Quality of Service in an Information Economy

R. BRAUMANDL and A. KEMPER
University of Passau, Germany
and
D. KOSSMANN
TU Munich, Germany

Accessing and processing distributed data sources have become important factors for businesses today. This is especially true for the emerging virtual enterprises with their data and processing capabilities spread across the Internet. Unfortunately, however, query processing on the Internet is not predictable and robust enough to meet the requirements of many business applications. For instance, the response time of a query can be unexpectedly high; or the monetary cost might be too high if the partners charge for the usage of their data or processing capabilities; or the result of the query might be useless because it is based on outdated data or only on parts (rather than all) of the available data. In this work, we show how a distributed query processor can be extended in order to support quality of service (QoS) guarantees. We propose ways to integrate QoS management into the various phases of query processing: (1) Query optimization uses a multi-dimensional assessment (cost, time and result quality) of query plans, (2) query plan instantiation comprises an admission control for sub-plans, and (3) during query plan execution the QoS of the query is monitored and a fuzzy controller initiates repairing actions if needed. The goal of our work is to provide an initial step towards QoS management in distributed query processing systems and do significantly better than current distributed database systems, which are based on a best-effort policy.

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Information networks*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Commercial services*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Design, Performance

Additional Key Words and Phrases: Quality of Service

1. INTRODUCTION

Network and database technology (among others) have made distributed data processing possible. It is now possible to access any kind of data from virtually

This research is supported by the German National Research Foundation under contract DFG Ke 401/7-1.
Authors' addresses: R. Braumandl, University of Passau, Innstrasse 30, 94030 Passau, Germany; email: braumand@db.fmi.uni-passau.de; A. Kemper, University of Passau, Innstrasse 30, 94030 Passau, Germany; email: kemper@db.fmi. uni-passau.de; D. Kossmann, TU Munich, Boltzmannstrasse 3, 85747 Garching, Germany; email: kossmann@in.tum.de.

any place in the world using the Internet and to build applications that involve data from many places. Furthermore, a new kind of economy is emerging on the Internet in which providers offer their data processing services on the Internet. Examples of such services are (1) storing and backing up data on foreign machines, (2) application service providing (e.g., market places), and (3) high-performance computing on the Internet by exploiting idle machine resources. In general, we will distinguish three kinds of providers that can participate in an information economy: *data providers*, which supply data, *function providers*, which offer operators and functions to process data, and *cycle providers*, which are contracted to execute operators and functions. The ObjectGlobe project [Braumandl et al. 2001], which is the basis for our work, enables such an open and distributed query processing services market.

In order to become commercially relevant, however, it is necessary to give guarantees on the services provided. Today, almost all open systems on the Internet are based on the "best-effort-principle" and nobody is willing to construct mission-critical applications in such an environment because those applications would simply not be reliable enough. Specifically, users would like to constrain the cost of running applications, the running times of applications, and the quality of the results obtained by running the applications using external data sources. To demonstrate these needs we give the following example:

> A realtor in the USA has an appointment with a customer, who wants to buy a villa in the Mediterranean area. The customer has mentioned his requirements regarding the maximum distance to the next airport and the amount of building area in advance. The realtor poses a query against a distributed query processing system which covers some European realtors as data providers. There also exists a data provider which has information about airports in that area. Since the requirements of the customer seem to be very selective the realtor requests that at least 70% of the registered data about estate offers should be considered in the query and at least 20 result tuples should be produced. The appointment is in 20 minutes; therefore the data should be available in no more than 10 minutes since the realtor wants to look at the offers before the meeting. Considering the budget for IT-services and the prospects of a successful deal, the realtor sets the upper bound for the query execution cost to 10 Dollars.

The SQL-query for this application computes a join between the real estate information and the data about airports. The join predicate uses a user-defined, external function which computes the length of the bee-line between two locations. The query also uses an external operator for scaling the images of the estate offers into a uniform size. We used the real estate DTD [Petit 1999] as a template for our real estate relation; the meaning of the attributes should be obvious.

```
select e.Price, e.Location.City, scale(e.Image,0.3), a.Name
from Estate e, Airports a
where e.building-area > 200 and
      geoDistance(e.Location,a.Location) < 10 and
      e.Location.region = 'Mediterranean';
```

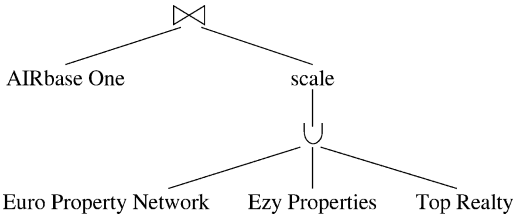

Fig. 1. Query evaluation plan for the example query.

In the ObjectGlobe system, external data sets are incorporated by so-called wrappers and internally represented in a nested relational format. Hence, the data of each European realtor would be incorporated as a partition of such a relation, here named *Estate*. For simplicity we assume throughout the whole paper that the partitions share the same schema. For the *Airports* relation we assume that there is only one data provider; therefore the completeness constraint for this relation is implicitly 100%.

A possible query evaluation plan for the given query is depicted in Figure 1. The leaves of the operator tree represent autonomous data providers that contribute their information to the query. The plan fragment consisting of the `union` and `scale` operation may be executed at a cycle provider that has a good network connection to the data providers beneath it. The `scale` operator itself could be loaded from a function provider, specialized on functions for image processing.

This example demonstrates the need to give guarantees at the "application level" and that giving guarantees involves coordinating the services of several providers. Obviously, it is not possible to fulfill all requests in such an environment since servers might be down, interconnects might be congested, or simply because the right data providers might not be found. However, the goal should be to fulfill as many requests as possible and to abort and inform the user as early as possible if a request cannot be fulfilled. This is the challenge we would like to address in this paper.

Specifically, we will pick up this challenge for query processing as required in this example. The data model of our ObjectGlobe query processor facilitates the integration of external data sources as provided by legacy applications or XML data sets with a format described by an XML DTD or an XML Schema. But in order to simplify the presentation we will restrict our descriptions to the relational data model. Accordingly, queries will be expressed in relational algebra with its usual set operators (union, set minus, selection, projection, Cartesian product, join[1]) enhanced by user-defined operators like the scale operator in the above example.

The next section lists related work, which exists in multimedia-, network-, and database literature. Section 3 introduces a query processing specific quality of service model and explains the basics of integrating the support for this model in a query processing system. The details of this integration are given in Section 4 for plan generation and in Section 5 for plan instantiation and

[1]Note that a join operation can be logically expressed as a Cartesian product followed by a selection.

execution. In Section 6 we discuss adaptations of query evaluation plans and their application for the enforcement of quality constraints during query execution. The results of experiments are summarized in Section 7. Section 8 provides a conclusion.

2. RELATED WORK

The term "Quality of Service" is mainly used in the area of networked multimedia applications. These applications need more high quality guarantees from the underlying devices in order to, for example, provide for smooth video playing. For multimedia data streams, precautions have to be taken in the server and the client machines as well as in the network infrastructure. Frameworks for managing QoS in a distributed multimedia application are presented, for example, in Li and Nahrstedt [1999] or Aurrecoechea et al. [1998]. With respect to our work, it should be noted that the nature of QoS parameters in multimedia applications and distributed query processing differ substantially. For multimedia applications, the QoS parameters mainly constrain the current execution of the application (Do we provide smooth playing at the moment?), whereas for query processing most QoS parameters refer to the end of the execution (Is the result set sufficiently large?).

Weikum [1999] gives a good motivation for the need to integrate the handling of service quality guarantees into information systems. Of the many aspects discussed in Weikum [1999], we concentrate on the quality of service in distributed query processing over autonomous data- and cycle-providers on the Internet. Work on service quality in the area of query processing is rare and the existing work particularly concentrates on constraints for the response time; examples are query scheduling techniques as presented in Beyer et al. [1998], Garofalakis and Ioannidis [1997], or Brown et al. [1994] or real-time databases [Pang et al. 1995]. Most work on QoS management for information systems classifies queries into groups and defines QoS for each group, rather than for each query. In this work, we would like to provide the flexibility to constrain the execution of each individual query.

Our work is related to the Mariposa project [Stonebraker et al. 1996]. In Mariposa a user can constrain the ratio between the running time of a query and its execution cost by means of a bidding curve. Mariposa tries to fulfill these constraints during its plan fragmentation step, which takes place after optimization. In our work, we consider the user-defined quality constraints during all phases of query processing. We think that this is necessary for achieving high rates of quality-conforming query executions in a distributed and open environment.

One important component of our work is to monitor the execution of a query and to adapt to changes in the system. There has been some work on similar techniques for adaptive query processing. Graefe and Ward [1989] and Ioannidis et al. [1992] propose the generation of query execution plans with embedded alternative sub-plans. The decision for a specific plan configuration is made at execution time, depending on the current load situation. In Kabra

and DeWitt [1998] and Ives et al. [1999] query execution steps are mixed with re-optimization steps. Query scrambling [Urhan et al. 1998] deals with delays in data delivery of remote data sources by re-scheduling executable sub-plans of the query. Antoshenkov and Ziauddin [1996] suggest starting competing sub-plans in parallel; after a "winner" plan has been identified, the "losers" are stopped. A more recent work on runtime adaptation can be found in Avnur and Hellerstein [2000]. This work proposes to decide for each tuple and for each of its processing steps which operator in a pipeline should process it next. This decision is based on the back pressure observed at the queues that are associated with each operator.

The adaptations proposed in this work supplement these techniques. In addition to resources like CPU, disk, and network bandwidth, we also consider data and function providers as resources.

3. THE QUALITY OF SERVICE MODEL

In this section, we present a model for quality constraints in order to describe and assess the quality of queries, query evaluation plans and query executions.

