Managing Long-Running Queries

Stefan Krompass[TUM], Harumi Kuno[HPL], Janet L. Wiener[HPL], Kevin Wilkinson[HPL]
Umeshwar Dayal[HPL], Alfons Kemper[TUM]

[TUM]Technische Universität München,
Munich, Germany
{firstname.lastname}@in.tum.de

[HPL]Hewlett-Packard Laboratories
Palo Alto, CA, USA
{firstname.lastname}@hp.com

ABSTRACT

Business Intelligence query workloads that run against very large data warehouses 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. Long-running queries are a known problem in all commercial database products. However, we have not seen a thorough classification of long-running queries nor a systematic study of the most effective corrective actions.

We present here a systematic study of workload management policies, including many implemented by commercial database vendors. Our goal is to enable a system to: (1) recognize long-running queries and categorize them in terms of their impact on performance and (2) determine and take (automatically!) the most effective control actions to remedy the situation.

To this end, we identify common workload management scenarios involving long-running queries, and create a taxonomy of long-running queries. We carry out an extensive set of experiments to evaluate different management policies and the relative and absolute thresholds that they may use. We find that in some scenarios, the right combination of policies can reduce the runtime of a workload by a factor of two, but that in other scenarios, any action taken increases runtime. One surprising result was that relative thresholds for execution control can compensate for inaccurate cost estimates, so that *Kill&Requeue* actions perform as well as *Suspend&Resume*.

1. INTRODUCTION

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 [8], Microsoft [15], and Oracle [17] 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 intuitions 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 [10]. 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. 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 estimations 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 in 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 paper, 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 newly-proposed "suspend" action [3, 5], and consider whether certain types of management policies should be combined in order to compensate their strengths and weaknesses.

The primary contributions of this paper are:

- We develop a taxonomy of long-running query types based on how they impact other queries.

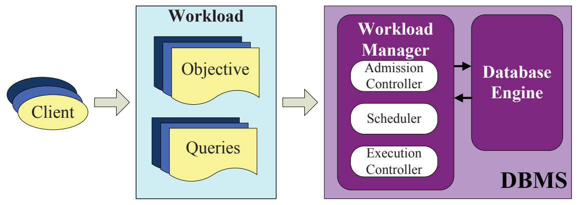

Figure 1: A typical workload management system includes three components: the admission controller, query scheduler, and execution controller.

- We have synthesized a core set of workload management policies including many offered by major commercial database vendors. We consider admission control, scheduling and execution control policies, but defer investigating priority-based policies to future work.

- We evaluate the ability of these core workload management policies to identify and act upon the problem scenarios described above using our experimental framework and database simulator. 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.

- 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.

The rest of this paper is organized as follows: Section 2 describes the components in a typical workload management architecture and gives an overview of the workload management in industry and academia. In Section 3, we present our taxonomy of long-running queries. We describe our workload management framework in Section 4. A description of our experimental setup follows in Section 5, while Section 6 shows our results and lessons learned. We conclude the paper in Section 7.

2. WORKLOAD MANAGEMENT OVERVIEW

In today's database systems, workload management is accomplished through the application of policies to workloads. 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. In this section, we overview current workload management techniques and related research.

2.1 Objectives, policies, actions

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.

Typically, workload management policies act at three control points — the admission control, scheduling, and execution control components shown in Figure 1. These components control which queries are admitted into the database system, the order in which admitted queries are queued for the database engine to run, and when to invoke execution control actions at runtime. The database engine compiles, optimizes, and executes the queries, and also provides runtime statistics to the execution controller.

Table 1: Metrics that scheduling can limit

Metric	Description
MPL	Multiprogramming level: Number of queries executing concurrently
Usage	Current resource usage
Costs	Current resource usage + expected cost of new query
Access	Number of queries accessing same table or database

2.1.1 Admission control

Admission control decides whether a newly arriving query should be admitted into the system, i. e., passed to the scheduling component, or rejected. The primary goal of admission control is to avoid accepting more queries than can be executed effectively with available resources.

