Figure 3.7.: Pseudo code of the Fisher-Krieger algorithm

```
KAFKA(R)
1  for all r in R
2        do p_r ← penalty at current time t
3           next_p_r ← penalty at the "next higher" step in the penalty function
4           et_r^est ← estimated execution time
5           compute Δ_r ← (next_p_r − p_r) / et_r^est
6  Sort the requests by decreasing order of Δ_r
```

Figure 3.8.: Pseudo code of KAFKA

rescheduled if its penalty function has reached the next step since the last scheduling event. Thus, the worst case time complexity is $O(n \log n)$ if all requests have reached their next step in the penalty function. We observed in our experiments that at each scheduling event only few requests had to be rescheduled.

Although MEFI already yields good results for scheduling the requests – as shown in Section 3.7 – the scheduling can be improved by additionally considering the time restrictions for an individual request. For example, consider two requests r_1 and r_2 having an execution time of 50 time units. Let

$$pf_1 = \begin{cases} 0 & \text{if } t \leq 10 \\ 100, & \text{otherwise} \end{cases} \quad \text{and} \quad pf_2 = \begin{cases} 0, & \text{if } t \leq 100 \\ 500, & \text{otherwise} \end{cases}$$

denote the penalty functions of r_1, r_2, i. e., no penalty is due if r_1 and r_2 are started before time 10 and 100, respectively. The MEFI algorithm considers the penalties

of the requests only, i. e., the order of execution would be to process r_2 and then r_1, resulting in a penalty for starting r_1 too late.

The Fisher-Krieger (FK) algorithm devised in [21] is an algorithm to schedule jobs without preemption on a single machine to maximize profit. The main idea of the algorithm is to use a linear approximation of the penalty function by considering the cost increase incurred by the request r between the scheduling time t and a "time horizon" T. FK is a heuristic approach for maximizing the sum of profits, which is equivalent to finding the minimal sum of incurred penalties from the perspective of optimization. Let et_r^{est} denote the estimated execution time of request r, $\mathcal{R}$ the set of queued requests, and $\mathcal{N} \subseteq \mathcal{R}$ the set of unscheduled requests. Furthermore, let T denote the "time horizon", i. e., the sum of estimated execution times of all requests in $\mathcal{N}$. The algorithm (see Figure 3.7) schedules the request first that causes the largest cost increase. The time complexity as indicated by the implementation above is $O(n^2)$.

We developed KAFKA to decrease the time complexity compared to the FK algorithm by applying a greedy approach for the request scheduling (w. r. t. the achievable profit per time unit). Figure 3.8 shows the pseudo code for KAFKA where p_r and $next_p_r$ for a request r are defined as above. In contrast to FK, where the penalty increase between the scheduling time and the time horizon is considered, KAFKA computes for all requests the increase of penalty between the current and the next step of the penalty function w. r. t. the estimated execution time. Afterwards, the requests are sorted in decreasing order according to the increase of penalty. Similar to MEFI, the worst case running time of KAFKA is $O(n \log n)$.

3.6. Experimental setup

This section describes the experimental environment that was used to evaluate the dynamic prioritization. For the performance evaluation, we chose the TPC-C benchmark [59] as a representative online transaction processing (OLTP) workload. The TPC-C benchmark models a wholesale supplier operating several warehouses that 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 [59]. For the experiments in Section 3.7.2 we control the load on the system by multiplying the keying and think times with a scale factor $f > 0$. Values $f > 1$ stretch the think and keying times, i. e., fewer transactions are started in a time window and, consequently, the load on the system is lower. Values $0 < f < 1$ shorten the think and keying times, i. e., transactions are started faster, resulting in a higher load on the system.

For our experiments, we implemented the TPC-C benchmark based on MaxDB 7.5.00.26 [41]. Appendix A contains the DDL statements we used to create the tables and indices. We loaded 20 warehouses into the database. The disk space consumption

Transaction	Relative frequency (in %)	Percentile objective (in sec)
NewOrder	45	5
Payment	43	5
OrderStatus	4	5
Delivery	4	80
StockLevel	4	20

Table 3.1.: Percentile objectives for the TPC-C transactions

of the tables is about 2GB, the indexes consume about 0.36GB of disk space. As a consequence, we set the number of terminals to 200 (TPC-C requires 10 terminals per warehouse). Using this setting results in maximum throughput where 99% of all transactions are still processed within their respective response time objectives in Table 3.1 with FIFO used as scheduling discipline, no throttling applied (i. e., the number of requests allowed in parallel was equal to the number of terminals), and the scale factor f set to 1. Increasing the number of warehouses (and, thus, the number of terminals) would result in a higher percentage of transactions that violate the response time requirements.

We conducted all experiments using a database server with 1GB RAM and a single Intel Xeon chip. The chip had a single core clocked at 2.8GHz and supports Intel's Hyper-Threading technology. The machine had a single 15000RPM disk with 36GB. As operating system, we were using SUSE Enterprise Linux 9 running a Linux kernel v2.6 that was shipped with the distribution. The terminals were executed on two different machines and submitted the requests via iJDBC. iJDBC is a generic wrapper around the JDBC driver provided by the database vendor to intercept database requests. Using iJDBC, it is possible to execute code before and after the execution of a request, e. g., it is possible to measure the execution time, wait time, and the response time of individual requests and entire transactions. Also, iJDBC communicates with the SLO component and the scheduler without changing the client code.

The SLO for a TPC-C transaction is based on the corresponding response time objective. For our experiments, we specified the SLOs using XML, similar to WS-Agreement [2], which has become a standard for establishing a service agreement between a service provider and a client. Our experiments are conducted with the step-wise SLOs. For each transaction, we define an SLO with a percentile and an deadline objective. The percentile objective requires 90% of the invocations to be processed within a specified response time objective. Table 3.1 shows the transaction mix and the time objectives, which are specified in [59]. A violation of this objective is fined with a penalty depending on the terminal representing the client that invokes the transaction, i. e., the SLO applies for the terminal and all transactions that are invoked from this terminal. In our test scenario, we chose a customer-mix where 15%

Algorithm	$n = 50$	$n = 100$	$n = 150$	$n = 200$	$n = 250$	$n = 1000$
MEFI	0.02	0.03	0.03	0.03	0.03	0.18
FK	3.06	11.70	26.07	45.44	70.73	–
KAFKA	0.03	0.03	0.04	0.05	0.04	0.21

Table 3.2.: Average elapsed time in milliseconds for determining the “best” request in a queue with n requests.

of the terminals incur high ($1000), 35% incur medium ($200), and the remaining terminals incur low penalties ($40) if the corresponding objective is violated. This customer mix models a service provider with a high number of regular customers that must be preferably processed compared to “normal” counterparts.

3.7. Experiments

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 requests, irrespective of their individual complexity [52].

3.7.1. Overhead

With the first set of experiments, we measure the overhead that is caused by the dynamic scheduling. We identify three components of overhead: The *architectural overhead* is caused by the network communication between the clients and the SLO component and between the clients and the scheduler. The *preprocessing overhead* includes the time needed for computing the penalty function for an individual request and the time needed for attaching the penalty information to the request. The *scheduling overhead* is the time needed for determining the order of the requests in the queue and depends on the complexity of the scheduling discipline.

For measuring the scheduling overhead, we filled a queue with $n - 1$ requests having randomly generated penalty functions. One measurement cycle consisted of adding the nth request with random penalty function, reordering the queue, and removing the first request from the queue. Table 3.2 shows the average elapsed time in milliseconds for 5000 add-schedule-remove iterations. We see from the table that even for small queues, the FK algorithm causes significant overhead while MEFI and KAFKA have low overhead even for longer queues. In our experiments (see below), we observed queue lengths of up to 150 requests, which results in a still acceptable overhead, considering transaction response times of a few seconds. However, for

queue lengths beyond 500, the overhead becomes dominant and FK is no longer applicable.

We measured the architectural overhead by running a TPC-C NewOrder transaction in isolation and comparing the response time when SLO component and scheduler were turned off and the response time with SLO component and scheduler turned on. We chose the NewOrder transaction because this transaction consists of the most requests. In these experiments, the response time of the runs with SLO component and scheduler turned on was about 20 milliseconds greater than the response times of the runs with the components turned off. We note that a very small fraction (about 0.2 milliseconds) of this overhead is due to scheduling because even if the requests are not held back, there is some time needed for inserting them into the queue and removing them immediately.

3.7.2. Evaluation of dynamic prioritization

In this section, we evaluate the different approaches for dynamic prioritization – MEFI and KAFKA – and compare them to the static prioritization schemes. For the remainder of this section, we concentrate on the NewOrder transaction, which is the central transaction in the TPC-C benchmark. We note that, however, we did not apply dynamic prioritization only to NewOrder but to all TPC-C transactions. Our metric to compare the different approaches is the sum of the penalties that are due for violated objectives of the NewOrder transactions.

When we ran the database at its limit, i. e., $f = 1$, we observed that the maximum number of transactions that were executed in parallel is ≤ 10 99% of the time (most of the time, there were even fewer active transactions). Consequently, we set the maximum number of transactions that are allowed to run in parallel to that number for our experiments. During low load phases, the queue of the scheduler was almost always empty. However, in the high load phases, there were up to 150 pending queries.

A single experimental run consists of several phases. We start with a warm-up phase where the database is operated at 80% load ($f = 1/0.8$) for 15 minutes. We chose the 80% under the assumption that no database system will be operated at its limit because of possible peaks. Subsequently, 8 minute high load periods (180% load, $f = 1/1.8$) alternate with 15 minute low load periods (80% load, $f = 1/0.8$).

No prioritization (FIFO) vs. static prioritization

Figures 3.9(a) to 3.9(c) show the end-to-end response times for the high, medium, and low priority terminals, respectively, in a 65 minutes interval for the FIFO algorithm. Note that Figures 3.9(a) to 3.9(c) do not all have the same vertical scale. The vertical gray bars indicate the high load-phases (minutes 15-23 and 37-45). During the high load-phases, the response times of all terminals increase to the same degree, i. e., the response times are almost identical, no matter whether the respective requests stem from a terminal with higher priority. We observe that even after the high load phase,

the response times do no fall back to low load-level because, since the arrival rate of requests was higher than throughput in the high load-phase, the database system must work off the backlog. Once the backlog has been cleared, the response times drop again. The peaks in the low load-phases (e. g., around minute 28 in the figures) occur when the arrival rate has a peak.

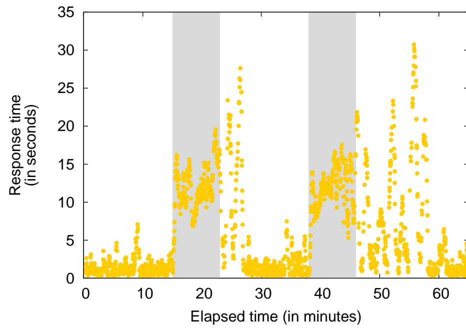
Each bar in Figure 3.9(d) shows the number of completed NewOrder transactions for high priority (yellow bars), medium priority (orange), and low priority (brown) terminals. The terminals in each priority-group completed between 130 and 150 NewOrder transactions in 65 minutes.

Figure 3.9(e) shows the deviation, defined as the difference between the 90%-objective and the actual compliance, for each terminal. Similar to the previous figures, the colors of the bars indicate the priority of the terminal. Positive values indicate that the compliance after 65 minutes exceeds 90% while negative values indicate a violation of the 90%-objective. The figure shows that the deviation varies between 16.3% and 40% (i. e., the compliance is between 50% and 73.7%). Since the response time objectives of the NewOrder transactions of all terminals are violated, the sum of penalties incurred in this scenario is $48000.

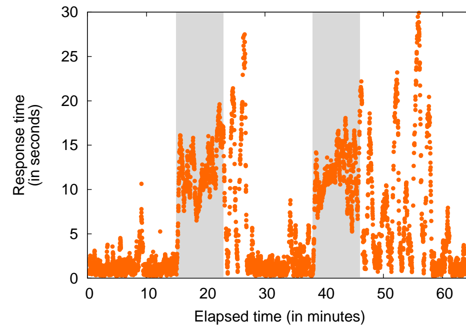
From the experiments with the FIFO scheduling discipline we can draw two conclusions: First, since FIFO treats all transactions identically, the total wait time of a transaction is independent of the priority of the transaction. Second, the database time of a request is not affected by its priority (because all requests have the same priority). Consequently, the total database time of a transaction (i. e., the sum of the database times of the individual requests) is also identical for the requests stemming from the different terminals.

The results of the experiments with static prioritization are summarized in Figure 3.10. The response time graphs (Figures 3.10(a)-(c)) show that the response time of the high priority clients is lower for higher priority clients. Note that the figures have different vertical scales. In the low load phases, all clients experience response times below 5 seconds. However, in the high load phases, there are not enough resources to process the all incoming requests. As a consequence, the response time of only the high priority clients remains low during the high load phase. The response time of the lower priority requests increases because they are only processed when there are no high priority requests in the queue. In particular, most of the low priority requests are delayed until after the high load phase, as indicated by the spikes at minutes 23 and 45 in Figure 3.10(c). The reason is that the probability that there are no requests with higher priority is low. The execution of the low priority requests after the high load phase leads to an increase of response time for the higher priority requests (high response times for high and medium priority after minutes 24 and 46). The reason is that our dual-queue scheduling favors the execution of already active requests. As a consequence, only the (low priority) transactions that became active after the high load phase are processed – delaying the higher priority transactions that haven't started yet.

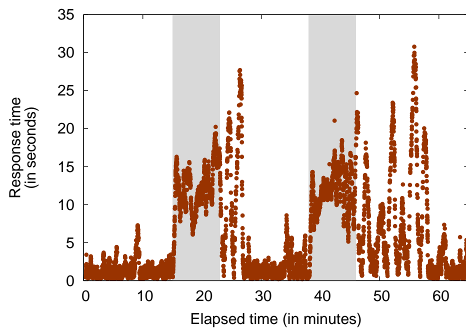
In summary, the static prioritization decreases the penalty for violated objectives compared to the FIFO scheduling to $11400 (37 medium priority and all low pri-



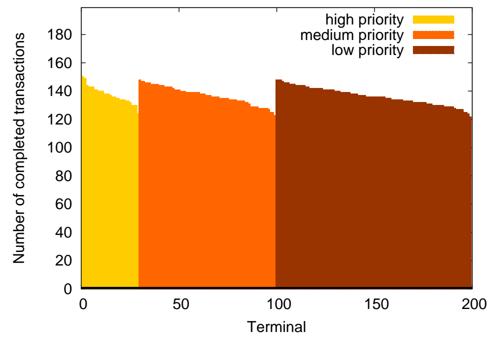
(a) Response times of the high priority NewOrder requests



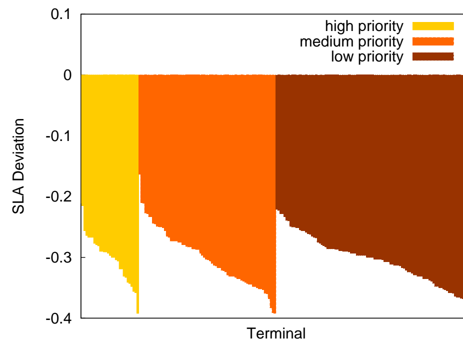
(b) Response times of the medium priority NewOrder requests



(c) Response times of the low priority NewOrder requests



(d) Number of completed transactions per terminal after 65 minutes (total: 27228)



(e) The deviation from the objectives after 65 minutes. Negative values denote an underfulfillment of the objectives.

Figure 3.9.: (FIFO, no deadlines) The graphs show the results after running a benchmark where the scheduler controls the parallelism in the database and puts the requests in a FIFO queue.

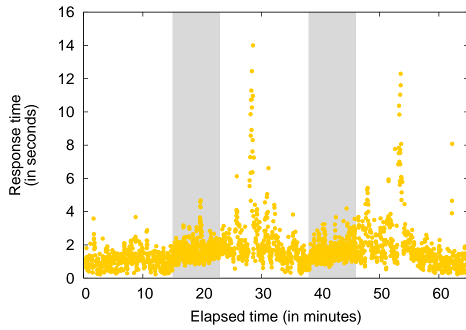
ority terminals violate the objectives for the NewOrder transaction). However, as Figure 3.10(e) indicates, the high priority and some of the medium priority terminals overexceed the objectives. Also, the number of NewOrder transactions completed depends on the priority. While all high priority terminals and most of the medium priority terminals complete more than 170 NewOrder transactions, the low priority terminals complete less than 110 transactions during in the 65 minutes.

Static prioritization vs. dynamic prioritization

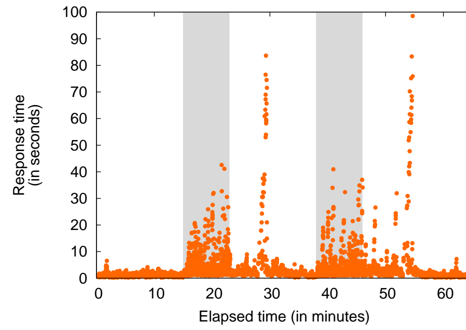
Figure 3.11 shows the results for the MEFI algorithm. With MEFI, the compliance for the high priority users has decreased – although all of them meet the objectives, as shown in Figure 3.11(e). Therefore, there are more resources to process requests from lower priority terminals, so that fewer objectives for medium priority terminals are violated compared to the static prioritization. In total, the objectives for 15 medium priority and all low priority terminals are violated, yielding a total penalty of $7000 for violating the NewOrder objectives.

At the beginning of the experiment, the priority of the requests is correlated with the priority of the terminal: higher priority requests are processed faster on average than their lower priority counterparts. In contrast to the static prioritization, the high priority requests are also delayed in the high load phase to allow lower priority requests to be processed faster. Figure 3.11(a) to 3.11(c) show the response times of the NewOrder transactions. Note the high variance of the response times in Figure 3.11(a): The execution of higher priority transactions is delayed until after the first high load phase — which we only observed for low priority terminals using the static prioritization. As a consequence, some high and medium priority terminals complete fewer NewOrder transactions compared to the static prioritization (Figure 3.11(d)). Since the total number of completed NewOrder transactions is similar to using static prioritization, the low priority terminals complete more transactions.

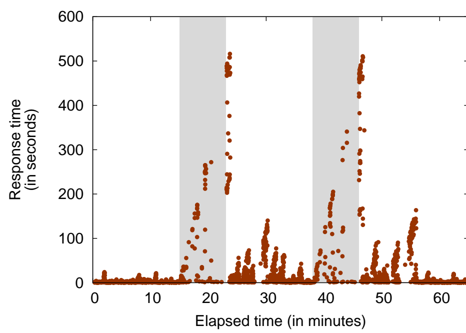
As described in Section 3.5, dynamic prioritization can be improved by considering not only the penalty for a request (i. e., the “height” of the step in the penalty function) but also the time when the request should be started (i. e., the “step width”). Figure 3.12 summarizes the experiments with the KAFKA algorithm. As indicated by Figures 3.12(a) and 3.12(b), more requests stemming from high and medium priority terminals are delayed compared to the MEFI approach, resulting in fewer completed NewOrder transactions. Although in the experiment using KAFKA more NewOrder transactions have been processed (compared to MEFI), a comparison of Figures 3.11 and 3.12 shows that due to the delay of higher priority requests, the percentage of completed low priority NewOrder transactions is higher when using KAFKA (about 45%) than when using MEFI (about 40%). Figure 3.12(d) also shows that the NewOrder transactions for some medium priority terminals are starved (not processed at all). The reason for the starved transactions is that a terminal first completes some NewOrder transactions, having a compliance near 100%. As a consequence, the requests have low penalty (due to the high conformance). With low penalty, there is a chance that the start time that is encoded in the penalty function



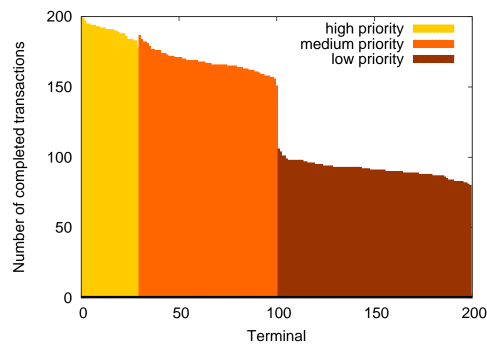
(a) Response times of the high priority NewOrder requests



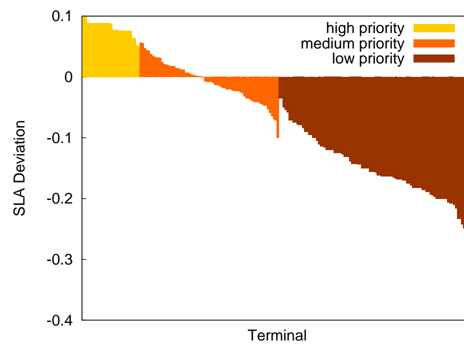
(b) Response times of the medium priority NewOrder requests



(c) Response times of the low priority NewOrder requests

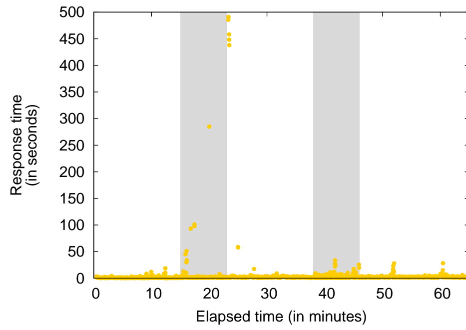


(d) Number of completed transactions per terminal after 65 minutes (total: 26664)

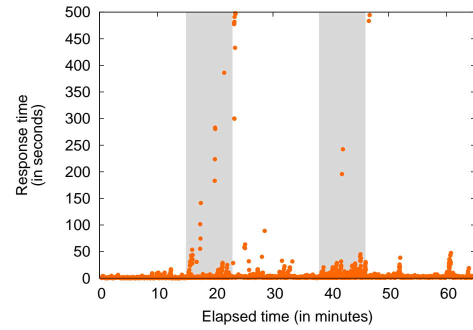


(e) The deviation from the objectives after 65 minutes. Negative values denote an underfulfillment of the objectives.

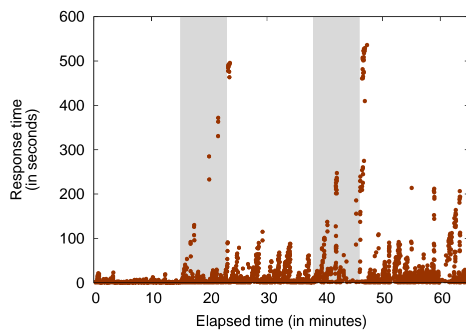
Figure 3.10.: (Static, no deadlines) The graphs show the results after running a benchmark where the requests have static priorities and the scheduler always executes the highest priority request in the queue.



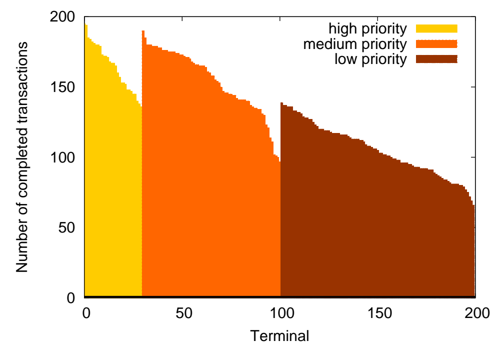
(a) Response times of the high priority New-Order requests



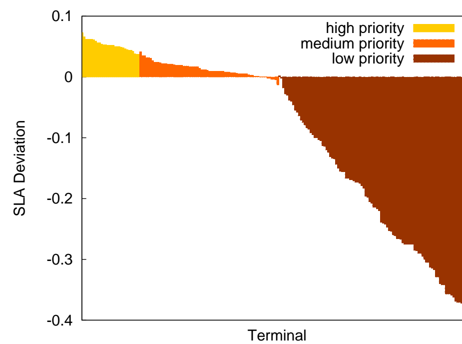
(b) Response times of the medium priority New-Order requests



(c) Response times of the low priority NewOrder requests



(d) Number of completed transactions per terminal after 65 minutes (total: 26277)



(e) The deviation from the objectives after 65 minutes. Negative values denote an underfulfillment of the objectives.

Figure 3.11.: (Dynamic, MEFI, no deadlines) The graphs show the results after running a benchmark where the requests have dynamic priorities and the queue is managed by the MEFI scheduling discipline.

Transaction	Deadline objective (in sec)		
	high	**medium**	**low**
NewOrder	15	25	50
Payment	15	25	50
OrderStatus	15	25	50
Delivery	80	400	800
StockLevel	60	100	200

Table 3.3.: Deadline objectives for the TPC-C transactions

passes, so that there is no benefit in processing the pending request. Consequently, the request remains queued until there are no more pending requests.

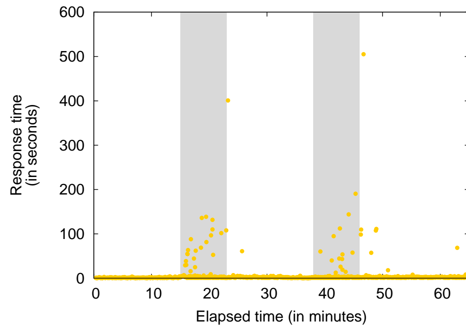
We also ran experiments with the Fisher-Krieger algorithm (results not shown in the figures). Both KAFKA and FK yielded similar results: none of the percentile objectives of the high and medium priority user loads were violated. While for KAFKA, the objectives for 81 low priority terminals were violated, yielding a total penalty of $3240, using the FK algorithm, 76 low priority terminals violated their percentile objectives.

From the dynamic prioritization experiments, we can draw the following conclusions: Using dynamic prioritization (MEFI, FK, or KAFKA) balances the objectives across all priority levels. However, delaying the transactions results in fewer completed transactions for high and medium priority terminals. Also, some of the transactions are delayed for a very long time (more than 500 seconds in our experiments), or not even processed at all, which is unacceptable in a real application. Therefore, we examine how dynamic prioritization behaves when we enforce a maximum execution time for transactions stemming from high and medium priority terminals.

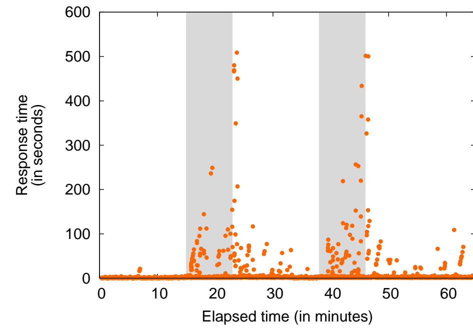
Dynamic prioritization with deadline objectives

In order to avoid long delays and starved transactions, we extended the objectives from the previous set of experiments with a deadline objective (Section 3.4.1) to enforce the execution of requests. In addition to the percentile objectives shown in Table 3.1 we configured the deadline objectives as shown in Table 3.3 to enforce the execution of a request.

