Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 00 (2011) 1–22
Future Generation Computer Systems

Utility-Driven Adaptive Query Workload Execution

Norman W. Paton, Marcelo A.T. de Aragão, Alvaro A.A. Fernandes

School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK.

Abstract

Workload management coordinates access to and use of shared computational resources; adaptive workload execution revises resource allocation decisions dynamically in response to feedback about the progress of the workload or the behavior of the resources. Where the workload contains or consists of database queries, adaptive query processing (AQP) changes the way in which a query is being evaluated while the query is running. In parallel environments, available adaptations may change the allocation of query fragments to a machine, for example to remove load imbalance or change the parallelism level. Most AQP strategies act on individual queries with the objective of reducing response times. However, where adaptations affect the usage of shared resources, or the principal goal is to meet quality of service targets rather than to minimize overall response times, locally beneficial decisions may have globally detrimental effects. This paper describes the use of utility functions to coordinate adaptations that assign resources to query fragments from multiple queries, and demonstrates how a common framework can be used to support different objectives, specifically to minimize overall query response times and to maximize the number of queries meeting quality of service goals. Experiments using simulation compare the use of utility functions with the more common heuristic control strategies, demonstrating situations in which significant benefits can be obtained.

Keywords: Autonomic Computing, Utility Function, Adaptive Query Processing

1. Introduction

This paper describes an approach to workload management for database queries, in which query workloads are adaptively scheduled on shared resources, with a view to maximizing a utility function that reflects users expectations. Although the paper focuses on query workloads, these can be seen as representative of composite requests that are amenable to pipelined and/or partitioned parallelism.

Large scale query workload evaluation commonly makes use of parallel platforms, and thus involves the allocation of query plan fragments to resources from multiple computational nodes, classically with a view to minimizing query response times. Access to high performance computational resources is often governed by quality of service (QoS) agreements, whereby different users or types of request have different response time expectations; such goals have implications for database administration (e.g. for ensuring that data warehouse update tasks are completed within a batch window [32]), for query evaluation (e.g. for guiding the selection among equivalent data sources in distributed query processing [31]), and for workload execution (e.g. for selecting queries for suspension [28]). This paper, building on research on grid query processing [37], addresses the problem of how to dynamically allocate computational resources to a set of queries, including the case where each query has a target response time goal resulting from a QoS agreement.

Parallel query processing stands to benefit from the use of shared computational resources, made available by way of grids (e.g. [34, 55]), local area networks (e.g. [50]) or the internet [6], in which a query may be running both at

the same time as other queries and in competition with tasks other than query evaluation. As a result, the environment in which parallel queries are run can be both changeable and unpredictable. It may therefore be difficult for a static query optimizer to construct plans when queries are submitted that correctly anticipate the computational resources that will be required or available to enable a query to meet its QoS goal. We note that this is not always a question of minimizing the individual or aggregate response times of all queries; minimizing global response time measures may not maximize conformance to QoS goals that are defined on a per-query basis. As such, there is a requirement to defer commitment to a specific resource usage pattern until requests are running, and to support adaptations that address different goals.

Although Adaptive Query Processing (AQP) helps to avoid premature commitment to potentially inappropriate strategies, little work has been carried out to determine how adaptive techniques can work together to obtain globally desirable behavior. Adaptive techniques are typically: *independent*, in that decisions are made without considering what other adaptations may be available; *selfish*, in that the goal of each adaptation is to bring a benefit (typically reduced response time) to a single query; and *heuristic*, in that decisions are made on the basis of rules and thresholds, the potential behaviors of which can be difficult to predict, both in isolation and in combination. Policies that direct decision-making in autonomic systems have been classified into three types [26]: *action policies*, in which the behavior of the system is captured using condition-action rules; *goal policies*, in which one or more desired states are identified and a planner identifies actions that should lead to that state; and *utility function policies*, in which the value of different outcomes is quantified, and an optimization activity seeks to identify actions that maximize utility. For the most part, current adaptive query processors, either explicitly or implicitly, deploy action policies.

This paper describes an approach in which utility functions inform which adaptations take place, with a view to coordinating the use of shared resources by queries. In the approach, rather than being independent, selfish and heuristic, adaptations are: *interdependent* – the potential benefits from several adaptive strategies involving multiple queries are considered together; *selfless* – the wider objective of meeting global QoS goals is given precedence over minimizing the response times of individual queries; and *model-driven* – the consequences of a collection of adaptations is predicted before individual adaptations take place. In this paper, the QoS goal takes the form of a response time target, which indicates the amount of time within which a query must be evaluated. However, the methodology, whereby the goal of the adaptation is expressed as a utility function and solutions are explored with a view to maximizing utility, is applicable to other notions of QoS.

This paper considers four adaptive strategies, as follows:

1. *Non-QoS-Aware Action Policy:* adaptations are defined as action policies, where actions adapt queries individually, with a view to minimizing individual query response times.
2. *QoS-Aware Action Policy:* adaptations are defined as action policies, where actions adapt queries individually, with a view to meeting the QoS goals of individual queries.
3. *Non-QoS-Aware Utility Policy:* adaptations are defined using utility policies, where the adaptations are coordinated across multiple queries, with a view to minimizing the sum of the query response times.
4. *QoS-Aware Utility Policy:* adaptations are defined using utility policies, where the adaptations are coordinated across multiple queries, with a view to maximizing the number of queries that meet their QoS goals.

These strategies play different roles in the paper. Both (1) and (2) are extensions of an existing strategy [54], and are included here to allow the behavior of the utility-based techniques to be compared with that of representative action-based methods. By contrast Methods (3) and (4), and the techniques by which they support utility-based workload execution, embody the key results of the paper. To the best of our knowledge, this is the first paper on adaptive query processing in which: (i) a single planning process coordinates adaptations over multiple queries; and (ii) queries are adapted to support objectives other than response time minimization. This paper can be seen as a successor to an earlier comparison of action-based methods for adaptive load balancing [46]; the contribution here is to demonstrate the use of utility-based techniques for adaptive query processing where previously only action-based techniques had been deployed.

We see the utility-based approach as being promising for query workload management because: (i) it enables explicit, declarative specification of the goals of the adaptation from which suitable adaptations can be identified using an optimization algorithm; (ii) it can support different goals through small-scale changes to utility function definitions; (iii) it supports coordinated optimization of multiple queries; and (iv) it is able to build on the large body of work on optimization algorithms [5].

The contributions of this paper are as follows:

1. A description of how utility functions can be used to rank alternative resource allocations for parallel query plans, including the case where the queries have individual QoS goals;
2. A description of an approach whereby the generation of parameter values for the utility function that coordinates adaptations across multiple queries is cast as an optimization problem; and
3. An empirical evaluation of the resulting techniques, in comparison with non-adaptive resource allocation and heuristic-based adaptive techniques.

The remainder of the paper is structured as follows. The relationship between the techniques described in this paper and other work on adaptive systems is discussed in Section 2. Section 3 describes the problem of adaptive query scheduling, and introduces terminology that is used later in the paper. Section 4 details the four adaptive techniques that are subsequently compared for a range of scenarios in Section 5, and conclusions are presented in Section 6.

2. Related Work

This paper investigates how the behavior of an adaptive query processor can be controlled in a way that reflects explicitly stated goals, including QoS goals represented as response time targets. This section reviews related work on *adaptive query processing*, *control of adaptive query processing*, on *the use of utility functions in adaptive systems*, and on techniques that address *explicit support for QoS* in databases.

In terms of *adaptive query processing*, strategies can broadly be classified as *plan preserving*, in which adaptation takes place continuously throughout evaluation over an essentially stable representation of a query; and *plan changing*, in which evaluation is halted, and some form of planning activity gives rise to a revised plan, with which evaluation is resumed. The *plan preserving* approach characterizes proposals such as Eddies [1], extensions of which have been developed for parallel settings [58]. In Eddies, the order in which data is routed through operators is determined dynamically on the basis of the costs and selectivities of the operations, and as a result join orders are adapted at query runtime. However, as Eddies require significant amounts of state to be maintained to allow flexible rerouting, their overheads can be considered to be significant. The *plan changing* approach characterizes proposals such as Tukwila [23], POP [38] and Rio [2]. Plans may be changed either by switching between statically determined alternatives (e.g. [11]) or by reinvoking the optimizer at runtime to create a new plan (e.g. [38, 15]). In plan changing approaches that rerun the optimizer, there is a need to generate a new plan that completes the work that remains to be done, and in many proposals including Dynamic Re-Optimization [25] and POP [38] the replacement plans can reuse materialized intermediate results from operators that have run to completion, but the results of any partially computed subqueries are discarded. As such, the unit of reuse is rather coarse, and unsuitable for use with pipelined evaluation, as used in this paper. In parallel query processing, adaptive techniques have often operated at quite a coarse grain, with some proposals only adapting the behavior of the system between queries (e.g. [14, 49]) or on operator completion (e.g. [58]), and thus address a coarser grain than more recent adaptive load scheduling techniques such as Flux [54] or DITN [50]. This paper, like Flux, addresses the problem of rebalancing load during operator execution, but differs in using utility functions to inform planning decisions, as discussed below. In carrying out adaptations that are essentially the same as those supported by Flux, which was developed for use in continuous query processing, we note that many of the results from this paper carry over to continuous query systems, where the requirement to support different qualities of service through adaptation has also been identified (e.g. [56]). However, in seeking to apply results more widely, we note that the architecture used to run parallel queries has a significant impact on the issues that affect adaptation; as an example, adaptation in XPRS [21] may occur during operator execution, but the shared-memory model avoids need for operator state movement that has a significant impact in shared-nothing settings.

