Dynamic Workload Management for Very Large Data Warehouses – Juggling Feathers and Bowling Balls

Stefan Krompass[†] Harumi Kuno[§] Umeshwar Dayal[§] Alfons Kemper[†]

[†]TU München
D-85748 Garching, Germany

[§]HP Labs
Palo Alto, CA, USA

ABSTRACT

Workload management for business intelligence (BI) queries poses different challenges than those addressed in the online transaction processing (OLTP) context. The fundamental problem is that the execution times of BI queries can range from milliseconds to hours, and it is difficult to estimate these times accurately. Key challenges raised by this problem are how to identify queries that are not performing properly and what to do about them.

We propose here a workload management system for controlling the execution of individual queries based on realistic customer service level objectives. In order to validate our proposal, we have implemented an experimental system that includes a dynamic execution controller that leverages fuzzy logic. We present results from a number of experiments that we ran using workloads based on actual industrial workloads and customer objectives that we gathered by interviewing industry practitioners.

Our experiments show that even a handful of moderately mis-behaving problem queries can have a significant impact on a workload consisting of thousands of queries. We were surprised when our experiments also demonstrated that false positives – incorrectly identifying a normal query as a problem – can also have significant consequences. For those reasons, it is very important that an execution controller be as accurate as possible – avoiding both false positives and false negatives. Our experiments also validate that our execution controller can markedly improve the execution of a workload that includes problem queries.

1. INTRODUCTION

Workload management of a very large data warehouse is like juggling thousands of opaque bags between hundreds of workers while sporadically the CEO of the company walks by and throws in a new bag, which then needs preferential treatment. Most of the bags contain feathers, some contain baseballs, and a few turn out to contain bowling balls. Heavy bags can slow a worker's juggling speed, making processing times unpredictable. There are financial consequences if various bags are not processed within given periods of time, but the consequences and deadlines are not visible to the workers.

Our goal is to enable the automatic scheduling and management of such a system. We believe that the key challenges that we must address for this are: first, we must be able to translate customer expectations into service level objectives; second, we must have strategies for scheduling jobs with uncertain resource requirements; and third, we must be able to accurately detect and handle "problem queries" – improperly functioning queries that may not complete and that may consume resources that could otherwise be used by properly functioning queries.

The very reasons that make workload management particularly relevant to data warehouses also make it especially challenging in that context. BI queries exhibit a huge variance in response times. Most queries are known to execute in under a minute, but some small number of them require hours of execution time. According to conventional wisdom, the presence of even a few poorly written or optimized queries can significantly impact the performance of a data warehouse system by taking up resources that could otherwise be used by properly functioning queries. It is not straightforward to estimate accurately how long a long-running query will take. Although customers may have service level agreements (SLAs) that spell out financial consequences, and although jobs have deadlines, the deadlines are not necessarily explicit, nor is it straightforward (or even necessarily possible) to link SLAs to deadlines.

Database administrators today thus struggle with questions like:

- How long should they wait before killing an unexpectedly long-running query?
- When should they run a newly arrived interactive query if the currently executing batch of queries is in danger of missing its deadline?
- What if the newly arrived interactive query was submitted by their CEO?

We believe that it is critical that workload management automate such decisions. Analysts currently predict that data warehouses are moving from the current scenario of scheduled batched workloads with hundreds of standard reports and limited numbers of interactive queries, towards a future scenario of mixed workloads characterized by continuous loading, thousands (or tens of thousands) of standard reports, and thousands of users [7].

There are three main components to our approach. First, we categorize workload types according to their service level objectives (SLO). We do this because these objectives dictate how the workloads should be treated. Second, we recognize the difficulty of estimating the execution times of BI queries in a multi-stream environment, and have considered how scheduling could accommodate this uncertainty in the face of service level objectives. Third, in order to identify and

control the execution of problem queries (queries that run much longer than expected), we recognized that our execution manager needs a flexible mechanism for identifying and handling problem queries. To this end, we have built an execution management component that leverages fuzzy logic to incorporate both quantitative measurements and heuristics in an intuitive manner.

We begin by discussing related research efforts and available commercial offerings in Section 2. We have implemented a prototype system for the purpose of validating policies and strategies for workload management. We describe the components of our approach and implementation in Section 3. Because to the best of our knowledge, service level agreements for data warehouses do not address query-level requirements, we interviewed a number of practitioners and distilled a characterization of customer expectations for BI workloads and how they map to job-specific penalties in Section 3.1. Our system features an execution controller that leverages fuzzy logic to provide an intuitive and flexible embodiment of management rules, which we describe in Section 4. In order to validate the effectiveness of our system, we used our prototype to test a variety of workload management strategies based on real workloads. We discuss this effort in Section 5. Finally, we present our conclusions, as well as ongoing work, in Section 6.

2. RELATED WORK

In this section, we consider two main areas of related work: (1) researchers who use workload management to address quality of service (QoS) in database systems (DBMS) and (2) efforts to address problem queries in DBMS workloads.

Most prior work in workload management has considered service-level performance objectives in the context of the OLTP, as opposed to BI, DBMS. Krompass et al. [9] present an adaptive QoS management that is based on an economic model which adaptively penalizes individual requests. Their model derives adaptive penalties for individual requests by differentiating between opportunity costs for underachieving an SLA threshold and marginal gains for (re-)achieving an SLA threshold. Their system includes a database component that schedules requests depending on their deadline and their associated penalty. Schroeder et al. [18] present a framework for providing QoS where the response time requirements are specified in an SLA. To meet the multiclass response time goals, the number of concurrently executing requests is dynamically adjusted using a feedback control loop which considers the available hardware resources and concurrently executing queries in the database.

If we think of SLOs as related to workload management techniques for resource allocation, we share a focus with researchers such as [9, 18, 6, 3, 13], who consider how to govern resource allocation for queries with widely varying resource requirements in multi-workload environments. For example, Davison and Graefe [6] present a framework for query scheduling and resource allocation that uses concepts from microeconomics to manage resource allocation.

A major difference between such work and ours is that we consider the case where some problem queries are not entitled to resources, and therefore in addition to admission control and scheduling, we also consider actions such as killing the problem queries. Also, their focus is OLTP, not BI, and thus they make assumptions such as transaction-specific SLAs, and do not consider workloads with a huge variance in uncertain execution times.