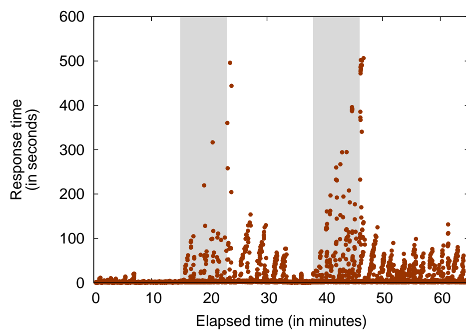
Figures 3.13(a) to 3.13(c) show the response times of the NewOrder transactions using the KAFKA algorithm and enforced executions. The figures show that the maximum response time is in the range of the deadline constraints. The response times greater than the deadline constraints occur, e. g., in the load peaks, because the deadlines only enforce the execution after waiting for the specified amount of time. However the response time also contains the time needed to execute the request. In addition, if there are two requests whose execution must be enforced, the first



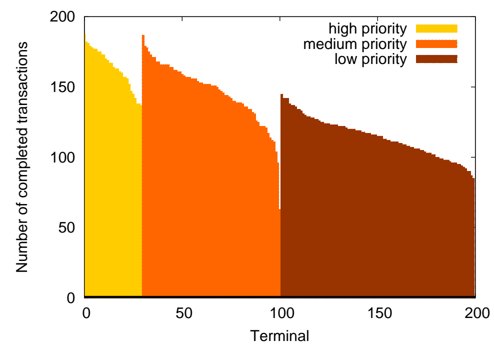
(a) Response times of the high priority NewOrder requests



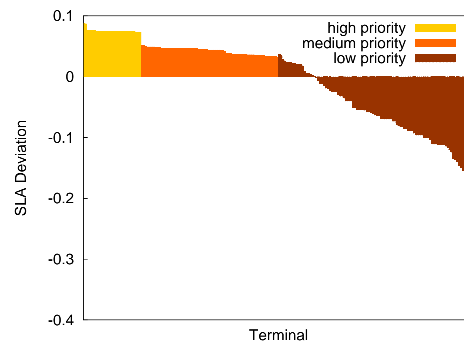
(b) Response times of the medium priority NewOrder requests



(c) Response times of the low priority NewOrder requests



(d) Number of completed transactions per terminal after 65 minutes (total: 26639)



(e) The deviation from the objectives after 65 minutes. Negative values denote an underfulfillment of the objectives.

Figure 3.12.: (Dynamic, KAFKA, no deadlines) The graphs show the results after running a benchmark where the requests have dynamic priorities and the queue is managed by the KAFKA scheduling discipline.

request may "block" the second one, which must wait some time for another currently executed request to complete.

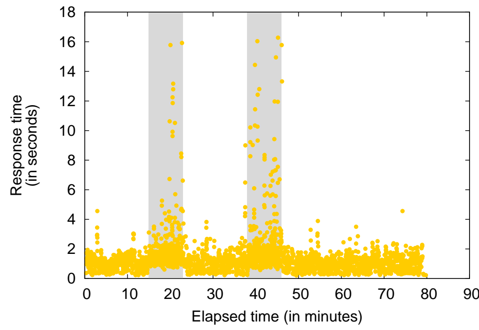
There are two notable results from the enforced execution of requests. First, the low priority requests complete more transactions compared to the experiments with no deadline because the requests are delayed for a shorter time due to the enforced execution, i. e., the requests are no longer delayed for several minutes. Similarly, there are no starved requests as we observed in the experiments with no deadlines. Second, the enforcement may result in more objective violations compared to the experiments without deadline objectives. Requests that are close to the step in their penalty function may be delayed by requests whose execution has been forced. Note that in this case, all requests whose execution must be enforced are executed before requests with penalties derived from their penalty functions: A request stemming from a low priority terminal may delay a request from a high priority terminal whose penalty function indicates very high priority. As a consequence, more percentile objectives are violated compared to the experiments with no deadline, i. e., no enforced request executions (Figure 3.13(e)). However, the total penalty of $3800 is only marginally greater than the penalty in the experiments without deadlines ($3240).

3.8. Conclusions and future work

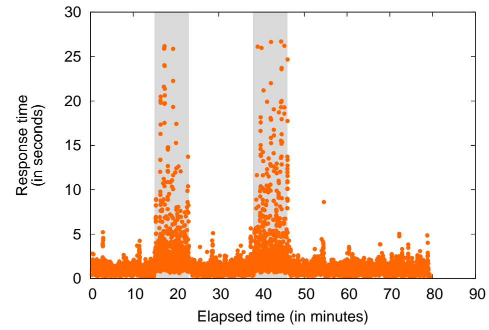
This section presented a dynamic prioritization approach to avoid overfulfillment of service level objectives. The dynamic prioritization is based on an economic model, which differentiates between opportunity costs and marginal gains. The architecture of our QoS management comprises an SLO component that computes a penalty function for individual requests using the economic model and annotates the requests with the penalty function. The requests are then managed by a scheduler that determines the order in which the requests should be processed. We devised MEFI and KAFKA, two algorithms to determine an order of the requests to minimize the incurred penalties. The prototypical implementation of the QoS management framework demonstrated the effectiveness of our dynamic prioritization approach to avoid overfulfilling service level objectives of high priority classes at the expense of lower priority classes.

We are currently working on extending the dynamic prioritization approach to an end-to-end QoS management in a multi-level infrastructure in operational transaction processing systems. In our setting the requests are processed at the Web server, application server, and database layer and each of the components is executed on a separate machine. Depending on changes in the workload and the load on the different machines, a different component may become the bottleneck so that scheduling must be applied at the respective layer. Since the service level objectives are only defined for the "top" layer of the architecture, i. e., the service invoked by the user, one challenge is to infer the objectives for each layer from the objectives of the layer above.

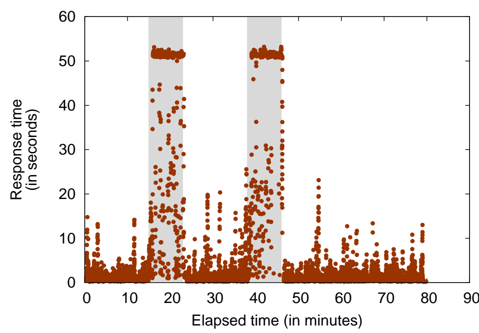
We also work on supporting dynamic deadlines to model the more complex objec-



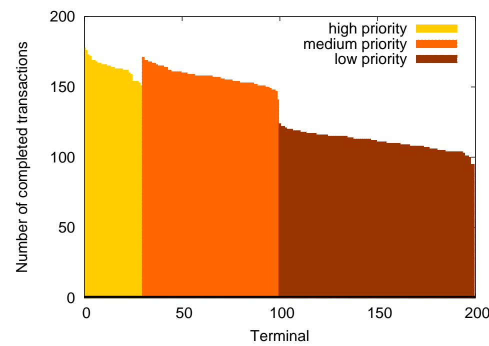
(a) Response times of the high priority NewOrder requests



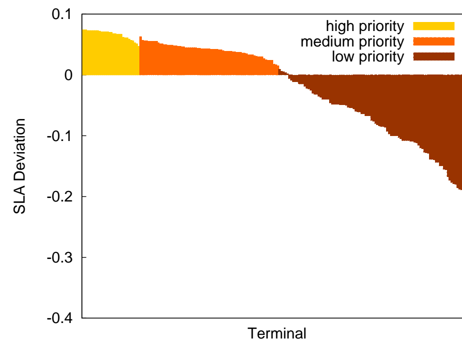
(b) Response times of the medium priority NewOrder requests



(c) Response times of the low priority NewOrder requests



(d) Number of completed transactions per terminal after 65 minutes (total: 27065)



(e) The deviation from the objectives after 65 minutes. Negative values denote an underfulfillment of the objectives.

Figure 3.13.: (Dynamic, KAFKA, with deadlines) The graphs show the results after running a benchmark where the requests have dynamic priorities and the queue is managed by the KAFKA scheduling discipline. The execution of requests is enforced after 15 seconds, 25 seconds, and 50 seconds for requests stemming from high, medium, and low priority terminals.

tives. For example, the TPC-C benchmark defines an objective where the average response time must be lower than the 90th percentile of the response time, which in turn must be lower than a pre-defined threshold. From the requirement "average response time $\leq 90^{th}$ percentile", we can compute a dynamic deadline for a transaction and, with the approach presented in this thesis, the deadlines for individual requests.

4. Query control for BI workloads

The previous chapter focused on OLTP-style workloads and how to manage them through scheduling by limiting the number of concurrently executing queries and adaptively setting the priority of requests. This chapter focuses on longer queries, business intelligence (BI) queries, for which there are more options for query control.

Long-running queries plague database administrators, who are forced to decide which queries are hurting system performance and what to do about them. Data skew, poorly-written SQL, poorly-optimized plans, and resource contention regularly lead to poorly-behaved, unpredictable queries. Business intelligence workloads make this task more difficult. A single workload may include short transaction-processing queries that take only milliseconds of CPU and I/O time as well as long, complex, analytic queries that run for hours as they access and process terabytes of data. Different workloads may have different objectives, such as query throughput, elapsed time for a set of queries, or an objective that measures both the queries completed and the ones aborted or not started.

Commercial systems support a number of actions, primarily focused on threshold-based admission control and user notifications of potential runtime problems. For example, workload management tools from IBM [17], Microsoft [44], and Oracle [47] will all alert the user if a query exceeds limits on estimated row counts, processing times, or joins by a given percentage. However, human experts are still responsible for choosing whether and how to act. These human practitioners talk about "problem" queries and have developed some intuition for dealing with them. However, we have not seen a thorough classification of long-running queries nor a systematic study of the most effective corrective actions.

We interviewed practitioners from a number of commercial database companies about workload management problems. Several said that any query that runs for too long (e. g., longer than 15 minutes) has a problem, such as a bad query plan. We therefore decided to focus on policies to identify and handle long-running queries. Note that the threshold to define a long-running query varies by workload and by system configuration and the job of administrators is to determine the value. We identify three common scenarios:

- **Unreliable cost estimates.** Early-detection policies that apply thresholds to cost estimates can have the biggest positive impact on performance either by preventing "problem queries" from starting or by postponing them to run last. However, optimizer cost estimates are known to be inaccurate – sometimes by multiple orders of magnitude.
- **Unobserved resource contention.** Workload management decisions are

based upon estimates and measurements of resource contention, but the measured resource may not be the major source of contention. Measuring CPU utilization does not address excessive contention for disks.

- **System overload.** Sometimes the database system is simply overloaded. Unlike the first two scenarios, no single query is at fault, and the only solution is to reduce the number of queries in the system.

In this section, we systematically evaluate the ability of existing workload management mechanisms to deal with these scenarios. In particular, we compare the effectiveness of various kinds of absolute and relative thresholds, consider the benefits of the "suspend" action [11, 12], and consider whether certain types of management policies should be combined in order to compensate their strengths and weaknesses. We use a simulator in order to run many more experiments and more methodically explore the space of policy combinations and workloads than would be possible using an actual database engine. Our evaluation uses a goodness metric that weighs both the queries completed *and* the queries left incomplete.

The primary contributions are:

- We develop a taxonomy of long-running query types based on how they impact other queries.
- We evaluate the ability of workload management policies to identify and act upon the problem scenarios described above using our experimental framework and database simulator.
- Finally, we make recommendations for which policies to use and demonstrate how to set their thresholds.

Our experimental results show that recognizing long-running queries early and acting upon them as soon as possible can halve overall workload times. We identify which combinations of policies work best if the goal is to eliminate "problem" queries from the system as soon as possible, and which work best if the goal is to complete the long-running queries while minimizing their impact on the rest of the workload. We also discuss which policies work best for predictable queries and which can handle the unexpected.

4.1. Related work

To our knowledge, few researchers explicitly consider long-running queries in workload management. Benoit [5] presents a goal-oriented framework that models DBMS resource usage and resource tuning parameters for diagnosing which resources are causing long-running queries and determining how to adjust parameters to increase performance. He does not address the evaluation of workload management mechanisms, nor does he model or manage the state of an individual query's execution.

	Query expected to be long	**Query progress reasonable**	**Uses equal share of resources**
expected-heavy	Yes	Yes	Equal share
expected-hog	Yes	Yes	> Equal share
surprise-heavy	No	Yes	Equal share
surprise-hog	No	Yes	> Equal share
overload	No	No	Equal share
starving	No	No	< Equal share

Table 4.1.: Query taxonomy: We distinguish types of long-running queries based on whether (1) we expected the query to take a long time, (2) the query is making progress toward completion, and (3) the query is receiving an equal share of measured resources, such as CPU time or disk I/Os.

Weikum *et al.* [62] discuss metrics appropriate for identifying the root causes of performance problems (e. g., overload caused by excessive lock conflicts). This was done in the OLTP context, not BI.

Query progress indicators attempt to estimate a running query's degree of completion. We believe such work is complementary to our goals and offers a means to identify various types of long-running queries at early stages, potentially before the workload has been negatively impacted. Existing approaches assume that the progress indicator knows the number of tuples already processed by each query operator [13, 14, 39]. Such operator-level information can be prohibitively expensive to obtain.

Luo *et al.* [40] leverage an existing progress indicator to estimate the remaining execution time for a running query in the presence of concurrent queries. They use these estimates to implement workload management policies, such as the ones that we study systematically.

4.2. Long-running query taxonomy

Effective workload management policies should be able to use cost estimates and simple runtime statistics to distinguish between a query that is a heavy user of system resources, one that is being starved by a heavy user, and one that is running in an overloaded system.

Table 4.1 shows our taxonomy of long-running queries based on how they contribute to system resource contention. First, we distinguish between queries expected

to take a long time and those that were not. Second, we look at whether the query is making reasonable progress. Third, we consider whether the query is using an equal share of resources relative to other queries, or whether it is getting significantly more or less of them. For example, if there are n concurrent queries, then each query is guaranteed at most $1/n$ of each resource. If a query needs less than $1/n$ of a resource, the share that is not used by the query is equally distributed among the other queries. We discuss how to measure these properties in Section 4.3.

Expected-heavy queries are predictable and allow other queries to make progress. *Expected-hog* queries are also predictably long, but use more than their share of the resources. They may interfere with concurrent queries.

Surprise-heavy and *surprise-hog* queries were expected to be short. These queries behave just like *expected-heavy* and *expected-hog* queries, respectively – but without warning. They are the most likely to cause problems for other queries and the most important to catch. Killing (and possibly requeuing) *surprise-heavy* and *surprise-hog* queries has the most impact on the completion time of the other queries in the workload.

Starving queries are those impeded by *expected-hog* and *surprise-hog* queries: they ought to be short, but are taking a long time because the *expected-hog* queries do not leave them enough resources. *Starving* queries that are killed and requeued when there is less contention will run faster. An alternative is to kill other queries that have made less progress. Finally, *overload* queries ought to be short, but there are simply too many queries in the system for any of them to make progress. The most commonly chosen way to relieve system overload is to reduce the number of concurrent queries.

4.3. Experimental framework

We believe that workload management policies informed by all three dimensions of our taxonomy (expectations, progress, and resource shares) can be more effective than those that consider only a single dimension, such as usage of a particular resource. We therefore built an experimental framework for workload management with which we can run thousands of realistic workloads under a variety of workload management policies while monitoring and controlling expectations (in the form of optimizer estimates), query progress, and resource share and measuring performance.

Our framework supports the workload management components depicted in Figure 4.1. Our workload manager implements different admission control, scheduling, and execution control policies and actions, which we synthesized from the policies of current commercial systems. Currently, we are not modeling different service classes. We implemented a simulator for the database engine that mimics the execution of database queries in a highly parallel, shared-nothing architecture. The simulator does not include components like the query compiler and the optimizer: we provide the query plans and the costs as input.

Using a simulated database engine was necessary. First, we investigate workloads

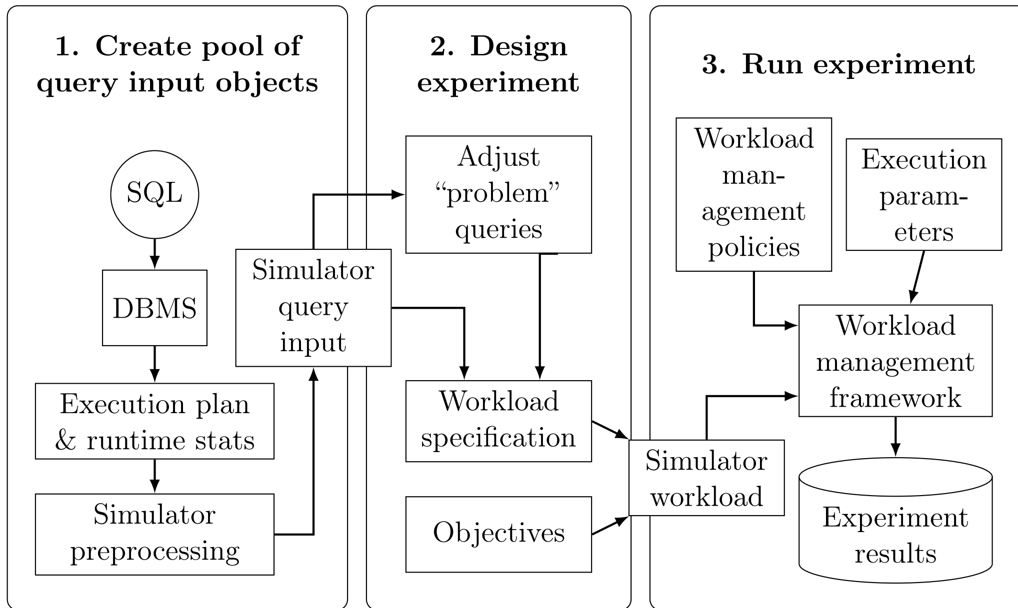


Figure 4.1.: Workflow of how we create and select from a pool of query objects, create workload input files, and specify parameters for our experiments.

that run for hours. Our simulated database engine "runs" these workloads in seconds, which let us repeat the workloads with many different workload management policies. Second, each workload management component in today's workload management systems implements only a subset of the possible workload management features described in Chapter 2. Using a real database would limit us to the policies that a particular product provides, contradicting our goal to experiment with an exhaustive set of techniques *and* to model features that are currently not available.

4.3.1. Workflow

Figure 4.1 sketches the workflow for our experimental framework. To create input, we first run queries in isolation on a real parallel database system – HP Neoview in our experiments – and collect their performance statistics. We then create a simulator input file that describes each query: the query plan and the CPU, disk, message, and other resource usage of each operator in the query plan.

We then design each workload by choosing a set of queries and adding objectives. For some workloads, we also inject "problem queries", i. e., hogs, into the workload. We give more details on how we model the problem queries below. Finally, we choose workload management policies and invoke the experimental framework. Each simulation run persistently stores a summary report for analysis. By running the same workload under various policies, it is possible to compare different workload management techniques.

4.3.2. Simulator implementation

The simulator must model query processing with enough detail to capture resource usage and contention but without needing to capture row-level data manipulation or specific query operator algorithms. Therefore, we simulate the resource consumption of individual operators in a query execution tree.

Query model

In a parallel database, a logical query operator, e. g., hash join, may be implemented as multiple instances of a physical operator: one instance of the hash join operator runs on each node. We use *operator* to refer to the physical operator that executes on a single node. We model each resource on each node (each CPU and disk) separately.

Each query has a tree of operators and each operator has its own resource costs. We model only the cost of the dominant resource for each operator, e. g., the CPU time of an aggregation operator, the number of disk I/Os of a table scan, and the network costs.

In order to run a simulated workload on the simulator, we need per-operator CPU and I/O time measurements. On our Neoview system, the measurement tools did not provide per-operator resource usage, so we had to estimate these from other metrics: Overall query CPU time was available so we allocated the CPU time to each operator instance in direct proportion to its input and output cardinalities (which were available). We estimated the disk I/O time by multiplying the actual number of rows accessed, which the tools did provide, by the disk speed. We estimated message time by multiplying the number of messages by the network bandwidth.

We simulate the operators of a query execution tree from the bottom up. An operator begins execution when all of its child operators complete. We did not model pipeline parallelism in these experiments.

Resource Sharing

By default, the simulator gives each query an equal share of each resource, e. g., two queries running concurrently on the same node would each get half the CPU. However, to model over-utilization, i. e., resource-hogging queries, a query may specify an unequal share. For example, one query may specify an 80% share of a CPU, which leaves 20% to be divided among the remaining "equal" share queries that are running.

4.3.3. Experiment input and output

The simulator input comprises a workload, a set of policies, and configuration information. Every query in the workload has an estimated cost and a "stretch" factor. To determine the actual resource usage, the simulator multiplies the stretch factor by the estimated cost. Thus, the stretch factor models optimizer estimation errors. By default, the "stretch" is set to 1 and the estimated cost is the actual number

of simulator time units that the query will consume. That is, the query will be of expected length. However, we also alter the "stretch" to create queries of unexpected lengths: we divide the query's estimated costs by 6 and set its stretch to 6. This technique is used to create *surprise-heavy* and *surprise-hog* queries.

Each query also has minimum and maximum resource requirements. For most queries, these parameters are 0% and 100%, respectively, and the query typically gets an equal resource share. To create *expected-hog* and *surprise-hog* queries, we set the minimum resource requirements to 60%. Other queries running concurrently get less than an equal share.

The simulator lets us model different machine configurations. A machine configuration specifies the number and maximum performance of the resources available for processing the queries.

During the execution of an experiment, the simulator outputs statistics to a management statistics database. The recorded data includes the start and end time of the workload, each query in the workload, and each operator of that query. The simulator also monitors the resources consumed by individual operators, e. g., the number of CPU cycles. In addition, it reports the status of each query, i. e., whether it is queued or running, and its outcome: whether it was rejected, killed, or completed successfully. All of these statistics are made available to the workload management components as they are produced. Since the simulator controls its own clock, writing statistics to the database does not impact the execution of the queries.

4.3.4. Validation against HP Neoview

To check its accuracy, we validated the simulator using HP Neoview as an example for a highly parallel, shared-nothing database. We performed two validation checks, one for query response time and a second for throughput. The validation workload was a subset of a workload used for the actual experiments.

To validate response times, we ran the queries serially on a four node HP Neoview database system and obtained the response time for each query. We then configured the simulator to mimic the database engine of the four node system (four CPUs, four pairs of disks, and the appropriate network bandwidth) and simulated the workload serially (MPL=1).

Figure 4.2 shows the elapsed times of 2130 queries. The x-axis plots their elapsed times when run in isolation (MPL=1) on the HP Neoview database. The y-axis plots their elapsed times in the simulator. A straight diagonal line would show perfect correlation and indeed, most points do fall on a straight line. The points that do not, in the lower left corner, correspond to queries that spend roughly equal amounts of time on disk I/O and in the CPU. On the Neoview system, the disk I/Os overlap substantially with CPU use, due to pipelining of operators. The simulator processes all of the disk-bound operators first, because they are the leaves of the query tree, before it starts the CPU-bound operators. Therefore, these short queries take approximately "twice as long" in the simulator.

To validate throughput, we measured queries processed per hour on Neoview and

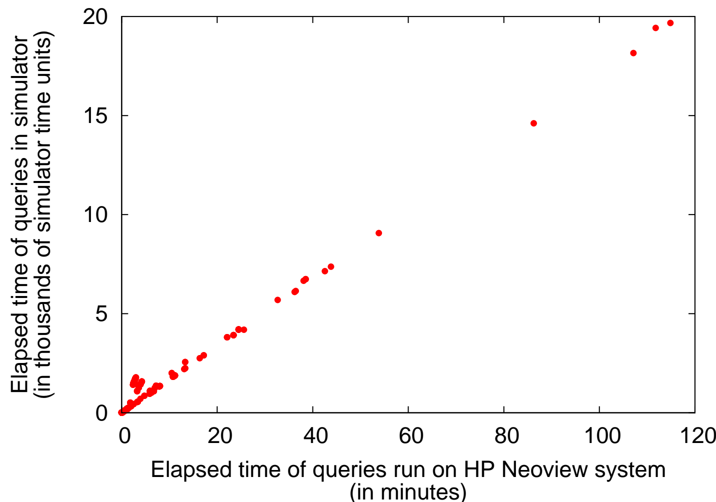


Figure 4.2.: Simulator validation: We compare the elapsed times of queries run on an HP Neoview database with their simulated elapsed times. A straight diagonal line of points would indicate a perfect correlation. Note that 5000 simulator time units corresponds to roughly 30 minutes of elapsed time on the real system.

the simulator as the MPL was increased: 1, 2, 4, and 8. For this test, we created eight different input streams of roughly equal numbers of queries and total duration.

Figure 4.3 shows the throughput for the real and simulated systems. Although the throughput (which is measured in different time units on the two systems) differs, the shapes of both curves are similar, indicating that the simulator does a reasonable job of modeling resource contention on a real system.

4.4. Experimental setup

We describe here the queries and workloads in our experiments, the specific thresholds we chose for the policies, and finally, the objective function we used to measure performance. In the next section, we will present our experimental results.

4.4.1. Queries and query types

Our experiments required a large pool of representative BI queries, including long-running "problem" queries. We started with the Decision Support benchmark TPC-DS [48]. To ensure our queries were CPU-bound, we created the database at scale factor 1. However, all of the queries produced by the TPC-DS templates completed in less than ten minutes on our four-node HP Neoview database system. We therefore created some new templates for the TPC-DS database to generate queries that ran

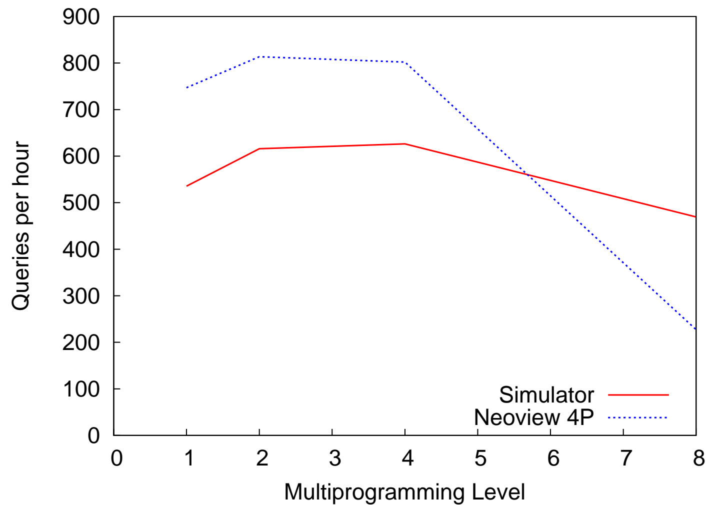


Figure 4.3.: Throughput (queries per hour, QPH) of the same workload when run on the Neoview four processor machine and on the simulator. For the simulator, we derived the QPH using the 30 minutes == 5000 simulator time units formula.

longer (on our system). These templates were based on “problem” queries from a Neoview production enterprise system. Using the combined set of templates, we generated thousands of queries and ran them at MPL=1 to get their query plans and performance statistics, as shown in Step 1 of Figure 4.1.

To characterize the variety of queries in our workloads, we defined three types of queries based on their runtimes. The query types *feather*, *golf ball*, and *bowling ball* roughly categorize the queries according to their costs. Although the boundaries between the different query types are somewhat arbitrary, they suffice to identify the long “problem” queries – the workload management policies should catch the bowling balls. Based on these query types, we created three query pools as shown in Table 4.2.

Most of these queries were CPU-bound. At scale factor 1, some of the TPC-DS database and nearly all of the space needed for sorting and hash tables fit in memory. The longer-running queries are dominated by join, aggregation, and sort operators, which were all CPU-bound. An example bowling ball has a five-way inner join plus a left outer join, a sort, an aggregation, and a nested subquery.