In terms of *control of adaptive query processing*, as mentioned in the introduction, most proposals implicitly or explicitly adopt action policies. Thus although there is some work on the use of control theory (e.g. [33]) or economic approaches (e.g. [47, 48]) for adapting query processing behavior or resource usage in databases at runtime, the principal focus here is on action strategies. Implicit support for action policies is implemented through changes to the code of a static query evaluator, for example through the introduction of operators that monitor progress and respond where necessary (e.g. [38]) or through the introduction of operators that encapsulate the complete adaptation (e.g. [54]). Explicit support for action policies is provided by rule languages that relate monitoring conditions to

adaptation actions (e.g. [22, 43]). Action policies can be effective, but are generally associated with heuristics and thresholds the precise behavior of which can be difficult to predict. For example, in Flux [54] the frequency and scale of adaptations is determined by the following heuristics: (i) a table is divided into a number of partitions, which specifies the minimum amount of state that may be moved between parallel nodes, and thus places a lower bound on the scale of any adaptation; (ii) at any adaptation, a node can receive or lose at most one partition, so the size of a partition also places an upper bound on the change in workload assigned to any machine; (iii) in any adaptation, at most half the partitions can be moved; and (iv) once an adaptation has taken place, no further adaptation can take place until query evaluation has been underway for at least as long as the time taken by that adaptation. In essence, these heuristics exist to minimize the effects of expensive adaptations taking place in response to temporary environmental changes, but their presence indicates that controlling action policies is not straightforward. Indeed, an evaluation of several adaptive load balancing techniques controlled using action policies identified circumstances in which all of the adaptive techniques performed worse than a static counterpart [46]. Although some adaptive techniques implemented using action policies seek to avoid over-adaptation through the development of models that make informed decisions as to when a plan may be sub-optimal (e.g. by recording with a plan the range of selectivities for which it was considered by the optimizer to be appropriate [38, 3]), even where such analyses can be carried out it is still possible for undesirable over-adaptation to take place, and [38] includes an upper limit on the number of adaptations that can be carried out on a query. Thus it seems to be the case that decision making in adaptive query processing can present significant challenges, and thus that there is scope for further work on techniques for controlling adaptive behaviors.

In terms of *the use of utility functions in adaptive systems*, we know of no other examples in adaptive query processing, although utility functions have been used to adaptively configure a middleware used to support the evaluation of stream queries [29]. Our use of utility functions in this paper is analogous to several other applications in adaptive systems in different areas; such applications have in common the identification of a quantifiable utility measure (in our case, either the total response time or the number of queries meeting their QoS target), the definition of functions that characterize that utility in terms of variables that can be measured or parameters that can be set, and the execution of an optimization algorithm that seeks to identify values for the parameters that maximize the utility function. Example applications reported in the autonomic computing literature have generally involved systems management tasks, such as service selection or resource allocation in data centers [4, 40, 57].

The provision of *explicit support for QoS* targets has been explored in several areas (e.g. [41, 42]), and in the database area, techniques have been investigated that adaptively configure system properties that influence query evaluation. For example, proposals have been made that adaptively configure buffer pools to reflect the needs of different kinds of e-commerce transactions [39], that dynamically alter the amount of resource allocated to different categories of query (OLTP or OLAP) depending on their service level objectives [44], and that dynamically replicate data resources onto which query fragments are allocated, where both replication and allocation take into account QoS requirements [31]. Such results fall into the general category of self-tuning database systems [7], in that the behavior of database server components is revised dynamically in a way that reflects QoS goals for all queries or for categories of queries, but few proposals have been made for supporting QoS goals within the query evaluator (e.g. [31]). We see work on the autonomic tuning of database management system parameters as complementary to adaptive query processing, where adaptations are applied to the queries rather than to the underlying infrastructure. Furthermore, although there has been explicit consideration of utility in static query optimization, in particular with a view to identifying plans that reflect the most likely runtime conditions [10], such an approach does not benefit from dynamically changing information about the environment during query runtime. Closer to the work described in this paper is the area of database workload management, in which the performance of collections of queries can be managed by [28]: (i) an *admission controller*, which seeks to identify and disallow access to potentially problematic requests; (ii) a *query scheduler*, which determines when jobs are released from a queue for execution; and (iii) an *execution controller*, which determines the level of resource allocated to queries while they are executing. In the context of workload management, this paper can be considered to be providing a utility-driven *execution controller*. In comparison with recent work on workload management, our utility-driven approach provides relatively fine-grained control over queries; for example, [27] describe an execution controller in which the actions carried out at query runtime are job-level (i.e., reprioritize, kill and resubmit), whereas the focus here is on finer grained decision making on resource allocation. The work described in this paper falls into the general category of adaptive query processing, in that adaptations take place within executing queries, rather than at the level of the platform on which the queries are running or the queries as a whole. To the best of our knowledge this paper is the first to direct adaptive query

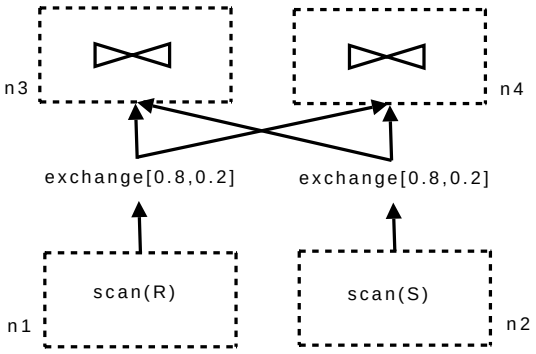


Figure 1: Example Parallel Execution Plan

processing using utility functions, and thus to investigate how utility functions can be used to support alternative adaptation goals. In so doing, we describe utility functions that reflect performance properties of individual queries and of groups of queries, thus addressing both types of service level objective identified in workload management [28].

3. Adaptive Workload Execution

3.1. Technical Context

The context for our contributions is one in which a set N of compute nodes is available for processing a set Q of queries in parallel. Queries are expressed using a parallel algebra consisting of the operators of the relational algebra (e.g. [17]) plus *exchange* [20]. In this paper the following operators are discussed directly:

- *scan*(R): return the tuples from the persistent collection R.
- $R \bowtie_p S$: return tuples obtained by concatenating the tuples from R and S, where the predicate p is satisfied; p is assumed to consist of a conjunction of equalities on attributes of R and S. Where the details of p are not relevant to the discussion, it is omitted.
- *exchange*[dv](R): redistribute the tuples from R over the n parent operators of the exchange, allocating tuples to parents following the fractions in the distribution vector dv, such that $dv = [v_1, \ldots v_n]$.

Using the above operators, a parallel query consists of a collection of plan fragments F, such that each fragment $f \in F$ is assigned to a compute node $n \in N$, and communication between fragments is represented by exchange nodes. For example, Figure 1 illustrates a query with four fragments (denoted by the dotted rectangles) each allocated to one of the nodes *n1*, *n2*, *n3* or *n4*. The query joins the persistent relations R and S, with the join being run in parallel on nodes $n3$ and $n4$. The *exchange* operators redistribute the data from R and S across the nodes $n3$ and $n4$ such that the fraction of the data sent to $n3$ is 0.8 and the fraction sent to $n4$ is 0.2. This uneven distribution might have been chosen to reflect the fact that $n3$ is more powerful than $n4$, or that $n4$ is more heavily loaded than $n3$. Tuple distribution in exchanges is typically carried out using a hash function on the join attributes, which must be implemented consistently on both exchange operators, thereby ensuring that all potentially matching $R-$ and $S-$tuples are sent to the same node.

In this setting, workload management determines how to allocate query fragments to nodes, and how data should be distributed across parallel fragments. As an example, consider the query $q = A \bowtie B$, where the join is parallelized into a collection of plan fragments $f_i = A_i \bowtie B_i$, for $1 \leq i \leq P$, where P is the level of parallelism being used to evaluate q. Each of the fragments f_i is allocated to a different computational node. Any delay in the completion of a fragment f_i relative to the other fragments delays the completion of the query as a whole. Thus load balancing aims to make the evaluation times of each f_i as similar (and as small) as possible, by matching the amount of work to be done on each node to the observable performance of the node.

Throughout this paper, without loss of generality, we focus on hash join operators. In a hash join, all tuples from one attribute are read and stored in a hash table indexed on the join attributes, subsequent to which the tuples from

the other operand are scanned and used to probe for matching tuples in the hash table. Load balancing is particularly challenging for stateful operators, such as hash join, because maintaining a balanced load involves ensuring that (i) the portion of the hash table on each node reflects the required work distribution, and (ii) changes in the work distribution to maintain load balance are preceded by corresponding changes to the hash table on each node. As a result, adaptively changing the load balance for stateful operators may incur significant cost, and thus adaptation is only beneficial when the benefits that result from improved load balancing outweigh the costs of transitioning to a balanced state.

We note that load balancing for stateless operators, such as calls to external operations, is more straightforward than for stateful operators, in that there is no need to ensure that the state of the operator is appropriately located before changing the flow of data through parallel partitions [19].

3.2. Possible Adaptations

In the non-adaptive case, for each query, once its distribution vector is determined, it remains unchanged until the end of the evaluation of the query. In the adaptive case, the initial distribution vector of each query may be changed.

In what follows, the proportion of the work associated with $q \in Q$ that is assigned to each node $n \in N$ is given by a distribution vector $d(q) = [v_1, v_2, \ldots, v_{|N|}]$, where $0 \leq v_i \leq 1$ and $(\Sigma_{i=1}^{|N|} v_i) \in \{0, 1\}$. Note that if the sum yields 0, this represents the suspension of the plan. Thus, at any point t in the evaluation of a query q, the evaluation strategy of q up to t can be characterized by a sequence of distribution vectors $D(q, t) = \langle d_1(q), d_2(q), \ldots, d_{|D(q)|}(q)\rangle$, where $d_1(q)$ is the initial distribution vector for q and each subsequent $d_j(q), 1 < j \leq |D(q)| \leq t$ is either the same as its predecessor or else the outcome of an adaptation. Let $D(q, t)$ be called the *(evaluation) trace* of q up to t. Note that this definition associates a distribution vector with a complete query, rather than with individual operators in the query.

The adaptations we consider in this paper can be defined in terms of *workload redistribution*. When workload is redistributed, some proportion of the overall work that is currently assigned to a compute node is reassigned to other node(s). The following adaptations can be seen as special cases of workload redistribution:

1. *increased parallelism* is the case in which a compute node that previously had no work assigned to it is given some proportion of the workload being redistributed from other node(s);
2. *reduced parallelism* is the case in which a compute node that previously had some work assigned to it is relieved of that workload, which is redistributed to other node(s); and
3. *suspension* is the case in which no work is assigned to any node and the evaluation of a query is temporarily halted.