Most other researchers have considered problem queries as something to be prevented via resource tuning (e.g., page replacement algorithms) or addressed manually, as opposed to something to be dealt with via workload management policies. Benoit [1] presents a goal-oriented framework that models resource tuning parameters as a resource tree. Their model includes knowledge of how resource parameters impact database performance and how all resources are interrelated. The goal is to model DBMS resource usage for the purposes of (1) diagnosing problem resources and (2) determining how to adjust parameters in order to increase performance; they do not address the evaluation of workload management mechanisms or model the state of an individual query's execution. Weikum et al. [21] discuss what metrics are appropriate for signaling a performance problem. They focus on tuning decisions at different stages: system configuration, database configuration, application tuning, adjustment of operational parameters.

We are very interested in work in query progress indicators, because we view such mechanisms as potentially providing valuable monitoring input to a workload management system such as ours. However, most current work in query progress indicators tends to assume that the progress indicator considers each query in isolation, A notable exception is Luo et al. [12], who propose a multi-query SQL progress indicator that can consider the impact of other queries running in the system. Furthermore, most also assume that the progress indicator has visibility into the remaining cost of each running query (e. g., [12]), or that the number of tuples processed by each query operator are known (e. g., [5, 10, 11, 4]). Such operator-level information can be prohibitively expensive to obtain when multiple queries are executing simultaneously. That said, we believe the area of multi-query SQL progress indicators is quite relevant to our goals and look forward to more developments and hope that as this technology becomes more practical we'd be able to incorporate it as an input to our workload management policies.

Most commercial database systems have developed their own mechanisms for dealing with problem queries. For example, the HP-UX Workload Manager [19], the IBM Query Patroller for DB2 [15], the SQLServer Query Governor [14], Teradata's Dynamic Workload Manager [20], and Oracle's Database Resource Manager [16] all provide functionality to control queries that exceed limits such as estimated row counts, processing times, or joins. IBM's Query Patroller for DB2 [8] and Oracle's Database Resource Manager [16] 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 or SLA conformance. Similarly, the SQLServer Query Governor prevents queries whose estimated query costs exceed a user-set upper cost limit from starting, as opposed to stopping them after they reach a predefined limit. However, these limitations are not associated with response time requirements nor with service level objectives, nor have we seen any systematic study validating the effectiveness of such strategies and mechanisms.

3. PROTOTYPE IMPLEMENTATION

We have implemented a prototype workload management

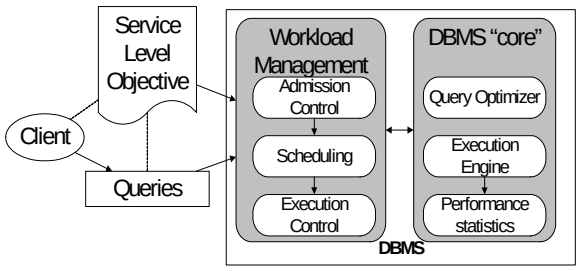


Figure 1: Workload Management Architecture

system in order to validate the various components of our approach. Figure 1 sketches a high level architecture of our workload management system. In this section, we walk through the components of our prototype.

3.1 Service Level Objectives

We model a workload as composed of one or more jobs. Each job consists of an ordered set of typed queries submitted by a client, and is associated with a service level performance objective as well as a submission style. Each query type maps to a tree of operators, and each operator in a tree maps in turn to its resource costs. Our current implementation associates the cost of each operator with the dominant resource associated with that particular operator type (e.g., disk or memory).

Before we can compare workload management strategies, we must have a model of the relative benefits of workload management. We distinguish between customer-facing service level objectives (e.g., the service level agreements commonly expressed in contracts) and job-facing service level objectives (e.g., adaptive penalty functions that can be used to optimize the scheduling of individual queries).

We interviewed a number of practitioners with experience in database workload management for a small variety of database products and customer environments, and found that customer-facing service level objectives seem to tend to fall into three categories: (1) *Deadline-driven*: In this scenario, the customer wants a given job to complete by a specified clock time. For example, a data load job could be required to complete before 5 am GMT. (2) *Concrete quantities of computing time*: In this scenario, the customer specifically expects a certain quantity of computing time or machine availability at a certain level of priority. (3) *Challenge*: In this scenario, a customer has a job which incurs known costs in an existing environment, and the requirement is that the job execute at least as well under a new environment with a specified set of constraints. Notably, we heard no constraints on query execution times.

For the purposes of this study, we focused on the first category – deadline driven objectives because that is the one in which we foresee that query scheduling and management could make the biggest impact. Within that scenario we further identified three categories of jobs:

1. Batch jobs composed of routine queries run at regular intervals where the entire job must be either completed or aborted by the soft deadline.

2. Batch jobs where at a minimum a certain percentage of the job must be completed by the soft deadline.

3. Interactive jobs that can be submitted at any time. Interactive jobs are submitted by special request for business reasons, and thus are potentially more valuable than a batch job.

We consider an open model in which we have scheduled batch jobs and interactive jobs that arrive according to a stochastic process. All jobs are associated with soft deadlines by which they must complete. Interactive queries are of greater importance than batch queries (approximately 15 times more "valuable," according to our interviews).

Each job consists of a set of one or more transactions. Each database transaction consists of a set of one or more typed queries that will be submitted sequentially to the database system and leave the system after being processed.

We model a job as a tuple (*clientId*, *typeId*, *slaId*, *startTime*), where *clientId* is an identifier for the client submitting the job, *typeId* is an identifier for the type of job (e.g., *batch* or *interactive*), *slaId* is an identifier for the service level objective (SLO) associated with that job, and *startTime* is the time that the job starts. In the case of a batch job, *startTime* is known a priori. In the case of an interactive job, *startTime* is set when the job arrives.

We model customer-facing service level objectives as a tuple, (*deadline*, *jobType*, *minWork*), where *deadline* is the soft deadline by which the job should be complete, *jobType* indicates whether the job is interactive or batch, and *minWork* indicates the minimum portion of the work that must be done. In the case of a batch job, the DBMS-facing service level objective is then to minimize the end-to-end execution time (*makespan*). In the case of an interactive job, the DBMS-facing service level objective is then to minimize the sum of the various queries' elapsed times (*flow time*).