We also created a pool of 34 disk-bound queries. These queries were originally feathers with complex query plans and their CPU time remains unchanged at under 3 minutes. However, we multiplied each query's disk usage by a randomly chosen number that makes the query's total elapsed time fall in the bowling ball range.

query type	size of query pool	queries per workload	elapsed time (hh:mm:ss)		
			mean	min	max
feather	2807	400	30 s	00:00:03	00:02:59
golf ball	247	23	10 min	00:03:00	00:29:39
bowling ball	48	3	1 hr	00:30:04	01:54:50

Table 4.2.: We created pools of candidate queries, categorized by the elapsed time needed to run each query on our 4-node Neoview database system.

4.4.2. Workloads

We created five batch workloads of 426 queries comprising 400 feathers, 23 golf balls, and 3 bowling balls, using random selection without replacement from the three CPU-bound query pools. The elapsed runtime for each workload at MPL=1 is approximately ten hours, and that time is proportioned roughly equally among the three query types.

We then created three variants of each workload using the techniques described in Section 4.3.3. In the Expected-Heavy variant, all queries have stretch of 1 and the bowling balls are *expected-heavy* queries. In the Surprise-Heavy variant, we alter (only) the bowling balls to be *surprise-heavy* queries. Finally, in the Surprise-Hog variant, the bowling balls are altered to be *surprise-hog* queries. The three variants of each workload are otherwise identical: they contain the same 426 queries in the same order. We did not create Expected-Hog variants since their behavior under resource contention should be like that of the Expected-Heavy and Surprise-Hog workloads.

We then created an additional Disk-Heavy variant of each workload. For this variant, we replaced each (CPU-bound) bowling ball with a query selected randomly from the pool of *disk-heavy* queries. Altogether, there were twenty workloads.

4.4.3. Workload management policies

The specific thresholds that will be most appropriate for a given workload depend on the queries in the workload. We now show how we chose the thresholds for the workload management policies in our experiments.

Admission control

Admission control policies must accept or reject queries based on their *estimated* costs. Figure 4.4 shows the expected vs. actual CPU costs in simulator cost units for all of the queries in our workloads, run at MPL=1. Most queries had estimated costs equal to actual costs. However, the *surprise-heavy* and *surprise-hog* queries (the line of triangles) were underestimated by a factor of 6.

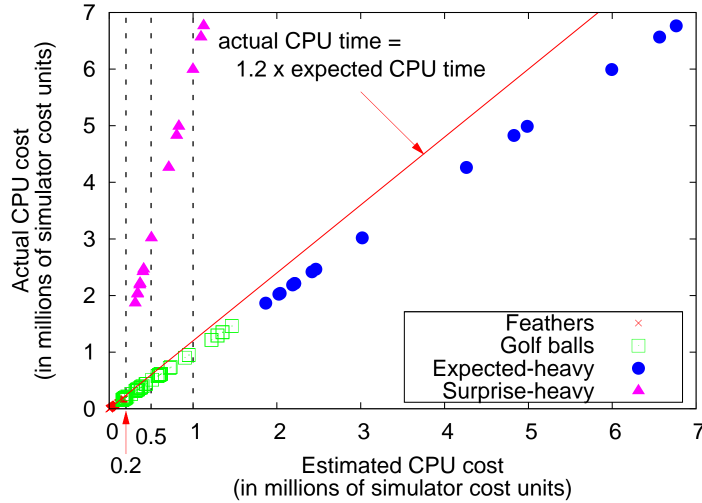


Figure 4.4.: Comparison of estimated and actual CPU time for each query: the CPU time of the *surprise-heavy* queries is underestimated. The dashed vertical lines indicate our admission thresholds and the solid diagonal line shows our kill relative threshold.

We chose four admission control policies for our experiments, *none*, which accepts all queries, and three *reject* policies with different thresholds. These thresholds are shown as vertical dashed lines in Figure 4.4.

Admission control with threshold *1.0m* (one million) simulator time units filters all *expected-heavy* queries but misses most of the *surprise-heavy* queries. It does catch two of the 15 *surprise-heavy* queries but also filters a few golf balls.

Admission threshold *0.5m* filters about half of the *surprise-heavy* queries, while *0.2m* filters all of them. However, the lower the admission threshold, the more golf balls, and even feathers, are rejected.

Scheduling

Scheduling policies control both the MPL of the workload and the number and type of queues used. We first ran the workloads (with no admission control or execution control) at different MPL≥1 to find the "ideal" multiprogramming level. Figure 4.3 shows that the ideal MPL for one simulated workload was 4. For different workloads, the ideal MPL varied between 3 and 5. We chose MPL=4 for most of our experiments since the elapsed time at MPL=4 was within a few percent of optimal for all workloads.

We then studied three scheduling policies. The first policy, *1Q*, uses a single FIFO queue for all queries and enforces MPL=4. When a query completes, *1Q* starts the query at the head of the queue. The other two policies use two FIFO

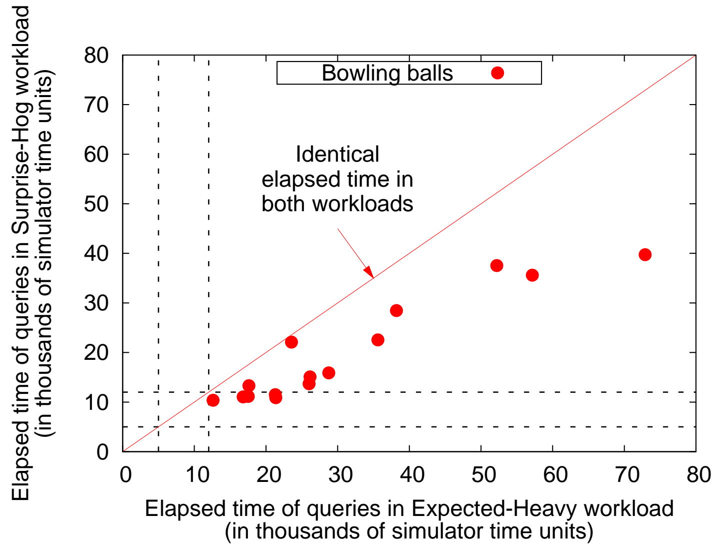


Figure 4.5.: Comparison of elapsed times of long-running queries in Expected-Heavy and Surprise-Hog workloads at MPL=4. All *surprise-hog* queries complete faster than their *expected-heavy* counterparts because they get a larger share of the resources. The dashed lines indicate our absolute kill thresholds.

queues. One queue holds short queries and the other longer queries, according to their CPU cost estimates. We chose the same threshold values of *0.5m* and *0.2m* as for admission control to decide where to enqueue a query (It only makes sense to use a scheduling threshold that is lower than the admission threshold, so, e. g., we only use a scheduling threshold of *0.5m* with an admission threshold of *1.0m* or none.). Both policies process all queries in the lower cost queue first. The policy *2Qs, both MPL 4* then runs the second queue's queries at MPL=4 while the policy *2Qs, different MPLs* runs those queries in isolation.

Execution control

The execution control policies we studied all based their conditions on the actual query CPU time (so far). We chose both absolute thresholds, which take action when a query's CPU time exceeds some fixed threshold and relative thresholds, which take action when query CPU time exceeds some function of its estimated cost. Absolute thresholds are more common because they do not rely on estimates, but relative thresholds are necessary to distinguish expected vs. unexpected runtimes.

To determine the thresholds for our execution control policies, we examined the elapsed times of queries in the Expected-Heavy workloads (when they had an equal share of the resources) and in the Surprise-Hog workloads (when they often did not). Each workload was run with MPL=4.

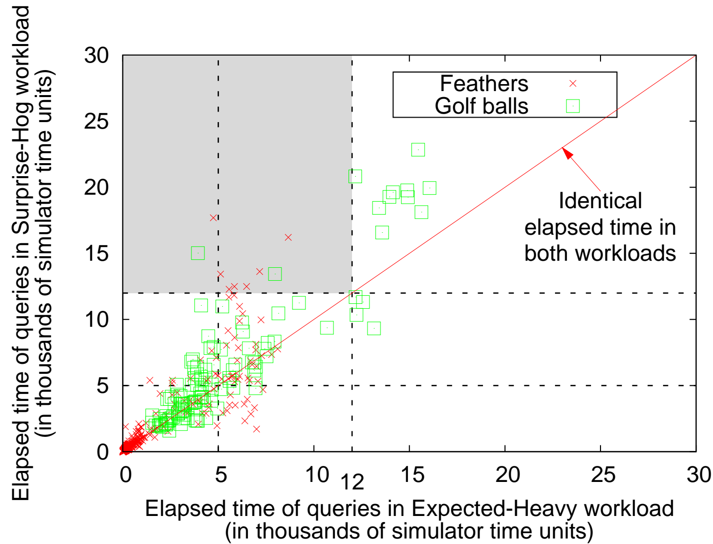


Figure 4.6.: Comparison of elapsed times of feathers and golf balls in Expected-Heavy and Surprise-Hog workloads at MPL=4. Queries above the diagonal run slower in the Surprise-Hog workload. The dashed lines indicate our absolute kill thresholds. The gray shaded area denotes *starving* queries.

Figure 4.5 shows these elapsed times for *expected-heavy* and *surprise-hog* queries and Figure 4.6 shows the times for golf ball and feather queries. We chose two absolute kill thresholds, both shown as dashed lines in Figures 4.5 and 4.6. The kill threshold of 12000 simulator time units catches all *expected-heavy* queries in the Expected-Heavy workloads and only 14 golf ball queries. However, the threshold is only slightly longer than many *expected-heavy* queries, so they are not identified until they have nearly completed (and used a lot of resources). Note that this threshold does not catch some *surprise-hog* queries in the *surprise-hog* workload; they are below the horizontal line in Figure 4.5.

The threshold of 5000 identifies the *expected-heavy* queries sooner, but kills 39 golf balls and 48 feathers. Note that resource contention at MPL=4 causes some feathers to be slower than some golf balls, even though they run faster in isolation.

Figure 4.5 also shows that each *surprise-hog* query, which is given a greater share of resources, completes faster than the corresponding *expected-heavy* query. Figure 4.6 additionally shows the impact of *surprise-hog* queries on the corresponding feather and golf ball queries in the Surprise-Hog and Expected-Heavy variants of the workloads. Any query above the diagonal line runs slower in the Surprise-Hog workload than in the Expected-Heavy workload. This is because the *surprise-hog* queries get a larger share of the resources so other queries running concurrently get a *much* smaller share. Some queries even ran concurrently with two *surprise-hog* queries.

The queries below the diagonal line complete faster in the Surprise-Hog workload. Such queries ran concurrently with a long-running query in the Expected-Heavy

workload but not in the Surprise-Hog workload, where the long queries completed faster. Furthermore, while the golf balls and longer queries use all four CPUs approximately 80% of the time, some feathers use only a single CPU resource. When two or more of these feathers run concurrently, they do not interfere with each other.

We also chose one relative threshold, based on the estimated and actual CPU times of the queries, shown as the diagonal line in Figure 4.4. We chose a very low value of 1.2x (i. e., the actual CPU time exceeds the estimated CPU time by 20%) to see how well a relative threshold can do. Since only *surprise-heavy* queries and *surprise-hog* queries exceed their estimates in our workloads, this threshold catches all and only those queries. In a non-simulated system, the relative threshold should not be set lower than the error typically made by the optimizer.

The *Kill* policies use their threshold to identify and kill queries. These queries do not get re-executed. The *Kill&Requeue* and *Suspend&Resume* policies return killed or suspended queries to a scheduling queue (a separate FIFO queue). When all of the queries in the first queue have finished or been moved to the second queue, we disable the execution control policy so that these queries are not killed a second time. We then run the queries at MPL=1, that is, one at a time. Our scheduling experiments in Section 4.5.2 show the impact of running them at MPL=1 rather than 4: it is negligible. These queries are able to fully use all four CPUs.

4.4.4. Workload objective functions

It is useful to have a single metric to measure performance and compare the effects of different policies. Workload performance is usually measured in terms of either throughput, the number of queries completed per unit time; or latency, the time to complete one or more queries. Makespan is the total latency for a set of queries. We use makespan as the primary objective function for our experiments, since we want to study the performance of whole workloads.

Policies that reject or kill more queries will have shorter makespans that policies that run all of them. However, we did not want policies that reject or kill non-problem queries to appear best. Consequently, we decided to modify the makespan metric to penalize policies for poor decisions. Our metric adjusts the makespan for the fraction of non-problem queries it did not complete: makespan is increased by the approximate amount of time it would have taken to run those queries. We call those queries *penalty queries* since we assess a penalty for not completing them. For our workloads, we define all queries derived from bowling balls as problem queries, and the rest as non-problem or *good* queries. (The term "good query" is derived from the notion of "goodput" in the networking community, which is the portion of throughput that does not include lost or discarded data packets or protocol overhead [3]).

For the modified metric, we first compute T_G ($Time_good$), the sum of the elapsed time of all good (non-problem) queries in the workload at MPL=1. We then compute T_P ($Time_penalty$), the sum of the elapsed time of all penalty queries at MPL=1. Since the penalty queries are a subset of the good queries, the *penalty* (T_P/T_G) is a fraction between 0 and 1, the fraction of useful processing that was not completed.

We then penalize the makespan M as follows: $M_{weighted} = M \cdot (1 + penalty)$.

4.5. Results

Our goal is to evaluate the ability of workload management policies to prevent long-running queries from disrupting the performance of the entire workload. The experiments in this section first evaluate the ability of admission control and scheduling policies to prevent different types of long-running queries from entering the system, then evaluate the ability of execution control policies to catch and handle them at execution time. For each set of experiments, we include a discussion of the lessons learned with regard to the scenarios and objectives described above.

In our experiments, we ran each policy and workload type combination on all five workloads of that type. Since all five workloads yielded comparable results, we present results from only one workload's run per policy/workload type combination. Unless otherwise stated, we used the *1Q* scheduling policy with MPL=4. We present both the makespan and *weighted makespan* for each workload as a stacked bar, where the upper portion indicates the *penalty*. The text *nA* and *nC* on top of the bars indicates the number of admitted and completed long queries (out of the three submitted in each workload). We also consider the makespan for completing 90% and 95% of the queries in the workload, by which we mean the first 90% (95%) to finish.

4.5.1. Admission control

The first set of experiments evaluates the effectiveness of rejecting queries based on their CPU cost estimates prior to execution. We used the admission thresholds *none*, *1.0m*, *0.5m*, and *0.2m* simulator cost units, as described in Section 4.4. We examine the ability of these policies to reject queries that require a lot of resources without also impacting queries with moderate resource requirements. We evaluate their effectiveness with both accurate and inaccurate cost estimates.

Figure 4.7 compares the elapsed times of the Expected-Heavy, Surprise-Heavy, and Surprise-Hog workloads. When no admission control is applied, the makespan and weighted makespan are identical because no queries are rejected. The makespans for the Expected-Heavy and Surprise-Heavy workloads without admission control are identical. Both workloads contain the same set of queries with identical runtime behavior. The Surprise-Hog workload runs slightly longer because the *surprise-hog* queries in the workload hog the resources and thus prevent other queries from making significant progress.

The admission threshold of *1.0m* rejects all three *expected-heavy* queries and three golf balls in the Expected-Heavy workload. Not admitting these six queries reduces the weighted makespan of the workload by about 33%, despite the penalty for not performing the golf balls. The reason for the significant drop of the weighted makespan is rejecting the *expected-heavy* queries. Stricter admission thresholds result in marginally decreased weighted makespans. The threshold of *0.5m* rejects another

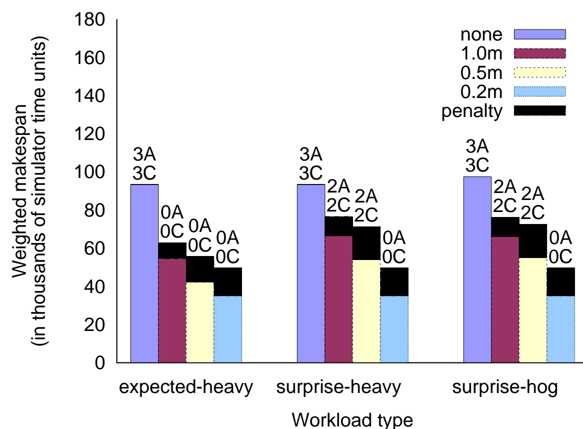


Figure 4.7.: Comparison of admission thresholds on different queries and workload types. The notations nA and nC above the bars indicate the number of admitted and completed long-running queries (out of the three submitted in each workload). Admission control is less effective when cost estimates are less accurate. Lower thresholds reject more "good" queries.

six golf balls, reducing the weighted makespan by another 13% (63k vs. 55k simulator time units). However, the penalty increases by 62% (8k vs. 13k simulator time units). Setting the threshold to *0.2m* rejects another seven golf balls and further increases the penalty.

The Surprise-Heavy workload demonstrates that even with the penalty for rejected "good" queries, a lower threshold that catches more long queries may be better. The admission threshold *1.0m* rejected only one of the *surprise-heavy* queries and three golf balls. At threshold *0.5m*, admission control rejects another six golf balls, but no additional *surprise-heavy* queries. The weighted makespan decreases significantly with threshold *0.2m*, which rejects all three *surprise-heavy* queries. The results for the Surprise-Hog workload are similar.

Figure 4.8 demonstrates the (in)effectiveness of CPU-based admission thresholds on long-running *disk-heavy* queries. The CPU-based admission control rejected only golf balls, i. e., it was completely ineffective at identifying long-running queries. Although the makespan decreases with stricter admission control, the weighted makespan remains constant.

Lesson: Unreliable cost estimates: Admission control is effective at reducing the workload makespan when resource cost estimates are accurate by preventing execution of the queries most likely to cause contention. However, when costs are underestimated, admission thresholds that can catch the long-running queries also reject many "good" queries.

Lesson: Unobserved resource contention: Admission thresholds are not effective against long-running queries that make heavy use of resources not measured by

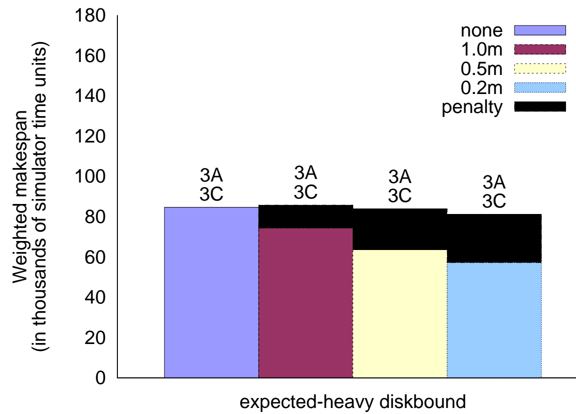


Figure 4.8.: Admission thresholds are much less effective when the workload includes long-running queries that make heavy use of resources not measured by the admission threshold. In this case, the queries were disk-bound but admission control looked at CPU time estimates.

the admission threshold. Therefore, workloads that contain a wide diversity of query types may need multiple policies with conditions on different resources.

4.5.2. Scheduling

The scheduling experiments evaluate the impact of the scheduling policies *1Q*, *2Qs both MPL 4*, and *2Qs different MPL* on the performance of the Expected-Heavy and Surprise-Heavy workloads (The results for the Surprise-Heavy and Surprise-Hog workloads are very similar.). We set the threshold for scheduling the queries in the expensive query queue to *0.5m* simulator cost units.

Our experiments show that regardless of admission control policy, both the makespans and the weighted makespans of the workloads vary by less than 1% across the different scheduling policies.

However, we observed a significant difference between policies when we looked at the makespans for a given percentage of completed queries. This is because the *2Qs* policy is similar to *shortest-job-first* (SJF), which is known to improve latency for short jobs [15].

Table 4.3 summarizes the time to complete 90%, 95%, and 99% of the queries in the Expected-Heavy and Surprise-Heavy workloads with admission threshold *none*. The *1Q* policy takes about twice as long to complete 90% and 95% of the queries as the *2Qs* policies. This result is not surprising: the "expensive" queue contains all of the long-running queries, which comprise about 35% of the total CPU time, plus nine of the golf balls. Removing them from the initial workload (by putting them in a separate queue) automatically makes it least 35% shorter. In addition, the shorter queries have less contention for resources, so they complete faster.

% of queries complete	*1Q*	*2Qs* both *MPL=4*	*2Qs* *different MPL*
Expected-Heavy, admission threshold *none*			
90	90.2	39.7	39.7
95	91.1	40.6	40.6
99	91.8	87.7	89.2
100 ("all")	93.4	100.5	100.9
Surprise-Heavy, admission threshold *none*			
90	90.2	50.9	50.9
95	91.1	52.0	52.0
99	91.8	81.9	87.6
100 ("all")	98.3	98.3	99.2

Table 4.3.: Time to complete a certain percentage of queries (in thousands of simulator time units) when trying to put expensive queries in a separate queue.

All three scheduling policies complete 99% of the queries in about the same amount of time. There is little difference between the *2Qs* policies: the queries in the expensive queue are able to use nearly all of the CPU for their entire duration, so saving that little idle time (by running them at MPL=4 instead of MPL=1) is not worth the extra overhead of running additional concurrent queries.

In contrast to the Expected-Heavy workload, it takes longer to complete 90%, 95%, and 99% of the queries in the Surprise-Heavy workload. Due to the cost estimate errors, the *2Qs* scheduling policy cannot identify the *surprise-heavy* queries and places them into the queue with the short queries. When the long-running queries are executed in parallel with short queries, the short queries take longer to complete.

Lesson: Minimizing makespan: Different scheduling policies have little effect on the total makespan of the workloads. Therefore, scheduling is not important for most batch workloads.

Lesson: Minimizing response time: However, because of the benefits of *shortest-job-first*, scheduling policies can have a significant positive impact on the latency of individual shorter queries.

Lesson: Unreliable cost estimates: Because scheduling does not reject or kill queries, it does not incur penalties for misidentifying queries; all queries eventually run. A *2Qs* policy thus could complement a lenient admission threshold. However, the benefits of the *2Qs* policy diminish with decreased accuracy of cost estimates, as more expensive queries are placed in the wrong queue.

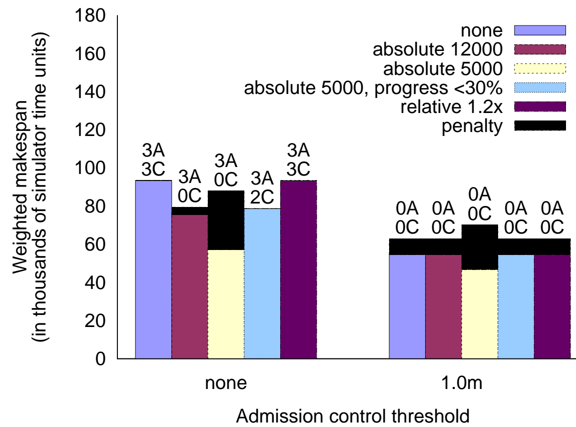


Figure 4.9.: Comparison of absolute and relative *kill* thresholds in the Expected-Heavy workload: The lower absolute threshold kills many more queries unless their progress is checked.

4.5.3. Execution control: kill thresholds

Admission control and scheduling policies are less effective when cost estimates are inaccurate. Execution control policies, on the other hand, look at runtime statistics to catch problem queries. In the following set of experiments, we compare the effectiveness of different execution control policies in identifying and handling long-running queries.

Figure 4.9 compares the makespan of the Expected-Heavy workload using different combinations of admission and execution control policies. With no admission control (*none*), the *absolute 12000* kill threshold kills the three *expected-heavy* queries and one golf ball when their elapsed time exceeds 12000 simulator time units. However, these queries have done most of their work by that time, so the makespan only decreases by about 15% (with negligible penalty for killing the one query).

The *absolute 5000* kill threshold kills the long-running queries much earlier, but also kills an additional eight golf balls and eleven feathers, yielding a weighted makespan that is 13% higher than the weighted makespan with the *absolute 12000* threshold. The *absolute 5000, progress<30%* threshold checks the progress of these queries before killing them. It only kills one (*expected-heavy*) query. No queries were killed using the relative threshold because estimated and actual CPU times are identical for *expected-heavy* queries.

With admission control set to *1.0m*, all of the *expected-heavy* (and 3 golf ball) queries in the *expected-heavy* workload are rejected. Therefore, no queries are killed except using the *absolute 5000* threshold, which kills 15 golf balls, resulting in a penalty that is 50% of the makespan. The Surprise-Heavy workload results (results not shown in the figures) follow from the Expected-Heavy workload, as well as the lessons learned from admission control: absolute kill thresholds are not impacted by

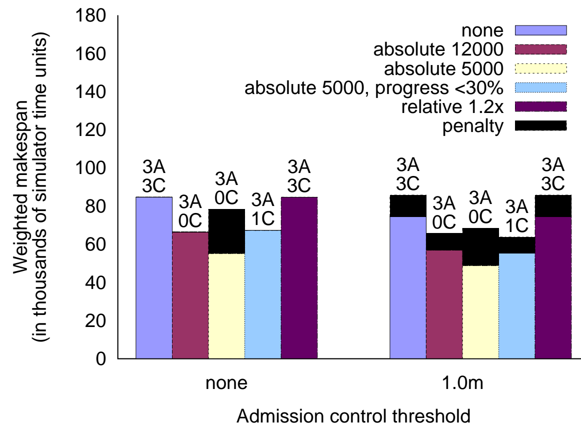


Figure 4.10.: Comparison of absolute and relative *kill* thresholds in the Disk-Heavy workload: the relative threshold compares actual and estimated CPU time and thus does not catch the *disk-heavy* queries.

estimates, and the effectiveness of progress thresholds depend on the accuracy of the estimates.

Figure 4.10 shows the performance of execution thresholds on the Disk-Heavy workload. Execution control performance with absolute elapsed time thresholds is similar to that for the Expected-Heavy workload. One noticeable difference is that fewer feathers and golf balls are killed in the Disk-Heavy workload, indicating that the *disk-heavy* queries contend less with the CPU-bound feathers and golf balls than the *expected-heavy* queries in the Expected-Heavy workload do. As expected, the relative threshold that compares the actual and estimated CPU times of a query does not kill any queries.

Lesson: Unreliable cost estimates: Execution control policies can detect and kill queries missed by admission control and scheduling, and are thus particularly useful for catching queries whose resource cost estimates are inaccurate. The two most effective policies for catching (only) queries that run unexpectedly long in our experiments were (1) a relative kill threshold and (2) a low absolute threshold combined with a progress check to let nearly-done queries finish.

Lesson: Unobserved resource contention: The longer a query runs before it is killed (the higher the kill threshold), the more work is "wasted" and the more it impedes other queries. However, the lower the threshold, the more "false positive" short queries are killed. Therefore, absolute thresholds may not work when contention or system overload can affect the measured values. Stopping a starving query and admitting another query will not improve system performance. A problem query might be using heavily a resource for which no cost estimate is available.

4.5.4. Execution control: different actions

These experiments compare the different execution control policies *Kill*, *Kill&Requeue*, and *Suspend&Resume*. The latter two policies complete killed or suspended queries at the end of the workload, i. e., they always complete all admitted queries. We note that we modeled the actions *Kill* and *Suspend&Resume* in the most charitable way possible. Killed and suspended queries immediately give up all resources. Also we did not model restart costs for resuming a suspended query.

Although we ran experiments with all of the admission control policies, we present the results for admission threshold *1.0m*; with the lower admission thresholds, so many queries are rejected that there is little to kill or suspend. The results for admission control *none* are similar to these results, but do not show a distinction between the Expected-Heavy and Surprise-Heavy workloads. We only show two of the kill thresholds from the previous section.