More formally, consider two successive states in the evaluation trace of a query, i.e., two distribution vectors, d_j and d_{j+1}. If, for a compute node i, $d_j[i] = d_{j+1}[i]$, then the strategy has led to a decision that no change is required in the proportion of work assigned to i. If, on the other hand, $d_j[i] \neq d_{j+1}[i]$, then the strategy has led to a decision that workload redistribution is necessary, i.e., that the proportion of work assigned to i needs to increase or decrease by $d_{j+1}[i] - d_j[i]$. The constraint that the proportions of work assigned to each node sum to either zero or one means that if an adaptation does not result in the query being suspended, then any increase (resp., any decrease) in the workload assigned to a given node implies a decrease (resp., increase) in the workload assigned to other node(s).

Note that an increase in parallelism is the case in which $d_j[i] \neq d_{j+1}[i]$ and $d_j[i] = 0$. Likewise, a decrease in parallelism is the case in which $d_j[i] \neq d_{j+1}[i]$ and $d_{j+1}[i] = 0$. Finally, a query q has been suspended at time j where $d_j(q)[n] = 0$ for all $n \in N$. Thus, at a conceptual level, changes resulting from any combination of adaptations can be represented in terms of the distribution vectors before and after each adaptation takes place. In this paper, the distribution vector is set at the level of the query, and thus applies to all operators in a query; as a result, any change to the distribution vector may involve redistribution of operator state from more than one operator.

In practice, any implementation of the above adaptations where partitions are stateful must employ a mechanism for relocating operator state in a way that reflects the changes to the distribution vector. Proposals have been developed in which such state is transferred between sibling partitions in Flux [54] and from upstream caches in OGSA-DQP [19]; in this paper, we adopt the former approach. In essence, each input table consists of a number of partitions, which is much higher than the parallelism level, and which is the unit of redistribution; a sensitivity analysis has demonstrated that overall performance of adaptive strategies is not sensitive to moderate changes in the number of partitions [46].

Given these partitions, when state needs to be relocated for any of the adaptations described above, a protocol is supported whereby evaluation is halted, state is relocated between sibling partitions to reflect the updated distribution

vector, and evaluation is resumed. A full description of such a protocol is provided in the paper on Flux [54]; the performance of different halting protocols is discussed in [19].

As such, the adaptations in this paper have the following features: (i) individual adaptations are potentially expensive, as they involve the relocation of operator state; (ii) adaptations become more expensive as a query progresses because hash tables become more fully populated; and (iii) adaptations do not discard work that has taken place, and thus never involve the repetition of work that has already been done. Both (i) and (ii) are challenging to address, and in a previous comparison of methods several scenarios were identified in which adaptations had a negative impact on performance [46]. In this paper, unlike [46], strategies only adapt when the adaptation is predicted to be beneficial using information from progress monitoring and a cost model. In the context of such cost and progress information, adaptations rarely reduce performance compared with static allocations in the experiments in this paper.

4. Adaptive Strategies

This section provides details of the four adaptive strategies outlined in Section 1. The adaptive strategies make use of some common definitions, which are provided here.

Monitoring Log. The outcome of runtime monitoring can be characterized as follows. Let L denote the log of information captured about the execution:

- $L.load[t, n]$ represents the average load on the node $n \in N$ during a period leading up to t; in the experiments the period is 0.1s. For example, if one job seeks to make full time use of a node and another job requires the node only half the time, the load on the node will be *1.5*; in essence, following UNIX, the load represents the average of the sum of the run queue length and the number of jobs running on a CPU.
- $L.startTime[q]$ denotes the time when query q was submitted for evaluation.
- $L.returned[q, o, t]$ represents the number of tuples returned by operator o in query q between $L.startTime[q]$ and t.

Progress Model. We assume that we have access to an estimate (e.g. from the optimizer's cost model) of the number of tuples that will be returned by the operator o in q, by way of the function $OperatorCardinality(q, o)$. We use the proportion of the total number of tuples returned by operators in a plan at time t as an estimate of overall query progress at that time:

$$progress(q,t) = \frac{\Sigma_{[o \in q]} L.returned[q,o,t]}{\Sigma_{[o \in q]} OperatorCardinality(q,o)}$$

For the pipelined select-project-join queries considered in the experiments, this has proved sufficiently accurate to allow informed decision making. Several models for progress estimation that can be used to estimate progress for a wider range of queries are described in the literature (e.g. [8, 18, 36]); we principally note here that we make use of progress information that is readily available, and that empirical evaluations of runtime progress monitoring approaches such as those citied have shown that runtime estimates can often quickly improve on predictions based on statically maintained statistics.

The function QT uses progress information to estimate, at a time t, $t > L.startTime[q]$ during the evaluation of q, the response time for a query q as:

$QT(q, t) =$
  let $elapsed = (t - L.startTime[q])$ // Time so far
  in $elapsed / progress(q, t)$

As adaptations incur costs, we also use a cost model that to estimate the cost of relocating operator state during adaptations. Given the current distribution vector $d_t(q)$, we use the adaptation-cost model, AT to estimate the cost of moving to a new distribution vector $AT(d_t(q) \rightarrow d_{t+1}(q))$. The adaptation time can be predicted using a cost model similar to that used to represent the performance of other database operations (e.g. [51]), including terms for extracting state from an operator, moving that state to another computational node, and updating the operator state in the destination node. We assume that if $d_{t+1}(q) = d_t(q)$ then $AT(d_t(q) \rightarrow d_{t+1}(q)) = 0$.

Predicted Response Time. In all the adaptive strategies defined later in this section, it is necessary to be able to predict when a query will complete. *PRT* represents the *predicted response time* of a query, and is defined as follows:

$$PRT(q,t) = QT(q,t) * S(q,t) + AT(d_t(q) \rightarrow d_{t+1}(q))$$

where QT represents the predicted response time of q at time t, S estimates the speedup or slowdown of q resulting from adaptations made to the distribution vectors of all the queries being considered together for adaptation at time t, and AT estimates the time taken to carry out the adaptation from $d_t(q)$ to $d_{t+1}(q)$.

The speedup or slowdown S represents the ratio of the execution time of the query with the proposed adaptation to the execution time without the adaptation, so where S is less than 1 the adaptation is predicted to reduce query execution times. The utility based strategies consider the *combined* effect of multiple adaptations, so to support the utility strategies S must be able to take into account the impact on q of the different adaptations that are being planned together.

The speedup or slowdown of a query in the context of an adaptation thus depends on: (i) the level of competition for resources its fragments are subject to using $d_t(q)$; and (ii) the level of competition for resources its fragments are subject to using $d_{t+1}(q)$. These in turn depend on: (i) the *external load* – the load that results from activities other than the evaluation of queries being considered together for adaptation; and (ii) the load that results from the *current allocation* of queries being considered together for adaptation to resources, and the load that results from the *candidate allocation* of those queries to resources proposed by an adaptive strategy. The load that results from the *current allocation* in combination with the *external load* is assumed to be available from in the Monitoring Log as *L.load* (see above), and the *predicted load* experienced by the *candidate allocation* is estimated based on *L.load* and the query allocations before and after adaptation. The specific calculations used to derive S are provided in Appendix A.

In the action strategies, individual queries are adapted independently, so for the action strategies the work being done by other queries is considered to be external. In contrast, by considering multiple adaptations together, the utility strategies are able to coordinate adaptations across multiple queries. This provides some measure of protection against ineffective adaptations, such as overwhelming a lightly loaded node with additional work from multiple queries so that it becomes a bottleneck.

Evaluation Model. Without loss of generality, we assume that query evaluation is pipelined, (e.g. making use of the iterator model [20]). At regular intervals (every *0.1*s in the experiments) during the evaluation of *exchange* operators, an operation $plan^{Strategy}$ is invoked, which computes a revised distribution vector for the exchange, as discussed in Section 3. The *Strategy* is one of *Non-QoS-Aware Action*, *QoS-Aware Action*, *Non-QoS-Aware Utility* or *QoS-Aware Utility*, as introduced in Section 1 and described in detail in the following subsections. Where the *plan* proposes a new distribution vector, the query is suspended, state associated with join operators is transferred between nodes in a way that matches the requirements of the new distribution vector, and execution is resumed; the paper on Flux provides a detailed description of a protocol for suspending evaluation, moving state, and resuming evaluation, as required by all the adaptive strategies used here [54]. Note that this means that the only difference between the strategies described is in their decision making (w.r.t. when to adapt and what to adapt to), as adaptations are carried out using the same protocol for query suspension and state movement.

4.1. Non-QoS-Aware Action Policies

Problem Statement. The problem we address in this section can be informally stated as follows: adapt in order to reduce the response time of every query independently, and independently of any prediction as to whether the final response time for the query in question will meet any set target.

More formally, given a query executing at time t, $q \in Q$, a set of compute nodes N, $|N| \geq 1$, and a log L, find a distribution vector d_{t+1} that defines how much work, in proportion, is to be done by each plan fragment of q, taking into account the loads on the compute nodes of N to which the fragment is allocated.

Action-Based Solution. An adaptation is assumed to be worth considering if an opportunity exists to reduce the predicted response time: (i) by increasing the level of parallelism up to some maximum; and/or (ii) by redistributing work to remove load imbalance.

fun $plan^A(q, N, t, L)$
  // Keep adding parallelism while it is predicted to be beneficial
  $current\_time = QT(q, t)$
  $current\_distribution = d_t(q)$
  do
    $best\_time = current\_time$
    $best\_distribution = current\_distribution$
    // Add the most lightly-loaded node to the resource pool
    $N_q = \{n \in N | d_t(q)[n] > 0\}$
    $N_q = N_q \cup$ **any** $\{n \in (N - N_q) |$
        $L.load[t, n] = min_{n' \in (N - N_q)}(L.load[t, n'])\}$
    // Compute alternative distribution vector for q
    $d_{t+1}(q) = pd(q, N_q, t, L)$
    $current\_time = PRT(q, t)$
    $current\_distribution = d_{t+1}(q)$
  until $current\_time \geq best\_time$ **or** at maximum parallelism level
  if $best\_time < QT(q, t)$ **return** $best\_distribution$
  else return $d_t(q)$
endfun

Figure 2: $plan^A$: compute the next distribution vector in Non QoS-aware action policies.

The function $plan^A$, given a query q, a collection N of nodes onto which the query can be assigned, the current time t, and the execution log L, returns the distribution to be used next, $d_{t+1}(q)$. If the result of $plan^A$ is equal to $d_t(q)$, no adaptation takes place.