Admission control policies can place different kinds of limits on the system, e. g., the number of queries running concurrently, the number of concurrent users, or the expected costs of the submitted queries. Typical admission control actions are: *Warn*, which accepts the query but signals a warning; *Hold*, which holds a query until the DBA releases it; and *Reject*, which rejects the query.

If a query passes all of the admission control policies then the query is admitted for scheduling. Some systems support policies that allow a high priority query to bypass admission control and scheduling and start executing immediately.

2.1.2 Scheduling

The main goal of the scheduling component is to avoid a state of system overload. The scheduler determines when to start the execution of a query. It maintains queues of pending queries and policies determine how the queues are managed. The most commonly used queue types used by schedulers include: *Priority*, separate queues for different query priorities; *Size*, separate queues for different expected runtimes; *One*, one FIFO queue for all queries; and *None*, all queries start immediately. Note that some policies include parameters and thresholds, e. g., to map expected runtime to the appropriate *Size* queue. Table 1 describes metrics that may be used in scheduler policies to decide whether to start the next query. If the metric is below threshold, then another query may start.

2.1.3 Execution control

Admission control and scheduling policies apply to queries *before* execution. Their decisions are based on estimated query cost. However, at runtime, a query may behave differently from its cost estimates. The task of execution control is to limit the impact of these deviations from expectations. Execution control uses both cost estimates and runtime information to make its decisions.

Different execution conditions may be evaluated by an execution policy, such as CPU time above a threshold or elapsed exceeding

Table 2: Execution control actions

Action	Description
None	Let the query run to completion
Warn	Print a message to a log; query continues
Reprioritize	Change the priority of the query
Stop	Stop processing query; return results so far
Kill	Abort the query and return an error
Kill & Requeue*	Abort the query; Put it in a scheduling queue to start over
Suspend & Resume*	Stop processing query; Put saved state in scheduling queue

an estimate by an absolute or relative amount. The administrator must choose the thresholds in the conditions to achieve workload objectives. Figure 2 describes some typical actions that might be used in execution control policies.

2.2 Policies in commercial systems

Table 3 shows workload management policies implemented by several commercial tools. We distilled the information from the documentation provided by the different vendors, e. g., IBM's Workload Management for DB2 [8], Microsoft's SQL Server [15], HP's Workload Management Services for Neoview [9], Oracle's Database Resource Manager [17], and Teradata's Dynamic Workload Manager [19]. We do not discuss these in detail but some features are noteworthy.

For admission control, IBM's Workload Management for DB2 allows administrators to define user groups and specify the maximum number of concurrent queries per group. Oracle can limit the number of threads used for query processing.

For execution control, IBM Workload Manager for zSeries [11] and Oracle Resource Manager [17] implement an aging mechanism that dynamically adjust the priority of a query (to lower priority) as it runs. Oracle Discoverer [16] supports *Stop*, which returns the first N results of a query. Teradata Workload Manager supports a variant of *Kill* that makes it easy for the DBA to resubmit the query.

2.3 Related research

To our knowledge, few researchers explicitly consider long-running queries in workload management. Benoit [2] 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. Weikum *et al.* [20] 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 [4, 6, 12, 13]. Such operator-level information can be prohibitively expensive to obtain.

Luo *et al.* [14] 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.

Table 3: Admission control, scheduling, and execution control policies implemented by commercial products

(a) Admission control

Admission policy	Action
Limit concurrent queries	Hold; Reject; Warn
Limit queries in queue	Hold; Reject
Limit logon sessions	Reject; Warn
Limit expected costs	Hold; Reject; Warn
Limit resource usage	Hold; Reject
Check access permissions	Reject; Warn

(b) Scheduling

Scheduling	Implementations
Queue types	None; One; Priority; Size
Query starts when under threshold	Access; Costs; MPL; Usage

(c) Execution control

Execution condition	Action
Elapsed time $>$ threshold	Kill; Reprioritize; Warn
Actual rows returned $>$ threshold	Kill; Reprioritize; Stop; Warn
Actual / estimated rows returned $>$ threshold	Warn
Actual resource consumption $>$ threshold	Kill; Reprioritize
Actual / estimated resource consumption $>$ threshold	Kill; Reprioritize; Warn

3. 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 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 weren't. Second, we look at whether the query is making reasonable progress. Third, we consider whether the query is using an equal share of resources to other queries, or whether it is getting significantly more or less of them. We discuss how to measure these properties in Section 4.