3.1 The Quality of Service Dimensions

As described in the introduction, in an information economy a user should be able to constrain the relationship between the quality of the result of a query, the execution of the query, and the monetary costs for the execution. In the following, we will describe the specific QoS parameters for each of these three dimensions.

3.1.1 *Query Result Quality.* We assume that a relation (data collection) S is partitioned into several partitions $S_1, \ldots, S_n$ that may be managed by independent data providers. In the example of the introduction, every realtor participating in the information economy, represents a data provider that contributes a partition of the relation *Estate*. Usually only a subset of these partitions is used in a specific query. For each relation S in a query, we can constrain the following QoS parameters:

—the oldest time stamp of the last update for a partition p of S or its maximum staleness factor as introduced in Mariposa. We call this parameter QR^{p}_{age}.

—the amount of data contributed by partitions used to answer the query in comparison to the total amount of data for relation S (QR^{S}_{comp}). We refer to this parameter as *completeness* in the remainder of this article.

The result cardinality can be characterized by the following parameters:

—a lower bound for the result cardinality ($QR_{\min\#}$). This parameter can be used to express that the user expects a minimum number of result tuples. If a query uses just a fraction of the available data for a relation, the system should incorporate as much data into query processing as is necessary to accomplish at least this result cardinality. If all the data for all relations in a query is used, this parameter need not be regarded.

—an upper bound for the result cardinality ($QR_{\max\#}$). Such a parameter corresponds to a *stop after* clause, whose support in query optimization and execution has already been studied in the literature [Carey and Kossmann 1998]. A *stop after* clause will typically be used in conjunction with an *order by* clause.

3.1.2 *Query Execution Time.* The execution of the query is characterized by the time spent in different phases of the execution of a query:

—QT_{first}. The time from the start of the query execution until the first tuple is delivered.

—QT_{last}. The time for producing all the result tuples.

3.1.3 *Query Evaluation Cost.* Since providers can charge for their services in our information economy, a user should be able to specify an upper bound for the respective consumption by a query. We propose the following quality parameters for this dimension:

—the cost for services of function providers ($QC_{function}$); the cost for leasing a function for the duration of the query.

—the cost for services of data providers (QC_{data}); the cost for reading the data at the respective data providers.

—the cost for services of cycle providers (QC_{cycle}); the cost for executing parts of the query at foreign cycle providers.

In our work, we do not assume specific business models for the different kinds of providers in an ObjectGlobe federation. Research in the area of electronic commerce, cost management and digital rights management is still a hot topic (e.g., see Sistla et al. [1998]), and there are no established standards. Therefore, we base our work regarding costs on two simple and sound rules: quality costs and work has to be paid. For example, we assume that an execution of a query on a high performance cycle provider costs more than an execution on a slow cycle provider and that a query has to pay for the number of CPU cycles it has consumed. We expect that further developments in the area of electronic commerce will basically adhere to these rules. Thus, our abstractions in this area should stay valid.

3.1.4 *Usage of QoS Parameters.* In many cases not all quality parameters will be interesting for a user (and therefore need not be specified) or perhaps just the sum of some of them, like the total cost or the total response time. The quality constraints for a specific quality parameter could also be expressed in the form of a continuous function over the space given by some or all other quality parameters. For instance, the user is willing to pay more money for the query execution, if more data was incorporated into the query processing. In our example, the realtor posed constraints on the total cost, total response time and the completeness of the *Estate* relation. Similar to real-time systems (hard real-time, soft real-time) some constraints on quality parameters could be strict and others could be handled in a relaxed way, by not aborting the query, if the

constraints are not fulfilled during execution. For example, users could limit the total response time for their queries, but they may still be interested in the result even if it takes longer. For batch queries users might only be interested in the total cost and the quality of the result because they do not wait for the completion of the query.

3.2 The Integration of QoS Management into Query Processing

It is currently not possible to construct a distributed and open query processing system on the Internet that can enforce the quality constraints of a query under all circumstances. The obstacles are either caused by the environment or by system immanent inaccuracies. Environmental factors, like network failures and unforeseeable load fluctuations on network links and cycle providers can obviously inhibit a QoS conform execution of a query. Likewise, difficulties in producing exact formulas for quality estimations and the limited accuracy of statistics for attribute value distributions of partitions or load distributions of resources can lead to overestimations of quality parameters. In this section, we first describe and motivate the overall goal of the QoS management component in our query processing systems. After that, we give an overview of the responsibilities and tasks of our QoS management component.

3.2.1 *The Goal of QoS Management.* Naturally, the ability to guarantee QoS constraints depends on the service quality guarantees one can get from the underlying resources. Among the common scheduling disciplines *best effort*, *priority-based scheduling*, and *reservation*, only the latter allows one to construct a QoS management that can absolutely guarantee the QoS constraints for an accepted query. However, since reservation entails over-booking and inefficient resource utilization and is therefore rarely used for scheduling computer or network devices, we avoid it. As a result of this, our goals for QoS management must take into account that queries that were admitted can fail during execution. Our goals for QoS management are as follows:

—The percentage of queries, whose quality constraints could be fulfilled, should be maximized. This percentage is calculated based on the overall number of queries that are issued and not only on the number of those for which a constraint-compliant query plan could be determined.
—The execution of queries that cannot fulfill their QoS constraints, should be stopped as early as possible. In this way the query does not waste the user's time and money. Of course, if the missed quality constraint is a soft one, the query should not be stopped but executed in a best-effort manner.

A query can only meet its quality constraints, if it gets a sufficiently good service from all involved providers. The difficulty in achieving a high percentage of QoS-compliant queries is to find at optimization time a query plan that uses providers capable of providing sufficiently good service at execution time. The optimizer uses estimates about the providers to construct such a query plan and the question is now, whether these estimates also hold during execution.

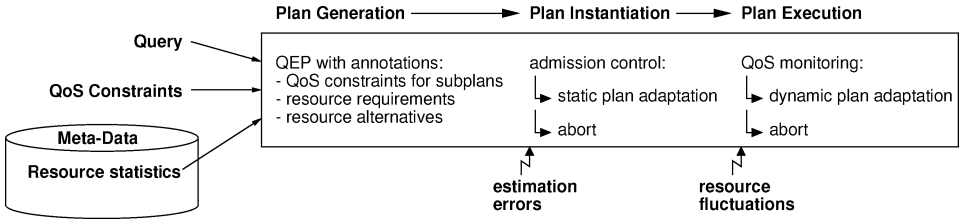

Fig. 2. The interaction of query processing and QoS management.

There are two possible solutions for obtaining these estimates:

—The optimizer could initiate a resource availability test for each of the resources during optimization. Since we are expecting such an information economy to consist of several thousands of providers and many of these would have to be regarded during plan generation, this approach is obviously not scalable enough.
—The remaining approach is then, to gather statistics about resource availability that is used during plan generation to estimate the current values for the resources in question.

In the latter approach imprecise estimations for resource availability could lead to failed quality constraints at run-time. In order to avoid a restrictive admission control, which could reduce the number of failed executions, we exploit the fact that different queries often have different demands at specific providers (e.g., batch queries in contrast to interactive queries). For every involved provider we explicitly state these demands in the form of resource requirements and quality constraints in the query plan. During query instantiation and execution we can use this information in order to check, whether the demands of a query can be satisfied, and we can react appropriately. For example, if a query seems to miss its quality constraints, it could try to get a greater share on the resources of the provider at the expense of queries that can work sufficiently with a smaller share. If such an adaptation is not possible and the query will not fulfill its QoS constraints our second goal causes us to stop the query in order to save the user's time and money.

3.2.2 *The Phases of Query Processing.* An overview of processing a query in the context of QoS management is depicted in Figure 2. The starting point for query processing is the query itself, the QoS constraints for it, and statistics about the resources that can be used for processing the query. In our scenario, cycle providers, the partitions from data providers, and the functions of function providers belong to these resources together with all the network links connecting them.

Figure 2 also shows the activities of our QoS management during the query processing phases of plan generation, instantiation and execution.

Plan Generation: The optimizer generates a query evaluation plan (QEP), which contains information about the used data, cycle and function providers,

and about the way their services are combined to compute a specific query. The optimizer has a search routine based on dynamic programming in our implementation. (Most other query optimizers are based on dynamic programming, also.) The optimizer assesses a very large number (potentially millions) of alternative plans with different providers by a quality model that provides formulas for estimating the quality parameters based on the structure of the plan and statistics about the providers. Every plan is constructed piecewise in a bottom-up manner and the quality parameters for every sub-plan that appears in such a process are computed with the quality model. Only a plan that fulfills all user constraints is considered for later phases and its plan description will be annotated with the quality estimations and resource requirements for every sub-plan. Additionally, if the optimizer can find approximately equivalent alternatives for resources used in the query evaluation plan, these are also annotated in the plan.

Plan Instantiation: During plan instantiation, sub-plans are distributed to cycle providers, functions are loaded from function providers and connections to data providers are established. When a sub-plan of a query uses the service of a specific provider, it is checked to determine whether the resource requirements resulting from the quality constraints for that sub-plan can be satisfied by this provider. For a cycle provider this would mean that if the optimizer underestimated the load on the respective cycle provider the newly arrived sub-plan will probably not be able to meet its constraints. Furthermore, all the other sub-plans on that cycle provider would be in danger of missing their quality constraints, if we execute the new sub-plan there. As a result of this admission control, the execution of the new sub-plan will be rejected or, if possible, the sub-plan will be adapted.

Plan Execution: During query execution, fluctuations regarding resource availability for example, CPU time or network bandwidth and estimation errors by the optimizer might violate the constraints on the quality parameters. In order to detect these violations, a monitoring component traces the current status of these parameters for every relevant sub-plan of the query. If this component detects a potential violation of the quality constraints for a sub-plan, it first tries to adapt the sub-plan so that it will again meet its constraints, or if this is not possible, it will abort the execution of this sub-plan. The plan adaptations during the instantiation phase can be performed rather easily because the plan is not yet instantiated. Here we need adaptations that can also be applied after a sub-plan has started to execute.