The algorithm for $plan^A$ in Fig. 2 iteratively extends N_q, the set of nodes used by q, with the most lightly loaded node on which q is not presently running. Given those nodes, a distribution vector is computed for q taking into account the loads on the nodes at time t.

The new distribution vector is computed using $pd(q, N, t, L)$ in Fig. 3 which, given a query q, a collection of nodes N to which the query is to be assigned, a time t, and the execution log L, returns a distribution vector for q for which its load would be balanced at t. In essence, given the load on a node, the definition assumes that if the load is less than 1, then this load comes from the query being evaluated, and that the complete resource on that node is available for the query. In contrast, if the load is greater than 1, then the assumption is that the query will be able to access the resource with equal rights to the other sources of load; thus if the load on the node is 2 then the evaluator will have access to half the capabilities of the node[1].

In essence, the Non-QoS Aware Action Policy described above and used in the experiments is Flux [54] extended with the ability to increase the parallelism level, with fewer restrictions on the maximum size of adaptation, and with the condition that adaptation only takes place if it is predicted to give rise to an improved response time. We note that the inclusion of the condition that the method should only adapt when this is predicted to be beneficial significantly improves the performance of the approach used in the paper compared with the original proposal.

4.2. QoS-Aware Action Policies

Problem Statement. The problem we address in this section can be informally stated as follows: adapt in order to reduce the response time of every query independently, but only if the final response time for the query in question is predicted to miss its set target.

[1] Different heuristics can be used for deriving a proposed distribution. For example, OGSA-DQP [19] takes into account both the rate at which data is being processed by a node and the load on a node in determining what fraction of the work to send to each node, whereas Flux [54] assigns work in inverse proportion to the load. The approach described in Figure 3 performed better in experiments than either of these existing proposals

fun pd(q,N,t,L)
  // Compute the availability of each node, which is the
  // fraction of the resource on the node to which a query
  // can expect to obtain access if required
  forall $n \in N$
    $a[n]$ = **if** $L.load[t, n] < 1$ **then** 1 **else** $1/L.load[t, n]$
  endfor
  // Derive the distribution vector from the availability,
  // so that more available nodes are assigned more work
  forall $n \in N$
    $d(q)[n] = a[n]/\Sigma_{j=1}^{N}(a[j])$
  endfor
  return $d(q)$
endfun

Figure 3: *pd*: compute the proposed distribution for a query at time t, taking machine loads into account.

More formally, if $TT(q)$ is the target response time (and thus the QoS goal) of a query q, the planner takes this into account when computing a new distribution vector $d_{t+1}(q)$. In particular, if the predicted response time $QT(q, t)$ is smaller than $TT(q)$, then $d_{t+1}(q) = d_t(q)$, i.e., no adaptation takes place.

Action-Based Solution. QoS-Aware planing can be implemented simply as:

fun $plan^{AQ}(q, N, t, L)$
  if $QT(q, t) \leq (TT(q) * k)$ **return** $d_t(q)$
  else return $plan^{A}(q, N, t, L)$
endfun

The constant k, $0 < k \leq 1$ allows for inaccuracies in QT, by enabling adaptation to take place when QT is within some fraction of TT[2]. So, the outcome of assessment in a QoS-aware adaptive query processor only leads to an adaptation taking place if the target response time is not currently on course to be met. This has the benefit of avoiding potentially costly state movement, and thus increasing overall resource usage, where the target is not being missed. The disadvantage of adapting purely on the basis of missing targets is that opportunities for reducing response times are not exploited unless a target is predicted to be missed.

To summarize, both the Non-QoS Aware and the QoS-Aware Action policies take decisions on the basis of basic monitoring information on individual queries, without seeking to predict the consequences of their actions, features they share with other representative adaptive scheduling techniques (e.g. [54, 19, 50]). This selfish independent behavior may lead to globally undesirable consequences.

4.3. Non QoS-Aware Utility Functions

For an adaptive strategy to bring about a coordinated response to events, it must consider the entire set of executing queries and possible adaptations together. We follow [26] in construing a utility function as a means of assigning a value to the desirability of each state that a planner may consider adapting to. Given a utility function, it is then possible to search the space of responses for the one that maximizes utility.

Problem Statement. The problem we address in this section can be informally stated as follows: adapt in order to minimize the sum of the response times of a set of queries, independently of any prediction as to whether that sum will meet any set target.

More formally, given a set of queries Q, $|Q| \geq 1$, and a set of compute nodes N, $|N| \geq 1$ to which query fragments can be allocated, minimize $\Sigma\{PRT(q, t)|q \in Q\}$, where PRT represents the *predicted response time* of a query, as defined in the lead-in to Section 4.

[2]In the experiments, k is set to *0.2*, so we consider adapting unless there is strong evidence that response time targets are going to be met.

We note that the declarative statement of the problem (namely that the goal is to minimize the sum of the predicted response times) gives rise in practice to desirable behaviors that may be overlooked by developers of action-based techniques. For example, the inclusion of the adaptation cost in the predicted response time prevents expensive adaptations from being carried out late during the evaluation of a query, as the cost of the adaptation will outweigh the predicted benefits.

Utility-Based Solution. Given that our goal is to minimize overall response times, the utility of a set of queries is in an inverse relation with the sum of their response times[3]. Thus we define the utility of a set of queries as follows:

$U^{RT}(Q,t) = 1/(\Sigma_{q\in Q} PRT(q,t))$

Then, the challenge is to maximize U^{RT} by searching the space of possible future distribution vectors $d_{t+1}(q) \in D$ for $q \in Q$. Such a search can be characterized as an optimization problem whose specification is given below, where Q is the set of executing queries, N is the set of available computational nodes, t is the time when the adaptation is being considered, D is the space of possible future distribution vectors, L is the log, and $U(Q,t)$ is bound to $U^{RT}(Q,t)$.

$plan^U(Q,N,t,D,L,U(Q,t)) =$
maximize: $U(Q,t)$ for some $d_{t+1} \in D$
subject to:
$0 \le d_{t+1}(q)[i] \le 1$, for $q \in Q$, $i \in N$ **and**
$(\bigwedge_{q\in Q}(\Sigma_{i\in N}(d_{t+1}(q)[i]) = (0 \vee 1)))$

The problem is then to find a new distribution vector that maximizes U^{RT} so that each $d_{t+1}(q)[i]$ is between 0 and 1, and either all the work is distributed or the query is suspended. These constraints allow for an increase/decrease of parallelism levels as well as temporary suspension of queries to be represented consistently and computed together in the same planning step.

In the experiments in Section 5, we have used for optimization a constraint solver that implements a sequential quadratic programming (SQP) method. SQP methods are designed to work on constrained, non-linear problems where the objective and constraints are both continuous and have continuous first derivatives. This is the case for the formulation above. These methods represent the state of the art in nonlinear programming methods. It has been shown [52] that SQP methods can outperform many other methods in terms of efficiency, accuracy, and percentage of successful solutions, over a large number of test problems. An overview of SQP is found in [16].

In practice, a starting point is provided to the optimization algorithms; in the experiments, the current distribution policy is provided as the starting point. In many cases the current policy is not improved upon – for example, if environment and query workload have changed little since the previous optimization.

The algorithmic complexity of the Utility-based solution depends on the following:

1. the complexity of the utility function $U(Q,t)$, which is simply $O(|Q| \times |D|)$;
2. the complexity of each constraint solving iteration; and
3. how many iterations are required for the constraint solver to converge onto a new distribution policy (or keep the current unchanged).

The analysis of items 2 and 3 can be summarized as follows. SQP methods rely on an approximation of the Hessian [45], i.e., the derivatives of $U(Q,t)$. There are efficient algorithms for deriving Hessians with quadratic complexity (i.e., $O((|Q|\times|D|)^2)$) using automatic differentiation or finite differencing, and in the experiments an interior-reflective Newton method [12] has been used that computes an approximation of the Hessian. This complexity implies, for example, that where adaptation is taking place in a setting with a fixed number of nodes, the number of utility function invocations in an iteration will grow linearly with $|Q|$, and this has been confirmed experimentally. Such methods give rise to slow growth in the required number of iterations [12], and in the experiments in Section 5, the number of iterations required by adaptations was generally less than *10*, and grows slowly with $|Q|$.

[3] Changing this notion of utility, while leaving other aspects of the method unchanged, would allow different response-time based objectives to be specified (e.g. a utility function could seek to minimize the maximum response time). Some caution is required, however; one desirable feature of the minimization of overall response times is that a change to the proposed evaluation strategy of *any* query is reflected directly in *overall* utility, thus providing the optimization algorithm with direct feedback across the whole of its search space.

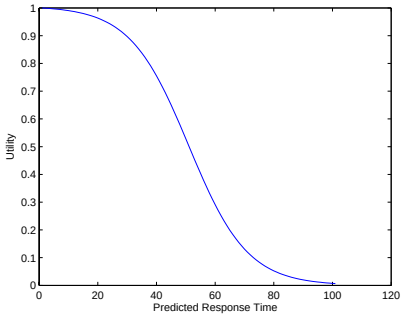


Figure 4: Utility for a target response time of *50*.

4.4. QoS-Aware Utility Functions

This section follows the same approach as Section 4.3, except that utility is defined in terms of the number of queries that meet their QoS goal.

Problem Statement. The problem we address in this section can be stated informally as follows: adapt in order to maximize the number of queries in a set that meet a response time target.

More formally, given a set of queries Q, $|Q| \geq 1$, a set of compute nodes N, $|N| \geq 1$, and for each q a target response time $TT(q) > 0$ that represents the QoS goal, maximize $|Q'|$ where $Q' = \{q \in Q | FT(q) \leq TT(q)\}$ and $FT(q)$ is the final response time of q, i.e., the observed response time at the end of the execution of q.

Utility-Based Solution. In principle, we can define the utility for a single query q, at a time t in its evaluation, $U^{QoS}_{PerQuery}(q)$, for a given distribution vector, as 1 if $PRT(q,t) \leq TT(q)$, and 0 otherwise. However, such a utility function is problematic during the search for effective distribution vectors, as every candidate distribution $d_{t+1}(p)$ that misses its QoS goal has the same utility of 0, no matter how near to or far from the target it is, and every query that meets the QoS goal has the same utility of 1 no matter how narrowly or comfortably the target is met. This makes it difficult for an optimization algorithm to rank alternative solutions. As a result, we use a utility function for queries that provides high and broadly consistent scores for meeting a QoS goal, and low but broadly consistent scores for missing a QoS goal, while also enabling improvements to be recognized during optimization.