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

Table 4: 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.

	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

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. 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 only way to relieve system overload is to reduce the number of concurrent queries.

4. 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 components depicted in Figure 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 groups, i. e., different subgroups in the workload to which the workload manager applies different policies. 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 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 databases implements only a subset of the possible workload management features described in Section 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.

4.1 Workflow

Figure 2 sketches the workflow for our experimental framework.

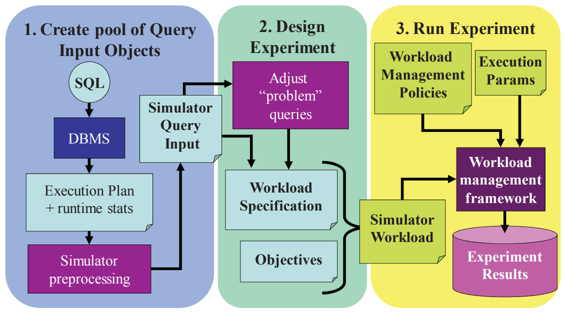

Figure 2: Workflow of how we create and select from a pool of query objects, create workload input files, and specify parameters for our experiments.

To create input, we first run queries in isolation on a real 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. 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.

4.2 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.

4.2.1 Query model

In a parallel database, a logical query operator, e. g., hash joins, may be implemented as multiple instances of a physical operator: one instance of the hash join operator run 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. Currently, we model only the cost of the dominant resource for each operator, e. g., the CPU time of an aggregation operator and the number of disk I/Os of a table scan.

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. Extending our simulator to mimic pipelined parallelism is on-going work.

4.2.2 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, 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 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 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.

4.4 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 3 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.

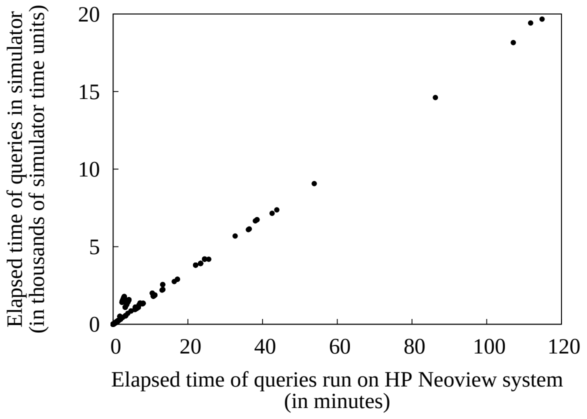

Figure 3: Simulator validation: We compare the elapsed times of queries run on an HP Neoview database with their simulated elapsed times. A straight diagonal line of points would indicate a perfect correlation. Note that 5000 simulator time units corresponds to roughly 30 minutes of elapsed time on the real system.

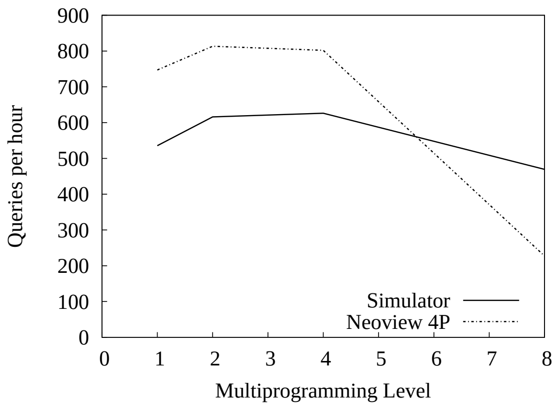

Figure 4: Throughput (queries per hour, QPH) of the same workload when run on the Neoview four processor machine and on the simulator. For the simulator, we derived the QPH using the 30 minutes == 5000 simulator time units formula.

To validate throughput, we measured queries processed per hour on the Neoview and simulator as the MPL was increased: 1, 2, 4, 8. For this test, we created eight different input streams of roughly equal numbers of queries and total duration.

Figure 4 shows the throughput for the real and simulated systems. Although the queries per hour (which are measured in different time units on the two systems) differ, the shapes of both curves are similar, indicating that the simulator does a reasonable job of modeling resource contention on a real system.