By estimating the necessary quality constraints for sub-plans of a QoS-compliant query plan the QoS management can monitor the development of the quality parameters with a fine granularity. As a result, potential quality violations can be detected as early as possible. For example, if there are pipeline-breaking operators in the plan, most of the work for the query could have already been done, when the first tuples arrive at the top of the plan. In our case, the plan beneath the pipeline breaker has its own quality constraints that must be monitored and enforced by the QoS management.

4. QUALITY OF SERVICE-ENHANCED PLAN GENERATION

In this section we discuss the necessary modifications of a classic, dynamic programming-based query optimizer for supporting QoS constraints during plan generation. We concentrate on the description of those parts of the optimization process that play an important role for QoS management and thus need modifications.

—In many cases, it will not be feasible to consider all possible data providers for a given data set because the resulting amount of data processed in a widely distributed environment would result in unacceptable running times. Thus, an additional task for QoS management is to select the most relevant data providers and find appropriate cycle providers that are able to efficiently process the data from the selected data providers.
—Query evaluation plans are constructed in a modular way using basic operators such as join, union, or user-defined operators. Analogously, the cost (and other properties) of a plan is computed in a modular way using cost functions for the individual operators. For QoS management, an extended framework is needed in order to construct plans and estimate the properties of plans in such a modular way.
—In order to estimate the running time of a plan, it is necessary to predict the load of resources (machines and network interconnects). Like traditional optimizers, we use statistics to estimate such loads. For QoS management, however, these statistics need to be evaluated in a different way because the probabilities for specific load situations need to be computed explicitly.
—The query optimizer enumerates alternative plans and prunes inferior plans based on their properties (e.g., estimated cost). A QoS-enhanced optimizer requires a special pruning metric in order to take all QoS parameters of a query into account.

In the following subsections, we will examine these issues. Finally, some post-optimization actions are described; this post-processing step is needed in order to generate information for admission control and monitoring at execution time.

4.1 Selecting Providers

Our prototype query processor uses a lookup service that contains meta-data about all the cycle-, data- and function-providers that are registered in the respective information economy. During the first step of plan generation we consult this lookup service. Providers of all kinds (data-, function- and cycle-providers) are selected according to their qualifications to contribute to the execution of the current query.

Relevant data partitions from data providers and functions from function providers can be searched by a lookup service query. The corresponding parameters for the query, like the requested type or freshness of the data and the class and signature of the function, can be directly retrieved from the query itself or its quality constraints. As a result we get sets of matching functions together with the cost formulas registered by the respective developers and sets

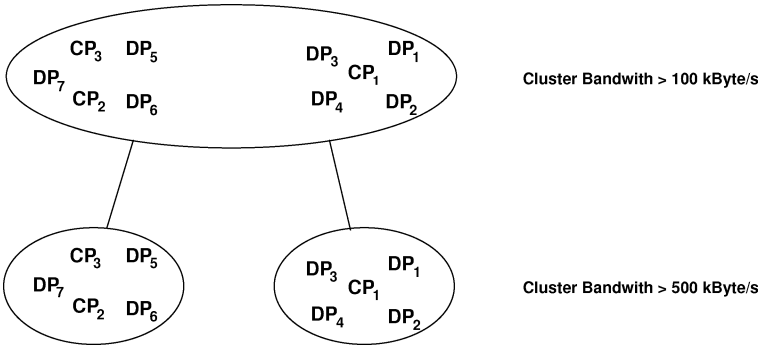

Fig. 3. A simple cluster tree with levels defined by minimum bandwidth interconnections.

of matching partitions together with information about cardinality, relevance, attribute value distributions (histograms) and average tuple size. This level of detail in the meta-data for the relations can be reached even in a heterogeneous environment by histograms constructed by query feedback [Aboulnaga and Chaudhuri 1999], frameworks that estimate data source relevance and overlap [Levy et al. 1996; Florescu et al. 1997] and mediator costing frameworks [Roth et al. 1999]. Furthermore, we also need statistics about the load characteristics of cycle providers and network links. Again, work on retrieving these statistics in a distributed environment has already been done [Gruser et al. 2000; Wolski et al. 1997].

These statistics are used by heuristics that select potentially useful cycle and data providers for a specific query. Heuristics are needed because all cycle providers (possibly thousands) can be used to execute a query and enumerating all possible combinations of data and cycle providers would simply be infeasible. These heuristics are based on a clustering algorithm that works on the interconnection graph of data and cycle providers with the bandwidth of a network connection as the weight of the corresponding edge. We search for a "minimum network diameter" cluster, whose data providers satisfy the constraints on staleness, completeness and cardinality for the relations referenced in the query. This cluster can be regarded as the hot spot for the specific query. In that cluster we then choose appropriate sets of data and cycle providers, again by building sub-clusters based on network connection and load characteristics. Figure 3 shows a resulting cluster tree where a cluster contains all the entries of its sub-clusters and the clusters at a lower level have a higher minimum interconnection bandwidth between the included cycle and data providers. Due to space limitations we do not present the heuristics in detail.

The information about useful clusters is also utilized, when we group the sets of partitions, cycle providers and functions in classes that we call similarity classes. The similarity classes of resources will be annotated in the corresponding plan. These classes define a kind of environment for the plan that can be utilized during plan distribution and execution to break up the resource binding of the optimizer by replacing the initial resource with a better suited one out of its respective environment. An informal description of these similarity

classes follows:

—The similarity class $Sim_{DP}^{R,\,dp}$ of a relation R and a data provider dp is the set of replicas of p or other partitions, currently not used in the query evaluation plan whose data providers appear in the same cluster as dp.
—The similarity class Sim_{CP}^{cp} of a cycle provider cp, is the set of cycle providers that, analogously, appear in the same cluster as cp.
—The similarity class Sim_{FP}^{f} of a function f, is the set of all functions in the same class as found in the lookup service.

4.2 Estimating QoS Parameters

The estimations of cost and time consumption at a cycle provider have in common that they are based on the amount of work W that induced by a specific plan. This dependency is obvious for time consumption. However, it is artificially introduced for cost consumption. We assume that the cost charged by cycle providers for the execution of plans depends on the work these plans induce.

The foundation for estimating the response time of query evaluation plans was established in Ganguly et al. [1992] and Garofalakis and Ioannidis [1997]. In the following, we refine this work by also considering parts of query evaluation plans that are executed in parallel with the rest of the plan but within which operators are executed in an interleaved way. Furthermore, we concentrate on structuring the quality model so that it can be easily implemented and extended.

4.2.1 *Estimated Processing Cost of an Operator.* The amount of work done by an operator depends on the processing information *PI* (e.g., definitions for selection or join predicates or other parameters that affect the processing by the operator), the available main memory M, and the properties *DProps* (e.g., cardinalities, attribute value distributions, mean attribute sizes) of the operator's input streams. The implementor of an operator must provide a *work function* *WF*, which can compute the work performed by the operator based on that information. This function must adhere to the following signature:

$$WF : PI \times M \times DProps \rightarrow W \tag{1}$$

Additionally, the implementor of an operator must provide a function *DPropsF*, which computes the properties of the operator's output stream in a specific context:

$$DPropsF : PI \times DProps \rightarrow DProps \tag{2}$$

For each operator the function *WF* is itself constructed out of two functions:

$$WF(pi, M, dprops) = WOF(pi, M, dprops) + WNF(pi, M, dprops) \tag{3}$$

The function *WOF* estimates the work performed in the so-called "open" phase of the operator, and *WNF* estimates the work in the so-called "next" phase. The open phase represents start-up work that needs to be carried out before the first tuple can be produced; tuples are then produced in the next phase. The

return values for all three functions *WF*, *WOF*, and *WNF* are equally dimensioned vectors (we call them *work vectors*), which means that the +-operation in Equation 3 corresponds to the usual summation on vectors. Each dimension in such a vector is associated with a resource, like CPU or disk. The unit of work for each dimension is determined by the type of the corresponding resource. For a CPU, work is measured by the number of instructions executed on it. If a more detailed model of the CPU should be used (e.g., modeling processor cache misses), we could integrate these details by adding dimensions to the work vector; for example, one dimension for each kind of CPU instruction. The work functions would need to measure the quantities for the different instructions separately for each operator. For disk drives, we introduce two dimensions in the work vector, one for the number of disk accesses and the other for the overall number of bytes transfered (read or written).

Network connections between plan fragments are modeled as virtual, unary operators that use a different work vector model than other operators. This work vector does not need information about disk accesses but has an entry for the number of needed network round trips for messages and the number of bytes that are transferred. The CPU related entries in the work vector of a network operator are replicated in order to consider the work done at the sender and receiver site separately. The CPU work estimated for a network operator also reflects the encryption or compression techniques that can be applied to the transfered data. Just as for normal operators, a network operator has different work functions for the open and the next phases of query execution.

4.2.2 *Estimating the Running Time and Monetary Cost of Operators.* The time and cost consumption of an operator *op* can be computed on the basis of the corresponding work vector. For each entry of such a vector the lookup service provides meta-data about the cost and time consumption per unit work for that resource (that is CpU and TpU, respectively) and the load (L) on it for the relevant range of time. For example, for a specific cycle provider we can obtain the information that the execution of an instruction lasts 5 μs and costs 10^{-8} \$ and the load on the CPU is 40%. Now, we can compute the time and cost of *op* for a specific resource *res*. In the formulae, we use the resource name for indexing the vectors for work, cost, and time consumption:

$$\textit{Cost Consumption (CC):}.\ CC_{res} = (WF(pi, mem, dprops))_{res} * CpU_{res}$$
$$\textit{Time Consumption (TC):}.\ TC_{res} = (WF(pi, mem, dprops))_{res} * TpU_{res} * 1/L$$

The overall response time and cost of an operator can then be computed as the summation of the respective values over all the resources. Due to the structure of WF, we can also compute the time consumption in the open (TOC) and the next (TNC) phases separately. In Section 4.3 we will provide more details on the right choice of the load parameters L of resources, which is correlated with the probability for load fluctuations on those resources.