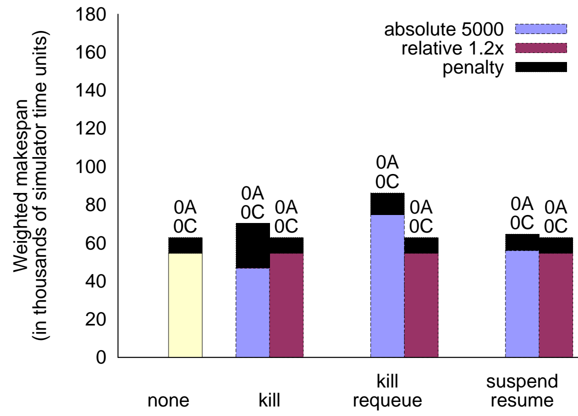
Figure 4.11(a) shows the makespans for the Expected-Heavy workload. The results for the execution policies *none* and *kill* are repeated from Section 4.5.3. While admission control rejects six queries (including all *expected-heavy* queries), the *absolute 5000* threshold catches an additional 15 queries and kills or suspends them. (Since the resource cost estimates are accurate for all queries, the relative threshold does not flag any queries.) When those 15 killed queries are rerun, the wasted 15×5000 time units of work must be repeated and so the total makespan is longer. However, when they are suspended and then resumed, the time is not wasted and the makespan is only 3% longer (because the *expected-heavy* queries ran in parallel with the rest of the workload for some time) than with no execution policy. An interesting observation is that the weighted makespan for *Suspend&Resume* is lower than the weighted makespan for *kill*. The former action has a lower penalty because the golf balls and feathers suspended are resumed at a later point in time.

Figure 4.11(b) shows similar results for the Surprise-Heavy workload. The makespans are slightly longer compared to those in Figure 4.11(a), since admission control misses the two *surprise-heavy* queries, but the kill and suspend thresholds catch them. There is therefore a slightly higher performance gain from the execution control policies than with the Expected-Heavy workload.

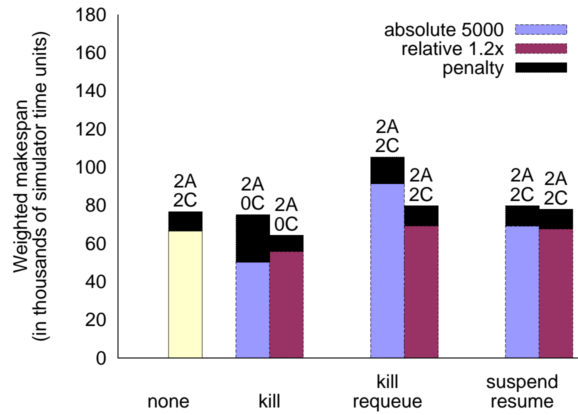
Table 4.4 also shows that by killing or suspending the longer queries, the makespan of the first 90% and 95% is greatly reduced. Note that the table does not report results for the 99%-case for the Expected-Heavy workload because admission control filters more than 1% of the workload. The makespan results are similar to those for scheduling longer queries to run later. However, by identifying the longer queries with an execution control policy, it is possible to catch the *unexpectedly* long-running queries.

Lesson: Minimizing makespan: *Kill* has more impact on makespan than other execution actions and should be the preferred action if it is acceptable not to complete all queries.

Lesson: Unreliable cost estimates: Since *Kill&Requeue* and *Suspend&Resume* policies identify and postpone long-running queries, they complete the less expensive



(a) Expected-Heavy workload: Rerunning the killed queries takes longer than if they were never killed, while suspending and resuming them does not.



(b) Surprise-Heavy workload: Killing queries that need to be run later increases the makespan compared to not killing them.

Figure 4.11.: Comparison of execution control policies with admission threshold *1.0m*.

% of queries complete	*none*	*Kill& Requeue*	*Suspend& Resume*
Expected-Heavy, admission control threshold 1.0m			
90%	51.0	44.5	44.7
95%	52.2	46.8	47.0
99%	—	—	—
Surprise-Heavy, admission control threshold 1.0m			
90%	63.2	47.7	47.7
95%	64.1	50.2	50.2
99%	66.5	91.4	69.1

Table 4.4.: Time (in thousands of simulator time units) to complete a certain percentage of queries.

queries first. Particularly when optimizer estimates are poor, they can be considered a kind of self-correcting shortest-job-first.

Lesson: Suspend&Resume: *Suspend&Resume* completes all queries significantly faster than *Kill&Requeue* with an absolute threshold (because it does not waste the work done by a query before execution control flags it). However, *Kill&Requeue* with a relative threshold is just as good (because it flags the unexpectedly long queries before they have done much work).

Lesson: Minimizing makespan: If the only metric of interest is makespan for all queries, e. g., for some batch workloads, then an execution control policy of *none* is the most effective of all.

4.5.5. Execution control: overload situations

The final set of experiments evaluates execution control policies in *overload* situations. Overload occurs when the actual MPL is significantly higher than the ideal MPL, either because scheduling does not constrain the MPL, the MPL is set to an appropriately high value, or because there are too many queries that bypass scheduling.

Figure 4.12 compares the impact of different kill thresholds on the Surprise-Heavy workload at MPL=4 and MPL=10 with the admission threshold set to *1.0m*. Admission control rejects no *surprise-heavy* queries. Since all queries take longer in the overload case, the policies with absolute thresholds kill more queries and have greater performance gains but also greater penalties. For example, the *absolute 12000* kill threshold improves makespan by 40% at MPL=10 compared to only 6% at MPL=4. However, it kills an additional 17 *starving* queries and has a much higher penalty

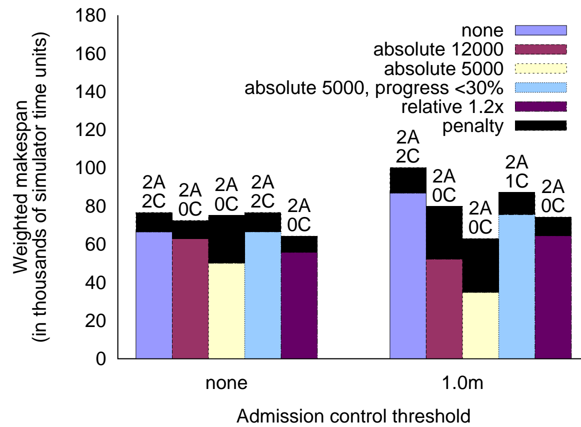


Figure 4.12.: Comparison of execution control policies using admission threshold *1.0m* at MPL=4 and MPL=10 (overload) for the Surprise-Heavy workload. All queries take longer at MPL=10, so policies with absolute thresholds kill more queries.

value. In contrast, the relative threshold and the absolute threshold with the check on progress kill far fewer queries (2 and 1, respectively, compared to 37 with *absolute 5000*). The lower number of killed queries almost makes up for the higher makespan of the query.

Lesson: System overload: Execution control policies are particularly ineffective in overload situations. They are more effective at catching long-running queries and reducing makespan, but also more likely to kill starving queries.

4.6. Conclusions

This chapter presented a systematic study of workload management policies that mitigate the impact of long-running queries on performance. We proposed a taxonomy that distinguishes between different types of long-running queries. We suggested a method for categorizing queries according to this taxonomy, using only cost estimates and simple runtime statistics. We then carried out a systematic series of experiments to investigate the effectiveness of known workload management policies on these different types of queries. We recommended particular combinations of policies for meeting several common workload objectives.

Admission control and scheduling policies that apply absolute thresholds to cost estimates can either prevent long-running queries from starting in the first place or postpone them to run later. When cost estimates are inaccurate, these policies can mistake good queries for problem queries and vice versa. However, execution control policies can correct for errors in admission control and scheduling. We find that when cost estimates are significantly off, the execution control actions *Kill&Requeue*

and *Suspend&Resume* function as a self-correcting *shortest-job-first* (SJF) and can effectively reduce the latency of individual queries. In addition, our experiments show that when using a relative threshold, *Kill&Requeue* performs as well as the presumably more expensive *Suspend&Resume* in terms of makespan.

When system overload occurs or when the measured resource is not the source of contention, thresholds that use the ratio of estimated to absolute values as a measure of query progress can distinguish between queries that are truly heavy users of resources and those that are starving. However, the disadvantage of relative thresholds is that they take longer to take effect, resulting in more "wasted work." We therefore recommend that policies be paired to compensate for the strengths and vulnerabilities of their underlying thresholds. For example, a less aggressive policy that uses cost estimates can be paired with a more aggressive policy that looks at runtime conditions. The optimal values for the aggressive and less-aggressive thresholds depends on the expected variance of the key metrics used in the workload.

5. Policy control for mixed workloads

Database systems provide measurements for different metrics to describe performance, e. g., average response time, throughput, and velocity. User loads submitted to the database system expect a certain performance. In order to quantify “acceptable” performance, the users who submit the loads formulate objective functions based on the performance metrics. For example, users may define an upper bound for the average response time and a lower bound for throughput for an OLTP-style user load, whereas users that submit reports would like to keep the execution time of the report below a deadline. At the database system level, the user loads are mapped to service classes that provide control parameters that control the processing of the respective user loads and, thus, indirectly affect performance. Examples for control parameters include the number of concurrently executed queries (multi-programming level, MPL) and estimated main memory consumption. For ease of presentation, we assume that there is a one-to-one mapping between user loads and service classes: all requests from a user load are mapped to the same service class and there are no requests stemming from different user loads mapped to the same service class. Note that the approach presented here can be easily generalized to support arbitrary mappings between user loads and service classes.

With a single service class, an objective that maximizes some performance metric is reasonable. However, with mixed workloads and service classes, it is not possible for every class to maximize some metric, which may result in severe overload. So the model in this chapter is pay-for-service, i. e., the objectives are constrained within some range so that users who are willing to pay more get better service. In other words, the goal is to set the control parameters to meet the objective rather than to “maximize” system performance. If the system is over-engineered (i. e., with excess capacity), it may be possible to meet the objective with an underutilized system. However, if the objective cannot be met when the system is saturated, pushing the system beyond its capacity will not result in improved performance.

For an example, let MPL be a control parameter and throughput be a metric. Assume two service classes, s_1 and s_2. For each of the service classes, the objective function is to keep throughput above a threshold. Consider the system is in a steady state but is not meeting the objectives. We will assume the system resources are underutilized because otherwise, there is nothing workload management can do to meet the objectives. Let m_1 and m_2 represent the MPL values for s_1 and s_2, respectively. As we increase the MPL for s_1 and/or s_2, the system moves from an underutilized state to a saturated state where resources are fully utilized and throughput is maximized. Increasing MPL beyond this point to an overload state will not increase resource utilization and, thus, will not increase throughput. In fact, throughput may

decrease due to resource contention. So, in this example, the objective function will maximize resource utilization and the algorithm to set the control parameters should avoid an overload state.

While it is relatively straightforward to identify the type of the control parameters for a given service class, it is more complicated to find values for the control parameters that result in a performance that meets the objectives. A database administrator assigned to control a workload on the database faces several difficulties: First, the workload may contain service classes with vastly different characteristics and different objectives. Second, although the administrator knows which service classes will be running, the information about the service classes and queries may be incomplete or inaccurate. For example, there may be no or little information about the time period a service class starts or ends, the resources that are used by a query, or when to expect a particular query. Additionally, it is not known prior to the execution of a set of queries how these queries will contend with one another for hardware (e. g., CPU, main memory, disk) and software (e. g., locks) resources. Third, there are non-linear relationships between the objectives and the control parameters. Changing the setting for a control parameter by a certain amount results in improved performance for some service classes and hurts the performance of others. However it is difficult to quantify the performance impact of the change. Also, changing the setting by the same amount does not necessarily lead to the same (absolute) change in performance. Fourth, the size of the search space is exploding with the number of control parameters to be considered. For our approach, we assume the control parameters to be discrete. Let $D_i, 1 \leq i \leq n$ denote the domain of the control parameter that controls the processing of the ith service class. For ease of presentation, we assume just one parameter per service class for this example. The domain of a control parameter represents the range of "reasonable" parameter settings. For example, although the MPL can take any non-negative integer value, it is not reasonable to consider MPLs in the thousands. Note that the bounds of the interval are not known prior to the execution. Thus, the size of the search space that contains an acceptable setting is $|D_1| \times \cdots \times |D_n|$. For three service classes with MPL control parameter values in the interval $[1, 32]$, there are $32^3 = 32768$ control parameter settings to be considered. Additionally, a probe in the search space is time-consuming and does not reveal information about where to search next.

5.1. Related work

We find that no single prior approach meets our goal of setting multiple control parameters to satisfy compound performance objectives for concurrently executing service classes. Below, we categorize prior approaches at query scheduling according to the control parameters, types of service classes, and objectives they support.

5.1.1. Single control parameter

Thiele *et al.* [58] address the problem of how to interleave the single-stream execution of a mixed workload composed of continuously arriving update and read-only queries in the context of a real-time data warehouse. They consider a single service class that has two objectives based on a single quality of service (QoS) metric (such as throughput, average response time, or stretch) and on a single quality of data (QoD) metric (lag-based, divergence-based, time-differential). They map this search to a knapsack problem, which they solve via a dynamic programming algorithm.

Abouzour *et al.* [1] show that Hill Climbing and Global Parabola approximation algorithms, as well as a hybrid algorithm that combines the two, can be used to select the MPL for a workload comprised of queries submitted in sequence from a series of service classes. The selected MPL applies to all queries running in the system, regardless of type. All of these algorithms work to optimize throughput, with a secondary goal to minimize MPL.

Schroeder *et al.* [53] present a framework for meeting QoS objectives where response time requirements are specified in an SLA. They use a feedback control loop to dynamically adjust the MPL to meet multi-class response time goals.

5.1.2. Multiple control parameters, single performance objective

Powell *et al.* [50] consider the problem of multiple service classes and multiple control parameters. Performance is controlled by throttling and slowing down queries via the amount of resources allocated to a service class. They propose a variety of controllers that decide how much a service class must be throttled to achieve the goals of the "important" service classes, including a simple controller, a black-box model controller, and a hybrid controller (black-box model for initial setting and simple controller for fine-tuning).

Pang *et al.* [49] schedule queries from both single class and multi-class workloads in real-time database systems so as to meet a single performance objective — minimizing the number of missed deadlines across service classes. Their approach dynamically adapts the system's admission, memory allocation, and priority assignment policies so as to minimize number of missed deadlines and to distribute deadline misses across service classes according to a pre-defined distribution (e. g., service class 1 is allowed three times as many deadline misses as service class 2).

5.1.3. Workload adaptation — maximize single objective

Of prior approaches, the workload adaptation approach [46] comes closest to our own goals. Workload adaptation defines a function that quantifies the *utility* $u_i(m)$ of a single performance measurement m towards meeting the objectives of service class s_i. The objectives are formulated relative to two thresholds: a threshold *goal* to define a performance goal and a threshold *worst* to define the "worst allowed"

performance:

$$u_i(m) = \begin{cases} > 0, & \text{if } m \text{ exceeds objective } \textit{goal} \text{ (and thus exceeds } \textit{worst}) \\ \leq 0, & \text{if } m \text{ violates } \textit{goal} \text{ and } m \text{ exceeds } \textit{worst} \\ -\infty, & \text{if } m \text{ violates } \textit{worst} \text{ (and thus violates } \textit{goal}) \end{cases}$$

Since we cannot directly control the performance measurements and, thus, the utility of a service class, we need a function to map control parameter settings to an estimate for the performance measurements at that setting. Niu [46] proposes a performance model that estimates how changes to the control parameter setting of a service class will impact performance of the service class, based on the knowledge of the last measured performance and the respective control parameter setting. In a nutshell, the performance model proposes a function for each performance metric that maps a control parameter value to a performance value estimate, i. e., the estimate is a function of the given control parameter value. For example, to estimate throughput tp_i^k at time k for service class s_i, for a given control parameter value c_i^k (e. g., the MPL for that service class), the model takes the previously measured throughput tp_i^{k-1} and scales it by the ratio of c_i^k divided by the previous control parameter value c_i^{k-1}. Thus, $tp_i^k = tp_i^{k-1} \cdot c_i^k / c_i^{k-1}$. Similarly, to estimate average response time at time k, the model takes the previously measured average response time, art_i^{k-1} and scales it by the ratio of the previous control parameter setting c_i^{k-1} divided by the proposed c_i^k: $art_i^k = art_i^{k-1} \cdot c_i^{k-1} / c_i^k$. Using the performance model, the (estimated) utility u_i can be expressed as a function of the control parameter. The overall utility of the system is expressed as an aggregate $f(u_1, \dots, u_n)$ of n concurrently running service classes, e. g., the sum of the utilities $u_1 + \cdots + u_n$.

[46] formulates the problem of finding the appropriate control parameter values as an optimization problem. The goal is to find control parameter values $c_1^k, \dots, c_n^k$ that maximize the expected overall utility:

$$\begin{aligned} \underset{c_1^k, \dots, c_n^k}{\textit{maximize}} \quad & u_1^k(c_1^k) + \cdots + u_n^k(c_n^k) \\ \text{subject to} \quad & c_1^k + \cdots + c_n^k = C \end{aligned}$$

where C is the total limit for the control parameter values that must be determined a priori and offline.

5.2. Problem statement

Users who submit queries to a database system formulate objective functions based on one or more performance metrics to express the desired quality of service. At the database system level, the queries stemming from users are mapped to *service classes* that define the level of service a query gets. Our work focuses on how to schedule queries, i. e., how to decide when to run which query, in an environment with multiple service classes so that the service level objectives are met. A service class

provides different types of parameters that control the number of concurrently active queries in the database system at any time. These control parameters indirectly affect the performance of the queries belonging to the service class. For our work, we assume that there is a single control parameter per service class to control the execution. Although we only assume a single control parameter per service class for ease of presentation, we note that the approach can be generalized to multiple control parameters per service class.

The measurements of a single performance metric of a service class can be illustrated by an $n + 1$-dimensional graph, n dimensions for the n control parameters and an additional dimension for the performance metric. The graph shows all measurements for all possible control parameter combinations. The hull formed by the control parameter-performance measurements is either convex or non-convex. A convex control parameter-performance hull can be observed when queries running concurrently only compete for resources, i. e., the queries do not benefit from running concurrently to other queries. For example, this behavior may be observed for OLTP requests where benefit from caching is not expected. A non-convex control parameter-performance hull may be observed for service classes where synergies may result in better performance for a service class even if more queries from another service class are admitted to the system. Sources of such synergies stem from caching behavior and cooperative approaches like reusing common sub-expressions in queries [18, 56] and shared scans [63].

To illustrate a convex control parameter-performance hull, we used TPC-CH [22], a new benchmark that combines TPC-C and TPC-H into a single benchmark. We note that the TPC-C-like transactions and the TPC-H-like queries access the tables on the same database instance. TPC-CH is basically the TPC-C schema extended with TPC-H tables *supplier*, *nation*, and *region* to support TPC-H-like queries (see Appendix B). We modeled two service classes s_C and s_H so that service class s_C comprised 32 TPC-C clients and service class s_H comprised 16 TPC-H clients. The clients submitted the requests to the database in random order with no wait times. For TPC-H we chose queries Q13, Q17, and Q19. We chose these queries to have a homogeneous query set, i. e., a set of queries with similar execution times when run in isolation. For measuring the response times and throughput of the requests, we used the iJDBC implementation described in Section 3.6. Although our algorithm would work with service classes with greater variance in execution times, greater variance in time requires longer observation periods in order to determine metrics such as average response times and average throughput, which would have caused our experiments to have taken longer to run. For this example, we ran multiple benchmarks based on a database with scale factor 24 (raw data size about 1.6GB) on a commercial DBMS (DB2 9.5). The average response time of the NewOrder transaction aggregated over all TPC-C clients (service class s_C) was about 100ms when run in isolation (TPC-C-only) and about 400ms when all TPC-C and TPC-H clients submitted their requests in parallel and the requests were immediately released from the queue. The average response time of service class s_H was about 2 seconds in isolation (TPC-H-only) and about 40 seconds when all clients were submitting queries in parallel. We separately

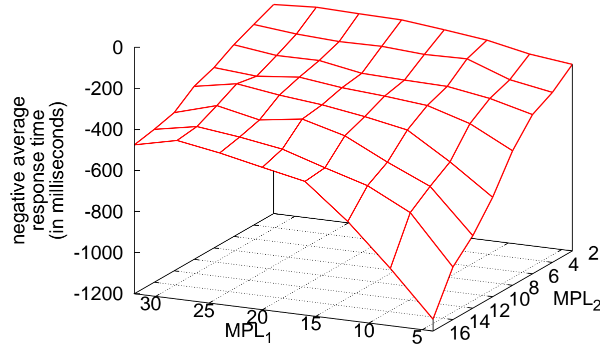


Figure 5.1.: Control parameter-negative average response time hull for service class s_C when executed in parallel with service class s_H at different $MPL_1 - MPL_2$-settings.

limited the number of concurrently active queries for each service class, i. e., there were two control parameters MPL_1 and MPL_2. We varied the MPL for the TPC-C (MPL_1) and MPL for the TPC-H queries (MPL_2) separately: $MPL_1 = 4, 8, 12, ..., 32$ and $MPL_2 = 2, 4, 6, ..., 16$, i. e., 64 MPL_1-MPL_2-combinations. For each setting, we executed the benchmark for 20 minutes and we measured the throughput and average response time for both service classes in the last 15 minutes, i. e., there was a 5 minute grace period that allows the system to return to stable state. Figure 5.1 shows the average response time-hull for the NewOrder transaction of service class s_C with different MPL_1-MPL_2-settings.

Consider a workload with n service classes $s_1, \dots, s_n$ where the performance of service class s_i is measured with a set M_i of performance metrics. In order to specify the desired performance for service class s_i, there is a lower (lb^i_m) and an upper (ub^i_m) bound for each metric $m \in M_i$. For example, we can measure the performance of a service class with throughput and define two objectives: throughput must exceed a minimum value and must not exceed a maximum value. Let $X = \{x_1, \dots, x_n\}$ denote a tuple of control parameter values where x_i denotes the control parameter value for service class s_i. Let $p^i_m(X)$ denote the performance based at metric m measured for s_i with control parameter settings X (e. g., p^i_{tp} denotes a throughput measurement). For ease of presentation, we assume that a higher value for p_m represents better performance. For example, rather than use response time as a metric, the inverse of the response time might be used. As a consequence, the objective requires the measurement to be in the interval $\left[lb^i_m, ub^i_m\right]$. An acceptable m-region A^i_m for service class s_i describes the set of tuples $X = \{x_1, \dots, x_n\}$ where the measurement p_m of

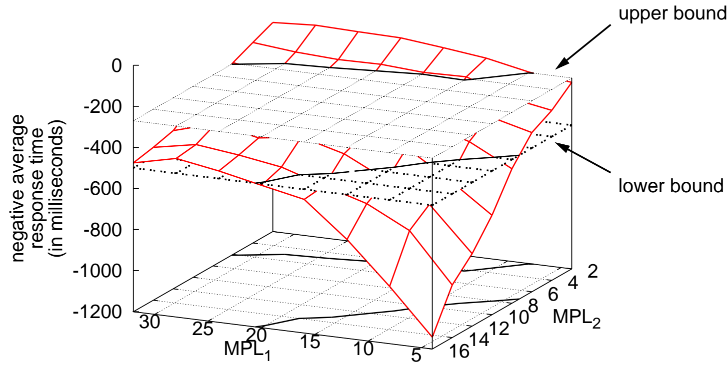


Figure 5.2.: Control parameter-negative average response time hull (from Figure 5.1) for service class s_C with lower and upper bounds. The resulting acceptable region is shown on the $MPL_1 - MPL_2$-plane.

performance metric m satisfies all objectives:

$$A_m^i = \left\{X = \{x_1, \ldots, x_n\} \mid lb_m^i \leq p_m^i(X) \leq ub_m^i\right\}$$

Note that an acceptable m-region may not exist, i. e., is empty, but if it exists, it is unique because the hull is convex. The acceptable region A^i for a service class s_i is defined as the overlap of all acceptable regions of service class s_i with A^i, i. e., $A^i = \bigcap_{m \in M_i} A_m^i$.

For example, consider a service class with two performance metrics, throughput (tp) and negative average response time ($nart$). Note that the inverse of the average response time takes negative values (the negative value of the average response time) because because we wanted higher performance values to describe "better" performance. There is one objective with a lower and an upper bound for throughput ($1000^{requests}/_{second} \leq tp$ and $tp \leq 2000^{requests}/_{second}$) and one objective with a lower bound for the inverse of average response time ($-200ms \leq nart$). The acceptable throughput-region A_{tp} for the service class denotes all control parameter settings where the throughput measured for the service class exceeds 1000 requests per second and does not exceed 2000 requests per second. Similarly, the acceptable average response time-region A_{nart} ($200ms$) describes the control parameter settings where the inverse of the average response time exceeds -200 milliseconds for the same service class.

An acceptable region can be illustrated with an n-dimensional plot, where n is the number of control parameters. The shape of the acceptable region is determined by the shape of the control parameter-performance hull. If the control parameter-performance measurements form a convex hull, the resulting acceptable region is also convex. For example, the lines in Figure 5.2 indicate the acceptable average

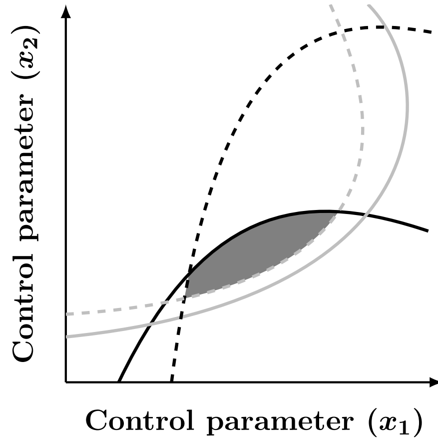


Figure 5.3.: Boundaries of the acceptable throughput- (solid) and average response time-regions (dashed) for classes s_1 (black) and s_2 (gray). The shaded area denotes the operating envelope.

response time region on the $MPL_1 - MPL_2$-plane for the NewOrder transaction in service class s_C. The planes parallel to the $MPL_1 - MPL_2$-plane indicate the upper and lower bounds, respectively, of the (inverse) average response time.

Two main factors influence the shape of the acceptable region. First, acceptable regions depend on the designated objectives. The acceptable region for less restrictive objectives contains the acceptable regions for more restrictive constraints formulated on the same metric. Second, the acceptable regions depend on the resources that are available to process a service class. Resource availability depends on (1) the hardware in the machine and (2) the number of concurrently active service classes competing for resources, and (3) the policies in effect that allocate resources to service classes.

From the workload management point of view, we are interested in the control parameter settings that satisfy all objectives of all service classes. Any control parameter setting that is located in the overlap of the acceptable regions of all service classes, satisfies this requirement. We define the *operating envelope* E for a workload consisting of the service classes $s_1, \ldots, s_n$ as the set of control parameter settings where all constraints of the service classes are met: $E = \bigcap_{i=1}^{n} A^i$

The operating envelope is empty if the objectives are formulated such that no thresholds can be found to satisfy the objectives. Figure 5.3 shows the operating envelope for a workload with two service classes s_1 and s_2, each having a negative average response time and a throughput objective. Note that the operating envelope is restricted by the acceptable average response time-region of s_1 and the acceptable throughput-regions of both service classes. The average response time of s_2 is always satisfied when the throughput objective is met.