5. 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.

5.1 Queries and query types

Table 5: We created pools of candidate queries, categorized by the elapsed time needed to run each query on our 4-node Neoview database system.

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

Our experiments required a large pool of representative BI queries, including long-running "problem" queries. We started with the Decision Support benchmark TPC-DS [18]. 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 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 2.

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 5.

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. A typical 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.

5.2 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. 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.

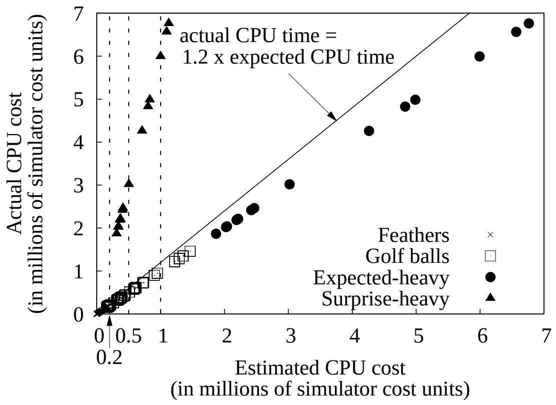

Figure 5: Comparison of estimated and actual CPU time for each query: the CPU time of the *surprise-heavy* queries is underestimated. The dashed vertical lines indicate our admission thresholds and the solid diagonal line shows our kill relative threshold.

We 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.

5.3 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.

5.3.1 Admission control

Admission control policies must accept or reject queries based on their *estimated* costs. Figure 5 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.

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 5.

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.

5.3.2 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 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.

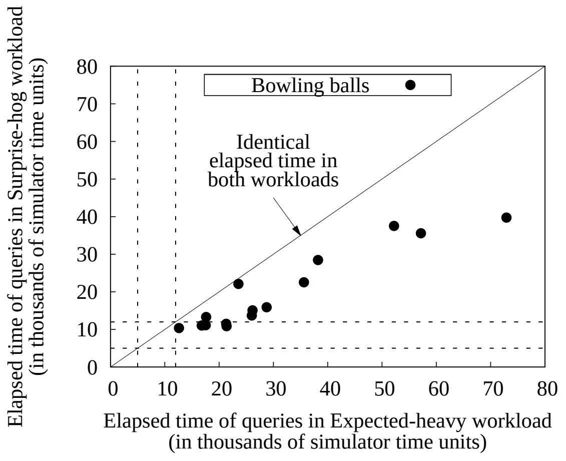

Figure 6: Comparison of elapsed times of long-running queries in Expected-Heavy and Surprise-hog workloads at MPL=4. All *surprise-hog* queries complete faster than their *expected-heavy* counterparts because they get a larger share of the resources. The dashed lines indicate our absolute kill thresholds.

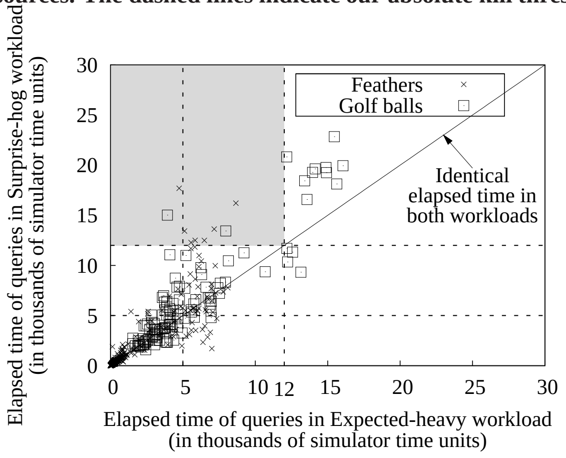

Figure 7: Comparison of elapsed times of feathers and golf balls in Expected-Heavy and Surprise-hog workloads at MPL=4. Queries above the diagonal run slower in the Surprise-hog workload. The dashed lines indicate our absolute kill thresholds. The gray shaded area denotes *starving* queries.

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 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. *2Qs, both MPL 4* then runs the second queue's queries at MPL=4 while *2Qs, different MPLs* runs those queries in isolation.