4.2.3 *Estimating Quality Parameters of Plans.* For each sub-plan considered during optimization a corresponding plan descriptor keeps the information about its estimated quality parameters, information about the properties

Table I. The Truth Table for the Parallel-Or

QoS Parameter	Description
QT_{first}	The time interval after which the first tuple has to be produced.
QT_{last}	The time after which all result tuples have to be produced.
QR^R_{comp}	The amount of contributed data for relation R in relation to the whole amount of data available for R.
$QR_{\min\#}$	The lower bound for the result cardinality.
QC_{cycle}	The costs for services of cycle providers.
QC_{data}	The costs for services of data providers.
$QC_{function}$	The costs for services of function providers.

Dprops of the intermediate result produced by the sub-plan, and the associated resource requirements. The plan descriptor not only contains entries for the time dimension of our quality model, but also for the cost- and result quality dimensions. An example plan descriptor is shown below. For simplicity, we omitted the resource requirements and the *Dprops* information and the result quality parameters are restricted to only one relation R.

$$PD := [\underbrace{QT_{first}, QT_{last}}_{\text{time sub-dimensions}}, \underbrace{QR^R_{comp}, QR_{\min\#}}_{\text{result quality sub-dimensions}}, \underbrace{QC_{cycle}, QC_{data}, QC_{function}}_{\text{cost sub-dimensions}}]$$

The meaning of the quality parameters is repeated in Table I. The computation of a plan descriptor for a sub-plan needs the information delivered by the *DPropsF* and *WF* functions of the operator that appears at the top of the sub-plan. This operator is called *root* in the following. Additionally, we need the plan descriptors $\{ipd_1, \ldots, ipd_n\}$ for the n sub-plans, which produce the input data streams of *root*. Together with this information the quality parameters, except for the time quality parameters and the resource requirements, are easy to compute:

—The properties of the intermediate result of the sub-plan are given by the *DPropsF* function of the *root* operator with parameters taken from $ipd_1, \ldots, ipd_n$. If *root* is a leaf node, the information can be found in the meta-data of the data source.

—The computation of the QR_{comp} parameter for a specific relation depends on the kind of *root*:
  —*root* = union: Add the respective parameters from $ipd_1, \ldots, ipd_n$.
  —*root* ≠ union: Take the minimum over the respective parameters from $ipd_1, \ldots, ipd_n$.
  Again, if *root* is a leaf node, the corresponding information can be retrieved from the meta-data.

—The cardinality estimations for the $QR_{\min\#}$ are done with histogram support in the usual way [Poosala et al. 1996] and the information for leaf nodes can again be found in the meta-data.

—The cost quality parameters for data, cycle, and function providers can be computed simply by summing the corresponding values from $ipd_1, \ldots, ipd_n$ and *root*.

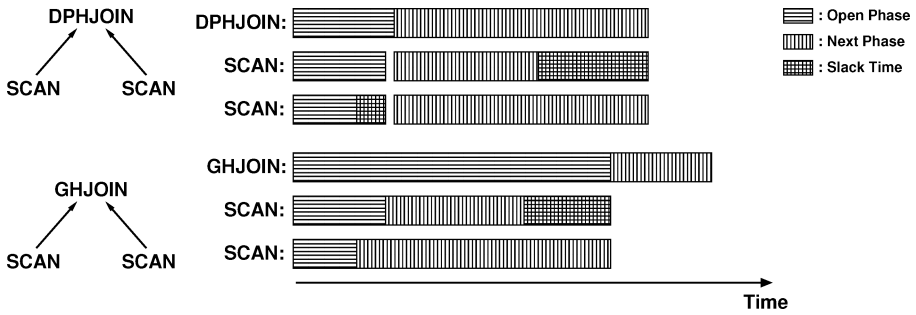

Fig. 4. Chronology of query executions with different scheduling models.

4.2.3.1 *The Scheduling Model of an Operator.* For the computation of the quality parameters of the "response time" dimension it is important to know how operators schedule the actions of their input sub-plans. For example, Figure 4 shows the chronology of two query evaluation plans with a join operator being fed with input data by two scan operators. In the upper plan a double-pipelined hash join (DPHJOIN) is used and in the lower plan a GRACE hash join (GHJOIN). As we can see in the picture, the GRACE hash join consumes its input streams in parallel during the open phase whereas the double-pipelined hash join does so in its next phase. Obviously, the QT_{first} parameter is mostly affected by the different scheduling behavior in this example.

We use a model, which we call the *scheduling model* of an operator, to describe for each operator in a plan how it schedules its own actions together with the actions of its input plans. This model does not only depend on the implementation of an operator but also on the kind of communication between the operators. Communication between two operators can be implemented by a synchronous interface, which results in a pipelined-sequential execution of the corresponding operators, or an asynchronous interface, which is used for communication across process and machine boundaries and results in a pipelined-parallel execution of the corresponding operators. The information about the degree of parallelism in the execution can be expressed in the model by two operators:

—The binary sequence operator ';' indicates that the execution of the left and right operands is equivalent to a sequential execution. By this definition the left and right operand are allowed to intermingle their execution as this occurs in a pipelined-sequential execution of the next phases of two operators.

—The binary parallel operator '||' indicates that the left and right operands are executed in a manner that in its time consumption is equivalent to a parallel execution.

The operands of these two operations can be

—*op.open* and *op.next*, which stand for the actions that are performed by operator *op* in its open and next phases respectively.

—$ipd_k.open$ and $ipd_k.next$, which stand for the actions performed by the subplan associated with the input plan descriptor ipd_k in its open and next phases respectively.

By definition, the sequence operator has a higher precedence than the parallel operator and that precedence can be modified by parentheses in the usual way. Both operators are associative.

We provide two different scheduling terms for an operator in order to reflect the scheduling behavior in its open phase and in its next phase. For example, the corresponding terms for the double-pipelined hash join and the GRACE hash join are:

DPHJOIN: *open: op.open*$\|$*ipd*$_1$*.open*$\|$*ipd*$_2$*.open*
next: op.next$\|$*ipd*$_1$*.next*$\|$*ipd*$_2$*.next*

GHJOIN: *open: op.open*$\|$(*ipd*$_1$*.open*; *ipd*$_1$*.next*)$\|$(*ipd*$_2$*.open*; *ipd*$_2$*.next*)
next: op.next

In these cases all connections between operators are assumed to be pipelined-parallel.[2] A completely pipelined-sequential scheduling model of the GRACE hash join is:

open:. op.open; *ipd*$_1$*.open*; *ipd*$_1$*.next*; *ipd*$_2$*.open*; *ipd*$_2$*.next*
next:. op.next

4.2.3.2 *Evaluating Scheduling Models of Plans.* For the computation of the time quality parameters of a plan we introduce a function *evalTime*, which is applied recursively to the scheduling models of the open and next phases of the operators that appear in the plan. Applied to the scheduling model of the open phase of the operator *op* it computes the QT_{first} parameter of the sub-plan rooted at *op*, applied on the scheduling model of the next phase it computes the value QT_{last}. This means, that for a given sub-plan that appears in the bottom-up optimization process, *evalTime* can be used to compute the QT_{first} and QT_{last} parameters of the plan descriptor. *evalTime* is recursively defined as follows (for a simpler presentation we omitted all the parameters of the function except for the scheduling model):

$$
\begin{aligned}
evalTime(x \| y) &= \max(evalTime(x), evalTime(y)) \\
evalTime(x; y) &= evalTime(x) + evalTime(y) \\
evalTime((x)) &= evalTime(x) \\
evalTime(ipd_k.open) &= ipd_k.QT_{first} \\
evalTime(ipd_k.next) &= ipd_k.QT_{last} \\
evalTime(op.open) &= op.TOC \\
evalTime(op.next) &= op.TNC
\end{aligned}
$$

[2]For the GRACE hash join, operator-internal parallelism is also needed to compute the join in the way the scheduling model suggests, but this can be captured in the scheduling model by encapsulating each input operator in a virtual operator which also performs some actions of the join operator.

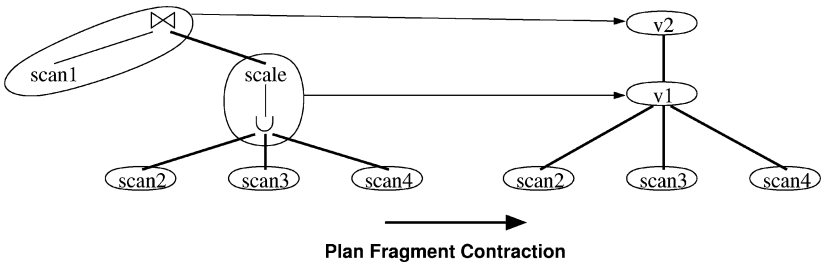

Fig. 5. Fragmentation of QEPs in parallel working plan fragments.

Although the definition of *evalTime* is quite complex, this model is not exact because it does not correctly assess the effects of parallelism in a complex plan. For example, consider the query evaluation plan in Figure 5. The thick lines denote pipelined-parallel connections and the thin lines pipelined-sequential ones. The evaluation of *evalTime* will take into account that the actions of the *union* operator are executed in parallel with those of the operators *scan2*, *scan3*, and *scan4*. Obviously, the actions of the *scale* operator also run in parallel with the three scan operators, but this will not be realized by *evalTime*.