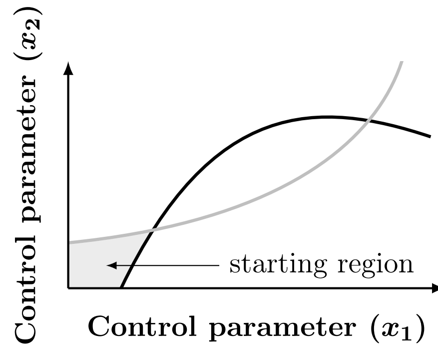


Figure 5.4.: The starting region for a workload with two service classes (and, thus, two control parameters)

5.3. Finding points in the operating envelope

This section describes the MsCoSearch (multi-service class, compound objectives search) algorithm, our approach to solving the multi-class, compound-objective search problem described earlier. We first describe how MsCoSearch locates a point when the workload is relatively constant, e. g., only small variance in the performance measurements occur. Then we show how to handle changes in the workload, e. g., change in characteristics of the incoming queries, arrival of new service classes, or a change in objectives.

The design goals of the algorithm are: (1) Eliminate human interaction in finding a point in the operating envelope. (2) The algorithm should be sound, i. e., find a point in the operating envelope if one exists or else if no such point exists, terminate gracefully. (3) Find a solution (or the lack thereof) in as few iterations as possible because probes in the search space are expensive (e. g., it takes time until system returns to a stable state after the change of control parameters).

The search begins in the *starting region*, defined as the settings of the control parameters such that no service class is in an acceptable region. For example, in Figure 5.4, the starting region is the area bounded by the origin and the first intersection of the two solid curves. To converge as fast as possible towards the operating envelope, we are interested in the "maximal" setting in the starting region, which is the setting where increasing a control parameter puts some service class in an acceptable region. Since the shape of the operating envelope and, thus, the starting region, is not known in advance, the search starts at a setting that is trivially in the starting region: setting $X^{(s)} = \left\{x_1^{min}, \ldots, x_n^{min}\right\}$ where the control parameters are set to their minimum values x_i^{min}. If additional knowledge is available, we can choose a setting that is closer to the operating envelope (we will show how to leverage previously gathered information in Section 5.5).

The goal of the algorithm is to find a minimal setting X in the operating envelope. We define $X = \{x_1, \ldots, x_n\}$ to be minimal w. r. t. the operating envelope E if for all

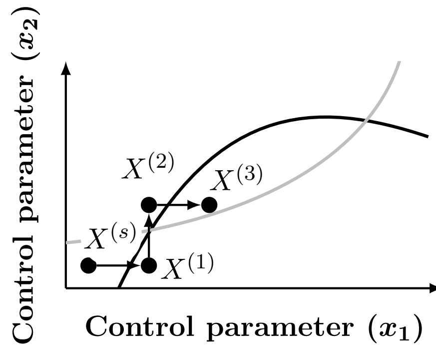


Figure 5.5.: Our algorithm moves along one dimension of the search space at a time. Points $X^{(1)}$ and $X^{(2)}$ denote the intermediate points, $X^{(3)}$ the final point.

settings $X' = \{x'_1, \ldots, x'_n\} \in E$: $x_i \leq x'_i$, i. e., all control parameters in X are less than or equal to the respective control parameters in X'. Even though the search problem does not require it, a minimal solution facilitates the location of a starting point if the workload or objectives change.

Starting at point $X^{(s)}$, the algorithm chooses a dimension along which it starts the search from the set of *candidate dimensions*, dimensions associated with a service class whose objectives are not satisfied. The SingleDimSearch algorithm, which we will describe in detail below, searches along the chosen dimension and either returns a setting $X^{(i)}$ in the acceptable region of service class s_i (if such a setting exists) or an "invalid" setting (e. g., NIL). The latter case indicates that the algorithm cannot find an operating envelope and the algorithm terminates. When $X^{(i)}$ denotes a "valid" setting in the search space, the algorithm checks whether $X^{(i)}$ is in the operating envelope, in which case it terminates. If the point is not in the operating envelope, the algorithm chooses another dimension to move along and then continues the search. We call the setting $X^{(i)}$ where the algorithm starts to move along a different dimension, an *intermediate point*. Figure 5.5 shows an example how the algorithm searches in a scenario with two service classes. Settings $X^{(1)}$ and $X^{(2)}$ denote the intermediate points where the algorithm moves along a different dimension while setting $X^{(3)}$ denotes a final point. Next, we describe how the intermediate points are located, i. e., how the algorithm moves along a single dimension in the search space.

Moving along the ith dimension, our algorithm produces a control parameter setting $X^k = \left\{x_1, \ldots, x_{i-1}, x_i^k, x_{i+1}, \ldots, x_n\right\}$ in the kth iteration by only changing the ith control parameter setting and leaving all other parameters constant. The goal of the search along dimension i is to find the minimal setting along that dimension that is in the acceptable region of the respective service class. We define a setting X to be minimal if there is no smaller value for the ith control parameter such that X is still in the operating envelope.

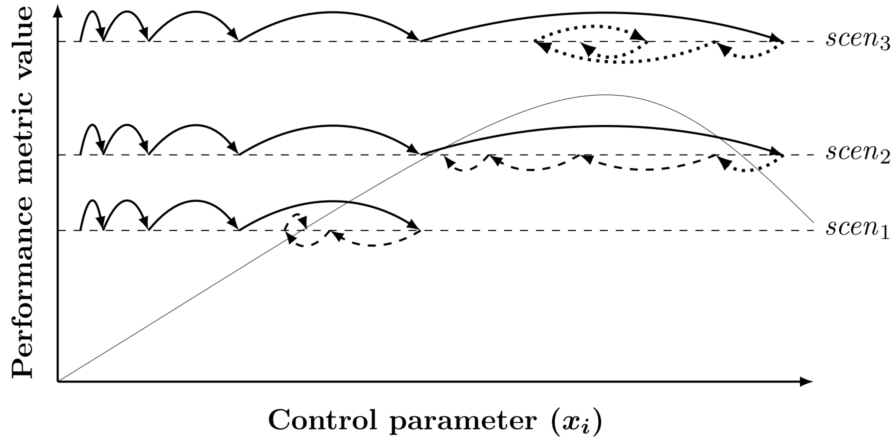


Figure 5.6.: Search steps made by the SingleDimSearch algorithm. Horizontal lines show three different scenarios (lower bounds). The arrows show the moves to the right (solid), binary search (dashed), and Fibonacci search (dotted). Binary search terminates when the minimum step width has been reached (e. g., last step in $scen_1$). Fibonacci search terminates when a measurement that exceeds the threshold ($scen_2$) or the maximum ($scen_3$) has been located.

5.3.1. Move along single dimension – single objective

While moving along dimension i in the search space, the SingleDimSearch algorithm makes two decisions in each iteration: First, it decides whether to increase or decrease the value of the control parameter, and second, how big the next step should be. The decision of whether to increase or decrease the control parameter value is based on the m control parameter-performance curves of the current service class. Note that these curves are not known in advance, so we must gather information about the curves during runtime. The algorithm can use each control parameter setting X to extend its information about the m curves concurrently.

We first describe how to change the control parameter based on a single performance metric. The description of the algorithm is based on two assumptions: First, we assume that higher performance values map to “better” performance. Second, increasing the control parameter value results in better performance when the system is in underload and leads to a performance decrease when in overload.

Figure 5.6 illustrates the three scenarios that can occur when searching a single dimension for a single performance metric. Each horizontal line represents a different setting for the objective. Recall that we assume the control parameter-performance curve is unimodal, i. e., an increasing region to the left of the maximum and a decreasing region to the right. In the first scenario, $scen_1$, the objective is reached searching only in the increasing region. The second scenario, $scen_2$, must search

both regions to find the objective. In the third scenario, $scen_3$, the objective is unsatisfiable. The arrows indicate the search steps the SingleDimSearch algorithm makes. We will describe the algorithm in more detail in the following.

For the following description, we denote with p^k_m the performance of metric m measured with setting X^k. If the metric is clear from the context then we omit subscript m. We also say that the algorithm moves left and right when it decreases and increases the control parameter, respectively.

As long as the current measurement p^k is below the objective and measurements p^{k-1} and p^k indicate an increase in performance with increasing control parameter value, we increase the control parameter value. When the algorithm increases the control parameter, there are three options for the next measurement p^{k+1}.

Case (1): objective violated, performance increase ⇒ move right

The performance increases further, i. e., $p^k < p^{k+1}$ but p^{k+1} still violates the objective. In that case, we must increase the control parameter value to approach the acceptable region of this performance metric. This case is illustrated in Figure 5.6 by the first four moves (solid arrows) for scenario 1 and the first five moves for scenarios 2 and 3.

There exist different options for determining how far to move to the right. We will briefly discuss the approaches here, noting that there may be more options for increasing the control parameter: First, increase the control parameter by 1. However, the small increments may take too long to approach the acceptable value of the performance metric. Second, exponentially increase the control parameter, i. e., make each step twice as big as the previous one. Third, take the last two measurements and the respective control parameter values to predict the control parameter value that satisfies the objective. This approach is similar to the performance model devised in [46]. We assume that when we move along a single dimension in the search space, the control parameter-performance curve has the well-known unimodal shape: performance increases for small control parameter values, reaches a plateau at "saturation point" and drops in overload (e. g., due to thrashing). Based on this assumption, we estimate the control parameter value that is located in the acceptable region of this performance metric applying a simple linear regression [19] using the last k measurements. Linear regression can be used for two reasons: (a) The curve can be approximated with a line if the difference between two control parameter values is not too big. (b) We account for variance in the measurements. For example, consider two measurements p and p' at the increasing side of the curve, taken with settings x and x' ($x < x'$), respectively. In a variance-free system we have $p < p'$. However, with variance, p could be larger than p' because of the variance in the measured performance, which may lead us to a wrong decision.

Case (2): objective violated, performance decrease ⇒ start Fibonacci search

Measurement p^{k+1} violates the objective and is lower than or equal to p^k. This case is illustrated in Figure 5.6 by the sixth move for scenario 2 and moves six to nine for scenario 3 (dotted arrows). When we observe two increasing measurements followed by a decreasing (or constant) one, i. e., $p^{k-1} < p^k$ and $p^{k+1} \leq p^k$, we can infer for unimodal curves that we overstepped the maximum of the curve, or we are at a plateau. Since performance is decreasing with increasing control parameter value and measurement p^{k+1} violates the objective, increasing the control parameter value would further decrease performance. As a consequence, we know that the maximum of the curve is located in the interval $\left[x_i^{k-1}, x_i^{k+1}\right]$. The Fibonacci search algorithm finds the maximum value of a unimodal curve given an interval that contains the maximum. When applying Fibonacci search, there are two possible outcomes.

First, Fibonacci search finds a control parameter value where the performance measurement satisfies the objective. As soon as it finds such a point, Fibonacci search stops and starts a binary search similar to case (3) below. Second, Fibonacci search terminates and the maximum performance value still violates the objective. In that case, the SingleDimSearch algorithm terminates, returning a value that indicates that there is no acceptable region for the service class.

Case (3): objective satisfied ⇒ start binary search

Measurement p^{k+1} satisfies the objective (e. g., after the fourth move in scenario 1). Since we are interested in the smallest control parameter setting along dimension i that satisfies the objective, we start a binary search in the interval $\left[x_i^{k}, x_i^{k+1}\right]$ to find the setting we are interested in. We use binary search (dashed lines in Figure 5.6 because we know due to the unimodality assumption that there is exactly one point where the performance measurement crosses the objective and that this point is in interval $\left[x_i^{k}, x_i^{k+1}\right]$.

5.3.2. Extension for compound objectives

If compound objectives are specified for a single service class s_i, the choice of which direction to move the control parameter is a function of all performance metrics in M_i: The algorithm locates the minimum point that satisfies *all* objectives. With multiple metrics, the rules from Section 5.3.1 are applied to each of the possibly different suggested moves (one move per metric): move right, start a Fibonacci search, or start a binary search. Consequently, a decision table is needed to resolve the different moves into a single "consolidated" move (Table 5.1). Let $move_j$, $1 \leq j \leq |M_i|$ denote the jth move and let $cons_j$ denote the consolidated move after considering the jth move. At the beginning, the consolidated move is initialized to *(none)*, i. e., $cons_0 = (none)$. To derive $cons_j$, the algorithm looks up the entry in Table 5.1 at position $(move_j, cons_{j-1})$. The value of the control parameter is changed according to the resulting consolidated move $cons_{|M_i|}$. The algorithm terminates for $j < |M_i|$

		Direction from currently considered acceptable region		
		right	**Fibonacci**	**binary**
Conso-lidated move	**(none)**	right	Fibonacci	binary
	right	right	stop	right
	Fibonacci	stop	Fibonacci	Fibonacci
	binary	right	Fibonacci	binary

Table 5.1.: Decision table for next move

when the moves are contradictory (the *stop* entries in the table), e. g., if one metric requires a move to the right and another requires a move to the left (i. e., Fibonacci search). In contrast to that, *binary* is compatible with every other move. Let m denote a metric that satisfies its objective at k, i. e., $X^k \in A_m$ and the move for that metric is *binary*. Note that from the perspective of that metric, the algorithm can either decrease, i. e., start a Fibonacchi search or a binary search, or increase the control parameter value. The choice for metric m depends on the moves for another metric m'.

If the suggested move for m' is to move *right*, the algorithm can safely choose to increase the control parameter value. Note that the increase of the control parameter might result in dropping out of the acceptable region for m again. However, in this case, the algorithm – in the next iteration $k + 1$ – would no longer allow a control parameter increase but would require to move to the left, i. e., start a Fibonacci search. If the suggested move for m' is to start a Fibonacci search, the algorithm chooses to start a Fibonacci search that is "driven" by m', i. e., the algorithm decreases the control parameter setting.

For an example how to read Table 5.1, consider a service class for which the rules in Section 5.3.1 result in the two moves *right* and *binary*. Using Table 5.1, the consolidated moves are $cons_1 = right$ (entry at *right* and *(none)*) and $cons_2 = right$ (entry at *binary* and *right*). Consequently, the algorithm decides to increase the control parameter setting for the service class.

Consolidated move: right

If the consolidated move $cons_{|M_i|} = right$, i. e., the control parameter value must be increased, the algorithm must figure out the step width. In this case, we use the maximum of the step widths derived in case (1) for the individual performance metrics. By choosing the maximum step width, the algorithm converges faster towards the acceptable region, but running the risk of either moving too far in the acceptable region of even overshooting it. However, the algorithm, as described in cases (2) and (3) for the single-objective case, can handle these cases and move left in order

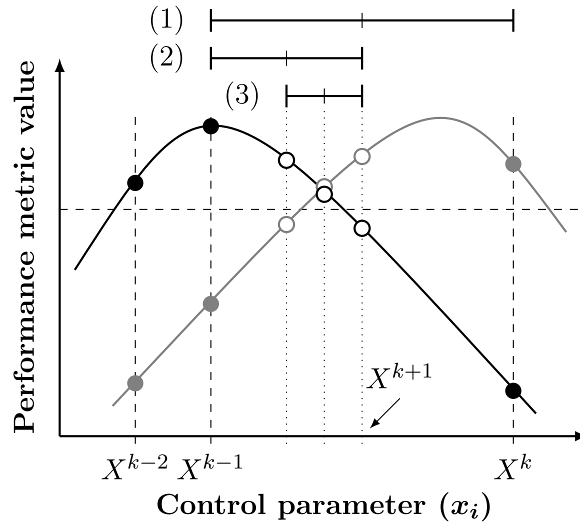


Figure 5.7.: Example to illustrate how to locate the minimum control parameter setting that satisfies the objectives of the two performance metrics using binary search.

to find the minimum control parameter setting.

Consolidated move: binary

If a binary search must be started (i. e., $cons_{|M_i|} = binary$), the challenge is that the different performance curves may peak at different control parameter values. For example, Figure 5.7 illustrates two performance metrics for a single service class. To simplify the figure, we use just a single horizontal line to represent the objective for both metrics. The vertical dashed lines show three control parameter settings X^{k-2}, X^{k-1}, X^k along with the performance measurements (filled dots). The open circles in the figure show the performance measurements during binary search. At control parameter setting X^k, the SingleDimSearch algorithm starts a binary search for the gray performance metric. The initial search interval for starting the binary search (bar with label (1) in the figure) is the minimum interval that – for each individual performance metric – contains a measurement that meets the respective objective.

Figure 5.7 gives an intuition why we use the "minimal" interval to start the binary search: From the position of the "gray" measurement at X^k relative to the "black" measurement at X^{k-1} and the unimodality assumption, we can conclude that the intersection of the acceptable regions for "gray" and "black" cannot be at settings $< X^{k-1}$ and $> X^k$. Note that the minimality requirement is not needed for correctness, it just makes the binary search interval smaller, i. e., the search terminates more quickly. The intuition why all service classes need a point in the respective acceptable

		Objective satisfied	
		mid ok	*mid nok*
Shape of performance curve	*inc*	either	right
	dec	either	left

Table 5.2.: Decision table to tell whether to search in the left or in the right half of the binary search interval.

region to start the binary search is as follows: If there is a performance metric for which we do not have a point in the respective acceptable region, we cannot guarantee that the overlap of the acceptable regions (if it exists) is in the search interval.

Based on Table 5.2, we determine for each performance metric separately whether the binary search continues in the left or the right half of the search interval. Similar to the decision whether to move left or right for a single service class, we consider whether or not the current measurement (i. e., the mid of the binary search interval) meets the objective. If it does, a valid solution may either be in the left or right half of the search interval. If X^{k+1} violates the objective, we distinguish two cases. If the current measurement is on the increasing part of the curve, we search the right half of the interval. If it is on the decreasing part of the curve, we search the left half. For example, consider the first binary search measurement at X^{k+1} in Figure 5.7 for the black performance metric. The measurement violates the objective. The measurements in interval (1) indicate that we are on the decreasing side of the performance curve, i. e., we approach the acceptable region for this performance metric by decreasing the control parameter value. As a consequence, we must continue the binary search in the left half of interval (1). The gray measurement at X^{k+1} satisfies its objective. As a consequence, we can continue to search either in the left or in the right half of interval (1). This gives us the binary search interval (2) and continuing in this fashion we get the binary search interval (3).

Consolidated move: Fibonacci

A similar approach can be taken when a Fibonacci search must be started (i. e., $cons_{|M_i|} = \mathit{Fibonacci}$). Since there is at least one metric $m \in M_i$ for which the rules in Section 5.3.1 indicated to start a Fibonacci search, the algorithm chooses one of the metrics. After each iteration, the algorithm checks, depending on whether the objectives are met, how to continue the search: If the Fibonacci search fails to locate a point in the acceptable m-region, the search terminates and we can conclude that there is no setting that satisfies all objectives (similar to scenario $scen_3$ in Figure 5.6). Otherwise, we continue the search as follows: First, if all objectives are violated at that setting, the Fibonacci search continues along the performance curve for m. Second, if all objectives are met at that setting, the algorithm starts a binary search

similar to the search described above. Third, if the objective based on m is satisfied (and at the objective for at least one other metric is violated), there are two options. If there is a metric $m' \neq m$ for which no setting in the acceptable m'-region has been observed so far, the algorithm must start a Fibonacci search for m'. Otherwise, there we know that there is a setting that satisfies the objectives for every metric in M_i and the algorithm starts a binary search, similar to the case ($cons_{|M_i|} = binary$.

5.4. Variance

The discussion so far assumes that the measurements taken follow the (unknown) control parameter-performance curves perfectly. However, in a real system, measurements are subject to variance (or "noise") so that the measurements do not perfectly follow the control parameter-performance curve. For example, in presence of variance, two measurements p_m^{k-1} and p_m^k taken with different control parameter settings x and x' ($x < x'$) may indicate a decreasing curve (dotted line in the figure) although they are on the increasing side of the curve.

In order to minimize variance, we assume that the metrics are chosen appropriately for a workload. For example use average response time for service classes that have similar queries. Also, for metrics based on sliding windows, the length of the sliding window should be chosen such that variance is smoothed out.

Even with taking precautions, variance cannot be reduced to 0. As a consequence, we tighten the bounds related to a performance metric m by a factor $f_m = (1 + \epsilon_m)$ with $\epsilon_m > 0$, i. e., we multiply the lower bound by f_m and divide the upper bound by f_m. We utilize the characteristics of the acceptable regions where acceptable regions associated to tighter bounds are completely contained in the acceptable region with less tight bounds. As a result, the operating envelope with the tighter bounds is also contained in the operating envelope with the less tight bounds. In our framework, we set the values for ϵ_m manually. An option is to determine the ϵ_m-values at runtime, based on performance measurements. The setting of ϵ_m is not in the focus of this work.

To avoid the problem sketched earlier, i. e., where two measurements show decreasing behavior even though the curve should be increasing, we designed our system to exhibit hysteresis. Hysteresis characterizes a system where the output depends not only on the current input but also on the history of the input. When the previous measurements indicate a performance increase, we assume a further increase if the current measurement p^k is greater than or equal to $\delta_m \cdot p_m^{k-1}$, with $0 < \delta_m \leq 1$, we assume a further increase. When the current measurement is smaller than $\delta_m \cdot p_m^{k-1}$, we assume that performance has decreased. Note that δ_m can be chosen small if the variance of the measurements is low. For higher variance, the δ_m value must be larger. However, too large values for δ_m result in errors when the performance is actually decreasing after an increase. Determining "good" values for δ_m is out of the scope of this work.

5.5. Operating envelope changes

When a workload change occurs, the current control parameter setting may result in performance outside the operating envelope. The task after such a change is to find a point where to initialize the search. Note that the presented algorithm assumes that the initial setting is in the starting region. The straightforward approach is to start from the setting where all control parameter values are set to their minimum values. Although in some cases, it is possible to start with a setting that is "closer" to the operating envelope so that the search terminates faster.

Below, we discuss the causes of operating envelope changes and how to find a setting $X^{(s)}$ to initialize the search after the change. Let E^b denote the operating envelope before the change and E^a the operating envelope after the change. With $X^b \in E^b$, we denote the setting before the change. Note that due to our algorithm, X^b is minimal w. r. t. E^b.

Workload change, objectives unchanged

A workload change occurs when the number of service classes changes (a new service class arrives or an active service stops executing) or the number of service classes remains constant but the characteristics of the queries of a service class changes. The latter occurs if, e. g., an updated version of an application that accesses the database system has new queries where the function is the same but the execution is expected to be more efficient. For simplicity, we assume that only one workload change happens at a time. We do consider that changes may happen within a short time. In that case, we restart the algorithm when we detect the next workload change. Note that the objectives of the service classes that were active before the workload change, remain constant. As a consequence, only the shape of the load-performance hulls change and result in new shapes for the acceptable regions, which in turn form a new operating envelope.

We first consider how to find a starting point when a new service class arrives. Then, we consider the problem when a service class departs. When the workload changes, information that we previously gathered, e. g., performance at certain control parameter settings, may become invalid. Let $X = \{x_1, \ldots, x_n\}$ denote the "largest" previously observed control parameter setting in the starting region before the new service class s_{n+1} arrived. The largest setting not in the operating envelope is defined as setting $X = \{x_1, \ldots, x_n\}$ where there is no other setting $X' = \{x'_1, \ldots, x'_n\}$ in the starting region where at least one control parameter value x'_i is greater than x_i. Note that in general, there may be more than one such setting. In that case, we can randomly choose one of the settings. When s_{n+1} arrives, setting $X^{(s)} = \left\{x_1, \ldots, x_n, x_{n+1}^{min}\right\}$ can be used as a starting point. However, it is possible that increasing the total load in the system from X to $X^{(s)}$ saturates the system or pushes it to overload. If we detect that setting $X^{(s)}$ is indeed an overload situation (for example when the performance along every dimension decreases when we increase the setting of the respective control parameter), we revert to setting

$X^{(s)'} = \{x_1^{min}, \ldots, x_n^{min}, x_{n+1}^{min}\}$ and restart the search from there.

Let $X^b = \{x_1, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n\}$ denote the measurement before service class s_j left the system (note that X^b was inside the "old" operating envelope). When a service class leaves the system, the operating envelope becomes bigger because the resources must be shared among fewer service classes. As a consequence, the performance at setting $X^{(s)} = \{x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n\}$ satisfies the lower bounds of the performance metrics. However, $X^{(s)}$ may be outside the new operating envelope E^a when the performance for at least one service class increases beyond the respective upper bound. Since $X^{(s)}$ meets all lower bounds, doing binary search along the individual dimensions, i. e., locating the smallest setting within the respective acceptable region, results in a setting in the operating envelope.

When the characteristics of the queries change but the number of service classes remains constant, we cannot reuse prior measurements to determine a point in the starting region. As a consequence, the initial point for the search is set to the "minimum setting" $X^{(s)} = \{x_1^{min}, \ldots, x_n^{\min}\}$.

Workload unchanged, objectives change

When the objectives change but the workload remains constant, we can reuse previously observed measurements to initialize the search for the new operating envelope E^a. For an objectives change, we consider the following scenarios: add/remove an objective and increase/decrease upper/lower bound.

When a new objective is added, e. g., an upper or a lower bound for a newly considered performance metric, we can make two observations: First, for all settings $X \notin E^b$, we also have $X \notin E^a$: A setting that was outside the operating envelope before the change will not be inside the operating envelope after the change upon adding a new objective. Second, a setting $X \in E^b$ may become "unacceptable" with the newly added objective. As a consequence, we can conclude that $E^a \subseteq E^b$. Since we know that setting X^b was minimal, we know that the starting point for the new search is $X^{(s)} = X^b$. If the operating envelope has not changed, i. e., $E^a = E^b$, no search is carried out.

When an objective is removed, the operating envelope E^a may become bigger than operating envelope E^b, i. e., $E^b \subseteq E^a$. Since setting X^b is inside the "old" operating envelope ($X^b \in E^b$), it will also be inside the "new" operating envelope ($X^b \in E^a$). In order to find the minimal setting inside operating envelope E^a, we can reuse information gathered previously (note that the performance curves have not changed because the load on the system has not changed). As the starting point, we use the "maximal" setting where the respective performance measurements are outside the operating envelope E^a. Figure 5.8 illustrates an example. The dots show the settings in the search space set by the algorithm. Note that for each setting, we also have the respective performance measurements. The solid lines show the "old" acceptable regions for service classes 1 (black) and 2 (gray). The gray dashed line denotes the shape of the acceptable region after an objective for service class 2 was removed. The shaded area denotes the starting region for the "new" operating

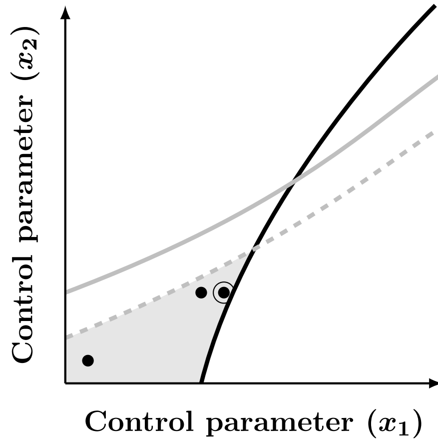


Figure 5.8.: Example for a change of the acceptable region when an objective for service class two is removed. The solid gray line denotes the acceptable region before, the dashed line the region after the change. The shaded area indicates the starting region for the search of the new operating envelope.

envelope. The dot marked with a circle ⦿ denotes the largest setting that is in neither of the acceptable regions.