We use a function definition from earlier work on resource allocation in data centers [4], which generates the curve illustrated in Fig. 4, for a target time of *50*:

$$UtilityCurve(q,t) = \frac{e^{-PRT(q,t)+TT(q)}}{1+e^{-PRT(q,t)+TT(q)}}$$

where $PRT(q,t)$ and $TT(q)$ are as defined above. In essence, in Fig. 4 any query that has a predicted response greater than the target response time of *50* has a utility of less than *0.5* (and where the query is predicted to miss the target by a substantial margin the utility is close to *0*), and any query that has a response time less than *50* has a utility greater than *0.5* (and where the query is predicted to meet the target by a substantial margin the utility is close to *1*).

The utility for an individual query is then defined as:

$$U^{QoS}_{PerQuery}(q,t) = UtilityCurve(q,t) + 1/S(q,t)$$

The incorporation of the term $1/S(q,t)$, representing the predicted change in the performance of the query as the result of a proposed adaptation and introduced in Section 4.3, causes the optimization algorithm to prefer distributions that are predicted to yield improved response times. In essence, where S is less than 1 there is predicted to be a speedup in q resulting from the adaptation, and where S is greater than 1 there is predicted to be a slowdown in q. This is relevant to optimization because in the absence of this term, queries that easily achieve (or significantly miss) QoS targets tend to yield rather similar values near to *0* (or *1*) in *UtilityCurve*; thus including S gives credit to proposed distributions that reduce response times even where there is no direct impact on the number of QoS goals met.

Given this definition for the utility of a single query, we define the utility for a set of queries Q as:

$$U^{QoS}(Q,t) = \Sigma_{q \in Q}(U^{QoS}_{PerQuery}(q,t))$$

Description	Value	Unit
Time to probe hash table	1e-7	s
Time to insert into hash table	1e-5	s
Time to add a value to fixed-size buffer	1e-6	s
Time to map a tuple to/from disk/network format	1e-6	s
CPU time to send/receive network message	1e-5	s
Size of a disk page	2048	bytes
Seek time/Latency of a disk	5e-3	s
Transfer time of a disk page	1e-4	s
Size of a network packet	1024	bytes
Network latency	7e-6	s
Network bandwidth	1000	Mb/s
Size of the caches on exchange operator	50000	tuples
Size of the disk cache for workloads	50	Mb

Table 1: Cost model parameters.

As in Section 4.2, the move from non-QoS-aware to a QoS-aware strategy implies refraining from responding to a problem if explicitly-set QoS targets are still being met, where the parameters are as defined for $plan^U$ in Section 4.3.

fun $plan^{UQ}(Q, N, t, D, L)$
  if $(\forall_{q \in Q}\ QT(q,t) \leq (TT(q) * k)$ **return** d_t
  else return $plan^U(Q, N, t, D, L, U^{QoS}(Q, t))$
endfun

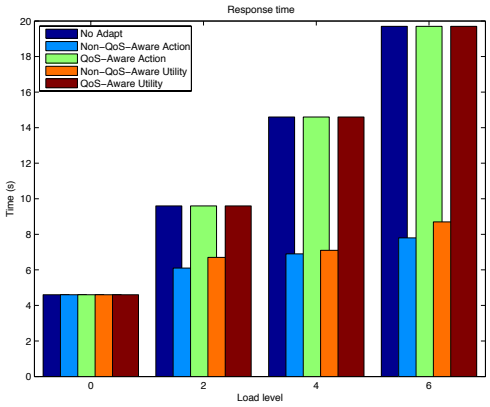
Note that the call to $plan^U$ in $plan^{UQ}$ uses the utility function U^{QoS}, and not U^{RT} as in Section 4.3.

If the set of current distribution vectors D is such that some QoS targets are being missed, our utility-based adaptive strategy is to search for the set of distribution vectors that maximize $U^{QoS}(Q, D, t)$ under the same set of constraints as described for the non-QoS-aware case.

5. Experimental Evaluation

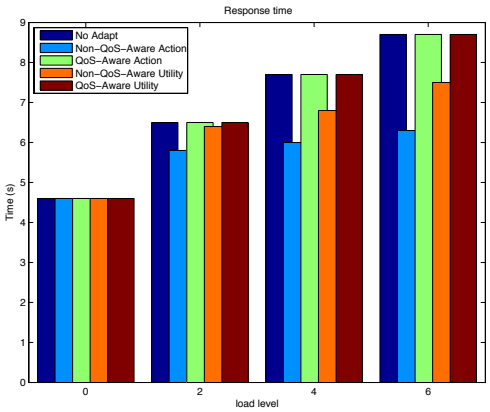
This section compares the adaptivity strategies from Section 4 across a range of experimental conditions in terms of the numbers of queries evaluating, the form of the external load experienced by those queries, and the stringencies of the QoS targets to be met. Performance comparisons are carried out using simulations of query evaluation; we use simulation because this allows the strategies to be compared in a controlled manner over a wide range of experimental conditions. The simulation uses a cost model consisting of a collection of parameterized cost functions based on those validated in [51], and extended to include the cost of adaptations; the parameters were obtained from the execution times of micro-benchmark queries, and are given in Table 1. The simulator is described in more detail in [46].

The following features characterize the environment in which the experiments are conducted. Query evaluation takes place in a shared nothing configuration – when queries have an initial parallelism level of i, the data in each table is uniformly distributed over all i machines, and all joins are initially run on i machines; where the parallelism level is increased during evaluation, this affects the number of nodes used to evaluate the joins but not the number of nodes across which the tables are distributed (i.e. the adaptations apply to joins but not scans). A single network (modeled as an Ethernet) and type of computer are used throughout, with physical properties as described in Table 1. There is assumed to be sufficient main memory in all computers to support the caches described in Table 1 and to hold join hash tables. All experiments report results using either query Q1 ($P \bowtie PS$) or Q2 (($P \bowtie PS$) $\bowtie L$) over the TPC-H database of scale factor 1 (where P has *200,000* tuples, PS has *800,000* tuples, and L has *6,000,000* tuples); the queries involve foreign key equi-joins, so Q1 returns *800,000* tuples and Q2 returns *6,000,000* tuples. As in Flux [54], the minimum unit of adaptation is a table partition (in the experiments, each table consists of *100* such partitions), and readapting is only considered when a period of normal evaluation has taken place that is at least as



(a) Expt 1: Q1 Response Time

Figure 5: Experiment 1: Constant imbalance for a single query.



Expt 2: Q1 Response Time

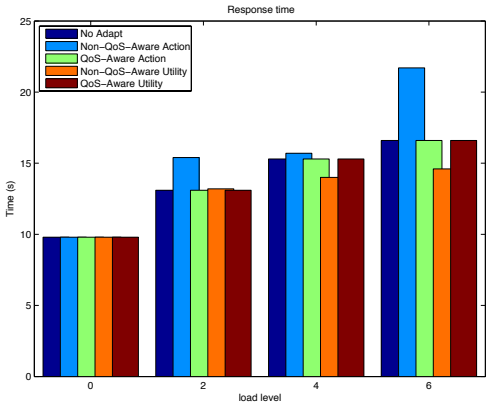
Figure 6: Experiment 2: Periodic imbalance for a single query, Q1.

long as the time taken by the most recent adaptation. This constraint on adaptation frequency essentially ensures that none of the strategies lead to more time being spent adapting than in query evaluation.

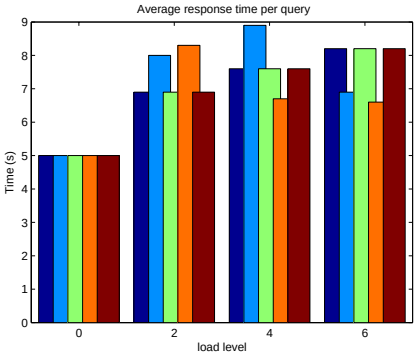
The following forms of load imbalance are considered:

1. *Constant:* A consistent external load exists on one or more of the nodes throughout the experiment. Such a situation represents the use of a machine that is less capable than advertised, or a machine with a long-term compute-intensive task to carry out. In the experiments, the *level* of the external load is varied in a controlled manner; the *level* represents the number of external tasks that are seeking to make full-time use of the machine.
2. *Periodic:* The load on one or more of the machines comes and goes during the experiment. In the experiments, the *level* is configurable and the *duration* and the *repeat duration* of the external load set to *1s*; the *duration* of the load indicates for how long each load spike lasts; and the *repeat duration* represents the gap between load spikes.

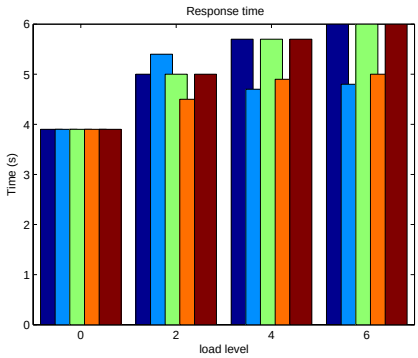
In practice, and as illustrated in [46], poisson arrival rates tend to give rise to adaptive behaviors that are similar to those for constant imbalance, and are not considered further here.



(a) Q2 Response Time for parallelism level 4



(b) Q2 Response Time for parallelism level 8



(c) Q2 Response Time for parallelism level 12

Figure 7: Experiment 2: Periodic imbalance for a single query, Q2, at different parallelism levels

5.1. Experiment Results

This section explores the extent to which the adaptive strategies from Section 4 are successful at improving query response times or meeting QoS targets in the presence of varying numbers of competing queries, and for varying levels and types of load imbalance. The experiments seek to determine: (i) the extent to which the strategies are effective at mitigating the consequences of load imbalance; and (ii) the contexts in which benefit can be observed from coordinated adaptive behavior using utility functions where the specific goal is to meet QoS targets.

We start in Experiments *1* and *2* using workloads containing a single query to show how the utility functions perform for simple settings in which no benefit can be derived from coordinating adaptations across queries, and then proceed to multi-query workloads to explore the effects of coordination.