Therefore, the time quality parameters are computed in two phases. First, we consider plan fragments that represent the largest components in the graph of the query evaluation plan, connected only by pipelined-sequential connections. These plan fragments are handled as virtual operators. The time consumption of such a virtual operator is computed by the application of *evalTime* to the plan fragments scheduling model. This scheduling model is constructed from the fragment's operators. An example for the relationship between those plan fragments and the corresponding virtual operators is shown in Figure 5. In the second phase, we merge the scheduling models of those operators of the plan fragment that have a pipelined-parallel input link in order to obtain a scheduling model for the virtual operator. This merge operation is rather simple since the *op.open* and *op.next* actions of the operators' scheduling models are unified to represent the corresponding actions of the virtual operator and the actions of the input plans are scheduled in the same way (parallel/sequential) as in the operators' scheduling models. This means that in the merge process '‖'-operators are merged before ';'-operators. In the virtual operator scheduling model the terms *op.open* and *op.next* refer to the time estimates of the plan fragment computed in the first phase. The time estimates of the resulting reduced plan are then computed by the *evalTime* function.

4.3 Managing Uncertainty in Resource Availability

In Section 4.2.2 we stated, that the optimizer needs meta data about the properties of the input data, the available main memory, and the load on resources used in a query evaluation plan. Obviously, some of these environmental parameters exhibit a somewhat random behavior in the changes of their values and can therefore be considered as random variables with an associated probability

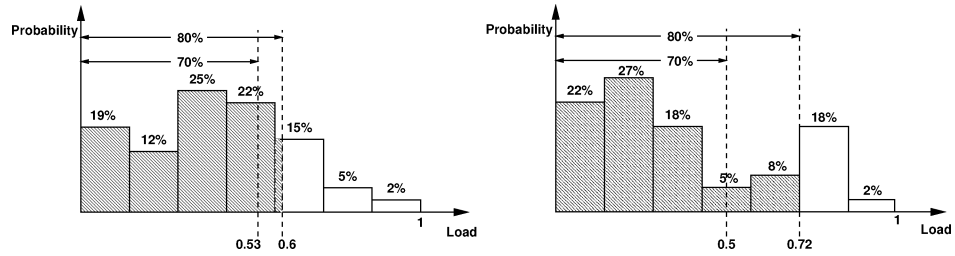

Fig. 6. Load distribution of different cycle providers.

distribution. This observation has also been made for central query processing, which resulted in a solution for finding a query evaluation plan with the least expected costs [Chu et al. 1999]. For QoS management with globally distributed cycle and data providers, it is even more important to take uncertainty in the parameter estimates into account. In the following, for ease of presentation, we will discuss load parameters of time shared resources (e.g., disks, CPUs, network); the availability of space shared resource (e.g., main memory) can be handled analogously.

For estimating the load on resources we use a statistical approach and roughly follow the work reported in Gruser et al. [2000] and Wolski et al. [1997]. Naturally, the activity of users and the resulting load on resources is based on time patterns that can be found within a day,[3] a week, a month, or a year. As described in Gruser et al. [2000], a learning algorithm can be used to identify periods with a rather stable usage of a specific resource. In spite of this classification of usage periods, the load on resources will also vary in such periods. Therefore, for each such period a histogram as shown in both graphs of Figure 6 is constructed to approximate the probability density function for the random variable load for the corresponding resource. The load measurements for the learning algorithm and the histogram construction are retrieved from a distributed resource monitoring framework for computational and network resources. The architecture of such a framework in a globally distributed environment is described in Wolski et al. [1997].

4.3.1 *Uncertainty in the Availability of one Resource.* When generating an evaluation plan for a query, we use the load histograms of resources for the period that corresponds to the scheduled execution time of the query. The question is how to use these statistics. The formulae for computing the time consumption of a query use a fixed load parameter for each affected resource and cannot operate on a histogram. One solution could be to compute the expected time consumption of a query as proposed in Chu et al. [1999] for central query processing. But due to the properties of the expectation, the plan will achieve a higher time consumption than the estimated one with a probability of 50%. If the estimated time consumption exactly hits the corresponding quality constraint of the user, the plan will fail this quality constraint with a probability of 50%. This shows, that a dependability probability for the time

[3]We assume that for the patterns within a day the load is logged with respect to a fixed time zone.

consumption estimates of about 50% is not adequate for a system that tries to enforce user-defined QoS constraints on this parameter.[4]

Therefore, we take the reverse path and permit the specification of a minimum dependability probability P_D for the time consumption estimates. This specification has to be translated into a concrete load estimate for every resource such that the given P_D holds. Now assume, that the whole plan uses just one resource. The optimizer then selects a maximum load value mlv so that the real load value lv falls below mlv according to the given histogram with probability P_D:

$$P(lv \leq mlv) = P_D$$

If we use this value for mlv in the computation of the time consumption, the query evaluation plan will find a resource availability during execution which is with a probability of P_D as good as that assumed during optimization or even better. In the two histograms of Figure 6 load distributions of two different cycle providers are depicted. The height of a bar in such a histogram gives us the probability that a load value between the minimum and the maximum value of the bar's extension on the load axis will be observed at run-time. The shaded areas in Figure 6 correspond to a value of 0.8 for P_D. The largest value on the load axis that belongs to a shaded area denotes the value for mlv. Therefore, the shaded area represents the aggregate probability that the load value will be less than mlv. We can see in the figure that a lower given value for P_D (0.7 in this example) results in a lower value for mlv and consequently in a larger required resource availability for the estimation process. Furthermore, the histograms suggest that for $P_D = 0.8$, the resource corresponding to the left histogram is more suitable than that for the right histogram; on the other hand, the resource depicted in the right histogram is favorable for $P_D = 0.7$.

4.3.2 *Uncertainty in the Availability of n Resources.* If a plan uses n resources and the optimizer proceeds as explained above for each resource, the overall dependability of the time consumption estimate will no longer be reached if we assume that the loads on the resources are independent from each other. Due to the independence assumption, the overall dependability can be computed by multiplying the individual probabilities (P_d for short) and the following holds:

$$P_D \leq (P_D)^n, \; with \; n > 0 \; and \; 0 \leq P_D \leq 1^5$$

To correct this divergence from P_D, we set the individual dependability probabilities for each resource to $\sqrt[n]{P_D}$. For growing n the value for P_d will approach the value 1 rather quickly, for example, $P_D = 0.9$ and $n = 10$ results in $P_d \approx 0.9895$. This means, that the mlv value for a resource grows with n, and consequently the availability and the utility of the resource for enforcing the time quality constraints decreases. In this way, the number of eligible data and cycle providers for the optimization process could be reduced considerably.

[4]Note, that the dependability probability itself depends on the accuracy of the resource statistics.
[5]For example, if each of 5 resources have P_d values of 0.9, then $P_D = 0.9^5 \approx 0.6$.

This effect can be mitigated in many cases:

—Obviously, not all resources have independent load distributions, especially not those resources (CPU, disk) that belong to the same provider. Thus, the optimizer treats these resources and all the incoming network connections for a provider that are present in a specific evaluation plan, as one resource in the computation of P_d. Furthermore, providers with a small net distance to each other (this can be checked with the cluster tree introduced in section 4.1) will most likely show a dependent load. These providers will often be located in the same time zone and will therefore be exposed to the same activity patterns. Hence, providers that appear in the same cluster at a user-defined level of the cluster tree, should also be regarded as one resource in the computation of P_d.

—A rather large value for P_d does not inevitably mean that we have problems in finding suitable providers to produce a QoS compliant query evaluation plan. For example, think of a cycle provider whose load never exceeds 40%. Even if P_d would have a value of 1, 60 % of the provider's resources would be available for the query. Depending on the resource requirements of the query and the nominal power of the provider, this resource availability can be much more than enough. Of course, with increasing P_D and n the number of providers that will not pass this kind of filter in the optimization process will also increase. But only the fittest providers for a given query will pass this filter and this is how it has to work in this setting.

4.4 Pruning Query Evaluation Plans

The quality dimensions span a space that we call QoS space, and the user-defined constraints determine an area in that space that we call QoS window. This is shown in Figure 7 for a (simplified) three dimensional QoS space. During optimization every enumerated plan is mapped to a point in that QoS space by estimating the value for every quality parameter that appears in the quality model. Only plans that lie within the QoS window, fulfill the constraints of the user. For our realtor example the QoS window is given by four intervals:

—[0 \$, 10 \$] is the valid range for the total cost.

—[0 min, 10 min] is the valid range for the total response time.

—[70%, 100%] is the valid range for the QR_{comp}^{Estate} parameter.

—[20, ∞] is the valid range for the $QR_{\min\#}$ parameter.

For multi-objective optimization, pruning works by partially ordering plans. The goal is to compute the Pareto-curve of equivalent plans [Papadimitriou and Yannakakis 2000] This is depicted in Figure 8, where we restricted the QoS space even further to only two dimensions in order to simplify the illustration. The figure shows, for example, that plan *P1* is superior regarding time and cost compared to plans *P4* and *P5*. Although *P1* produces the query result faster, its execution is cheaper than the execution of *P4* and *P5*. *P1*, *P2* and *P3* are incomparable, but only *P1* and *P2* are candidate plans because *P3* lies outside of the QoS window. The arrows emanating from these incomparable plans mark the area in the QoS space that is dominated by the respective plan. The plan

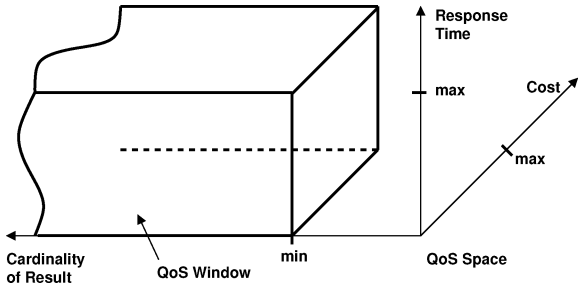

Fig. 7. The QoS space and the QoS window.