3.2 Workload Management

The left-hand side of the box labeled DBMS lists components that provide fundamental workload management functionality – admission control, scheduling, and execution control. Each of these modules represents a knob which can be adjusted to select from a variety of workload management policies and algorithms. The right-hand side of the box contains the query optimizer, execution engine, and runtime monitor. These components provide core database functionality and supply information to workload management components but do not implement workload management policies.

We distinguish between jobs that are measured according to makespan (end-to-end execution times) and those that also seek to minimize flow time (the cumulative elapsed time-per-job). Both the OLTP and BI environments include queries whose times can range from milliseconds to hours. In the OLTP environment, workloads that seek to minimize flow time tend to feature small, short lived, predictable queries (e.g., an automatic teller machine). Long-running queries, such as index creation or backup activities, tend to be system batch jobs that are measured by makespan.

In order to provide maximum flexibility, our system decouples the task of controlling the work to be done by each resource from the task of detecting and responding to overload situations. The first task is controlled through admission control and scheduling policies; the second task is accomplished via execution controller and the rules it implements. Our system supports the insertion and specification of policies for each of the elements of workload management:

admission control, scheduling, and execution control. Our intent is to enable the comparison of the effectiveness of various algorithms for different types of workloads.
Admission control policies determine the submission of queries to the execution engine, and thus play three functions in workload management. First, when a new job arrives, admission control evaluates the DBMS's multiprogramming level, and either submits or enqueues each of the job's queries. Second, our framework can be configured to support multiple admission queues. Admission control policies regulate the distribution of queries among these queues, for example adding queries to queues based on estimated cost or dominant resource. Third, the SLA Manager uses the requirements formulated in the service level agreements (SLA) to calculate workload-specific service level objectives for the workload. The SLA Management uses this SLO information to perform admission control and assign workload components to appropriate queues maintained by an Execution Control. Next, when the execution engine has finished processing a query, admission control selects the next query for execution. In our system, this is done by evaluating the amount of work waiting in each queue (if there are multiple queues), and selecting the first query from the queue with the most work remaining. Once queries have been enqueued, the Scheduler's policies determine the ordering of the queries within a queue – for example, by estimated cost. Scheduling algorithms determine in which order which jobs should be processed by which resources. As noted by [2], the complexity of a database system prohibits establishing an optimal schedule dynamically, and realistically, heuristics must suffice. Furthermore, [24, 25] demonstrate that under lax timing conditions (1.5 times the minimum) even the simplest heuristics achieved nearly 100% performance, and that although simple heuristics perform poorly under strict timing conditions where the scaled time is 1.1 times the minimum time, even in that case combining simple heuristics with simple backtracking leads to significant improvement.
We focus at this time not on finding a better scheduling heuristic, but rather on how classes of scheduling algorithms apply to customer SLOs. [2] distinguish between three classes of time-critical database scheduling algorithms, based on the degree of knowledge they have regarding resource requirements: (1) algorithms with incomplete resource requirement knowledge, where an upper time-bound is impossible to achieve (e.g., priority scheduling); (2) algorithms with full knowledge of resource requirements (e.g., algorithms that detect and resolve conflicts among tasks over resources, such as conflict-avoiding transaction class preanalysis); and (3) hybrid optimistic algorithms that do not require full knowledge of resource requirements and compensate by responding to overload situations (where the scheduler must cope with with unschedulable tasks) by increasing throughput.
In our context, we have incomplete resource requirement knowledge, because we do not know a priori exactly how long each job will use each resource. That said, the customer objectives give us objective criteria we can use to decide how to respond to overload situations:

- Rewards and penalties (tardiness penalty, earliness reward). This is an explicit factor for batch jobs, and is implied for interactive jobs.
- Makespan, meaning the total amount of time end-to-end needed to finish a given set of jobs). This is a priority for both batch and interactive jobs.
- Average flow time, meaning the total time including wait time for all jobs. This is a priority for interactive jobs, but not for batch jobs. Note that if an interactive job contains only one query, then that job's flowtime will be the same as its makespan.

We model choice of scheduling algorithm as a tuple of the form, (*jobtype*, *schedulingalgorithm*, *configuration*) for each type of job, *jobtype*, supported by our workload management system, where *schedulingalgorithm* is an identifier for a scheduling algorithm implemented by our system and *configuration* is a set of (*parameter*, *value*)-tuples that capture configuration choices for that scheduling algorithm. As we describe in Section 5, we have included two classic scheduling algorithms in our initial implementation: *first-in-first-out* (FIFO) and *two queues*.
Finally, we implement our execution control using a fuzzy execution controller that takes as input a configurable set of rules for how to detect and respond to overload situations based on the type of job, the rewards and penalties associated with the job, and the values placed on flow time and throughput. We defer discussion of the execution controller to Section 4.

3.3 Execution Engine / DBMS Core

The Execution Engine performs the operations required to execute the query. We chose to build an execution engine simulator to implement the execution engine component. One reason for this decision was that it would otherwise require a prohibitive amount of time to experiment with workloads that take hundreds of hours to run. Second, we needed to insert problems into our workloads in a controllable manner for our experiments, Finally, due to the scale and complexity of BI systems, performance might otherwise not be repeatable due to the difficulty in controlling the execution environment.
Implementing the execution engine in this way enables us to drive our experiments from arbitrary workloads of varying numbers of queries and query sizes generated from data gathered from real workloads. Our simulated execution engine also introduces the imprecision into the costs of the queries as derived from the query optimizer (estimated costs). We have written a set of tools to translate log/output files generated by the execution of commercial workloads into an XML input file that specifies the objectives, jobs, queries, resource usage, and problems of an experimental workload. We can thus build arbitrary workloads of varying numbers of queries and query sizes from data gathered from real workloads.
The Execution Manager monitors runtime statistics, and is capable of submitting control commands to the Execution Engine. Our system establishes a feedback loop between the database management system and the Execution Manager. For example, if the Execution Manager decides that a query's current execution cost falls significantly outside of the expected distribution for execution costs for that query's type, then it may choose to take action (e.g., kill) one or more of the currently running queries.