5.3.3 Execution control

The execution control policies we studied all base 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 6 shows these elapsed times for *expected-heavy* and *surprise-hog* queries and Figure 7 shows the times for golf ball and feather queries. We chose two absolute kill thresholds, both shown as dashed lines in Figures 6 and 7. 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 6.

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 6 also shows that each *surprise-hog* query, which is given a greater share of resources, completes faster than the corresponding *expected-heavy* query. Figure 7 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 5. 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 6.2 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.

5.4 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 [1]).

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: $M_{weighted} = M \cdot (1 + penalty)$.

6. 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 in Section 1.

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.

6.1 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 5. 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.

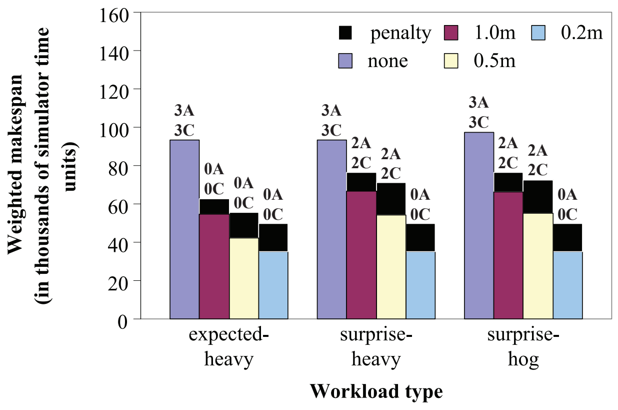

Figure 8: Comparison of admission thresholds on different query and workload types. The notations *nA* and *nC* above the bars indicate the number of admitted and completed long queries (out of the 3 submitted in each workload). Admission control is less effective when cost estimates are less accurate. Lower thresholds reject more "good" queries.

Figure 8 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 six golf balls, reducing the weighted makespan by another 13% (63k vs. 55k simulator time units). However, the penalty increases by 38% (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 9 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.

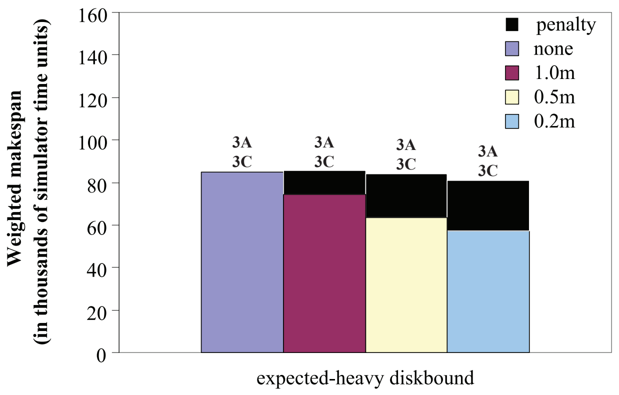

Figure 9: Admission thresholds are much less effective when the workload includes long-running queries that make heavy use of resources not measured by the admission threshold. In this case, the queries were disk-bound but admission control looked at CPU time estimates.

Table 6: Time to complete a certain percentage of queries (in thousands of simulator time units) when trying to put expensive queries in a separate queue.

% of queries complete	1Q	2Q both MPL=4	2Q 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

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

6.2 Scheduling

The scheduling experiments evaluate the impact of the scheduling policies *1Q*, *2Q both MPL 4*, and *2Q 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. (We omit the graphs due to space constraints.)

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 *2Q* policy is similar to *shortest-job-first* (SJF), which is known to improve latency for short jobs [7].

Table 6 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 *2Q* 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.

All three scheduling policies complete 99% of the queries in about the same amount of time. There is little difference between the *2Q* 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 *2Q* 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 *2Q* policy thus could complement a lenient admission threshold. However, the benefits of the *2Q* policy diminish with decreased accuracy of cost estimates, as more expensive queries are placed in the wrong queue.