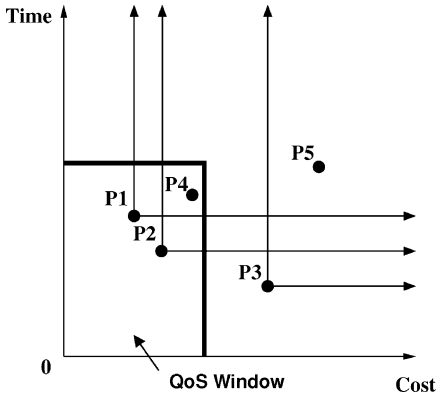

Fig. 8. The partial order for plans.

P4 lies inside the QoS window (i.e., the plan fulfills the user constraints), but it is no candidate plan, because it is dominated by the plans *P1* and *P2*, both of which are superior to *P4* in *all* dimensions of the QoS space. Thus, *P1* and *P2* are the only plans "surviving" pruning.

The definition of the partial order $\leq_p$, which is used to compare alternative plans is shown below. Naturally, $\leq_p$ uses the plan descriptors that contain the quality estimates of the sub-plans generated during the optimization process.

$$PD_1 \leq_p PD_2 \iff \begin{cases} PD_1.QT \leq_t PD_2.QT \\ PD_1.QR \leq_r PD_2.QR \\ PD_1.QC \leq_c PD_2.QC \end{cases}$$

$\leq_t$, $\leq_r$, and $\leq_c$ are defined using the components of each of the three quality dimensions Section 3.1. For example, $\leq_r$ is defined as follows:

$$PD_1.QR \leq_r PD_2.QR \iff$$
$$PD_1.QR_{comp} \geq PD_2.QR_{comp} \ \wedge \ PD_1.QR_{\min\#} \geq PD_2.QR_{\min\#}$$

As mentioned in Ganguly et al. [1992], in the worst case, the complexity of the optimization process increases exponentially with the number of dimensions of the plan descriptor. The worst case is that the values for the different dimensions are distributed independently. In a distributed system, the plan descriptor typically has many dimensions, but fortunately the values of several dimensions

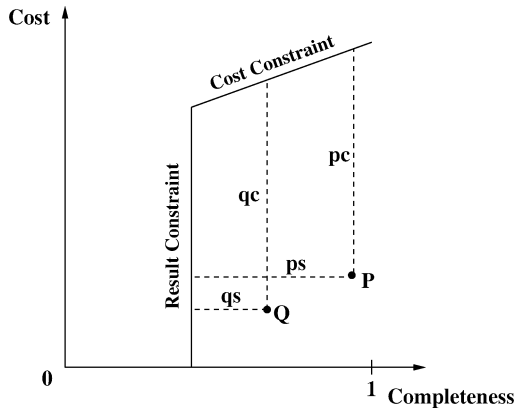

Fig. 9. Plan descriptors in the QoS window.

are correlated. The dimensions QT_{first}, QT_{last}, and QC_{cycle}, for instance, are all based on the work performed by an operator. In addition, the dimensions QR_{comp}, $QR_{\min\#}$, and QC_{data} depend on the selection of data providers for the plan. Furthermore, the possibilities for varying the value for the $QC_{function}$ dimension will be rather limited and nearly equal for all plans compared in a pruning step.

If there is more than one Pareto-optimal plan in the QoS window at the end of the optimization process, as in Figure 8, we need heuristics to choose one winner plan. We should choose a plan that is robust against quality violations caused by resource fluctuations or side-effects of adaptations in the plan instantiation phase or at execution time. Therefore, for every plan descriptor in the QoS window we determine the minimal distance to each of the user-defined borders of the window. Beforehand we normalize the values for the parameters with the maximal values occurring for the respective parameter in these plan descriptors. Otherwise the distances could not be compared in a fair way across dimensions. For our example we get:

$$\begin{aligned}\text{Dist}(PD) = \min(&costConstraint - PD.QC,\\ &timeConstraint - PD.QT,\\ &resultConstraint - PD.QR)\end{aligned}$$

Again, the subtraction operation on each quality dimension is defined appropriately, based on the corresponding sub-dimensions. We then choose that plan descriptor PD out of the set $PD_1, \ldots, PD_n$ of plan descriptors in the QoS window for which the following holds:

$$\text{Dist}(PD) = \max(\text{Dist}(PD_1), \ldots, \text{Dist}(PD_n))$$

For the simplified example of Figure 9, this definition results in the selection of plan P because $\text{Dist}(P) = ps$, $\text{Dist}(Q) = qs$ and $ps > qs$.

4.5 Relaxing Some Constraints on Sub-Plans

As described in the previous section, the optimizer typically selects a plan that is better than what the user expects. Therefore, the execution of a plan has credits

in most quality parameters that result from the differences between the user-defined quality constraints and the estimates of the optimizer. For example, in Figure 9, plan P has a credit of pc cost units for the cost dimensions and of ps percent for the completeness parameter. These credits are maintained in a so called remainder account for the whole plan. We need these credits at execution time in order to decide whether adaptations to a plan are necessary and to detect which adaptations are feasible without violating the quality goals.

The response time dimension must be treated in a special way. During optimization, the response time is estimated by a critical path analysis (Section 4.2.3). Thus, a plan fragment that is not on the critical path, may consume more time than initially estimated by the optimizer because plan fragments executing in parallel consume more time than this one. For example, in Figure 4 two simple join plans are depicted, one with a double-pipelined hash join operator [Wilschut and Apers 1991; Ives et al. 1999] and one with a grace hash join operator. Both join operators drive their input plans in parallel and due to the differing time consumption of the respective input plans, an extra time called *slack time*, can be added to the time constraints of the faster input plans.

Slack time cannot be distributed in a perfect manner during a processing step after optimization, when in the corresponding plan slack time would have to be shared among sequential actions. For example, consider the GRACE hash join plan in Figure 4. Available slack time for the `open` phase of this join operator can be distributed among the `open` and `next` phases of both input plans. But the `open`- and the `next` phase of an input plan are executed sequentially and we do not know in advance the needs of each of the phases for extra time. At runtime we could assign all the available slack time first to the `open` phase and after its execution, the remaining slack time can be assigned to the `next` phase.

However, estimating a plan fragment's resource requirements that are needed for admission control, demands a priori knowledge of its allowed running time. Furthermore, run time distribution of slack time in a distributed environment is a complex task. Thus, we use heuristics to distribute slack time statically after optimization. These heuristics work top-down on the scheduling model of the contracted plan (see Figure 5) and distribute slack time to sequentially executed actions proportionally to their estimated running time. We only have to define how the parallel and the sequence operator in the scheduling model of a plan have to be handled. Assume, that t is a term in the scheduling model. The available slack time for t, which is passed from the operator above in the model is denoted as $t.st$ and the estimated time consumption as $t.et$. The slack time distribution is then defined by the following two formulas:

$$t = t_1; \ldots; t_n \;\Rightarrow\; \forall i \in \{1, \ldots, n\} : t_i.st = t.st \cdot \frac{t_i.et}{\sum_{j=1}^{n} t_j.et}$$

$$t = t_1 \| \ldots \| t_n \;\Rightarrow\; \forall i \in \{1, \ldots, n\} : t_i.st = \max(t_1.et, \ldots, t_n.et) - t_i.et$$

A slack time distribution example for the term $critical\ path \| (t_1\,;\, t_2\,;\, t_3)$ is shown in Figure 10. At the beginning, the extra time from the corresponding remainder account is assigned to the whole scheduling model of the plan as slack time.

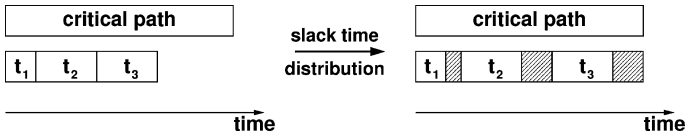

Fig. 10. Slack time distribution.

5. QOS ENFORCEMENT DURING PLAN INSTANTIATION AND EXECUTION

The tasks of QoS management in these two phases are somewhat similar. The assumptions (resource or quality) determined by the optimizer are checked and if a violation is detected, the plan is adapted, rejected or aborted.

5.1 Plan Instantiation and Admission Control

The assumptions of the optimizer are stated in the form of the resource requirements annotated in the plan description. They will be checked by the admission control directly before the respective service is activated. The following resource parameters are rather easy to check, since a simple comparison of values is sufficient:

—The freshness and the size of requested data.

—The cost factors for CPU cycles, usage of functions and data consumption.

—The availability of main memory.

Admission control is more complicated for the resource requirements that affect the time quality parameters of the query. The optimizer estimates the work performed by plan fragments on resources like the CPU or the disk and with the help of operator scheduling models the response time of all plan fragments. As a result, the optimizer produces a resource vector (W, T) for each plan fragment and each affected resource; W denotes the work performed by the fragment on each resource and T the maximum amount of time the plan fragment is allowed to take in order to produce its complete output. As in Garofalakis and Ioannidis [1997] we assume that the work performed by an operator and consequently by a plan fragment is equally distributed during its execution time. When a new plan fragment is to be admitted, admission control checks whether for each resource the work that still has to be performed for all admitted queries permits the execution of the new plan fragment within its deadline.