4. EXECUTION CONTROL

We implemented the execution control component to support fuzzy logic-based rules. To this end, we leveraged the

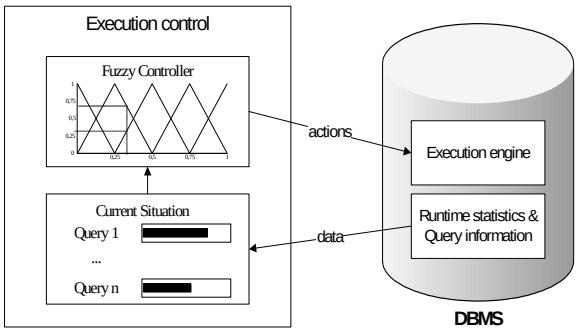


Figure 2: The Architecture of the Execution Controller

open source *Fuzzy Engine for Java*, a Java library for interpreting fuzzy rules expressed as text strings [17].
Our workload management system relies on the execution controller to govern the flow of the running system at execution-time. Figure 2 shows the architecture of the execution controller. As queries execute, the execution controller uses runtime statistics to identify potential workload problems. The fuzzy controller component of the execution control then executes corrective actions to rectify the workload problems.
Before explaining the details of our execution controller, we motivate why we explored this direction and give an introduction of the fuzzy logic basics.

4.1 Motivation for Using a Fuzzy Controller

We identify three pragmatic issues that we must address in order to automate the management of overload situations in BI workloads. One, the classification of queries according to their expected behavior involves a degree of uncertainty because query execution times are not entirely predictable. A query may have identical SQL code as one submitted yesterday, but during execution it becomes apparent that because of new, heavily skewed data, we can no longer be certain of its execution time. Two, there are numerous factors that must be considered when governing query execution. Practitioners can attest to the difficulty of both database sizing (which attempts to predict the capabilities of a system as a whole with regard to a proposed workload) and query progress estimation (which focuses on the performance of a single query). Capturing management logic in an accessible framework is thus a practical challenge. One of our interviewed practitioners lamented that he didn't want to be told about input and output cardinalities at the various stages of an operator tree for all the queries being currently executed – he needs to know how the system is running and if there is a bad query that he should cancel. Three, due to the complexity of data warehouse systems, complete knowledge about the state of the system and the queries running in it is not available. Even if a database *could* monitor everything, the performance overhead would be prohibitive.
We believe that the fuzzy logic paradigm addresses all these issues. Fuzzy logic is designed to reason about sets whose members belong to a given set with some degree. Rules are expressed in terms of membership in these sets; it accommodates intermediate degrees of truth [22]. In addition, the fuzzy controller does not need complete knowledge about the queries it manages. The degree to which knowledge is

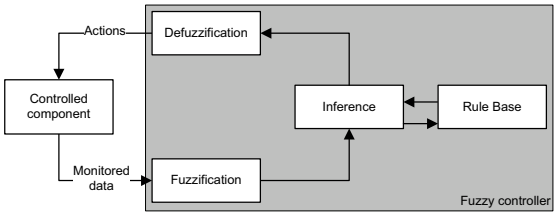


Figure 3: Architecture of a Fuzzy Controller

incomplete is an inherent part of the model. Furthermore, a fuzzy controller can be extended incrementally when new knowledge does become available by adding new linguistic variables and new rules, respectively. Thus one of the strengths of fuzzy logic is that it provides an implementation framework for building management interfaces that are easy to understand (enabling database administrators to govern the system using intuitive sentences instead of rules) [22].

4.2 Fuzzy Logic Basics

Fuzzy controllers are special rule-based systems based on *fuzzy logic* [23]. Figure 3 summarizes the workflow of a fuzzy controller. The fuzzy controller obtains monitored data from the component to be managed. Using these values, the controller converts the monitored values into appropriate fuzzy sets in the *fuzzification* step. The *inference engine* uses the fuzzy sets to evaluate the fuzzy rule base and generate the sets for the output variables, which are converted into a vector of crisp values during the *defuzzification* step. The defuzzified values represent the actions the fuzzy controller uses to control the workload.
The membership grade of elements of fuzzy sets ranges from 0 to 1 and is defined by a membership function. Let X be an ordinary (i. e., crisp) set, then

$$A = \{(x, \mu_A(x)) \mid x \in X\} \text{ with } \mu_A : X \to [0, 1]$$

is a fuzzy set in X. The membership function μ_A maps elements of X into real numbers in $[0, 1]$. A larger (truth) value μ denotes a higher membership grade.
Linguistic variables are variables whose states are fuzzy sets (*linguistic terms*). A linguistic variable is characterized by its name, the set of linguistic terms, and a membership function for each linguistic term. Figure 4 shows an example for the linguistic variable *progress* and the assigned trapezoid membership functions for the three linguistic terms *low*, *medium*, and *high*.
During the fuzzification phase, the controller maps the crisp values of the measurements (e. g., the progress of a query) to the corresponding linguistic input variables (e. g., *progress*) by calculating membership grades using the membership functions of the linguistic variables. For example, based on the membership functions in Figure 4, a progress $p = 0.3$ (30%) for a query is 33% low ($\mu_{low}(p) = 0.33$), 67% medium ($\mu_{medium}(p) = 0.67$), and 0% high ($\mu_{high}(p) = 0.0$).
In the inference phase, the fuzzy rule base is evaluated using the fuzzified measurements. We show two example rules.

```
IF relDatabaseTime IS high AND progress IS high
THEN reprioritize IS applicable

IF relDatabaseTime IS high AND
  (progress IS low OR progress IS medium)
```

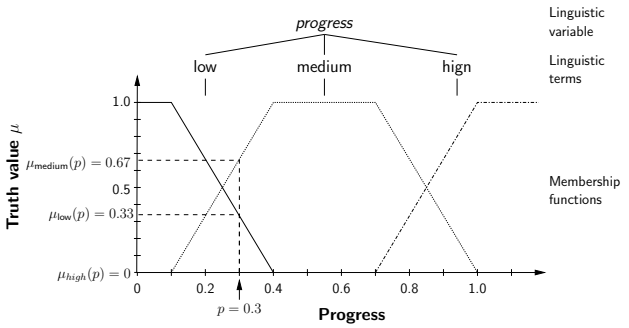


Figure 4: Linguistic Variable *progress*

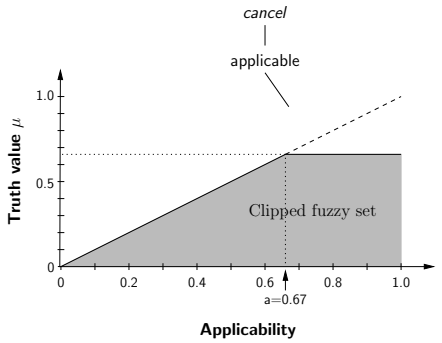


Figure 5: Max-Min Inference Result

```
THEN cancel IS applicable
```

where *relDatabaseTime* and *progress* denote input variables and *cancel* and *reprioritize* output variables described in Section 4.3. The second rule states that "hopeless" queries (i.e., the ones that are late but for which not much work has been done) can be canceled. In contrast to that, the first rule allows a query to get higher priority if it is late but almost complete.