Experiment 1. *Effectiveness of different strategies in the presence of constant imbalance.* This experiment involves *4* machines, with an initial parallelism level of *4*, where an external load is introduced that affects one of the nodes being used to evaluate the join. In all the experiments, queries with an initial parallelism level of p have an initial distribution vector that assigns work evenly across the p nodes. The increasing level of imbalance simulates the effect of having *0* to *6* other jobs competing for the use of one of the compute nodes. The QoS target is easily met for all imbalance levels.

Fig. 5 illustrates the results for single-join query *Q1*; the following observations can be made:

1. When adaptation is switched off (*No Adapt*, referred to as the *base case* hereafter), response times increase linearly with increasing external load.
2. The QoS-Aware strategies do not adapt, because the conditions in $plan^{AQ}$ in Section 4.2 and $plan^{UQ}$ in Section 4.4 identify that QoS goals can be met without adapting.

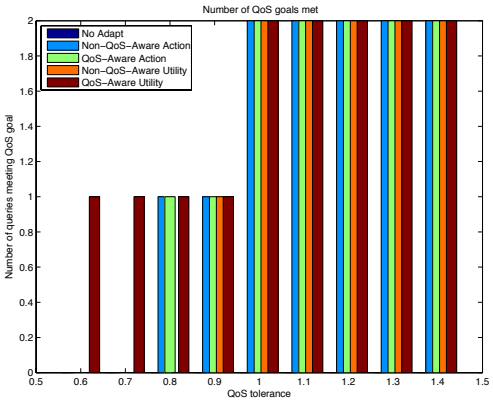


Figure 8: Expt 3: QoS for two queries with periodic imbalance.

3. The Non QoS-Aware strategies both significantly out-perform the base case, successfully moving work from the externally loaded machine to the other machines. The Non-QoS-Aware Action and Non-QoS-Aware Utility strategies perform similarly in this case. Overall, this is a straightforward setting for adaptation, as the environment is stable. In addition, because the imbalance is present from the start, an adaptation early in the evaluation of the query not only provides lasting benefits, but also only needs to relocate small amounts of operator state because join hash tables are only partially populated.

Experiment 2. *Effectiveness of different strategies in the presence of periodic imbalance.* This experiment covers the same ground as *Experiment 1*, except that the load is periodic rather than constant, and also explores different parallelism levels. The increasing level of imbalance simulates the effect of having increasingly severe load spikes on one of the compute nodes.

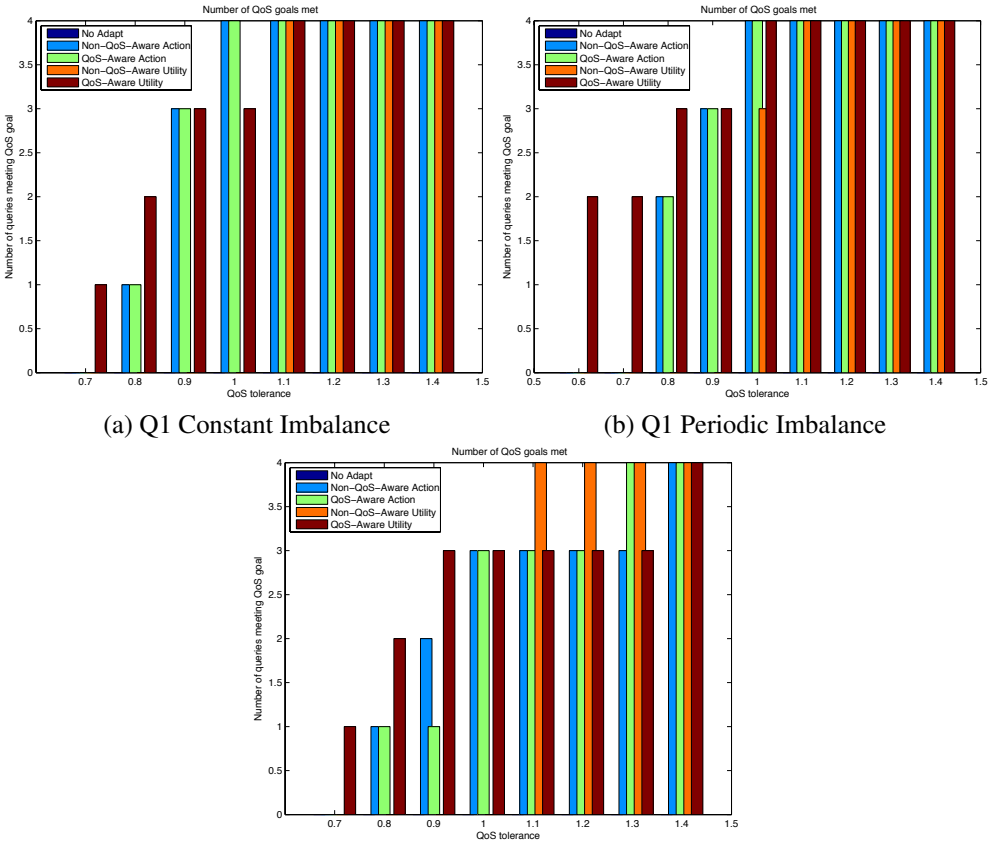
Figures 6 and 7 illustrate the results for queries *Q1* and *Q2*, respectively; the following observations can be made:

1. The performance degradation in the base case is less marked than for constant imbalance, as the external load is not present all the time.
2. The precise timing of adaptations influences their effectiveness, and can lead to adaptations having a negative impact on query response times. For example, the Non-QoS-Aware Acton strategy has a negative effect on response times in *6* of the *9* cases in which there is an external load in Fig. 7, as it adapts frequently in response to transient effects. The consequences of unfortunately timed adaptations are more marked in *Q2* than in *Q1*. This is because in *Q2*, which involves two joins compared with one in *Q1*, during evaluation of the second join, any adaptation must relocate the operator state of *both* joins, the fully populated lower join hash table and the partially populated upper join hash table; as a result, adaptations late during the evaluation of *Q2* are expensive.
3. The different parallelism levels tested in Fig. 7 suggest that changing parallelism levels has little influence on the conclusions of the experiments.

Experiment 3. *Effectiveness at meeting QoS goals involving two queries, in the presence of periodic imbalance.* This experiment involves two *Q1* queries being run simultaneously with an initial parallelism level of *4*, a maximum parallelism level of *8* across *8* machines and a periodic imbalance level of *4* on one machine, where a base QoS target of *8s* is multiplied by values in the range *0.6 to 1.4*, thus enabling the behavior of the strategies to be compared given QoS goals from *4.8* (8×0.6) to *11.2* (8×1.4). This allows the methods to be compared across a range of QoS goals, some of which are easy to meet and some of which are considerably more challenging.

Fig. 8 illustrates the number of queries that have met their QoS goals for a range of QoS goals. The following observations can be made:

1. The QoS-Aware Utility strategy meets more QoS targets than its Non QoS-Aware counterpart for stringent QoS targets because it benefits from selectively preferring one plan over the other. In practice, one query can be given preference over another through suspension, reduced parallelism, or less even workload distribution.



(a) Q1 Constant Imbalance

(b) Q1 Periodic Imbalance

(c) Q2 Periodic Imbalance

Figure 9: Expt 4: QoS for multiple queries.

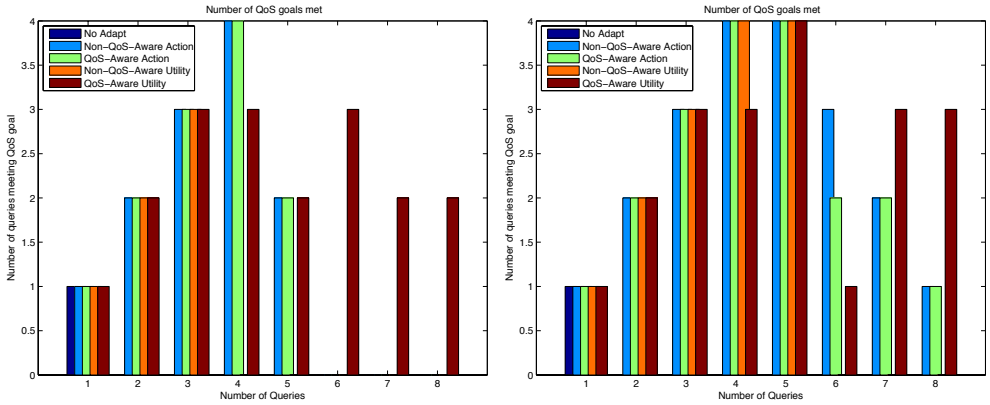
2. Only the QoS-Aware Utility strategy generates different plans within the given range of QoS tolerances; this is because it is the only strategy that takes account of how many QoS targets are predicted to be met in the derivation of distribution vectors. The QoS-Aware Action strategy only uses predictions on the number of targets that will be met when deciding whether or not to adapt.
3. The Non-QoS-Aware Utility strategy yields similar response times for the two queries, and thus in most cases either both or neither of the queries meet the QoS target.

Experiment 4. *Effectiveness at meeting QoS goals involving four queries.* This experiment involves four queries being run simultaneously with an initial parallelism level of *4* and a maximum parallelism level of *8* across *8* machines, where one of these machines has an imbalance of level *4*, and where a base QoS target is multiplied by values in the range *0.6 to 1.4*, thus enabling the behavior of the strategies to be compared for workloads containing larger numbers of queries, given QoS goals with different stringencies.

Fig. 9 illustrates the number of queries meeting the varying response time targets; in particular, (a) shows results for four Q1 single-join queries with Constant Imbalance, (b) shows results for four Q1 queries with Periodic Imbalance, and (c) shows results for four Q2 two-join queries with Periodic Imbalance. The results for Q1 involve a base QoS target of *15s* and those for Q2 involve a base QoS target of *25s*.

The following observations can be made:

1. Where there is Constant Imbalance, in Fig. 9(a), all the strategies except for Non-QoS-Aware Utility gracefully reduce the number of queries meeting their targets as the targets become more stringent. The rapid drop the number of Non-QoS-Aware Utility queries meeting their response time targets results from the fact that all the



(a) Q1 QoS Goals (b) Q2 QoS Goals

Figure 10: Expt 5: Varying number of queries for a fixed QoS goal and constant imbalance.