6.3 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 10 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

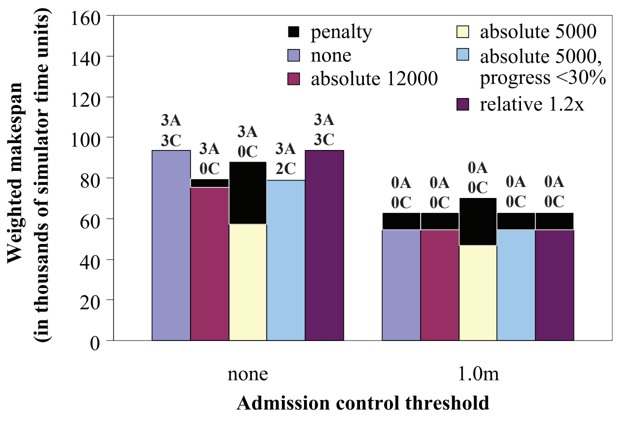

Figure 10: Comparison of absolute and relative *kill* thresholds in the Expected-Heavy workload: The lower absolute threshold kills many more queries unless their progress is checked.

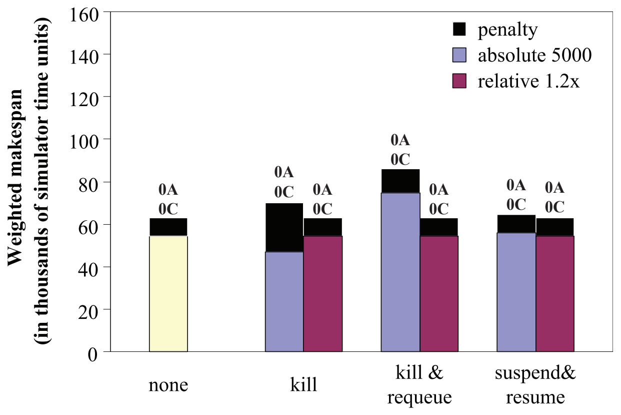

Figure 12: Comparison of execution control policies with admission threshold *1.0m* for Expected-Heavy workload. Rerunning the killed queries takes longer than if they were never killed, while suspending and resuming them does not.

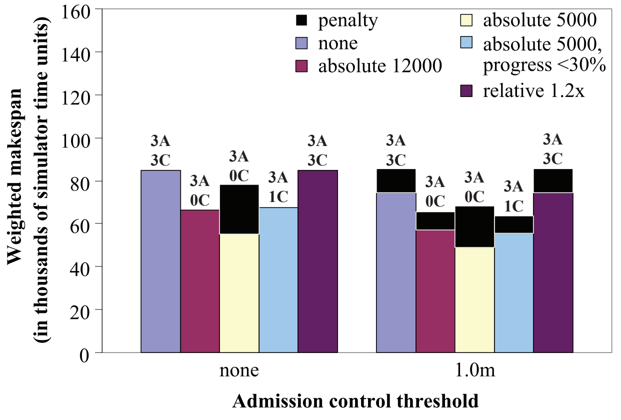

Figure 11: Comparison of absolute and relative *kill* thresholds in the Disk-Heavy workload: the relative threshold compares actual and estimated CPU time and thus does not catch the *disk-heavy* queries.

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 (omitted due to space constraints) follow from the Expected-Heavy workload, as well as the lessons learned from admission control: absolute kill thresholds are not impacted by estimates, and the effectiveness of progress thresholds depend on the accuracy of the estimates.

Figure 11 tests 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.

6.4 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.

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 12 shows the makespans for the Expected-Heavy workload. The results for the execution policies *none* and *kill* are repeated from Section 6.3. 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

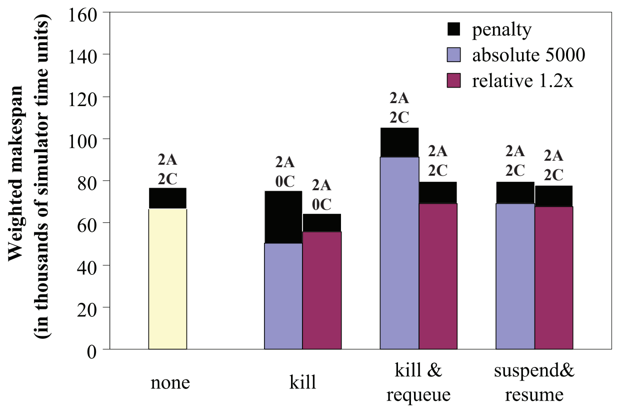

Figure 13: Comparison of execution control policies with admission threshold *1.0m* for Surprise-Heavy workload. Again, killing queries that need to be run later increases the makespan compared to not killing them.