Fuzzy logic evaluates conjunctions and disjunctions of truth values in the antecedent of a rule using the minimum and maximum function. For our example, we assume that the relative database time of the monitored query takes a value r and that the membership grades for the linguistic variable *relDatabaseTime* are $\mu_{low}(r) = 0.0$, $\mu_{medium}(r) = 0.3$, and $\mu_{high}(r) = 0.7$ (membership functions for the linguistic variable *relDatabaseTime* are not shown in a figure). Thus, the truth values of the antecedents in the example rules above evaluate to $\min(0.7, \max(0.67, 0.33)) = 0.67$ and $\min(0.7, 0.0) = 0.0$.

The implication rule in classical logic that the consequent is true if the antecedent evaluates to true is not applicable for fuzzy logic because the truth value of the antecedent is a decimal value between 0 and 1. Thus, literature proposes several different inference functions for fuzzy logic. We use the popular *max-min* inference function that clips off the fuzzy set specified in the consequent of a rule at a height corresponding to the degree of truth of the rule's antecedent. After rule evaluation, all fuzzy sets referring to the same output variable are combined using the fuzzy union operation: $\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$ for all $x \in X$

The resulting combined fuzzy set is the result of the inference step. The fuzzy set for the consequent of the first example rule is shown in Figure 5.

During the defuzzification phase, a crisp output value is calculated from the fuzzy set that results from the inference phase. Again, literature lists several defuzzification methods, from which we chose the established *maximum method*. This method determines the smallest value at which the maximum truth value occurs as result. As shown in Figure 5, the crisp value for the applicability of action cancel is 0.67. Assuming that the applicability for action reprioritize is defined analogously, action reprioritize is not applicable at all. Since the execution controller will execute the action with the higher applicability, the respective query will be canceled.

4.3 Execution Control Details

Our execution controller leverages information gathered at runtime to manage the queries concurrently running in the database. As mentioned in Section 4.2, there are several metrics that are monitored and used to compute the input variables for the rules, respectively. We recognize that an actual system may not support all these. For example, we understand that monitoring work per operator could be prohibitively expensive. That said, we wanted our system to support the possibility of such monitoring to be done so that its costs and benefits could be evaluated.

Our list of monitored metrics include the following (ordered by the probable cost of obtaining the information):

Priority The priority of a query has impact on the resource allocation. The higher the priority, the more resources the query gets, compared to other queries.

Number of cancellations A count for the number of times an individual query has been canceled.

Operator progress This abstract metric is monitored on the operator level for each currently active operator. Depending on the type of the operator, this metric represents for example the number of tuples read from disk or the number of I/O operations. The query optimizer returns an estimate for this metric for each operator. Using the monitored (actual) data divided by the estimate, we get the relative work of an operator. This allows us to (1) get an estimate of the progress of a query and (2) decide whether or not the estimates for a query are way off. We assume an intelligent query cost estimator that updates its estimates during run-time.

Resource contention The number of queries that are competing with a query for resources.

Resource Progress Ratio of the work performed thus far on a given resource, compared to the expected work to be performed by a query on that resource. This metric should be considered in perspective of the monitored/expected work

Database time We define the database time for a query as the time the query is executing in the database (i.e., no wait times in the queue before the database). If the query is still running, the elapsed time is the time between start of the execution and the current point in time. Analogous to the relative work of an operator,

we define the relative database time of a query as the actual elapsed database time for a query divided by the estimated database time for that particular query. This information can then be used to help determine whether or not a query's performance is suffering.

Based on these metrics, we are focusing on three actions that can be executed on a query in order to control the workload (an implicit fourth action is "noop" which denotes that none of the other actions is executed).

Reprioritize This action increases the priority of the query. If a query is re-prioritized, the resources are redistributed immediately among the queries according to the priorities of the individual queries.

Kill This action kills a running query and immediately frees the resources used by this query. Any intermediate results generated during the execution of the query are disposed. After killing a query, pending queries (if any) are admitted to the database and the resources are redistributed.

Resubmit A killed query can be resubmitted to the system, i.e., the query is enqueued again. This action can either be executed transparently to the user (i.e., the user does not see that a query has been killed) or be user-induced (i. e., the user has been notified and he manually resubmits the query). For the rest of this paper, we denote killing queries without automatic resubmission as *abort* while *cancel* denotes killing and automatically resubmitting the query. For the latter case, we identified three options for how to treat resubmission, which we describe here along with their drawbacks.

The query can be immediately re-enqueued. Nevertheless, if the query was blocked by a problem query that hogs the resources of the system, the query would be delayed again if the problem is still running if the killed query re-enters the system.

Thus, the resubmission of a killed query could be delayed for a specified amount of time. Nevertheless, there is no guarantee that the conditions that caused the initial delay will have disappeared.

For this purpose, a killed query can be resubmitted when the queries that were running simultaneously with the killed query have left the system. This approach increases the makespan of the delayed queries, possibly unnecessarily.

For the purposes of our implementation, we chose the first option. We use FIFO to schedule queries, and our experiments show that the delay until the query is processed again is generally sufficient to prevent the previous contention.

5. EVALUATION

In order to characterize the potential impact of unexpected behavior (unexpectedly heavy items in a workload), as well as the ability of our fuzzy execution controller to react to these problems, we have performed a number of experiments, some of which we present here.
In particular, we explore the following:

- **Impact of problem queries.** In these experiments, we show the impact of problem queries on a workload without any execution control on both interactive and batch objectives.
- **Execution controller.** These experiments show the impact of the execution controller with the multi-programming level (MPL) set to one.
- **False positives: Accuracy requirements for the execution controller.** In this experiment, we show the impact of an over-aggressive execution control policy on both batch and interactive objectives.