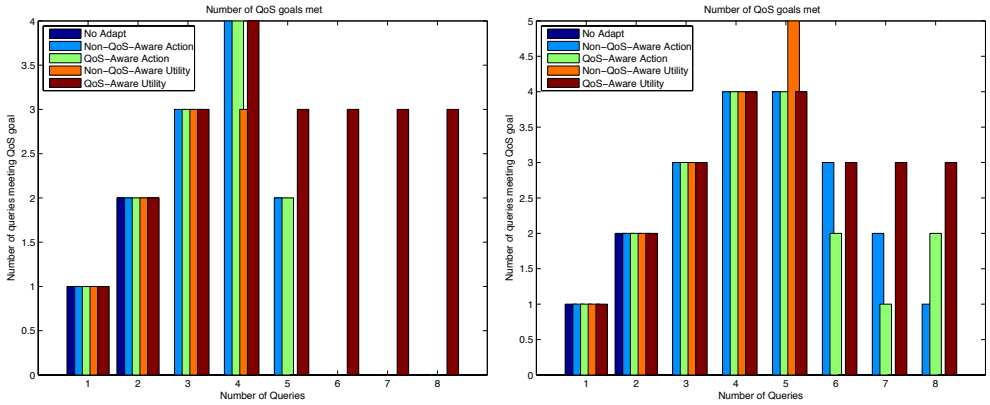
queries have similar response times under this strategy – as a result, for most QoS goals either all or none of the queries meet the target. The most (by a small number) QoS targets are met by the QoS-Aware Utility strategy, where queries are selectively suspended in the presence of challenging targets. For a QoS tolerance of *1*, the QoS-Aware Utility strategy is less effective than the action strategies; this results from the utility strategy being pessimistic about how many plans would meet their targets, and discriminating against one; the QoS-Aware Utility strategy is particularly dependent on the accuracy of the response time predictions because one of its most important strategies is to suspend queries that it predicts will miss targets. The Action-based strategies, in contrast with the Non-QoS-Aware Utility strategy, executes the queries in a way that leads to them having rather variable response times. As a result, as stringencies tighten there are incremental reductions in the numbers of queries meeting their targets.

2. Where there is Periodic Imbalance, in Fig. 9(b) and (c), the QoS-Aware Utility strategy is the most successful where there are stringent QoS targets, enabling several queries to meet their targets over a range of targets where the other methods failed. In this experiment, the more stringent targets can only be met by preferring some queries to others, and the QoS-Aware Utility strategy benefits by selectively suspending some queries in order to allow others to complete on time. This is clear evidence of the benefits that can result from coordinating adaptations across multiple queries.
3. As in the case of Constant Imbalance, for Periodic Imbalance the Non-QoS-Aware Utility strategy performs better than the action strategies where the targets are not especially stringent, but by scheduling queries in an even-handed manner, leads to fairly consistent response times and thus to a rapid drop-off in the number of queries meeting QoS goals as these goals become more stringent.
4. For several QoS targets in Fig. 9(c), the Non-QoS-Aware Utility strategy meets more QoS targets than the QoS-Aware Utility strategy. This is because the predicted completion times produced by the QoS-Aware strategy are slightly pessimistic in some cases, which causes one query to be suspended when in fact this is not necessary. Overall, however, the progress estimates have been accurate enough to allow effective decision making, and the QoS aware strategy meets response time targets with *22* queries in this experiment, whereas the Non-QoS-Aware Action strategy, which is the next most effective strategy, is successful with *19* queries.

Experiment 5. *Effectiveness of different strategies for variable numbers of queries in the presence of constant and periodic imbalance*. This experiment has an initial parallelism level of *4* and a maximum parallelism level of *8*, an imbalance of level *4* on one machine, a base QoS target of *15s* for *Q1* and of *35s* for *Q2*, and a varying number of queries.

Figures 10 and 11 illustrate the the number of QoS Goals met for *Q1* and *Q2*, with constant and periodic imbalance, respectively; the following observations can be made:

1. In the absence of well targeted adaptations, the expected behavior is that the number of QoS targets met will



(a) Q1 QoS Goals (b) Q2 QoS Goals

Figure 11: Expt 5: Varying number of queries for a fixed QoS goal and periodic imbalance.

grow from *1* with the increasing number of queries executing until such time as the total load increases to the extent that typical query response times exceed the QoS target, subsequent to which the number of queries meeting the target will decline. This overall pattern of behavior is exhibited by all the strategies, although in the case of QoS-Aware Utility the number of goals met is maintained better for large numbers of competing queries.

2. As in Experiment 4, the QoS-Aware Utility strategy meets the most QoS targets as the targets become increasingly challenging. This is because selective suspension of queries allows some queries to meet their QoS goals at the expense of others. Across the experiment, the QoS-Aware Utility strategy meets *83* QoS goals, and the nearest challenger is the Non-QoS-Aware Utility strategy with *63*.

6. Conclusions

This paper extends earlier work on adaptive query execution control by using utility functions both to make explicit the purpose of adaptation and to coordinate adaptations across query plans.

The contributions of this paper are as follows:

1. A description of how utility functions can be used to rank alternative resource allocations for parallel query plans. Utility functions have been defined that seek (i) to minimize total response times for a set of queries, and (ii) to maximize the number of queries that meet a QoS target. This is the first occasion we know of where AQP has been used to address QoS goals.
2. A description of an approach whereby the generation of parameter values for the utility functions is cast as an optimization problem. The two utility functions share: (i) a means of predicting future response times based on readily available monitoring information that enables each query adaptation to take into account the resource implications of the changes being made to other queries; (ii) a common representation for the decision space, in the form of a collection of distribution vectors; and (iii) a response mechanism, whereby operator state is transferred between sibling operators, as in Flux [54]. An optimization algorithm then uses a sequential quadratic programming method [16] to generate schedules that reflect the predictions from (i), are described using the representation from (ii) and are put in place using the strategy from (iii). This is the first occasion we know of in which adaptations involving multiple queries are coordinated to meet shared goals.
3. An empirical evaluation of the resulting techniques, in comparison with heuristic-based adaptive techniques. The evaluation compared the two utility-based strategies with action-based counterparts, and demonstrated: (i) that the utility based techniques frequently provided improved performance compared with their action-based counterparts for meeting QoS goals; and (ii) how an adaptive infrastructure can be configured by utility functions to prioritize behaviors that reflect specific requirements. As well as response times and QoS targets,

the utility-based approach could be revised to reflect other priorities, such as the financial cost of accessing different resources.

We note that the approach applied here to query workload management has several generic features that can be applied for other types of workload: (i) utility functions are defined that reflect the measurable objective of adaptation; (ii) the utility functions make use of a model that estimates progress in the evaluation of the workload – a key aspect of the model is that it considers together the consequences of the allocations of every component in the workload; and (iii) an optimization algorithm searches for a resource allocation that maximizes utility by exploring assignments, the consequences of which can be estimated using the model.

If other workload types are to be supported, the same components must be designed; this paper has provided concrete examples of each of these features of utility driven workload management, for the challenging case of query workload adaptation with stateful operators. A methodology for applying this approach more widely has been developed [46], and has also been applied to adaptive workload management for workflows [24]. In clouds, although there has been work on autonomic tuning of parameters that control scheduling decisions [9], this is essentially a heuristic approach, and we are not aware of utility-based models that make fine grained decisions with a view to meeting specific goals.

The work in this paper complements the use of utility functions for system management tasks (e.g. [4, 57]), thus providing further evidence for the wide applicability of the approach. One of the benefits of the utility approach described in this paper is that it adapts multiple queries together; while there are cases where this improves on single query optimization, it leaves unresolved the question as to how to optimize many different types of task together. There has been some work on optimizing resource allocation decisions across different types of job to meet QoS goals in clouds [30], but at a granularity of decision making that is too coarse to capture the adaptations described in this paper; an open question for the utility-based approach is the selection of the most suitable granularity and an optimization strategy that accommodates multiple types of task. Three general strategies for accommodating multiple types of task would be the use of multi-dimensional optimization [13], the use of more abstract models to which optimization can be applied [30], and a focus on shared underlying computational models such as MapReduce [53].

APPENDIX A: Predicting the Impact of Distribution Policy Changes

As outlined in Section 4, the adaptive strategies make use of a model of resource usage to estimate $S(q,t)$, the speedup or slowdown experienced by q as a result of a transition from $d_t(q)$ to $d_{t+1}(q)$. The definition of S depends on a model of the consequences of proposed changes to the use of shared resources. Computational resources are shared between queries and tasks that are external to query evaluation. The following concepts are used in the definition of S.

The *recent load*, $RL_t[n]$ at time t is the average load on node $n \in N$ over the period from the last adaptation to t, and represents the level of contention for the resource while the current distribution policy has been in place. The *recent load* can be computed from *L.load* in the Monitoring Log.

The *current allocation*, $A_t[n]$ is the weighted sum of the distribution vectors as they affect $n \in N$ at time t:

$$A_t[n] = \Sigma_{q \in Q}(w_q[n] * d_t(q)[n])$$

where w_q scales the distribution vector for q so that its highest value is 1 rather than the sum being 1, i.e.

$$w_q[n] = d_t(q)[n] * (\frac{1}{max_{m \in N}(d_t(q)[m])})$$

Thus the current allocation estimates the level of demand for each node (i.e. if $A_t[i] > A_t[j]$ then a greater fraction of the query workload has been assigned to node i than node j).

The *external load*, $EL_t[n]$ on a node $n \in N$ at time t is estimated by subtracting the *current allocation* from the *recent load*:

$$EL_t[n] = max(0, (RL_t[n] - A_t[n]))$$

A *candidate allocation*, A_{t+1} is the weighted sum of a possible future distribution vector $d_{t+1}(q)$, and is defined analogously to A_t above.

The *predicted load*, $PL_t[n]$ at time t is the sum of the candidate allocations for that node and the external load:

$PL_t[n] = A_{t+1}[n] + EL_t[n]$

Thus the *predicted load* on each node at time $t + 1$ is computed from the monitored load by subtracting an estimate representing the query load before adaptation and adding an estimate representing the query load after adaptation. As a result, where several queries are being adapted together in the utility-based approach, their predicted response times after adaptations take into account the new allocations of the other queries.