If the lower bound for a performance metric is increased, the operating envelope becomes smaller, where the explanation is similar to the "add objective" case. Due to the minimality of setting X^b w. r. t. the operating envelope E^b, we start the search from $X^{(s)} = X^b$ to locate the minimum point in the "new" operating envelope. If the lower bound for a performance metric decreases, we know that $E^b \subseteq E^a$. Although $X^b \in E^a$, we locate the minimum setting w. r. t. the operating envelope E^a. No search is necessary if the upper bounds change. Since our algorithm moves along the lower bounds for the performance metrics, changing the upper bound does not invalidate the current setting.

5.6. Experiments

In the following we describe the results of the experiments which compare our MsCoSearch algorithm to an extension of the workload adaptation approach.

5.6.1. Extension of the workload adaptation approach

The workload adaptation algorithm [46] was devised to solve a multi-class, single objective problem. In order to handle compound objectives, we extended the algorithm to use *dominant* objectives for optimization. We observed that, given a specific range of control parameters, e. g., MPL less than 40, it is possible that satisfying one objec-

tive implies that all other objectives are also satisfied by those conditions, i. e., one objective dominates the others. We note that, however, it is not possible in general to know a priori which objective dominates the other objectives. Our experiments show that using the "wrong" dominant objective results in control parameter settings that are outside the operating envelope even though the envelope is not empty.

5.6.2. Experimental setup

For the experiments, we defined three service classes s_1, s_2, s_3 that comprise 32, 64, and 8 users, respectively. Each user interactively submits queries to the system with zero wait time, i. e., it sends the next query only after receiving the result of the previous one. As a consequence, the throughput in the database is not limited by the arrival rate unless all queries are admitted to the database system as they arrive – a situation we avoid by appropriately designing our experiments. Each query accesses a small amount of data on a single disk, reading between zero (data is cached) and two pages from disk, and returns between 1 and 4096 result tuples. Users associated with service class s_1 choose their next queries from a pool of 12 queries; similarly, the streams from service classes s_2 and s_3 choose from a pool of 15 and 8 queries, respectively (the query pools are disjoint, but queries from s_1, s_2, and s_3 will access the same table). We use one MPL control parameter per service class to control the load on the system.

The performance metrics considered for the service classes are throughput and average response time. Throughput counts the number of completed queries in the last 90 seconds. Similarly, the average response time aggregates the query response times in the last 90 seconds. The objectives for each service class are described with the individual experiments. We use the multiprogramming level MPL_i, i. e., the number of concurrently active queries stemming from a service class s_i, to control the share of the system resources assigned to a service class.

For the experiments, we configured the workload adaptation algorithm to use the correct dominant resource for all service classes. The workload adaptation algorithm requires setting a priority value for each service class. To get the threshold for the "worst allowed" performance (*worst*, see Section 5.1), we divided the performance objective by 2. Note that for metrics where lower values denote "better" performance, we used the inverse so that we can treat all performance metrics identical. We describe for each experiment separately how we determined the system cost limit that is required by the workload adaptation algorithm.

For MsCoSearch, we ran some offline experiments to determine the values for δ_m and ϵ_m (introduced in Section 5.4) to account for the variance. For the experiments, we set $\delta_m = 0.05$ and $\epsilon_m = 0.02$ for all metrics of all service classes.

In the following, we present two selected scenarios to evaluate our search algorithm. Each experiment starts with MPL set to 1 for each service class. For every experiment, our controller invoked an MPL change every 100 seconds. Considering the 90 second time window for computing the performance values, there is a 10 second grace period in which the system can stabilize after an MPL change. This grace

period proved to be sufficient for the experiments we ran. For different workloads, a different time window may be necessary. Last but not least, our algorithms terminate the search once they locate a point in the operating envelope or else detect that the envelope is empty. When a change in the workload occurs, another search is initiated for the operating envelope.

5.6.3. Experimental framework

We briefly describe the experimental framework we used to run our experiments. The framework extends the framework described in Section 4.3.

Similar to the framework introduced in Section 4.3, we use a simulated database engine. Acquiring the measurements used to create Figures 5.1 and 5.2 by running TPC-CH queries on a commercial DBMS while controlling the MPL of each service class took around 32 hours. Testing our algorithms required many more experiments, we therefore used a simulated, instead of an actual, database engine. Furthermore,the simulated database engine allows us to re-run experiments with repeatable results. The simulator also facilitates the implementation of the algorithms because we have complete control over the execution of the queries and the simulated database engine. For implementing and evaluating the algorithms, we can build arbitrarily complex (or simple) queries that either heavily interact with each other or have almost no interaction. Using the simulator, we can also run experiments with different configurations (e. g., four-nodes vs. 16-nodes), which cannot be easily achieved with a real database engine (buying a many-node system may be prohibitively expensive and "down-grading" it to a system with fewer nodes may incur tedious reconfiguration). Last but not least, since we are interested in the load that is caused by processing a query and not in the actual results, we achieve a speedup. Note that our simulator is not a full-fledged database engine but just gives us resource utilization measurements similar to a real database engine.

Our simulated database engine is implemented using CSIM [54] and extends the simulator described in Section 4.3.2. Our new database engine simulator has a more detailed model for query execution: First, the simulated database engine models pipeline-parallelism. When an operator has produced an output tuple, this tuple is sent to the parent operator, which starts processing the incoming tuple. Second, the simulator models CPU and disk consumption on a more detailed level. The CPUs and disks are modeled as facilities that can be used by a single operator at a time. Operators that want to access the facility are put in the waiting queue of the facility. When the facility is idle, the next operator to be admitted to the facility is chosen by a scheduling discipline (i. e., FIFO). An operator that has access to the facility, exclusively "locks" the facility for a predefined amount of time (time slice). When an operator has used its time slice, it is evicted from the facility and scheduled for execution again (if more work has to be done). If an operator completes before the time slice is over, a new operator is admitted to the facility. One operator can access one facility at a time (note that a logical operator can have multiple physical operators on different nodes). Third, the simulator allows a single physical operator

	Min throughput (queries per second)	Max avg. response time (seconds)
s_1	125	0.35
s_2	50	0.6

Table 5.3.: Bounds for scenario 1

to access multiple resources to produce an output tuple, i. e., there is no "dominant" resource. Fourth, the simulator models the exchange of tuples between operators that run on different nodes over the network. Instead of sending individual tuples, the producer operator groups multiple tuples into fixed-size messages. A message is sent over the network if (1) it is full or (2) a timeout has been reached. The time it takes a network message to travel from the producer to the consumer operator is determined by the bandwidth of the network.

Details on the workflow to create the input for the database engine simulator, for creating workloads for the simulator, and for running an experiment can be found in Section 4.3.1.

5.6.4. Scenario 1: two service classes, no changes

The goal of the experiments in scenario 1 is to show how the workload adaptation algorithm and our search algorithm work when there are no changes (neither workload nor objective) and the operating envelope is not empty. For this scenario, we use service classes s_1 and s_2, as described above. Each of the service classes has an average response time and a throughput objective with a single bound that requires a "minimum" performance. Table 5.3 summarizes the performance requirements for the two service classes. For ease of presentation, the table displays the performance metrics as they would have been entered by a user instead of the negative average response times. Note that we chose the performance metrics such that the operating envelope is not empty and such that admitting all queries as they arrive is not an option.

MsCoSearch We first evaluate the three approaches to move along a single search dimension mentioned in Section 5.3.1: increment the control parameter by a fixed value, exponentially increase the control parameter, and apply linear regression to compute the next setting. Figures 5.9(a) to 5.9(c) show the probes in the search space using the different approaches. As shown in Figure 5.9(b), using the approach to increment the control parameter by one locates the operating envelope after 31 probes. With this increment, the algorithm never has to start binary search: once we cross the border of an acceptable region, we know that the current setting is the smallest one and we can move along another dimension. With a bigger increment (>1,

not shown in the figures), the algorithm approaches the acceptable region faster but always has to start a binary search when it finds finds a setting in the acceptable region. For the experiments we ran, a greater increment resulted in more probes in the search space.

With exponential increase, the algorithm quickly moves towards the acceptable region. However, the large steps in the exponential increase require a binary search to locate the minimum setting. As shown in Figure 5.9(c), using the exponential approach terminates after 40 probes. So the “costs” for doing binary search dominate the benefit from moving faster towards the acceptable region.

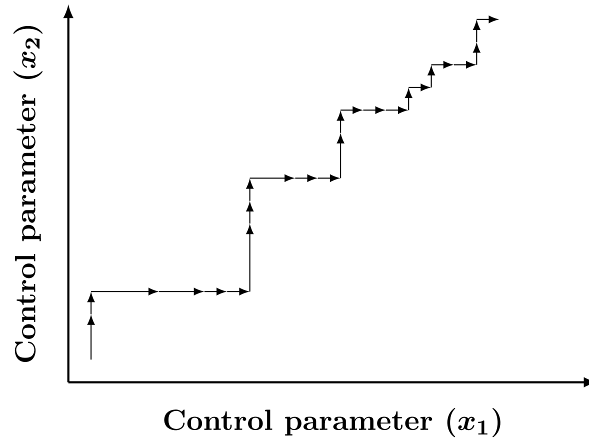
None of the two approaches above considers the distance from the current probe point to the acceptable region. The linear regression approach makes big steps to approach the acceptable region, making smaller step sizes the closer it approaches the region, trying to avoid binary search after locating the acceptable region. Figure 5.9(a) shows the 25 probes in our experiment. We note that, although the figure shows no binary search, the linear regression approach can not always avoid binary search.

We ran experiments with different workloads that showed that linear regression is the fastest approach to locate the operating envelope. As a consequence, we use linear regression for our MsCoSearch algorithm.

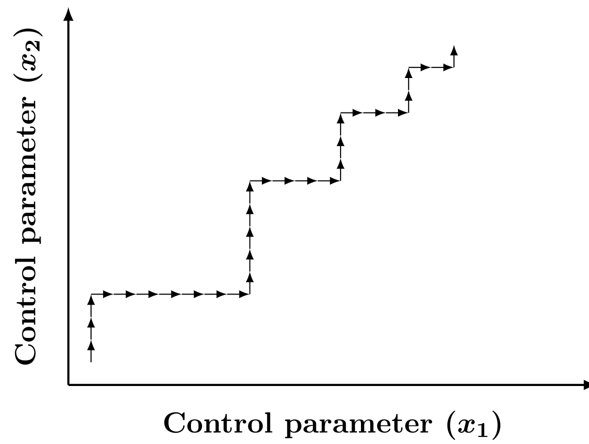
Figure 5.10 shows the details of running MsCoSearch with linear regression. The performance graphs (Figures 5.10(a) and 5.10(b)) show the course of the performance over time (solid lines) and the lower bounds defined for the respective performance metric (dotted lines). The shaded areas in the figures show the time the algorithm searches for a point in the operating envelope. The algorithm terminates after 25 probes (time 2500) at setting $(19, 16)$, satisfying all constraints of all service classes.

Workload adaptation In order to determine the system cost limit, we created a dummy service class that contains all queries from service classes s_1 and s_2. In an offline experiment, we increased the MPL for this single service class until throughput hit a performance plateau at MPL=36, i. e., the throughput increase for a further MPL increase is less than 3%. Thus, increasing the MPL beyond 36 does no longer increase performance.

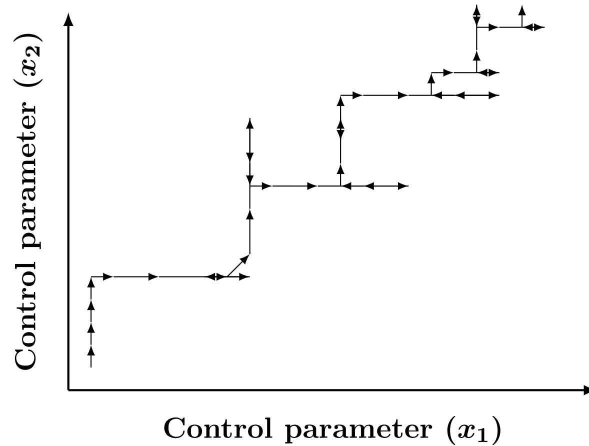
For this experiment, we assumed that we have complete information about the dominance of the objectives. We determined the dominance offline by a comprehensive search, i. e., running all different MPL_1-MPL_2 combinations. Figures 5.11(a) and 5.11(b) show for which MPL_1-MPL_2 settings the throughput and average response time (light green), only throughput (dark pink), only average response time (orange), or none of the objectives (white) are satisfied. For s_1, throughput is the dominant objective: whenever the throughput objective is satisfied for the settings shown in Figure 5.11(a), the average response time objective is as well. However, not all settings that satisfy the average response time objective also satisfy the throughput objective. Similarly, Figure 5.11(b) indicates that average response time is dom-



(a) Linear regression (25 probes)

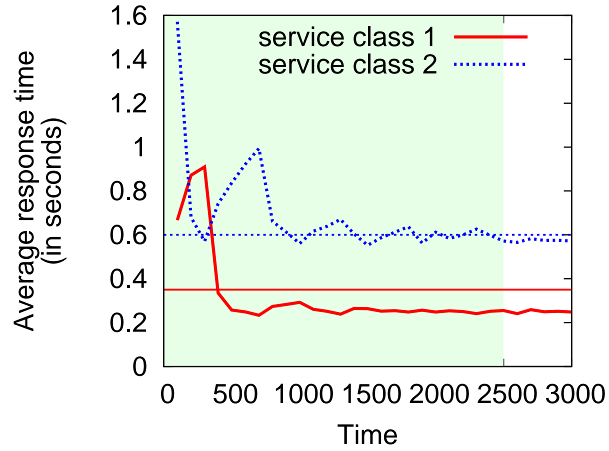


(b) Increment by 1 (31 probes)



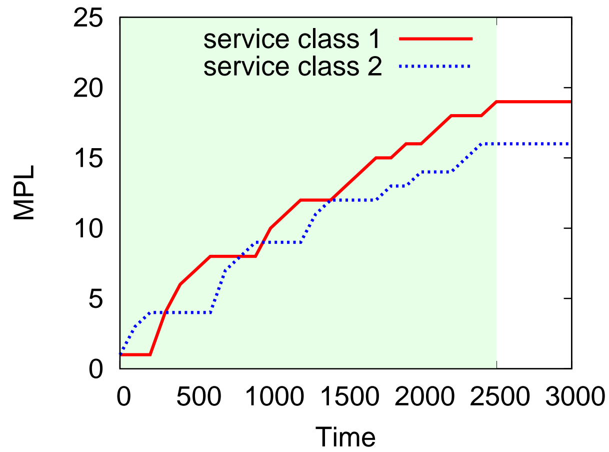
(c) Exponential increase (40 probes)

Figure 5.9.: Search paths using different techniques to find the next probe along a search dimension.



(a) Average response time

(b) Throughput



(c) Threshold changes

Figure 5.10.: Results of our MsCoSearch algorithm in scenario 1. The horizontal dotted lines for the performance graphs show the bounds for the respective service classes. The shaded areas indicate the time the algorithm is active.

	Time obj. valid	Min throughput (queries per second)	Max avg. response time (seconds)
s_1	$0 \leq t < 3000$	125	0.35
	$3000 \leq t \leq 10000$	90	0.6
s_2	$0 \leq t < 4000$	50	0.6
	$4000 \leq t \leq 10000$	100	0.45
s_3	$0 \leq t < 5500$	not started	not started
	$5500 \leq t \leq 10000$	30	0.4

Table 5.4.: Bounds for scenario 2

inant for service class s_2. As a consequence, the workload adaptation algorithm used throughput and average response time as dominant objectives for s_1 and s_2, respectively.

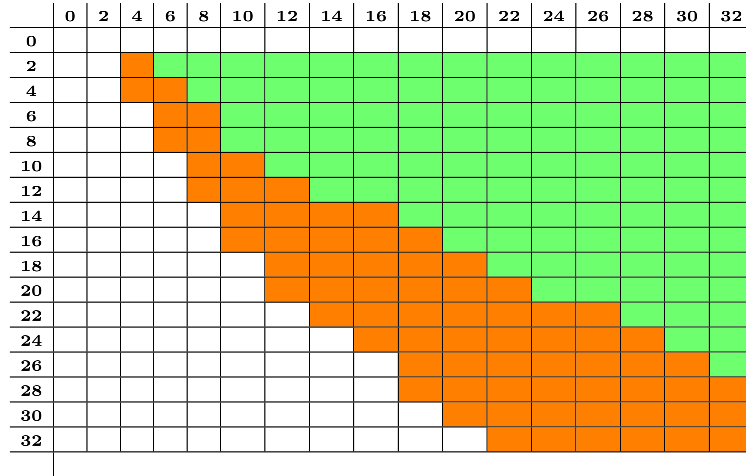
Figures 5.12(a) and 5.12(b) illustrate the results of the workload adaptation algorithm in scenario 1. The algorithm successfully locates a point in the operating envelope after six probes (600 seconds). The resulting point $(20, 16)$ satisfies the throughput and average response time objectives of service classes s_1 and s_2. Note that for this experiment we assumed knowledge about which resource is dominant for a service class. As stated earlier, this information may not be available prior to running the algorithm.

In a variation of the experiment we set the workload adaptation algorithm to use throughput as dominant objective for both service classes. In this variation the workload adaptation terminates after four probes at point $(25, 11)$, i. e., near the operating envelope but fails in finding a point inside the envelope, even though one definitely exists. The results in Figure 5.13 indicate that the throughput for service classes 1 and 2 at the resulting setting far exceeds the objectives (Figure 5.13(b)). However, the average response time objective for s_2 is violated at that setting.

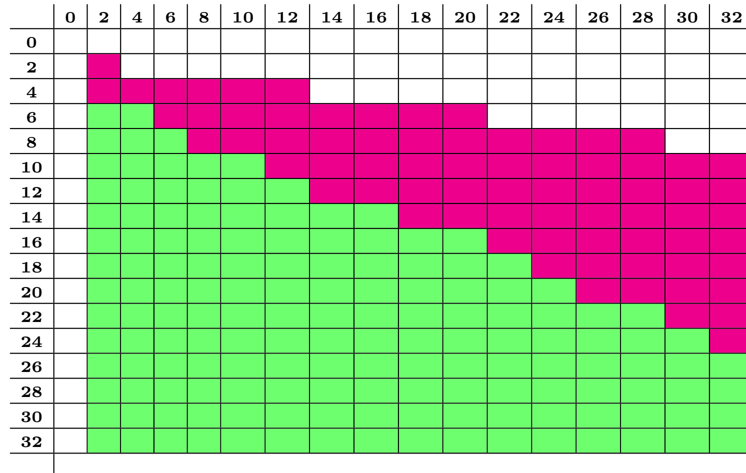
5.6.5. Scenario 2: change objectives, add new service class

The goal of this experiment is to evaluate how the workload adaptation and our MsCoSearch algorithm work when the operating envelope changes. Similar to scenario 1, we first start with service classes s_1 and s_2. After relaxing the objectives for service class s_1 at time 3000 and tightening the objectives for s_2 at time 4000, we add a new service class s_3 to the workload at time 5500. Table 5.4 summarizes the settings for the bounds and the changes to the objectives.

Figure 5.14 and Figure 5.15 summarize the results for running the algorithms in scenario 2 in the time interval between 0 and 10000 seconds. The vertical dashed lines

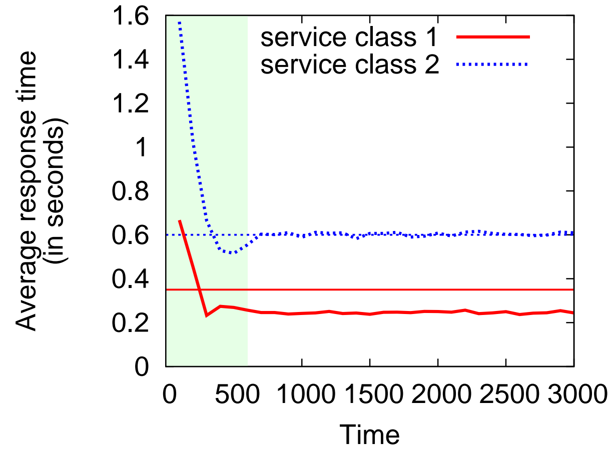


(a) Service class s_1



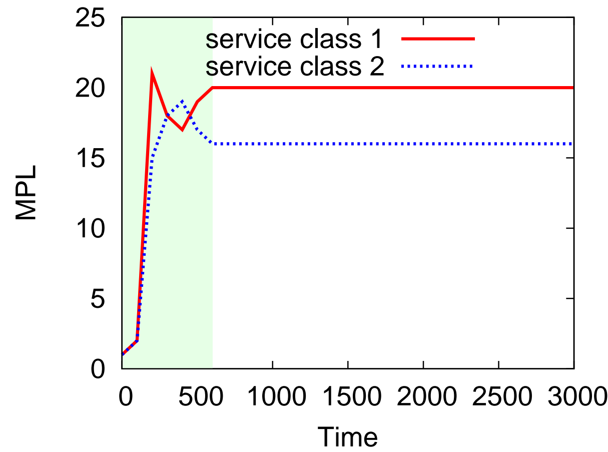
(b) Service class s_2

Figure 5.11.: The settings where none of the objectives are met (white), the throughput but not the average response time objective is met (dark pink), the average response time but not the throughput objective is met (orange), or both objectives (green) are met for service classes s_1 and s_2, respectively. The axes show the values for MPL_1 (x-axis) and MPL_2 (y-axis).



(a) Average response time

(b) Throughput

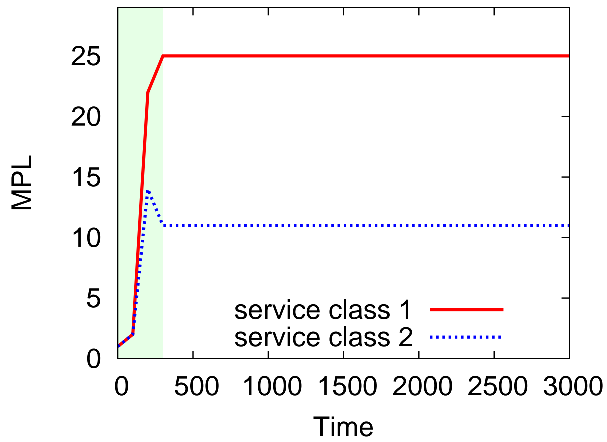


(c) Threshold changes

Figure 5.12.: Results of the workload adaptation algorithm in scenario 1 with correct information about dominant objectives.

(a) Average response time

(b) Throughput



(c) Threshold changes

Figure 5.13.: Results of the workload adaptation algorithm in scenario 1 with throughput as dominant objective for both workloads. With incorrect or incomplete knowledge about the dominance, the workload adaptation algorithm fails in finding the operating envelope.

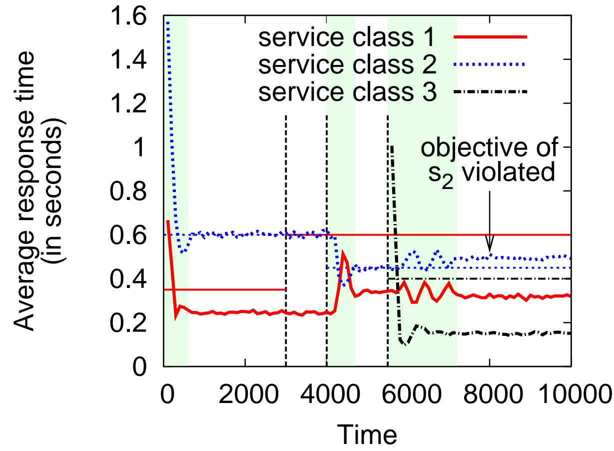
in the figures denote the times when the changes in either the objectives (times 3000 and 4000) or in the workload (time 5500) occur. Similar to Figure 5.12 and 5.10, the shaded areas denote the time the algorithm is active.

Workload adaptation The workload adaptation algorithm used the average response time as dominant objective for s_1 and s_3 and the throughput as dominant objective for s_2. We set the system cost limit to MPL=36 for the workload containing s_1 and s_2 as well as for the workload containing all three service classes (value determined as described in scenario 1).

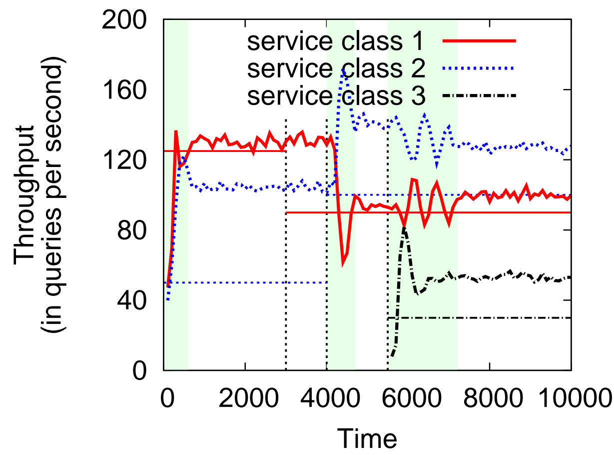
Similar to the first scenario, the workload adaptation algorithm starts at setting $(1, 1)$ and terminates after 600 seconds at $(20, 16)$, a point in the operating envelope. When the objectives of s_1 change at time 3000, the algorithm is executed again but does not move to a different point in the search space: The point $(20, 16)$ is still considered "optimal" after the objective change. Since the objectives of service class s_1 were relaxed, the resulting setting is still inside the operating envelope. After changing the objectives of service class s_2, the algorithm initiates a new search for the operating envelope because the objective change makes setting $(20, 16)$ unacceptable. The algorithm terminates at time 4700 at setting $(14, 22)$, which is inside the "new" operating envelope again, i. e., all objectives for service classes s_1 and s_2 are met. When the new service class arrives at time 5500, the workload adaptation algorithm initializes a search starting from setting $(14, 22, 1)$. The performance at the resulting setting, $(13, 17, 6)$, meets the throughput objectives of all three service classes and the average response time objectives of service classes s_1 and s_3 (Figures 5.14(a) and (b)). However, the resulting setting is not in the operating envelope (although there definitely is one) because the average response time objective of s_2 is violated (Figure 5.14(a)).

We ran some variants of scenario 2 with the workload adaptation algorithm, increasing the system cost limit to MPL values greater than 36. Although we do not show detailed results of these experiments, we note that we had to increase the system cost to 64 to get a point in the operating envelope using the workload adaptation algorithm. Setting the MPL to 64 does not overload the system. However, the throughput at MPL=64 is similar to the throughput at MPL=36 (note that we chose MPL=36 because it is the MPL setting where throughput "levels off", i. e., increasing the MPL does not lead to a significant throughput increase).