Note that although the workloads driving our experiments are derived from actual commercial workloads, the execution times derived from the simulation are not intended to map literally to actual time units. Since the focus of our experiments is to show the relative impact of problem queries and the effectiveness of managing the problems, we present the times in this section in the artificial time unit given by the simulation.

5.1 Experimental Workloads

Before discussing the experiments in detail, we first describe the two workloads used for the experiments. Both workloads contain two jobs that are executed in parallel. The *interactive* job contains about 1100 queries with relatively low estimated costs. The processing time for nearly all of the queries in the workload we observed for deriving the data for the interactive job ranged from milliseconds to seconds; about 40 ran for longer than a minute, and a small handful of those ran for around ten minutes. The mean costs of the queries in the *batch* job are about 1000 times higher than the mean costs of the interactive job. The queries in the commercial workload we observed for deriving the data for the batch job completed after several minutes to hours.
Our experimental framework allows us to generate arbitrary workloads from the observed data of workloads executed on a real database system. Thus, we created a *problem* workload, which is based on the *normal* workload described above.
The *problem* workload models a workload where some batch queries run longer than expected. Our model comprises two parameters. The first parameter denotes how much the execution of a problem query is "stretched" compared to a normal query. For example, an unexpected high execution time in the database can be caused by data skew, i.e; the optimizer underestimated the actual costs of the query. The second parameter describes whether a query shows a problem behavior after it is killed and resubmitted. For example, this behavior can be observed when there is not enough main memory for the execution of the query. After killing and resubmitting the query, there might be enough resource for that particular operator, such that the query can complete without exhibiting problem behavior.
In total, we added 75 problem queries (about 4.5%) to the batch job in the *problem* workload. These problem queries consisted of 50 queries with a stretch factor of 10 (meaning that they would run about ten times longer than expected), 20 queries with a stretch factor of 100, and 5 queries with a stretch factor of 1000. With each of the problem queries, we associated a probability that the problem prevails if the query is canceled (and resubmitted).

The interactive jobs in both workloads begin arriving shortly after the processing of the batch job starts. In our model, the interactive jobs arrive one at a time – an interactive query is submitted only after its predecessor has completed and returned results to the submitting client. As the interactive jobs consist of a single query each, flowtime becomes equivalent to makespan. Furthermore, we configured the simulated execution engine such that it admits pending interactive queries before their batch counterparts. All batch queries are maintained in a FIFO queue, i. e., in the order they were registered at the database system. This reflects the settings that could be implemented with today's commercial database systems.

5.2 Impact of Problem Queries

We first investigated the workload in the presence of problem queries at various multi programming levels. Figures 6(a) and 6(b) show the makespan of the jobs executed in the presence of a small number of problem queries (*problem* workload) compared to the makespan of executing the jobs in the *normal* workload. Notice that all of these curves show a "knee". At low MPLs, the system is underutilized. After reaching an optimal range of MPL, the system becomes over-utilized, i. e., the queries suffer too much from resource contention.

As expected, the makespan for the batch workload increases in the presence of unexpectedly long-running queries (Figure 6(a)). But, as indicated in Figure 6(b), the long-running batch queries in the problem workload also impact the interactive workload. The main reason for the makespan increase is because the average wait time for an interactive query to enter the system increases due to the increased average processing time of the batch queries.

5.3 Evaluation of Workload Management

Based on the findings of the problem query experiments, we wanted to evaluate the effectiveness of the workload management actions proposed in Section 4. For the following experiments, we used the relative database time, progress, and the number of cancellations of a query as metrics for deciding whether or not to abort or cancel a query.

5.3.1 Effectiveness of Workload Management Actions

First, we want to evaluate the effectiveness of the workload management actions with near-perfect knowledge of the system. For controlling the precision of the estimated processing times for queries – which is a complex task even in our controlled environment – we set the MPL to 1. At this MPL, the execution of batch and interactive queries is interleaved because each time an interactive query completes, a batch query is admitted to the database and the successor of the completed query has to wait for the batch query to complete. Queries are identified as long-running using the monitored value r for the relative database time. This is defined as an interval $[1, 1.1]$, i. e., the truth value for *relative database time is high* is 0 for values $r < 1.0$ and 1 for values $r > 1.1$. For $1.0 \leq r \leq 1.1$ the truth value increases linearly. We do not consider the progress of queries because in this setting, we assume that only the processing time for problem queries exceeds the respective estimate.

The makespan of the interactive job decreased from 0.68 (*problem* workload in Figure 6(b)) to 0.17 due to decreased wait times. With workload management actions taken, it is guaranteed that the time a query stays in the system is in the range of its estimated processing time, irrespective of whether it is a problem query or not. "Normal" queries run to completion, problem queries are either aborted or canceled. Thus, especially in the presence of queries with an unpredicted long processing time, the wait time of pending interactive queries is bounded because they do not have to wait for their problem counterparts to complete.

Applying workload management actions, the makespan of the batch job in presence of long-running queries is decreased from about 1.41 (Figure 6(a), looking at MPL=1) to about 0.33. For handling the workload problems, the execution controller triggered 119 actions for the 75 problem queries. No false positives were identified. From these 119 actions, 101 cancel operations were executed. One would expect that canceling a query would increase the makespan of the batch job because the amount of work done before the cancellation is lost and the processing starts from scratch after resubmission. However, this effect is over-compensated by aborting 18 queries which still showed the problem behavior after canceling them twice.

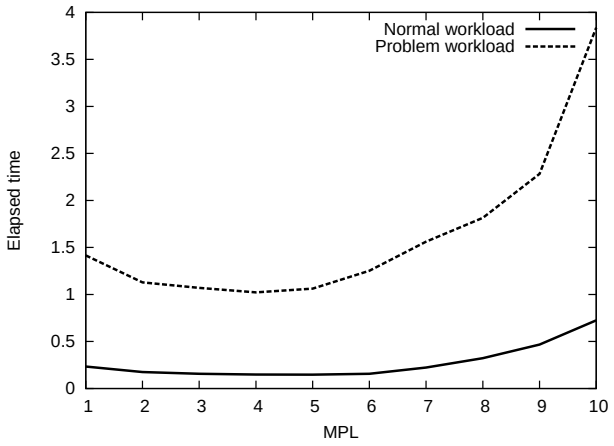
5.3.2 Impact of False Positives

Ideally we would like to cancel just the problem queries. However, we cannot assume the perfect knowledge to achieve this goal. Thus, we wanted to evaluate the impact of workload management actions on an environment with less accurate knowledge about the processing time of a query. Therefore, we conducted an experiment at MPL=4. Even in our controlled environment, estimating the query execution time cannot be done with arbitrary precision when several queries are running simultaneously. In order to derive reasonably accurate processing time estimates, we calibrated our times for non-problem queries by running them in our simulated environment under specific MPL and measuring execution times. The main focus of this experiment is to understand the impact of handling problem queries too aggressively.