Given the above characterization of overall resource usage before and after adaptation, the next step is to work out the consequences for the predicted response time of each query. A query completes when the last of its parallel fragments completes. The *recent load factor*, $RLF_t[q, n]$ of a node n for a query q at time t is the product of the amount of work from q assigned to n and the *recent load* on n:

$RLF_t[q, n] = d_t(q)[n] * RL_t[n]$

The recent load factor represents the relationship between the level of demand being placed on a node by a query and the recent load on the node. Thus the recent load factor associated with a query on a node is high when the distribution vector of the query has assigned a significant amount of work to a highly loaded node, and is low if little work was assigned and/or the node is lightly loaded.

The *predicted load factor*, $PLF_t[q, n]$ of a node n for a query q at time t using a new distribution vector d_{t+1} is the product of the allocation by the query to the node and the *predicted load*:

$PLF_t[q, n] = d_{t+1}t(q)[n] * PL_t[n]$

In each case, the node with the maximum load factor is predicted to be the rate-limiting node. As a result, we can predict the fractional speedup or slowdown $S(q, t)$, as required to predict the response time above as:

$S(q, t) = \frac{max_{n_1 \in N}(PLF_t[q,n_1])}{max_{n_2 \in N}(RLF_t[q,n_2])}$

When combined with an estimation of the progress of a query at time t, S can be used to estimate the predicted response time of the query, as described in Section 4.3. In common with Luo *et al.* [35], we estimate the progress of each query taking account the presence of others. In contrast with [35], the emphasis here is on shared use of distributed resources, as our goal is to support fine-grained resource-allocation decisions.

References

[1] R. Avnur and J.M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *ACM SIGMOD*, pages 261–272, 2000.
[2] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-Optimization. In *Proc. SIGMOD*, pages 107–118, 2005.
[3] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive Re-optimization. In *SIGMOD Conference*, pages 107–118, 2005.
[4] M.N. Bennani and D.A. Menasce. Resource allocation for autonomic data centres using analytic performance models. In *Proc. 2nd ICAC*, pages 229–240. IEEE Press, 2005.
[5] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.
[6] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, and S. Seltzsam ans K. Stocker. ObjectGlobe: Ubiquitous Query Processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.
[7] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, pages 3–14, 2007.
[8] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. Estimating Progress of Long Running SQL Queries. In *SIGMOD Conference*, pages 803–814, 2004.
[9] Q. Chen, D. Zheng, M. Guo, Q. Deng, and S. Guo. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environments. In *Proc. 10th IEEE Intl. Conf. on Computer and Information Technology*, pages 2736–2743, 2010.
[10] F. Chu, J.Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *PODS*, pages 138–147, 1999.
[11] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. SIGMOD*, pages 150–160, 1994.
[12] Thomas F. Coleman and Yuying Li. An interior trust region approach for nonlinear minimization subject to bounds. *SIAM Journal on Optimization*, 6(2):418–445, 1996.
[13] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001.
[14] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. VLDB*, pages 27–40, 1992.
[15] K. Eurviriyanukul, N. W. Paton, A. A. A. Fernandes, and S. J. Lynden. Adaptive join processing in pipelined plans. In *EDBT*, pages 183–194, 2010.
[16] R. Fletcher. *Practical Methods of Optimization*. John Wiley&Sons, 1987.
[17] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008. Second Edition.
[18] A. Gounaris, N.W. Paton, A.A.A. Fernandes, and R. Sakellariou. Self-monitoring query execution for adaptive query processing. *Data Knowl. Eng.*, 51(3):325–348, 2004.

[19] A. Gounaris, J. Smith, N. W. Paton, R. Sakellariou, A. A. A. Fernandes, and P. Watson. Adaptive workload allocation in query processing in autonomous heterogeneous environments. *Distributed and Parallel Databases*, 25(3):125–164, 2009.

[20] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. SIGMOD*, pages 102–111, 1990.

[21] Wei Hong. Exploiting Inter-Operation Parallelism in XPRS. In *SIGMOD Conference*, pages 19–28, 1992.

[22] Z.G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *SIGMOD Conference*, pages 299–310, 1999.

[23] Z.G. Ives, A.Y. Halevy, and D.S. Weld. Adapting to Source Properties in Data Integration Queries. In *Proc. SIGMOD*, pages 395–406, 2004.

[24] N. W. Paton K. Lee, R. Sakellariou, and A.A.A. Sakellariou. Utility functions for adaptively executing concurrent workflows. *Concurrency and Computation: Practice and Experience*, 23(6):646–666, 2011.

[25] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. SIGMOD*, pages 106–117, 1998.

[26] J.O. Kephart and R. Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[27] S. Krompass, U. Dayal, H. A. Kuno, and A. Kemper. Dynamic workload management for very large data warehouses: Juggling feathers and bowling balls. In *VLDB*, pages 1105–1115, 2007.

[28] S. Krompass, A. Scholz, M.-Cezara Albutiu, H. A. Kuno, J. L. Wiener, U. Dayal, and A. Kemper. Quality of service-enabled management of database workloads. *IEEE Data Eng. Bull.*, 31(1):20–27, 2008.

[29] V. Kumar, B.F. Cooper, and K. Schwan. Distributed Stream Management Using Utility-Driven Self-Adaptive Middleware. In *Proc. 2nd Intl. Conf. on Autonomic Computing*, pages 3–14, 2005.

[30] J. Li, J. Chinneck, M. Litoiu, and G. Iszlai. Performance model driven qos guarantees and optimization in clouds. In *ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 15–22. IEEE, 2009.

[31] W.-S. Li, V. S. Batra, V. Raman, W. H., and I. Narang. Qos-based data access and placement for federated information systems. In *VLDB*, pages 1358–1362, 2005.

[32] W-S Li, D. Gao, R. Bhatti, and I. Narang. Deadline and QoS Aware Data Warehouse. In *Proc. VLDB*, pages 1418–1421, 2007.

[33] S. Lightstone, M. Surendra, Y. Diao, S. S. Parekh, J. L. Hellerstein, K. Rose, A. J. Storm, and C. Garcia-Arellano. Control theory: a foundational technique for self managing databases. In *ICDE Workshops*, pages 395–403, 2007.

[34] D.T. Liu and M.J. Franklin. GridDB: A Data-Centric Overlay for Scientific Grids. In *Proc. VLDB*, pages 600–611. Morgan-Kaufmann, 2004.

[35] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query sql progress indicators. In *EDBT*, pages 921–941, 2006.

[36] G. Luo, J.F. Naughton, C. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *Proc. ACM SIGMOD*, pages 791–802, 2004.

[37] S. J. Lynden, A. Mukherjee, A. C. Hume, A. A. A. Fernandes, N. W. Paton, R. Sakellariou, and P. Watson. The design and implementation of OGSA-DQP: A service-based distributed query processor. *Future Generation Comp. Syst.*, 25(3):224–236, 2009.

[38] V. Markl, V. Raman, D.E. Simmen, G.M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. SIGMOD*, pages 659–670, 2004.

[39] P. Martin, W. Powley, H-Y Li, and K. Romanufa. Managing database server performance to meet qos requirements in electronic commerce systems. *Int. J. on Digital Libraries*, 3(4):316–324, 2002.

[40] D. A. Menascé and V. K. Dubey. Utility-based qos brokering in service oriented architectures. In *ICWS*, pages 422–430, 2007.

[41] D. A. Menascé, H. Ruan, and H. Gomaa. Qos management in service-oriented architectures. *Perform. Eval.*, 64(7-8):646–663, 2007.

[42] K. Nahrstedt. End-to-end qos guarantees in networked multimedia systems. *ACM Comput. Surv.*, 27(4):613–616, 1995.

[43] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel. Dynamic Query Re-Optimization. In *SSDBM*, pages 264–273, 1999.

[44] B. Niu, P. Martin, W. Powley, P. Bird, and R. Horman. Adapting mixed workloads to meet slos in autonomic dbmss. In *ICDE Workshops*, pages 478–484, 2007.

[45] J. N. Nocedal and S. J. Wright. *Numerical Optimization*. Prentice Hall, 1999.

[46] N.W. Paton, J. Buenabad-Chavez, M. Chen, V. Raman, G. Swart, I. Narang, D.M. Yellin, and A.A.A. Fernandes. Autonomic Query Parallelization using Non-dedicated Computers: An Evaluation of Adaptivity Options. *VLDB Journal*, 18:119–140, 2009.

[47] F. Pentaris and Y. E. Ioannidis. Query optimization in distributed networks of autonomous database systems. *ACM Trans. Database Syst.*, 31(2):537–583, 2006.

[48] F. Pentaris and Y. E. Ioannidis. Autonomic query allocation based on microeconomics principles. In *ICDE*, pages 266–275, 2007.

[49] E. Rahm and R. Marek. Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In *Proc. VLDB*, pages 182–193, 1993.

[50] V. Raman, W. Han, and I Narang. Parallel querying with non-dedicated computers. In *Proc. VLDB*, pages 61–72, 2005.

[51] S. Sampaio, N.W. Paton, J. Smith, and P. Watson. Measuring and Modelling the Performance of a Parallel ODMG Compliant Object Database Server. *Concurrency and Computation: Practice and Experience*, 18(1):63–109, 2006.

[52] K. Schittkowski. NLQPL: A fortran-subroutine solving constrained nonlinear programming problems. *Annals of Operations Research*, 5:485–500, 1985.

[53] S. Sehgal, M. Erdélyi, A. Merzky, and S. Jha. Understanding application-level interoperability: Scaling-out mapreduce over high-performance grids and clouds. *Future Generation Comp. Syst.*, 27(5):590–599, 2011.

[54] M.A. Shah, J.M. Hellerstein, S.Chandrasekaran, and M.J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. ICDE*, pages 353–364. IEEE Press, 2003.

[55] J. Smith, A. Gounaris, P. Watson, N.W. Paton, A.A.A. Fernandes, and R. Sakellariou. Distributed query processing on the grid. *Intl. J. High Performance Computing Applications*, 17(4):353–368, 2003.

[56] T. M. Sutherland, Y. Zhu, L. Ding, and E. A. Rundensteiner. An adaptive multi-objective scheduling selection framework for continuous query processing. In *IDEAS*, pages 445–454, 2005.

[57] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proc. ICAC*, pages 70–77. IEEE Press, 2004.

[58] Y. Zhou, B.C. Ooi, K-L Tan, and W.H. Tok. An Adaptable Distributed Query Processing Architecture. *Data & Knowledge Engineering*, 53(3):283–309, 2005.