Therefore, admission control gathers for each of the running plan fragments the information about the remaining work W_r the fragment still has to perform and the time T_r the fragment is still allowed to run. This information is deduced by monitor operators from the original (W, T) vector of the plan fragment. The fraction W_r/T_r denotes the minimum average resource usage the corresponding fragment can tolerate. The vertical bars for *Query* 1 to 3 in Figure 11 represent these resource usages for all active plan fragments. Now, admission control can gather the information about available "resource packages," shown by the shaded boxes in the figure. The size of these resource packages will increase in the future development because already running plan fragments will finish their execution and consequently release occupied resources. This gathering

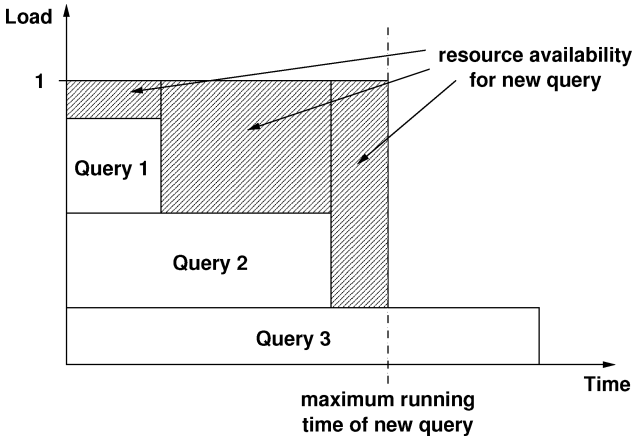

Fig. 11. Gathering available shares on a resource during admission control.

process is limited to the point in time when the maximum running time T of the new fragment is reached. If the size of the collected resource packages does not exceed the work information W for the new fragment, admission fails and the plan fragment is rejected. If this check succeeds for all types of resources affected by the fragment, the plan fragment can start execution on this cycle provider.

If admission control fails, the estimates of the optimizer were too optimistic. The decision whether the plan fragment is changed or the whole query is rejected is made using the credits for all quality parameters. We will describe how this decision is made in Section 6. The same rules apply for admission control as for adaptations at execution time.

If there is a violation in the resource requirements and there are no useful alternative resources, then admission control aborts the plan fragment. As a result, the whole query needs to be reoptimized with adjusted meta-data and restarted.

5.2 Plan Execution and Monitoring

Query evaluation plans in our system have a "natural" fragmentation, which is given by the thread and machine boundaries that appear in the instantiated operator tree. For example, the plan fragmentation as shown in Figure 12 arises when the operators in the two fragments are executed in two different threads on the same machine or on two different machines. Monitoring and as we will see later also adaptation is done on the basis of plan fragments. Every plan fragment monitors its inputs—these are leaf operators of the operator tree or the inputs from other plan fragments—and its output by the use of special monitor operators. For operators that represent pipeline breakers we also introduce monitor operators within a plan fragment in order to monitor the inputs of the pipeline breaker separately. A monitor operator traces the actual quality parameters and forecasts these parameters for the end of the execution. The corresponding optimizer estimates for the respective sub-plan represent the

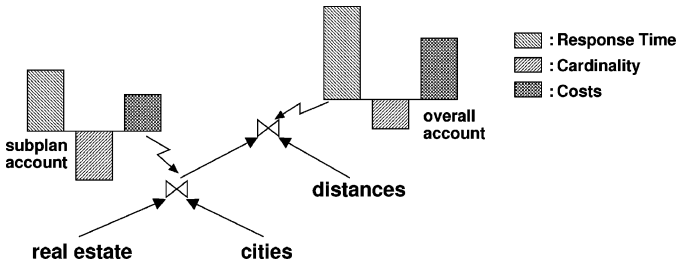

Fig. 12. QoS accounts of a query plan.

target values for the forecasted values and a comparison of these values shows whether the corresponding sub-plan is still within its quotas.

During query execution, monitor operators keep track of the number of tuples produced, the time and cost consumption of the execution and some rates like cost or time consumption per produced tuple. These rates are used for computing the forecasts. For example, the formula

$$T_N + R_{TN}(C_E - C_N)$$

uses the time consumption for the production of all the tuples so far (T_N), the time consumption per produced tuple (R_{TN}) and the estimated and current result cardinality, C_E and C_N, to compute an estimate for the overall time consumption of the sub-plan. For the calculation of the mentioned rates we use a moving average computation in order to detect changes in the execution behavior in a prompt way. The window size of the moving average computation determines how fast we can detect changes in the execution behavior. A larger window size results in a slow, but more reliable forecast; a smaller window size results in a fast, but less reliable forecast because the forecasting mechanism is more susceptible to bursty tuple production and skew in the data. Thus, we adapt the window size during the execution according to the riskiness of a sudden change to our quality parameters. This means that we start with a relatively large window size, which is decreased constantly during the course of the execution.

Additionally, we gather runtime information for plan fragments, which helps in the analysis of critical points in a query execution with jeopardized quality parameters. This information can also be used to reason about the effects of specific adaptations executed on the plan fragment. Some of the information we gather during runtime is listed below:

State Size: The size of the internal state of all operators in the plan fragment that would have to be transmitted, if the plan fragment were moved to another cycle provider.

Execution Progress: This parameter comes in two flavors: the remaining time until the execution must be finished and the number of result tuples, the plan fragment still needs to produce in order to meet the user requirements.

Buffer Pressure: In order to determine, whether a plan fragment is itself a bottleneck or just suffers from slow plan fragments below and above it, we

compare the fill rates of the input and output buffers. The difference between the fill rates of the input buffers and the fill rate of the output buffer determines the buffer pressure. Therefore, a high buffer pressure means, that the corresponding plan fragment is probably the bottleneck in the execution. Conversely, if the input buffers have a low fill rate and the output buffer has a high fill rate, we can deduce that the buffer pressure is low and the current plan fragment is certainly not the bottleneck and therefore adaptations to improve the response time are useless for this plan fragment.

Fragment Selectivity: The monitor measurements for the cardinality of the inputs and the output together with the respective cardinality estimates of the optimizer can be used to compute a current selectivity value for the whole plan fragment. This can be used to detect skew in the data.

In summary, the gathered statistics should help to determine if there is a problem with a quality constraint and if the current plan fragment is responsible for this problem. In the next section we show, how a plan fragment can react to such a problem.

6. PLAN ADAPTATIONS AND CONTROL

Plan adaptations are used during the query instantiation and the query execution phase if it becomes apparent that the user requirements cannot be met with the current plan. In the following we will concentrate on the query execution phase because the adaptations for the query instantiation phase are similar.

6.1 Adaptations

The adaptations that we employ in our system to react to QoS violations, operate on the resource allocation of the query evaluation plan. The corresponding resources are, for example, cycle providers, partitions from data providers and functions from function providers.

6.1.1 *Prevention of Response Time Violation.* If a plan fragment seems to miss its constraints on the response time, we can use adaptations on the machine resource- or the application resource-level. On the machine resource-level we can change the priority of the respective thread (e.g., by a so called *increasePriority* adaptation), the main memory allocation for the respective operators (e.g., by a so called *increaseMemory* adaptation) or we can renegotiate the network service quality (e.g., by a so called *alterNetServiceQuality* adaptation), if the underlying network itself supports QoS handling like an ATM network. The adaptations on the application resource level comprise the activation of compression at runtime for the data sent through a network link (the *useCompression* adaptation) or the movement of plan fragments together with their state from one cycle provider to another (the *movePlan* adaptation)—again, during the runtime of the query. For example, if a monitor detects that a plan fragment suffers from a lack of computing power at its current cycle provider, the QoS management component can decide to move this plan

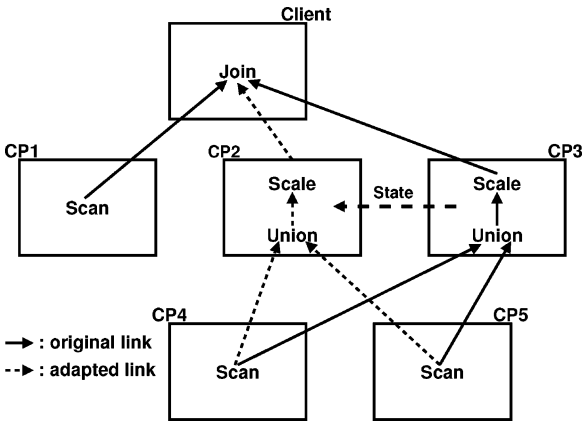

Fig. 13. Moving a plan fragment from one cycle provider to another.

fragment with all its state information to another, better suited cycle provider. This adaptation is depicted in Figure 13, where the scale-union plan fragment is moved from the cycle provider $CP3$ to $CP2$. The remainder of the plan fragment's work is then performed on the new cycle provider. None of the other plan fragments of the same query above and beneath that plan fragment are affected by this move operation, because the relevant communication links between them are disconnected and reestablished automatically by the runtime system of our query processor.

6.1.2 *Prevention of Cardinality or Completeness Violation.* If the cardinality constraints or the completeness constraints are in danger, a possible adaptation is *addSubPlan*, which integrates additional data sources in the query execution that were not involved in the original query execution plan. Of course the information about these additional data sources has to be annotated in the plan during the optimization phase. To accommodate this adaptation we have a union operator that can dynamically establish an additional sub-plan at runtime.

6.1.3 *Prevention of Cost Violation.* If the costs of a plan fragment seem to exceed the corresponding limit, we can try to reduce the amount of processed data, for example, by a so called *reduceCompleteness* adaptation that stops input plans before they are finished. Other ways for reducing cost consumption are the movement of a plan fragment to a cycle provider that charges less for the execution (e.g., to the client's site), or the exchange of externally loaded functions, like thumbnail encoders, with versions that use a more lossy compression technique but which consume less CPU time. The type of the latter adaptation is called *reduceStrength*.