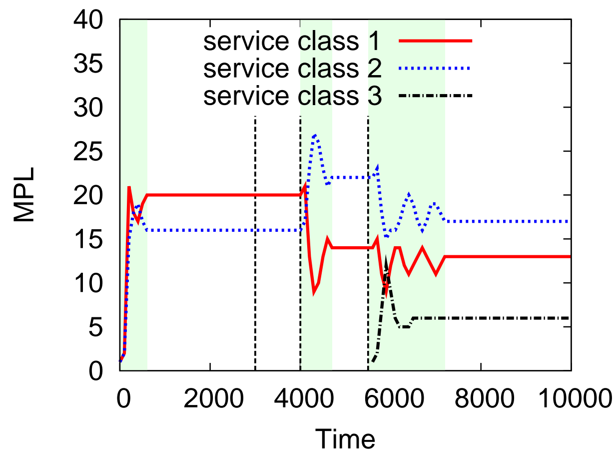
MsCoSearch Figure 5.15 shows the results of running the MsCoSearch algorithm in scenario 2. The threshold settings and the performance measurements are similar to scenario 1: the algorithm starts at point $(1, 1)$ and locates setting $(19, 16)$ in the operating envelope. When at time 3000 the objectives of service class s_1 relax, the algorithm reuses previously observed measurements to determine the MPL setting where to start the search for the new operating envelope. As described in Section 5.5, we use the largest previously probed MPL setting that is in neither acceptable region (i. e., that does not satisfy all objectives of service class s_1 or s_2) (setting $(4, 4)$ in



(a) Average response time

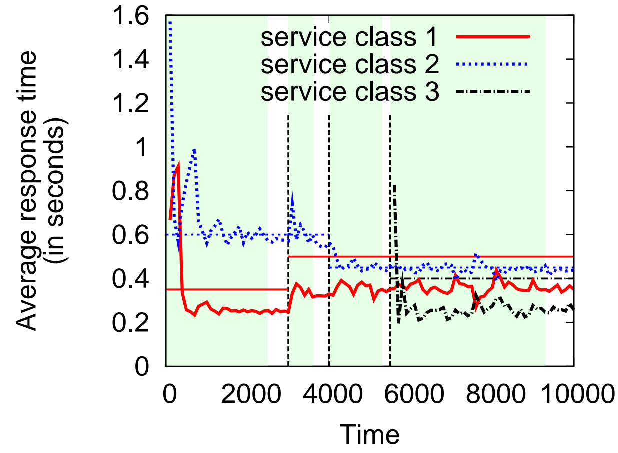


(b) Throughput



(c) MPL changes

Figure 5.14.: Results of the workload adaptation algorithm in scenario 2. The vertical dashed lines indicate the times at which the objectives and the workload change. The horizontal dotted lines for the average response time and throughput graphs show the bounds for the respective service classes.



(a) Average response time

(b) Throughput

(c) MPL changes

Figure 5.15.: Results of the MsCoSearch algorithm in scenario 2.

this experiment). Starting from $(4, 4)$, the algorithm restarts the search, resulting in setting $(6, 8)$. For the objective change of service class s_2 at time 4000, locating the starting point for the new search is straightforward: Since the objective was tightened, the current setting $(6, 8)$ is used as starting point for searching the operating envelope. After MsCoSearch returns the system to a stable state in the operating envelope at $(9, 16)$, the new service class s_3 starts executing at time 5500. After the workload change, MsCoSearch starts a new search in the now three-dimensional search space and finally terminates in the operating envelope at setting $(21, 33, 6)$. The spikes in Figure 5.15(c) stem from measurements taken in a near-saturated system, i. e., all system resources are almost fully utilized. In that case, increasing the MPL by a small value (e. g., 1), also results in a small performance increase. As a consequence, the algorithm estimates that it takes a big step to reach the acceptable region. However, MsCoSearch overshoots and then needs to dial back to find the smallest setting in the respective acceptable region.

5.7. Dashboard

As summarized in Chapter 2, commercial database workload management systems implement admission control, scheduling, and execution control policies. Chapters 3 and 4 explore the effectiveness of several static scheduling and execution control policies in the query control loop. This loop can remedy short-term fluctuations in the workload such as a single long-running query that unexpectedly hogs resources or lock contention that creates a convoy. However, it cannot change the policies themselves.

We envision our policy controller to be part of a workload management dashboard with the following capabilities:

- Overview of the performance metrics over time
- Allow manual execution of workload management actions
- Automatically execute workload management actions
- Manual policy control
- Automatic policy control
- Support an "instructor mode" that allows an "instructor" to inject workload management problems into the workload (e. g., add new service class, add new query, change availability of hardware (hardware failure))

We implemented a prototype of such a workload management dashboard on top of our policy controller. Our dashboard implementation supports two views: the administrator view and the instructor view. The *administrator view* consists of three screens. The *policy control screen* (Figure 5.16) displays the currently active rules and the threshold for the rules. If the policy controller is turned off, a human

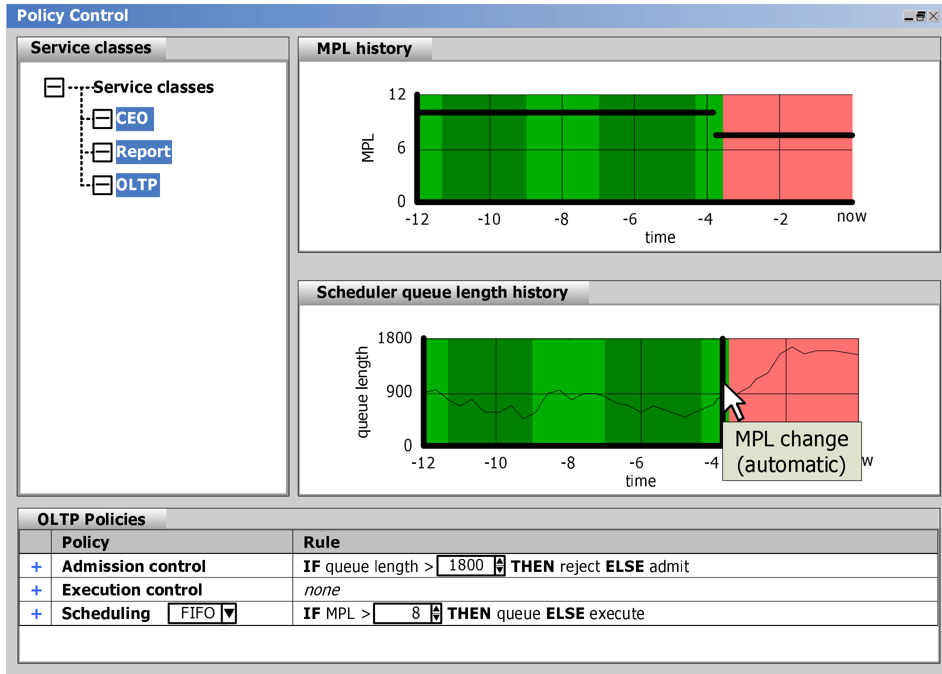


Figure 5.16.: Policy control screen

administrator can add and remove rules, and set the thresholds for active rules. If the policy controller is turned on, rules are (de)activated automatically and our MsCoSearch algorithm is applied to locate the operating envelope. The impact of changes of workload management policies can be observed in the *performance objective* (Figure 5.17) and the *system resource utilization screen* (Figure 5.18). The former shows information about performance metrics and how well the objectives of the service classes are met. The system resource utilization screen summarizes the resource utilization in the system.

The dashboard is also intended to help train database system administrators in managing workloads in order to meet the objectives of the service classes. As a consequence, an instructor can use the dashboard to manually change the workload. For example, the instructor can start or stop service classes, turn individual resources on and off, change the arrival rate of queries, or change the objectives of one or more service classes.

We illustrate an example of the dashboard using a scenario with three user loads:

- The “CEO” user load has hand-written, ad hoc queries written on behalf of a company executive. They arrive at unpredictable times. The only information the workload manager has about the expected resource usage and behavior of these queries are the optimizer’s cost estimates. The objective for each query is to complete it promptly — as long as its cost estimates are accurate. In terms of

Figure 5.17.: Performance objective screen after problem injection shows the OLTP service class not meeting objectives.

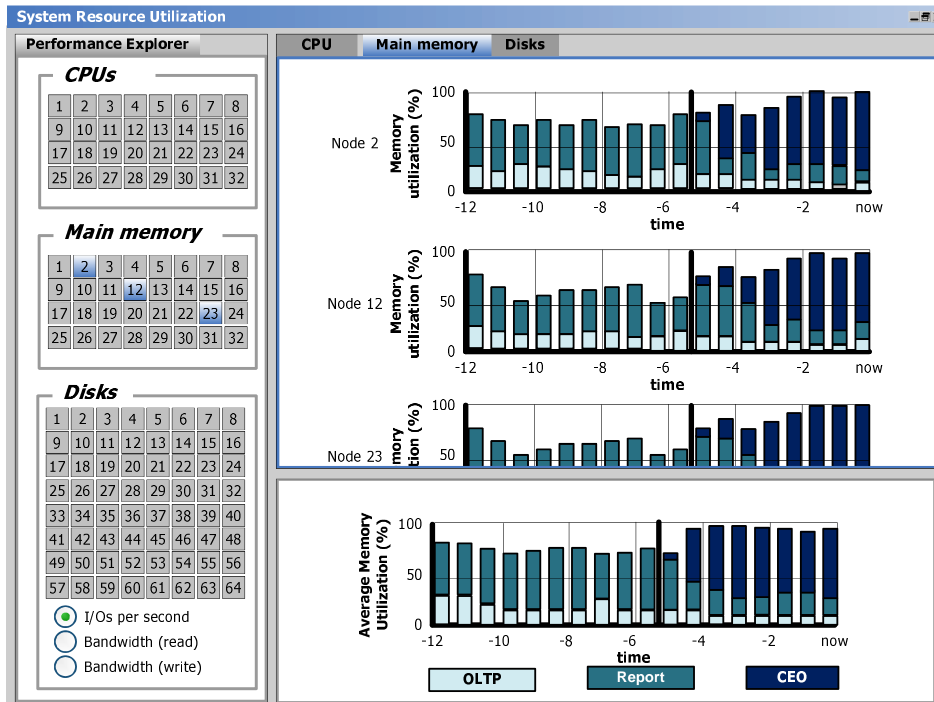


Figure 5.18.: In the system resource utilization screen, the administrator can see that after the arrival of a long-running heavy-weight CEO query, resource contention interferes with the rest of the workload.

workload management policies, (1) each query should be immediately admitted and scheduled, (2) these queries have higher priority than other queries, and (3) a query may be killed if actual resource usage exceeds twice that estimated.

- The “report” user load comprises medium-sized, roll-up report queries with an objective to complete all of the queries before a deadline. These queries are also well-understood.
- The “OLTP” user load has queries that are short and have a fixed arrival rate. The queries are well-understood, and we have high confidence in how we expect them to behave. The objectives for these queries require the throughput to be above a certain transactions per second threshold and the average response time to be lower than another threshold, expressed in terms of milliseconds.

Each of the three user loads is mapped to a separate service class.

5.7.1. Problem injection

For the example, we assume that the automatic policy controller is turned off and that the system is in steady state initially, with the OLTP requests running and meeting

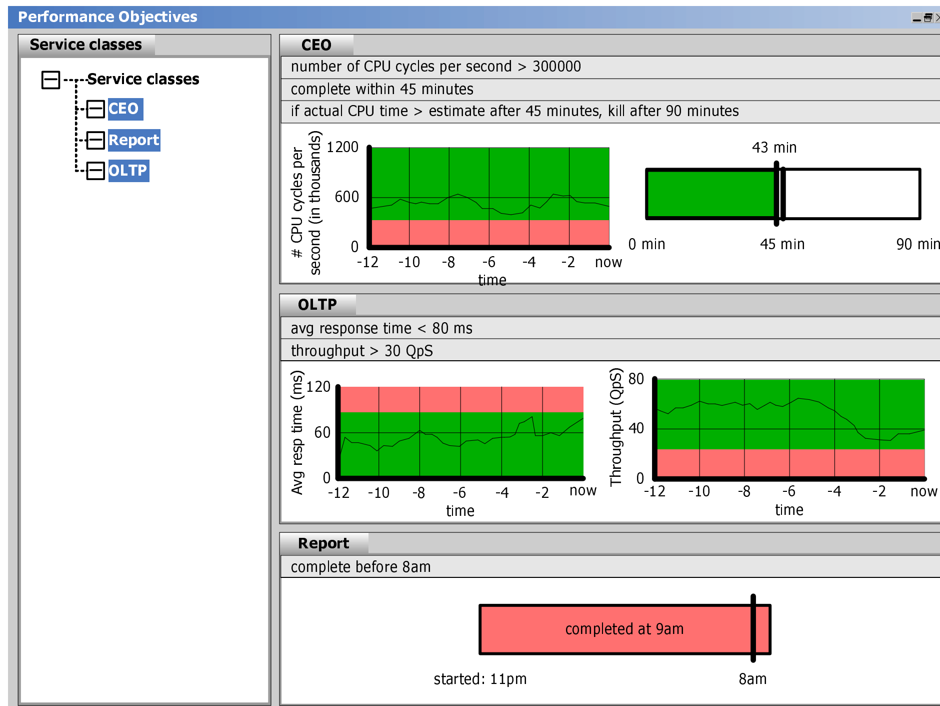


Figure 5.19.: Performance objective screen after a run has completed.

objectives, the report query batch running and on-track to meet its objective, and no ad hoc CEO queries in the system. Then an ad hoc CEO query arrives, is admitted, and starts executing. After a brief delay, system performance degrades and the OLTP requests are in danger of not meeting their objectives, as can be seen in the performance objective window in Figure 5.17, which shows how well each service class is meeting its objectives. Multiple CEO queries may execute simultaneously (causing even more contention), while there may be no such queries in the system at other times. The workload management policies and thresholds may need adjusting with each increase or decrease in these queries: resources that are reserved for these queries may be wasted when none are executing.

5.7.2. Attempt at manual correction

For a manual adjustment of the policies, an administrator can diagnose the situation in the resource utilization window (Figure 5.18) that shows excess resource utilization (memory and CPU) by the CEO queries. Note that diagnosing the cause of degraded performance is itself a challenge – it is not always obvious what or where the problem is.

There are multiple possible effective policy changes to reduce contention. For example, reducing the scheduling threshold (MPL) for the batch report queries will

reduce resource contention. In addition, it may be necessary to kill or suspend some active report queries to achieve the new, lower, threshold. What makes this task particularly difficult is that re-allocating resources can have unexpected effects. For example, slowing down a long-running query, e. g., by lowering its priority, may mean that it continues to occupy system resources for a longer time. The administrator can use our interface to adjust various thresholds and policies and see the impact of their actions on system performance and service objectives.

In this simple example, one option is to suspend the report class temporarily to enable the CEO class to complete. Another option is to reduce the priority of the OLTP class. However, the best strategy is actually to reduce the scheduling MPL for the OLTP requests just a little (e. g., from 10 to 8 or 9) when the executive's ad hoc query starts executing. If the administrator lets the system load stay too high or else reduces the MPL too much, then the OLTP requests fail to meet their throughput objectives. If the administrator does not reduce the MPL enough, then the ad hoc query fails to meet its response time objective. If the human administrator takes too long to respond, then multiple objectives are missed.

In addition, as the CEO queries complete, the administrator should raise the MPL for the OLTP and/or report queries, so that resources are not left idle. After an administrator makes adjustments in the policy control window, the impact of those changes may be observed in the performance objective window. Figure 5.19 shows how the window might look at the end of a run. In the figure, one can see that although the OLTP and CEO queries did meet their throughput and response time objectives, the administrator was not aggressive enough in limiting the number of OLTP requests processed in parallel, and the report workload has missed its completion deadline.

5.7.3. Policy control feedback loop

After the administrator corrects the policies during execution of an entire workload, our demo replays the workload with our policy controller automatically adjusting the policies. The same GUI interfaces then show the administrator's actions and their effects side-by-side with those of the policy controller. At the end of the workload, both are scored based on their ability to meet the workload objectives.

5.8. Conclusions and future work

One major challenge of managing mixed workloads is that it is difficult to allocate resources amongst different classes of diverse queries when each class of queries has its own unique set of performance objectives. This chapter addressed this challenge by modeling the solution as a search in a geometric space. This model lets us structure our search in such a way that we can reconcile how changes to control parameters affect the performance of diverse workload components. We use this model to propose a new algorithm for solving the general search problem. We describe the experimental

framework we built to evaluate our algorithms and the metrics we chose to quantify how well they work, and our experiments demonstrate promising results.

Our prototype implementation of the dashboard helps administrators understand the impact of policies on mixed workloads and provides some positive examples of how to set policies. In addition, our demonstration framework allows us to create new workload scenarios so that we can study how our policy controller adapts to unexpected situations. This helps us to devise better policies and meta-policies.

It is future work to devise an algorithm that can use any setting in the search space as starting point. The challenge is to detect in which direction to move to locate the acceptable region. A benefit would be that we can more easily use information that narrows the search space. For example, we can use the workload adaptation algorithm to find a starting point for our MsCoSearch algorithm.

6. Conclusions

Most workload management research has focused on simple, static workloads with a single user load. Only recently, more attention has been paid to more complex, mixed workloads that comprise multiple user loads, e. g., sets of queries with associated characteristics and service level objectives. Most of the research was driven by commercial vendors that used their workload management tools to control the complex workloads. However, even though administrators have tools that allow them to control mixed workloads, many problem remain unsolved. This thesis described some of main challenges in mixed workload management:

1. There is no common “workload management” terminology such as is needed to systematically approach such a complex topic.
2. There is a “mismatch” between the user side, where the performance expectations are formulated, and the database side that has no information about the performance expectations but where a workload manager controls the execution of the queries. Consequently, the administrator must not only map incoming queries to service classes, but also define the policies for the service classes in order to meet the objectives.
3. Defining the policies and the respective thresholds must be done in the presence of inaccurate and incomplete information about the workload. For example, even though the set of queries that must be started at a certain time is known, almost no information is available how these queries interact – especially when the queries have been submitted ad hoc.
4. Workloads are often dynamic, e. g., new user loads may start at unpredictable times or queries do not arrive at a constant rate. Note that even a scheduled user load may start or end unpredictably. The workload management policies must be adjusted as the workload changes.
5. Since workload management is an after-thought to the design of database systems, database systems do not provide a standardized interface workload managers could use for applying workload management. The lack of such an interface also impedes the implementation of generic workload management components, which could be used to run systematic workload management experiments with different database systems.

Although this list of workload management is not exhaustive, it illustrates the challenges currently faced in managing mixed workloads.

This thesis devised a workload management terminology that describes the different components of a mixed workload management framework. In particular, we identified three different control loops where workload management is applied: the query control loop, the policy control loop, and the business control loop. The terminology was the basis for discussing three workload management problems in managing mixed workloads:

First, we focused on dynamic prioritization of OLTP-style business transactions. A commonly used service objective defines constraints on the percentile of the response times of transactions submitted on behalf of a user and a deadline to avoid the starvation of transactions. The constraints on the percentile of the response time incur a penalty while the deadline enforces the execution of requests, i. e., avoids starvation. We devised a dynamic prioritization scheme which adaptively penalizes individual requests. In order to minimize the penalties incurred for violating the constraints on the percentile response times, the MEFI and KAFKA algorithms have been proposed. Our prototypical implementation comprises an SLO component that computes the penalty function and annotates the individual requests with the penalty information. It also includes a scheduler that queues the queries and reorders them in order to minimize the penalty incurred for violating the constraints on the percentile response time. Our experiments showed that with static prioritization, where the priority of a request is positively correlated with the penalty that is due for violating the percentile constraint, the service level conformance for "high priority" transactions exceeds the desired threshold at the cost of their lower-priority counterparts. With our dynamic prioritization, the objectives for more transactions are met and, thus, the incurred penalties are reduced by a factor of 2.

Second, we considered a workload with long-running queries that may negatively impact the performance of other queries. We built a taxonomy for these long-running queries based on how they impact the other queries in a mixed workload. We identified how to detect and handle long-running queries using query control loop policies under three scenarios: queries with inaccurate cost estimates, unobserved resource contention, and system overload. We described an experimental framework based on a simulated database engine to systematically evaluate the ability of existing workload management mechanisms to deal with the three scenarios. We used the framework to methodically explore the space of policy combinations with different workloads. We showed that inaccurate cost estimates cause admission control and scheduling to mistake good queries as problem queries and that execution control is needed to compensate for errors in admission control and scheduling. The experiments also demonstrated that in the presence of overload, absolute thresholds for the execution control policies are ineffective if the "wasted work" is considered. Our recommendation is to pair admission control, scheduling, and execution control policies. We showed how to set the thresholds for the different policies.

Third, we looked at managing mixed workloads where user loads have compound objectives. The challenge is how to configure workload management to allocate the system resources so that all objectives of all service classes are met. The allocation of system resources and therefore the performance of the service classes is done

by setting thresholds for the workload management policies. We formulated the problem as a search problem and devised a model to describe the solution space. Based on the model, we described an algorithm that automatically locates a setting in the operating envelope, i. e., that sets the thresholds for the workload management policies so that the performance objectives of the service classes are met. We used our extended experimental framework to compare our algorithm to an extended version of the workload adaptation algorithm that was devised to solve a similar problem with simple (non-compound) objectives. Our experiments showed that the extended algorithm needs information about dominant objectives, which may not be available a priori. The experiments also show that, although our algorithm takes longer to approach the operating envelope, it can find a control parameter setting in the envelope, if such a setting exists.

Although this thesis has addressed some of the workload management problems described above, there are still some open problems to be solved. A future research challenge is to devise a generic workload management interface so that a single workload manager could interface to database systems from different vendors. There could be a similar interface as the ODBC/JDBC interface that is used for sending requests to the database system. The iJDBC implementation that was used for experiments in Chapters 3 and 5, which supports generic admission control and scheduling, is a first step in the direction of generic workload management.

Another research topic would be to devise a benchmark that could be used for evaluating mixed workload management approaches. There are multiple challenges in defining such a benchmark: First, the workload of the benchmark must comprise different user loads where each user load has its own service level objectives. Second, the workload should be dynamic, e. g., it could define user loads starting and completing at different times, changes in the characteristics of the user loads (e. g., changes in the arrival rates of queries), and changes to the objectives of the user loads. In order to evaluate the workload management approaches for such a mixed workload, a metric is needed to quantify the goodness of the approaches. For example, the metric may not only consider how fast the performance converges towards an "acceptable" performance but also if the changes made to the system result in "smooth" performance changes or the performance values vary widely.

A. TPC-C

This chapter shows the DDL statements used for creating the TPC-C tables and indexes on MaxDB 7.5.00.26.

```
create table warehouse (
    w_id int not null,
    w_name varchar(10) not null,
    w_street_1 varchar(20) not null,
    w_street_2 varchar(20) not null,
    w_city varchar(20) not null,
    w_state char(2) not null,
    w_zip char(9) not null,
    w_tax numeric(4,4) not null,
    w_ytd numeric(12,2) not null,
    primary key (w_id))

create table district (
    d_id int not null,
    d_w_id int not null references warehouse(w_id),
    d_name varchar(10) not null,
    d_street_1 varchar(20) not null,
    d_street_2 varchar(20) not null,
    d_city varchar(20) not null,
    d_state char(2) not null,
    d_zip char(9) not null,
    d_tax numeric(4,4) not null,
    d_ytd numeric(12,2) not null,
    d_next_o_id int not null,
    primary key (d_w_id, d_id))

create table customer (
    c_id int not null,
    c_d_id int not null,
    c_w_id int not null,
    c_first varchar(16) not null,
    c_middle char(2) not null,
    c_last varchar(16) not null,
    c_street_1 varchar(20) not null,
    c_street_2 varchar(20) not null,
    c_city varchar(20) not null,
    c_state char(2) not null,
    c_zip char(9) not null,
    c_phone char(16) not null,
    c_since timestamp not null,
    c_credit char(2) not null,
    c_credit_lim numeric(12,2) not null,
    c_discount numeric(4,4) not null,
    c_balance numeric(12,2) not null,
    c_ytd_payment numeric(12,2) not null,
    c_payment_cnt numeric(4,0) not null,
    c_delivery_cnt numeric(4,0) not null,
    c_data varchar(500) not null,
    primary key (c_w_id, c_d_id, c_id),
    foreign key (c_d_id, c_w_id) references district(d_id, d_w_id))
```

```
create table history (
    h_c_id int not null,
    h_c_d_id int not null,
    h_c_w_id int not null,
    h_d_id int not null,
    h_w_id int not null,
    h_date timestamp not null,
    h_amount numeric(6,2) not null,
    h_data varchar(24) not null,
    foreign key (h_c_id, h_c_d_id, h_c_w_id) references
            customer(c_id, c_d_id, c_w_id))

create table orders (
    o_id int not null,
    o_d_id int not null,
    o_w_id int not null,
    o_c_id int not null,
    o_entry_d timestamp not null,
    o_carrier_id int,
    o_ol_cnt numeric(2,0) not null,
    o_all_local numeric(1,0) not null,
    primary key (o_w_id, o_d_id, o_id),
    foreign key (o_c_id, o_d_id, o_w_id) references
            customer (c_id, c_d_id, c_w_id))

create table neworder (
    no_o_id int not null,
    no_d_id int not null,
    no_w_id int not null,
    primary key (no_w_id, no_d_id, no_o_id),
    foreign key (no_o_id, no_w_id, no_d_id) references
            orders(o_id, o_w_id, o_d_id))

create table stock (
    s_i_id int not null,
    s_w_id int not null references warehouse(w_id),
    s_quantity numeric(4,0) not null,
    s_dist_01 char(24) not null,
    s_dist_02 char(24) not null,
    s_dist_03 char(24) not null,
    s_dist_04 char(24) not null,
    s_dist_05 char(24) not null,
    s_dist_06 char(24) not null,
    s_dist_07 char(24) not null,
    s_dist_08 char(24) not null,
    s_dist_09 char(24) not null,
    s_dist_10 char(24) not null,
    s_ytd numeric(8,0) not null,
    s_order_cnt numeric(4,0) not null,
    s_remote_cnt numeric(4,0) not null,
    s_data varchar(50) not null,
    primary key (s_w_id, s_i_id))

create table orderline (
    ol_o_id int not null,
    ol_d_id int not null,
    ol_w_id int not null,
    ol_number int not null,
    ol_i_id int not null,
    ol_supply_w_id int not null,
    ol_delivery_d timestamp,
```

```
    ol_quantity numeric(2,0) not null,
    ol_amount numeric(6,2) not null,
    ol_dist_info char(24) not null,
    primary key (ol_w_id, ol_d_id, ol_o_id, ol_number),
    foreign key (ol_o_id, ol_w_id, ol_d_id) references
            orders(o_id, o_w_id, o_d_id),
    foreign key (ol_i_id, ol_supply_w_id) references
            stock(s_i_id, s_w_id))

create table item (
    i_id int not null,
    i_im_id int not null,
    i_name varchar(24) not null,
    i_price numeric(5,2) not null,
    i_data varchar(50) not null,
    primary key (i_id))

create index cust_index on customer (c_last, c_d_id, c_w_id)

create index order_index on orders (o_w_id, o_d_id, o_c_id)

create index stock_index on stock (s_quantity, s_w_id)
```

B. TPC-CH benchmark

This chapter gives details on the schema of the TPC-CH benchmark and the SQL statements of the OLAP query suite of the TPC-CH benchmark. [22] gives more information on the benchmark.