Table 7: Time (in thousands of simulator time units) to complete a certain percentage of queries.

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

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 13 shows similar results for the Surprise-Heavy workload. The makespans are slightly longer compared to those in Figure 12, 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 7 also shows that by killing or suspending the longer queries, the makespan of the first 90% and 95% is greatly reduced. These 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,

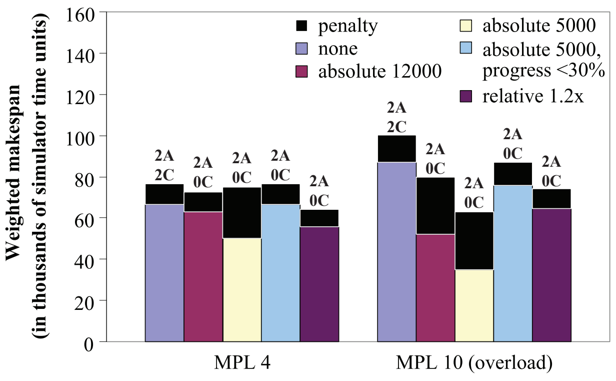

Figure 14: Comparison of execution control policies using admission threshold *1.0m* at MPL=4 and MPL=10 (overload) for the Surprise-Heavy workload. All queries take longer at MPL=10, so policies with absolute thresholds kill more queries.

they complete the less expensive 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.

6.5 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 (as described in Section 2.1.1).

Figure 14 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 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.

7. CONCLUSIONS

In this paper, we present a systematic study of workload management policies that mitigate the impact of long-running queries on performance. We propose a taxonomy that distinguishes between different types of long-running queries. We suggest a method for categorizing queries according to this taxonomy, using only cost estimates and simple runtime statistics. We then carry out a systematic series of experiments to investigate the effectiveness of known workload management policies on these different types of queries. We recommend 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 last. 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* policies 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.

8. REFERENCES

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

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

[3] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query Suspend and Resume. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2007.

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

[5] S. Chaudhuri, R. Kaushik, R. Ramamurthy, and A. Pol. 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.

[6] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating Progress of Execution for SQL Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 803–814, 2004.

[7] C. Chekuri and S. Khanna. *Approximation Algorithms for Minimizing Average Weighted Completion Time*, chapter 11. CRC Press, 2004.

[8] W.-J. Chen, B. Comeau, T. Ichikawa, S. S. Kumar, M. Miskimen, H. T. Morgan, L. Pay, and T. Väättänen. Workload Manager for Linux, Unix, and Windows. `http://www.redbooks.ibm.com/redbooks/pdfs/sg247524.pdf`, May 2008.

[9] HP Neoview Workload Management Services Guide, August 2007.

[10] S. Krompass, H. Kuno, U. Dayal, and A. Kemper. Dynamic Workload Management for Very Large Data Warehouses: Juggling Feathers and Bowling Balls. In *Proc. of the 33rd Conf. on Very Large Database (VLDB)*, 2007.

[11] N. Lei. Workload Management for DB2 Data Warehouse. `http://www.redbooks.ibm.com/redpapers/pdfs/redp3927.pdf`.

[12] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a Progress Indicator for Database Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2004.

[13] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Increasing the Accuracy and Coverage of SQL Progress Indicators. In *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE)*, 2005.

[14] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL Progress Indicators. In *10th Intl. Conf. on Extending Database Technology (EDBT)*, 2006.

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

[16] Oracle Discoverer Administrator Administration Guide 10g Release 2 (10.1.2.1). `http://download.oracle.com/docs/pdf/B13916_04.pdf`, July 2005.

[17] Oracle Database Resource Manager. `http://download.oracle.com/docs/cd/B28359_01/server.111/b28310/dbrm.htm#i1010776`, March 2008.

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

[19] Teradata. Teradata Dynamic Workload Manager User Guide, Release 13.0.0.0 (B035-2513-088A), August 2008.

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