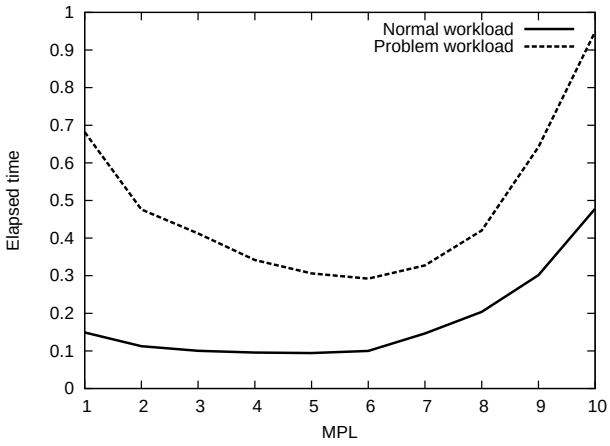
Figure 7(a) demonstrates that rectifying the workload management problems aggressively is done at the costs of executing more actions and canceling and aborting false positives – queries that are actually not problem queries. The 'x' axis of the figure measures "aggressiveness" in terms of when we consider the relative database time as "high". Aggressiveness is negatively correlated with the relative database time; i. e., the lower the values in the interval, the more aggressive the execution control is. False positives occur because the actual processing time for some normal queries is up to 1.3 times higher than predicted. That is to say, when the threshold value for the variance in time of a "problem" versus "non-problem" query overlaps with the natural variance in query execution times, then false positives are likely to occur. When the wait time for identifying problem queries lies outside normal variance, then the number of false positives drops.

A number of queries in the experiment above were false positives because the execution control did not consider the progress of a query before killing it. Thus, the execution controller executed actions even for queries that were almost complete. We therefore experimented with different thresholds for the progress and kept the time for identifying long-running queries constant at $[1, 1.1]$.

Figure 7(b) presents the number of false positives incurred, along with the number of actions executed, for that experi-

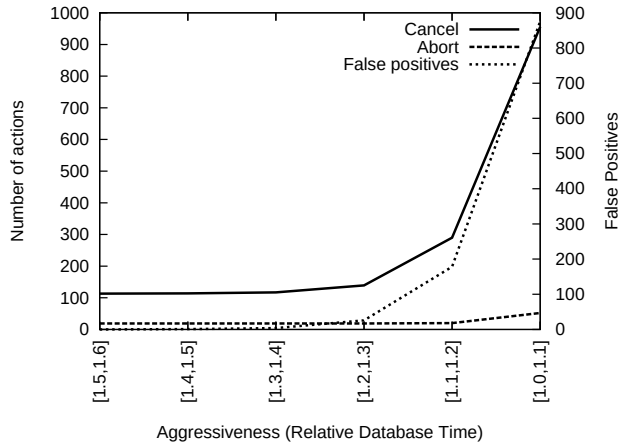


(a) Makespan of Batch Job

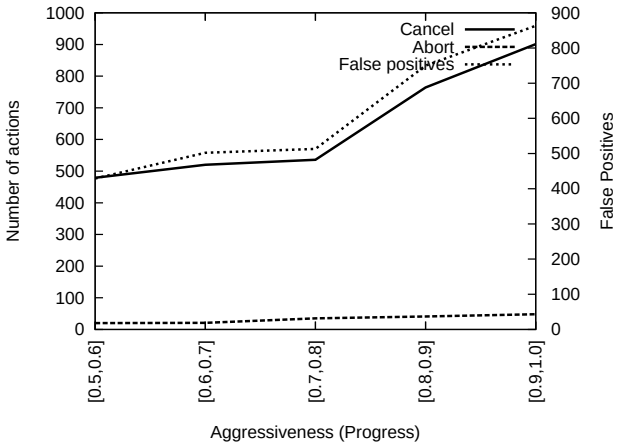


(b) Makespan of Interactive Job

Figure 6: Impact of Problem Queries



(a) Relative Database Time



(b) Progress

Figure 7: Number of False Positives and Number of Actions Executed

ment. Similar to the previous experiment, the left-hand 'y' axis denotes the number of actions taken and the right-hand 'y' axis the number of false positives. The 'x' axis plots aggressiveness as the threshold indicating the amount of progress made after which we would not kill a long-running query. Queries whose progress is below this threshold are canceled. In contrast to Figure 7(a), in this figure aggressiveness is positively correlated with progress. Higher numbers in the intervals indicate that the progress of the query must be higher in order to "survive". As can be seen, considering the progress helps to control the number of false positives and, thus, the number of actions executed – but, again, only if the controller does not execute actions too aggressively.

Figure 8 illustrates how progressively aggressive control policies impact makespan in the realistic case that cancel and resubmit actions are associated with costs. In the figure, we report the makespan for the experiment shown in Figure 7(a).

Similar to the experiments at MPL=1, the makespan of both jobs drop. The makespan of the batch and interactive jobs

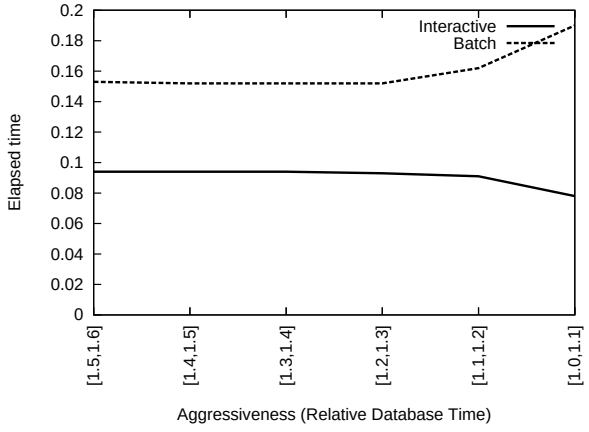


Figure 8: Makespan for Batch and Interactive Jobs

with problem queries but no actions takes are 1.02 and 0.34 (see Figures 6(a) and 6(b)), looking at MPL=4) compared to values between 0.15 and 0.19 (batch), and 0.08 and 0.1 (interactive) with actions taken. This drop is due to not completing some of the long-running queries.