B.1. DDL for TPC-CH

Figure B.1 shows the schema of the TPC-CH benchmark. The tables originating from TPC-H (*region*, *nation*, and *supplier*) are shown as bold rectangles, all other tables originate from TPC-C. TPC-CH keeps the TPC-C entities and relationships unchanged. The arrows in the figure denote the foreign keys constraints. The number below the names of the tables indicate the number of entries for each table. As specified in TPC-C, the size of some tables depends on the number of warehouses W The solid arrows denote the foreign keys that are enforced by the DDL statements (we reused the statements in Appendix A and below). The dashed arrows indicate "implicit" foreign keys to model the relationships between *customer* (originating from TPC-C) and *nation* (TPC-H), and *stock* (TPC-C) and *supplier* (TPC-H). In TPC-CH, a *customer*'s nation is identified by the first character of the field *c_state*. TPC-C specifies that this character can have 62 different values (upper-case and lower-case letters, and digits). Consequently, we populated table *nation* with 62 entries. The primary key *n_nationkey* is an identifier according to the TPC-H specification. Its values are chosen such that their associated ASCII value is greater is a letter or digit: $n_nationkey \in [48, 57] \cup [65, 90] \cup [97, 122]$. An entry in *stock* is uniquely associated with one of the 10000 entries in *supplier* through the relationship $stock.s_i_id \cdot stock.s_w_id \mod 10000 = supplier.su_suppkey$.

Since TPC-CH was designed to leave the TPC-C schema unchanged, we used the DDL statements in Appendix A to load the "TPC-C-part" of the TPC-CH schema. The tables from TPC-C are extended with the likewise unchanged relations from the TPC-H benchmark. The DDL statements for creating the "TPC-H-part" of the TPC-CH schema on a DB2 9.5 database system are shown below.

```
create table region (
    r_regionkey int not null,
    r_name char(55) not null,
    r_comment char(152) not null,
    primary key (r_regionkey))

create table nation (
    n_nationkey int not null,
    n_name char(25) not null,
    n_regionkey int not null references region(r_regionkey),
```

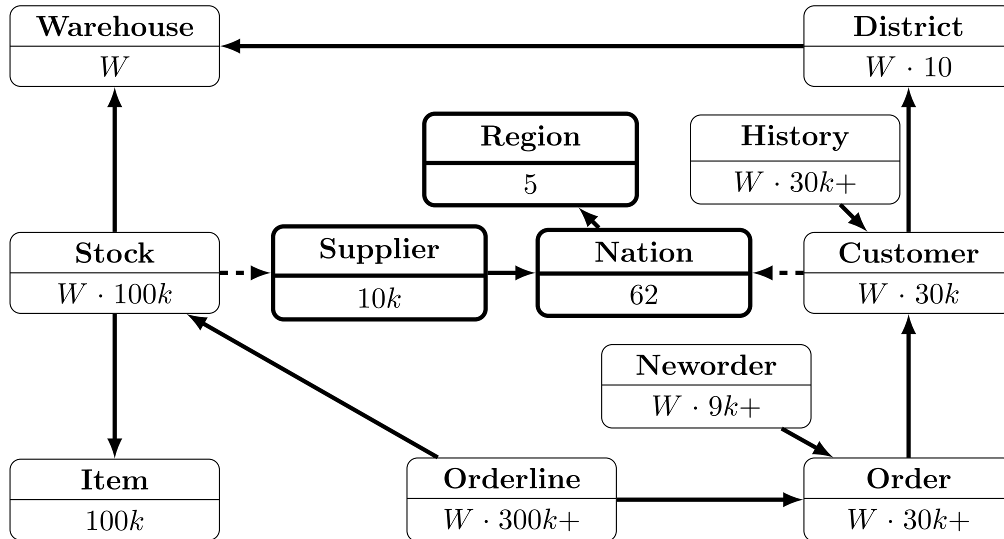


Figure B.1.: Schema of the TPC-CH benchmark

```
    n_comment char(152) not null,
    primary key (n_nationkey))

create table supplier (
    su_suppkey int not null,
    su_name char(25) not null,
    su_address varchar(40) not null,
    su_nationkey int not null references nation(n_nationkey),
    su_phone char(15) not null,
    su_acctbal numeric(12,2) not null,
    su_comment char(101) not null,
    primary key (su_suppkey))
```

B.2. OLAP query suite in the TPC-CH benchmark

In the following, we show the OLAP query suite used in the experiments on a DB2 9.5 database system.

Q1: Generate orderline overview

```
select ol_number,
       sum(ol_quantity) as sum_qty,
       sum(ol_amount) as sum_amount,
       avg(ol_quantity) as avg_qty,
       avg(ol_amount) as avg_amount,
       count(*) as count_order
from orderline where ol_delivery_d > '2007-01-02␣00:00:00.000000'
group by ol_number order by ol_number
```

Q2: Most important supplier/item-combinations (those that have the lowest stock level for certain parts in a certain region)

```
select su_suppkey,
       su_name,
       n_name,
       i_id,
       i_name,
       su_address,
       su_phone,
       su_comment
from item, supplier, stock, nation, region,
     (select s_i_id as m_i_id,min(s_quantity) as m_s_quantity
      from stock, supplier, nation, region
      where mod((s_w_id*s_i_id),10000)=su_suppkey
      and su_nationkey=n_nationkey
      and n_regionkey=r_regionkey
      and r_name like 'Europ%'
      group by s_i_id) m
where i_id = s_i_id
and mod((s_w_id * s_i_id), 10000) = su_suppkey
and su_nationkey = n_nationkey
and n_regionkey = r_regionkey
and i_data like '%b'
and r_name like 'Europ%'
and i_id=m_i_id
and s_quantity = m_s_quantity
order by n_name, su_name, i_id
```

Q3: Unshipped orders with highest value for customers within a certain state

```
select ol_o_id,
       ol_w_id, ol_d_id,
       sum(ol_amount) as revenue,
       o_entry_d
from customer, neworder, orders, orderline
where c_state like 'A%'
and c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and no_w_id = o_w_id
and no_d_id = o_d_id
and no_o_id = o_id
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and o_entry_d > '2007-01-02 00:00:00.000000'
group by ol_o_id, ol_w_id, ol_d_id, o_entry_d
order by revenue desc, o_entry_d
```

Q4: Orders that were partially shipped late

```
select o_ol_cnt, count(*) as order_count
from orders
where o_entry_d >= '2007-01-02 00:00:00.000000'
and o_entry_d < '2012-01-02 00:00:00.000000'
and exists (select *
            from orderline
            where o_id = ol_o_id
            and o_w_id = ol_w_id
            and o_d_id = ol_d_id
```

```
            and ol_delivery_d >= o_entry_d)
group by o_ol_cnt
order by o_ol_cnt
```

Q5: Revenue volume achieved through local suppliers

```
select n_name, sum(ol_amount) as revenue
from customer, orders, orderline, stock, supplier, nation, region
where c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and ol_o_id = o_id
and ol_w_id = o_w_id
and ol_d_id=o_d_id
and ol_w_id = s_w_id
and ol_i_id = s_i_id
and mod((s_w_id * s_i_id),10000) = su_suppkey
and ascii(substr(c_state,1,1)) = su_nationkey
and su_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'Europe'
and o_entry_d >= '2007-01-02 00:00:00.000000'
group by n_name
order by revenue desc
```

Q6: Revenue generated by orderlines of a certain quantity

```
select sum(ol_amount) as revenue
from orderline
where ol_delivery_d >= '1999-01-01 00:00:00.000000'
and ol_delivery_d < '2020-01-01 00:00:00.000000'
and ol_quantity between 1 and 100000
```

Q7: Bi-directional trade volume between two nations

```
select su_nationkey as supp_nation,
       substr(c_state,1,1) as cust_nation,
       year(o_entry_d) as l_year,
       sum(ol_amount) as revenue
from supplier, stock, orderline, orders, customer, nation n1, nation n2
where ol_supply_w_id = s_w_id
and ol_i_id = s_i_id
and mod((s_w_id * s_i_id), 10000) = su_suppkey
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and su_nationkey = n1.n_nationkey
and ascii(substr(c_state,1,1)) = n2.n_nationkey
and (
     (n1.n_name = 'Germany' and n2.n_name = 'Cambodia')
     or
     (n1.n_name = 'Cambodia' and n2.n_name = 'Germany'))
and ol_delivery_d between '2007-01-02 00:00:00.000000' and
                          '2012-01-02 00:00:00.000000'
group by su_nationkey, substr(c_state,1,1), year(o_entry_d)
order by su_nationkey, cust_nation, l_year
```

Q8: Market share of a given nation for customers of a given region for a given part type

```
select year(o_entry_d) as l_year, sum(case
                                        when n2.n_name = 'Germany'
                                            then ol_amount
                                        else 0
                                    end) / sum(ol_amount) as mkt_share
from item, supplier, stock, orderline, orders, customer,
     nation n1, nation n2, region
where i_id = s_i_id
and ol_i_id = s_i_id
and ol_supply_w_id = s_w_id
and mod((s_w_id * s_i_id),10000) = su_suppkey
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and n1.n_nationkey = ascii(substr(c_state,1,1))
and n1.n_regionkey = r_regionkey
and ol_i_id < 1000
and r_name = 'Europe'
and su_nationkey = n2.n_nationkey
and o_entry_d between '2007-01-02␣00:00:00.000000' and
                      '2012-01-02␣00:00:00.000000'
and i_data like '%b'
and i_id = ol_i_id
group by year(o_entry_d)
order by l_year
```

Q9: Profit made on a given line of parts, broken out by supplier nation and year

```
select n_name, year(o_entry_d) as l_year, sum(ol_amount) as sum_profit
from item, stock, supplier, orderline, orders, nation
where ol_i_id = s_i_id
and ol_supply_w_id = s_w_id
and mod((s_w_id * s_i_id), 10000) = su_suppkey
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and ol_i_id = i_id
and su_nationkey = n_nationkey
and i_data like '%BB'
group by n_name, year(o_entry_d)
order by n_name, l_year desc
```

Q10: Customers who received their ordered products late

```
select c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone, n_name
from customer, orders, orderline, nation
where c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and o_entry_d >= '2007-01-02␣00:00:00.000000'
and o_entry_d <= ol_delivery_d
and n_nationkey = ascii(substr(c_state,1,1))
```

```
group by c_id, c_last, c_city, c_phone, n_name
order by revenue desc
```

Q11: Most important (high order count compared to the sum of all order counts) parts supplied by suppliers of a particular nation

```
select s_i_id, sum(s_order_cnt) as ordercount
from stock, supplier, nation
where mod((s_w_id * s_i_id),10000) = su_suppkey
and su_nationkey = n_nationkey
and n_name = 'Germany'
group by s_i_id
having sum(s_order_cnt) > (select sum(s_order_cnt) * .005
                           from stock, supplier, nation
                           where mod((s_w_id * s_i_id),10000) = su_suppkey
                           and su_nationkey = n_nationkey
                           and n_name = 'Germany')
order by ordercount desc
```

Q12: Determine whether selecting less expensive modes of shipping is negatively affecting the critical-priority orders by causing more parts to be received late by customers

```
select o_ol_cnt,
       sum(case
               when o_carrier_id = 1 or o_carrier_id = 2 then 1
               else 0
           end) as high_line_count,
       sum(case
               when o_carrier_id <> 1 and o_carrier_id <> 2 then 1
               else 0
           end) as low_line_count
from orders, orderline
where ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and o_entry_d <= ol_delivery_d
and ol_delivery_d < '2020-01-01 00:00:00.000000'
group by o_ol_cnt
order by o_ol_cnt
```

Q13: Relationships between customers and the size of their orders

```
select c_count, count(*) as custdist
from (select c_id, count(o_id)
      from customer left outer join orders on (
           c_w_id = o_w_id
           and c_d_id = o_d_id
           and c_id = o_c_id
           and o_carrier_id > 8)
      group by c_id) as c_orders (c_id, c_count)
group by c_count
order by custdist desc, c_count desc
```

Q14: Market response to a promotion campaign

```
select 100.00 * sum(case
                        when i_data like 'PR%' then ol_amount
                        else 0
```

```
                    end) / 1+sum(ol_amount) as promo_revenue
from orderline, item
where ol_i_id = i_id and ol_delivery_d >= '2007-01-02␣00:00:00.000000'
and ol_delivery_d < '2020-01-02␣00:00:00.000000'

\subsubsection{Q15: Determine the top supplier}

with revenue (supplier_no, total_revenue) as (
    select mod((s_w_id * s_i_id),10000) as supplier_no,
           sum(ol_amount) as total_revenue
    from orderline, stock
    where ol_i_id = s_i_id and ol_supply_w_id = s_w_id
    and ol_delivery_d >= '2007-01-02␣00:00:00.000000'
    group by mod((s_w_id * s_i_id),10000))
select su_suppkey, su_name, su_address, su_phone, total_revenue
from supplier, revenue
where su_suppkey = supplier_no
and total_revenue = (select max(total_revenue) from revenue)
order by su_suppkey
```

Q16: Number of suppliers that can supply parts with given attributes

```
select i_name,
       substr(i_data, 1, 3) as brand,
       i_price,
       count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
from stock, item
where i_id = s_i_id
and i_data not like 'zz%'
and (mod((s_w_id * s_i_id),10000)) not in (select su_suppkey
                                           from supplier
                                           where su_comment like '%bad%')
group by i_name, substr(i_data, 1, 3), i_price
order by supplier_cnt desc
```

Q17: Average yearly revenue that would be lost if orders were no longer filled for small quantities of certain parts

```
select sum(ol_amount) / 2.0 as avg_yearly
from orderline, (select i_id, avg(ol_quantity) as a
                 from item, orderline
                 where i_data like '%b'
                 and ol_i_id = i_id
                 group by i_id) t
where ol_i_id = t.i_id
and ol_quantity < t.a
```

Q18: Rank customers based on their placement of a large quantity order

```
select c_last, c_id o_id, o_entry_d, o_ol_cnt, sum(ol_amount)
from customer, orders, orderline
where c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o_ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, o_entry_d
```

Q19: Machine generated data mining (revenue report for disjunctive predicate)

```
select sum(ol_amount) as revenue
from orderline, item
where ( ol_i_id = i_id
        and i_data like '%a'
        and ol_quantity >= 1
        and ol_quantity <= 10
        and i_price between 1 and 400000
        and ol_w_id in (1,2,3))
or ( ol_i_id = i_id
     and i_data like '%b'
     and ol_quantity >= 1
     and ol_quantity <= 10
     and i_price between 1 and 400000
     and ol_w_id in (1,2,4))
or ( ol_i_id = i_id
     and i_data like '%c'
     and ol_quantity >= 1
     and ol_quantity <= 10
     and i_price between 1 and 400000
     and ol_w_id in (1,5,3))
```

Q20: Suppliers in a particular nation having selected parts that may be candidates for a promotional offer

```
select su_name, su_address
from supplier, nation
where su_suppkey in (select mod(s_i_id * s_w_id, 10000)
                     from stock, orderline
                     where s_i_id in (select i_id
                                      from item
                                      where i_data like 'co%')
                     and ol_i_id=s_i_id
                     and ol_delivery_d > '2010-05-23␣12:00:00'
                     group by s_i_id, s_w_id, s_quantity
                     having 2*s_quantity > sum(ol_quantity)  )
and su_nationkey = n_nationkey
and n_name = 'Germany'
order by su_name
```

Q21: Suppliers who were not able to ship required parts in a timely manner

```
select su_name, count(*) as numwait
from supplier, orderline l1, orders, stock, nation
where ol_o_id = o_id
and ol_w_id = o_w_id
and ol_d_id=o_d_id
and ol_w_id = s_w_id
and ol_i_id = s_i_id
and mod((s_w_id * s_i_id),10000) = su_suppkey
and l1.ol_delivery_d > o_entry_d
and not exists (select *
               from orderline l2
               where l2.ol_o_id = l1.ol_o_id
               and l2.ol_w_id = l1.ol_w_id
               and l2.ol_d_id = l1.ol_d_id
               and l2.ol_delivery_d > l1.ol_delivery_d)
and su_nationkey = n_nationkey
and n_name = 'Germany'
group by su_name
```

```
order by numwait desc, su_name
```

Q22: Geographies with customers who may be likely to make a purchase

The workload TPC-CH of the unmodified TPC-C transactions. However, the TPC-H queries had to be modified in order to be executed on the TPC-CH schema. The SQL statements for the TPC-H-like queries is shown below.

```
select substr(c_state,1,1) as country,
       count(*) as numcust,
       sum(c_balance) as totacctbal
from   customer
where substr(c_phone,1,1) in ('1','2','3','4','5','6','7')
and c_balance > (select avg(c_BALANCE)
                 from customer
                 where c_balance > 0.00
                 and substr(c_phone,1,1) in ('1','2','3','4','5','6','7'))
and not exists (select *
                from orders
                where o_c_id = c_id
                and o_w_id = c_w_id
                and o_d_id = c_d_id)
group by substr(c_state,1,1)
order by substr(c_state,1,1)
```

Bibliography

[1] Mohammed Abouzour, Kenneth Salem, and Peter Bumbulis. Automatic Tuning of the Multiprogramming Level in Sybase SQL Anywhere. In *Proc. of the 2010 Workshop on Self-Managing Database Systems (SMDB)*, 2010.

[2] Alain Andrieux, Kark Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). http://www.ogf.org/documents/GFD.107.pdf, 2007.

[3] Jim Basney. *Network and CPU Co-Allocation in High Throughput Computing Environments*. PhD thesis, University of Wisconsin-Madison, 2001.

[4] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying Business Processes with BP-QL. In *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, 2005.

[5] Darcy G. Benoit. Automated Diagnosis and Control of DBMS Resources. In *EDBT PhD. Workshop*, 2000.

[6] Reinhard Braumandl, Alfons Kemper, and Donald Kossmann. Quality of Service in an Information Economy. *TOIT*, 3(4), 2003.

[7] Kurt Brown, Manish Mehta, Michael Carey, and Miron Livny. Towards Automated Performance Tuning for Complex Workloads. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases (VLDB)*, 1994.

[8] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *Proc. of the 19th Intl. Conf. on Very Large Data Bases (VLDB)*, 1993.

[9] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-Oriented Buffer Management Revisited. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1996.

[10] Michael J. Carey, Rajiv Jauhari, and Miron Livny. Priority in DBMS Resource Scheduling. In *Proc. of the 15th Intl. Conf. on Very Large Data Bases (VLDB)*, 1989.

[11] Badrish Chandramouli, Christopher Bond, Shivnath Babu, and Jun Yang. Query Suspend and Resume. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2007.

[12] Surajit Chaudhuri, Raghav Kaushik, Abhijit Pol, and Ravi Ramamurthy. Stop-and-restart Style Execution for Long Running Decision Support Queries. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases (VLDB)*, 2007.

[13] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. When Can We Trust Progress Estimators for SQL Queries? In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2005.

[14] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating Progress of Execution for SQL Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2004.

[15] Chandra Chekuri and Sanjeev Khanna (edited by Joseph Leung). *Approximation Algorithms for Minimizing Average Weighted Completion Time*, chapter 11, pages 1–30. CRC Press, 2004.

[16] Ken Chen and Paul Muhlethaler. A Scheduling Algorithm for Tasks Described by Time Value Function. *Real-Time Syst.*, 10(3), 1996.

[17] Whei-Jen Chen, Bill Comeau, Tomoko Ichikawa, S Sadish Kumar, Marcia Miskimen, H T Morgan, Larry Pay, and Tapio Väättänen. Workload Manager for Linux, Unix, and Windows. http://www.redbooks.ibm.com/redbooks/pdfs/sg247524.pdf.

[18] Sunil Choenni, Martin Kersten, and Johan van den Akker. A Framework for Multi-query Optimization. In *Proc. of the 8th Intl. Conf. on Management of Data (COMAD)*, 1997.

[19] Allen L. Edwards. *An Introduction to Linear Regression and Correlation.* W. H. Freeman & Co Ltd, 1976.

[20] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-commerce Web Sites. In *Proc. of the 13th Intl. World Wide Web Conf. (WWW)*, 2004.

[21] Marshall L. Fisher and Abba M. Krieger. Analysis of a Linearization Heuristic for Single-Machine Scheduling to Maximize Profit. *Mathematical Programming*, 28:218–225, 1984.

[22] Florian Funke, Alfons Kemper, and Thomas Neumann. Benchmarking Hybrid OLTP&OLAP Database Systems. In *Database Systems for Business, Technology and Web (BTW)*, 2011.

[23] Nicolas Gibelin and Mesaac Makpangou. Efficient and Transparent Web-Services Selection. In *Proceedings of the 3rd International Conference on Service Oriented Computing*, 2005.

[24] Daniel Gmach, Stefan Krompass, Andreas Scholz, Martin Wimmer, and Alfons Kemper. Adaptive quality of service management for enterprise services. *ACM Transactions on the Web (TWEB)*, 2(1), 2008.

[25] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[26] Greenplum Database 4.0 Administrator Guide. http://gpn.greenplum.com/, 2010.

[27] HP Neoview Workload Management Services Guide, April 2008.

[28] IBM Query Patroller Administration and User's Guide. `ftp://ftp.software.ibm.com/ps/products/db2/info/vr9/pdf/letter/en_US/db2dwe90.pdf`, 2006.

[29] David G. Kleinbaum, Lawrence L. Kupper, and Keith E. Muller, editors. *Applied Regression Analysis and Other Multivariable Methods.* PWS Publishing Co., Boston, MA, USA, 1988.

[30] Achim Kraiss, F. Schön, Gerhard Weikum, and Uwe Deppisch. Towards Response Time Guarantees for E-Service Middleware. *IEEE Data Engineering Bulletin*, 24(1), 2001.

[31] Achim Kraiss, Frank Schön, Gerhard Weikum, and Uwe Deppisch. With HEART Towards Response Time Guarantees for Message-Based E-Services. In *Proceedings of the 8th International Conference on Extending Database Technology*, 2002.

[32] Stefan Krompass, Daniel Gmach, Andreas Scholz, Stefan Seltzsam, and Alfons Kemper. Quality of Service Enabled Database Applications. In *Proc. of the 4th Intl. Conf. on Service-Oriented Computing (ICSOC)*, 2006.

[33] Stefan Krompass, Harumi Kuno, Janet Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. A Testbed for Managing Dynamic Mixed Workloads. In *Proc. of the 35th Intl. Conf. on Very Large Data Bases (VLDB)*, 2009.

[34] Stefan Krompass, Harumi Kuno, Janet Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Managing Long-running Queries. In *Proc. of the 12th Intl. Conf. on Extending Database Technology (EDBT)*, 2009.

[35] Stefan Krompass, Harumi Kuno, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Adaptive Query Scheduling for Mixed Database Workloads with Multiple Objectives. In *Proc. of the 3rd Intl. Workshop on Testing Database Systems (DBTest)*, 2010.

[36] Stefan Krompass, Andreas Scholz, Martina-Cezara Albutiu, Harumi Kuno, Janet Wiener, Umeshwar Dayal, and Alfons Kemper. Quality of Service-Enabled

Management of Database Workloads. *IEEE Data Engineering Bulletin*, 31(1), 2008.

[37] Jan Karel Lenstra, A.H.G. Rinnooy Kan, and Peter Brucker. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[38] Wen-Syan Li, Vishal S. Batra, Vijashankar Raman, Wei Han, and Inderpal Narang. QoS-based Data Access and Placement for Federated Information Systems. In *Proc. of the 31st Intl. Conf. on Very Large Data Bases (VLDB)*, 2005.

[39] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Increasing the Accuracy and Coverage of SQL Progress Indicators. In *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE)*, 2005.

[40] Gang Luo, Jeffrey F. Naughton, and Philip S. Yu. Multi-query SQL Progress Indicators. In *Proc. of the 10th Intl. Conf. on Extending Database Technology (EDBT)*, 2006.

[41] MaxDB. `http://www.mysql.com/products/maxdb/`.

[42] Michael Maximilien and Munindar P. Singh. Toward Autonomic Web Services Trust and Selection. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.

[43] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proc. of the 20th Intl. Conf. on Data Engineering (ICDE)*, 2004.

[44] Microsoft SQL Server 2005 Books Online. `http://msdn2.microsoft.com/en-us/library/ms190419.aspx`, September 2007.

[45] Managing SQL Server Workloads with Resource Governor. `http://msdn.microsoft.com/en-us/library/bb933866.aspx`, December 2008.

[46] Baoning Niu. *Workload Adaptation in Autonomic Database Management Systems*. PhD thesis, Queen's University, Kingston, Ontario, Canada, 2008.

[47] Oracle Database Resource Manager. `http://download.oracle.com/docs/cd/B28359_01/server.111/b28310/dbrm.htm`, March 2008.

[48] Raghunath Othayoth and Meikel Poess. The Making of TPC-DS. In *Proc. of the 32nd Intl. Conf. on Very Large Data Bases (VLDB)*, 2006.

[49] HweeHwa Pang, Michael J. Carey, and Miron Livny. Multiclass Query Scheduling in Real-Time Database Systems. *IEEE Trans. on Knowledge and Data Engineering*, 7(4), 1995.

[50] Wendy Powley, Patrick Martin, Mingyi Zhang, Paul Bird, and Keith McDonald. Autonomic Workload Execution Control Using Throttling. In *Proc. of the 2010 Workshop on Self-Managing Database Systems (SMDB)*, 2010.

[51] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The Oracle Database Resource Manager: Scheduling CPU Resources at the Application Level. http://research.microsoft.com/~jamesrh/hpts2001/submissions/, 2001.

[52] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, and Erich M. Nahum. Achieving Class-Based QoS for Transactional Workloads. In *Proc. of the 22nd Intl. Conf. on Data Engineering (ICDE)*, 2006.

[53] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich M. Nahum, and Adam Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *Proc. of the 22nd Intl. Conf. on Data Engineering (ICDE)*, 2006.

[54] Herb Schwetman. CSIM19: A Powerful Tool for Building System Models. In *Proc. of the 33rd Winter Simulation Conference (WSC)*, 2001.

[55] Stefan Seltzsam, Daniel Gmach, Stefan Krompass, and Alfons Kemper. AutoGlobe: An Automatic Administration Concept for Service-Oriented Database Applications. In *Proc. of the 22nd Intl. Conf. on Data Engineering (ICDE)*, 2006.

[56] Subbu Subramanian and Shivakumar Venkataraman. Cost-based Optimization of Decision Support Queries Using Transient-views. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1998.

[57] Teradata Dynamic Workload Manager User Guide (Release 13.0.0.0). http://www.info.teradata.com/eDownload.cfm?itemid=082330034, 2008.

[58] Maik Thiele, Andreas Bader, and Wolfgang Lehner. Multi-objective Scheduling for Real-time Data Warehouses. In *Database Systems for Business, Technologie und Web (BTW)*, 2009.

[59] TPC Benchmark C – Standard Specification, Revision 5.4. http://www.tpc.org/tpcc.

[60] Yi-Cheng Tu, Sunil Prabhakar, Ahmed Elmagarmid, and Radu Sion. QuaSAQ: An Approach to Enabling End-to-End QoS for Multimedia Databases. In *Advances in Database Technology, 9th International Conference on Extending Database Technology*, 2004.

[61] Jinggang Wang and Binoy Ravindran. Time-Utility Function-Driven Switched Ethernet: Packet Scheduling Algorithm, Implementation, and Feasibility Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 15(2), 2004.

[62] Gerhard Weikum, Christof Hasse, Axel Mönkeberg, and Peter Zabback. The COMFORT Automatic Tuning Project. *Information Systems*, 19(5):381–432, 1994.

[63] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases (VLDB)*, 2007.