As shown in the figure, when cancel and resubmit actions incur costs, the benefit of canceling the problem queries is balanced by the cost of the control action, and impact makespan and flowtime. If we cancel many queries, the makespan for the interactive job even drops – but at the costs of the batch job. In our experiment, overaggressively canceling queries resulted in a 25% makespan increase for the batch job compared to a less aggressive controller, even for moderate costs of executing workload management actions.

In conclusion, our experiments demonstrated that, first, the presence of even a few problem queries can have a negative impact on makespan. Second, that if the execution controller is overly-aggressive in treating problem queries, then it is likely to invoke a large number of superfluous control actions against harmless individual queries. Third, that a progress indicator, if one is available, can significantly help reduce the number of spontaneous actions. And finally, fourth, that when control actions incur penalties, those penalties can outweigh the benefits of canceling actual problem queries when a control policy is overly aggressive.

6. CONCLUSION AND ONGOING WORK

In summary, we designed and implemented a workload management system that can automatically admit, schedule, and control the execution of BI workloads according to the expectations of the customers who own them. To this end, we have interviewed practitioners and present some insights regarding the nature of customer expectations and how they translate to the service level objectives that a workload management system would consume. Because we recognize that query cost estimation and query progress estimation are open problems, we have utilized fuzzy logic in the design of our execution controller. Fuzzy logic enables us to capture in a format that humans can easily understand the many trade-offs that the execution controller makes when making its decisions. In order to test our system, we acquired examples of both batch and interactive industrial workloads, along with multiple traces from their execution at various multi-programming levels.

Our experiments bore out conventional wisdom's assertion that the presence of even a few poorly written or optimized queries can significantly impact the performance of a database system. More surprisingly, our experiments also demonstrated the considerable impact of "false positives" – properly executing queries that are incorrectly identified as problems. This impact reflects the cost of canceling and rerunning a properly executing query that was in fact performing slowly in reaction to the presence of an actual problem query.

Because false positives and false negatives both carry significant penalties, it is tremendously important that problem queries be identified with great accuracy. We show that our execution controller can identify problem queries without incurring an excess of either false negatives or false positives.

For next steps, we want to investigate problem queries by interviewing practitioners and building a taxonomy to characterize the impact that problem query have on a database system. We also plan to add new workload management policies and control actions to our prototype, and perform experiments to validate their effectiveness. Finally, we would like to perform a large number of experiments so as to better understand the impact of variance in query costs on system performance.

7. ACKNOWLEDGEMENTS

We would like to thank the following people for generously giving their help and perspectives: Thomas Anderson, Malu Castellanos, Paul Denzinger, Rich Folsom, Peter Friedenbach, Goetz Graefe, Chetan Gupta, Kannan Govindarajan, Meichun Hsu, Pierre Huyn, Lily Jow, Joseph Karam, Abhay Mehta, Brian Thome, Bob Wall, Janet Wiener, Hans Zeller, and Alex Zhang. We also particularly thank Peter Friedenbach, Chetan Gupta, and Abhay Mehta for their help collecting workload run data.

8. REFERENCES

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

[2] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal. Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 470–480, Washington, DC, USA, 1989. IEEE Computer Society.

[3] M. J. Carey, M. Livny, and H. Lu. Dynamic Task Allocation In A Distributed Database System. In *ICDCS*, pages 282–291, 1985.

[4] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When Can We Trust Progress Estimators For SQL Queries? In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 575–586, New York, NY, USA, 2005. ACM Press.

[5] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating Progress Of Execution For SQL Queries. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 803–814, New York, NY, USA, 2004. ACM Press.

[6] D. L. Davison and G. Graefe. Dynamic Resource Brokering for Multi-user Query Execution. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 281–292. ACM Press, 1995.

[7] D. Feinberg. Database Management Systems Technology Trends. Technical report, Gartner, October 2006.

[8] IBM DB2 Query Patroller. `http://www-306.ibm.com/software/data/db2/querypatroller/`.

[9] S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper. Quality of Service Enabled Database Applications. In *Service-Oriented Computing - ICSOC 2006, Proceedings*, volume 4294 of *Lecture Notes in Computer Science (LNCS)*, pages 215–226, December 2006.

[10] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a Progress Indicator For Database

Queries. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 791–802, New York, NY, USA, 2004. ACM Press.

[11] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Increasing The Accuracy and Coverage of SQL Progress Indicators. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 853–864, Washington, DC, USA, 2005. IEEE Computer Society.

[12] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL Progress Indicators. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology*, pages 921–941, 2006.

[13] M. Mehta and D. J. DeWitt. Dynamic Memory Allocation for Multiple-Query Workload. In *Proc. of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[14] Query Governor Cost Limit Option. `http://msdn2.microsoft.com/en-us/library/ms190419.aspx`.

[15] B. Niu, P. Martin, W. Powley, R. Horman, and P. Bird. Workload Adaptation In Autonomic DBMSs. In *CASCON '06: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, 2006.

[16] Oracle Database Resource Manager. `http://www.oracle.com/technology/deploy/availability/htdocs/rm_overview.html`.

[17] E. S. Sazonov. Open Source Fuzzy Inference Engine for Java. `http://www.intelligent-systems.info/FuzzyEngine.htm`.

[18] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. M. Nahum. Achieving Class-Based QoS for Transactional Workloads. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006*, page 153, 2006.

[19] I. Subramanian, C. McCarthy, and M. Murphy. Meeting Performance Goals With The HP-UX Workload Manager. In *WIESS'00: Proceedings of the 1st Conference on Industrial Experiences with Systems Software*, 2000.

[20] Teradata. Teradata Dynamic Workload Manager User Guide, September 2006.

[21] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. The COMFORT Automatic Tuning Project. *Information Systems*, 19(5):381–432, 1994.

[22] O. Wolkenhauer and Jim M. Edmunds. A Critique Of Fuzzy Logic In Control. *International Journal of Electical Engineering Education*, July 1997.

[23] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

[24] W. Zhao and K. Ramamritham. Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints. *Journal of Systems and Software*, August 1987.

[25] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers*, August 1987.