Naturally, there is also an *abort* adaptation, which is initiated when a quality violation in one of the quality dimensions seems inevitable. The decision about the execution of an adaptation depends on specific conditions like the forecasts of the quality parameters or information about the state of the query plan. For example, if for a sub-plan the time quality parameter is jeopardized and there

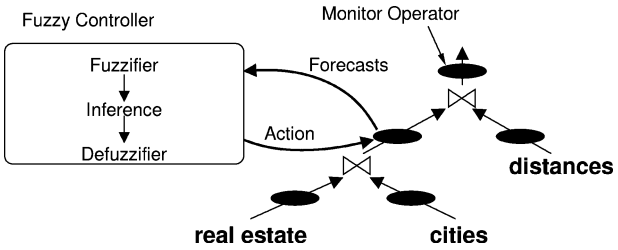

Fig. 14. Feedback loop for QoS adaptations.

is also little scope for the cost quality parameter, it does not make sense to move this sub-plan to a faster, but more expensive cycle provider.

6.2 Fuzzy Control

Plan adaptation during runtime is controlled by the monitor operators themselves because they are placed at the most interesting positions for adaptations in the query evaluation plan and also gather most of the information that is needed for this task. As shown in Figure 14, a monitor operator uses a rule-based fuzzy controller that gets the forecasts of the quality parameters and other state descriptions of the respective plan fragment as input. After the application of a rule-based fuzzy control technique, which we describe later, the controller determines, if an adaptation should be applied and what adaptation this should be. There are proposals in the literature [Ives et al. 1999] to control query plan adaptations by event-condition-action (ECA) rules, which also permit construction of a flexible controller. But such a controller is not suited for making decisions on the basis of uncertain information, which is common in a distributed and heterogeneous environment.

The application of fuzzy controllers has been studied in different application areas, for example, in Li and Nahrstedt [1999] for a visual tracking system. One reason for the use of a fuzzy controller in our system is that our runtime adaptations are discrete and this is quite easy to support in the inference rules that form the basis of the fuzzy controller's decisions. Furthermore, estimation errors and resource fluctuations introduce some uncertainty factors in the control process, which can be modeled by fuzzy logic [Klir and Yuan 1995] quite easily. The inference rules of a fuzzy controller also provide a highly configurable means to incorporate expert knowledge for the adaptation process in a very intuitive way. Therefore, it becomes possible to experiment with varying adaptation strategies by just changing the inference rules and perhaps the definition of the underlying fuzzy sets. In the following we present the three components of a fuzzy controller in our context: fuzzifier, fuzzy inference rules and defuzzifier.

6.2.1 *Fuzzifier.* The task of the fuzzifier is to transform the input of the controller into a representation that can be used to trigger the inference rules. In our case the input consists of values for forecasts of quality parameters and values for state information of the plan fragment. These numeric values must be transformed to *linguistic values* of *linguistic variables*. For each input

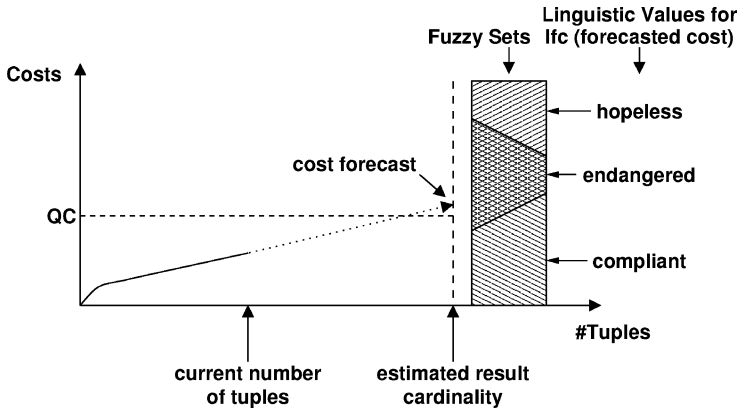

Fig. 15. Mapping quality parameter forecasts on fuzzy sets.

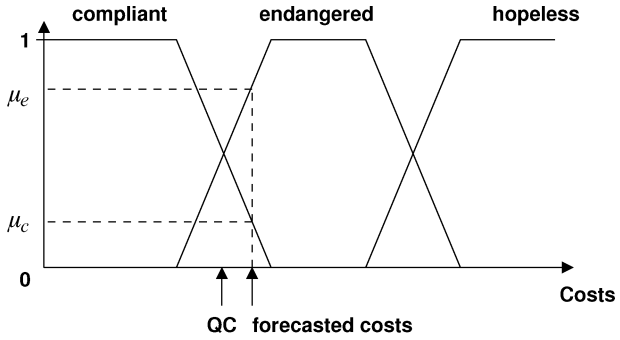

Fig. 16. The fuzzy sets for the cost forecast value.

variable of the controller there exists one such *linguistic variable*. This transformation is depicted for the forecasted cost parameter in Figure 15—we call the corresponding linguistic variable *lfc*. We see, that the domain of the input is partitioned into different fuzzy sets. Each fuzzy set represents a linguistic value of *lfc*, and the *weights* of the forecasted value that specify the level of containment to the fuzzy sets, can be determined. The linguistic values that correspond to these sets are normally adjectives that qualify the corresponding input variable. In our case the linguistic values *hopeless*, *endangered* and *compliant* qualify the chance of meeting the QoS constraints for this quality parameter. Figure 16 shows in more detail the partitioning of an input variable in fuzzy sets. A fuzzy set is described by its member function; an input value need not belong to exactly one set, but can belong to different sets with different weights—the input value in the figure belongs to the *endangered* fuzzy set with weight μ_e and to the *compliant* fuzzy set with weight μ_c.

6.2.2 *Fuzzy Inference Rules.* The inference rules of a fuzzy controller are of the form

$$\textbf{if } X_1 \text{ is } A_1 \text{ and } \ldots \text{ and } X_n \text{ is } A_n \textbf{ then } Y \text{ is } B$$

where $X_1, \ldots, X_n$ and Y are linguistic variables and $A_1, \ldots, A_n$ and B are linguistic values with accompanying fuzzy sets. A small portion of an example rule set is shown below. Here *lfc*, *lfr* and *lft* are the linguistic variables for cost-, result- and time-quality parameters; *lep*, *lbp* and *lss* correspond to the execution progress, the buffer pressure and the state size.

if *lfc* is *endangered* and *lep* is *late* **then** *abort* is *preferred*

if *lft* is *endangered* and *lbp* is *high* and *lss* is *small* **then** *movePlan* is *preferred*

if *lft* is *endangered* and *lep* is *middle* and *lbp* is *high* **then** *increasePriority* is *possible*

if *lfr* is *endangered* and *lep* is *early* and *lfc* is *compliant* **then** *addSubPlan* is *preferred*

The variables in the **then** part of a rule represent specific adaptations. Every adaptation defines its own linguistic variable and the corresponding values, for example, *not recommended*, *possible* and *preferred*, denote the applicability of the adaptation. Thus, our rule base consists of rules, whose heads describe the premise of specific adaptations. The combined weights of the linguistic variables of a rule's premise then determine the weight of its conclusion. In our case, the *and* operator in the premises of the rules is implemented by a *minimum* operation on the weights of the linguistic values. For example:

$$\mu_{endangered}(lfc) = 0.8 \ \wedge \ \mu_{late}(lep) = 0.9 \implies \mu_{preferred}(abort) = minimum(0.8, 0.9) = 0.8$$

6.2.3 *Defuzzifier.* After all rules have been applied, all the resulting weights for an output linguistic variable are combined with an application-specific defuzzification method in order to get an overall value for the corresponding, virtual applicability scale of an adaptation. Here we assign every linguistic value a corresponding singleton out of the applicability scale; for our example, -1 is assigned to *not recommended*, 0.5 to *possible* and 1 to *preferred*. The overall applicability value x_a for an adaptation a is then determined by:

$$x_a = \mu_{not\ recommended}(a) * (-1.0) + \mu_{possible}(a) * 0.5 + \mu_{preferred}(a) * 1.0$$

We then choose the adaptation with the highest applicability value x_a, if this value is above 0.5. If there is no such adaptation, nothing will be done. For example, when for a certain adaptation, say *movePlan*, the combination of the inference rules yields $\mu_{not\ recommended}(movePlan) = 0.6$ and $\mu_{preferred}(movePlan) = 0.7$ then

$$x_{movePlan} = 0.6 * (-1.0) + 0.7 * 1.0 = 0.1$$

The adaptation will not be executed because $x_{movePlan}$ is smaller than the threshold of 0.5. This shows, that during defuzzification the output of different rules regarding the same adaptation are balanced. This is a special feature of a fuzzy controller in comparison to a pure ECA-rule based mechanism, where the rules are evaluated in isolation. The balancing of all rules for a certain adaptation helps in the design of the rule base. For example, for specific exceptional

cases we can add "*veto rules*," which inhibit the activation of the corresponding (default) adaptation.

7. PERFORMANCE EXPERIMENTS AND RESULTS

Naturally, if the optimizer is perfect at compilation time, enforcing QoS constraints during the execution time is not necessary. As mentioned in the previous sections, however, there are many situations in which the optimizer cannot possibly make the right decisions. In the experiments presented in this section, we would like to demonstrate how the QoS management techniques devised in Sections 5 and 6 perform in such situations. The focus of our experiments is to show some of the scenarios in which QoS management is particularly useful and how much benefit our techniques achieve in such situations.

7.1 The Effectiveness of Adaptations in a Distributed Environment

In the first set of experiments, we study to what extent monitor operators can forecast quality violations and whether our adaptations can be used to react to such violations. We use three Internet hosts as cycle providers located in *Passau* (Germany), *Mannheim* (Germany), and *Maryland* (USA, east coast). In the following, we use the locations as names for the hosts. Each host is used as a cycle provider and additionally as a data provider for the *order* and *lineitem* relations of the TPC-D benchmark. In the first experiment, we also use wrappers for the data providers HotelBook (`www.hotelbook.com`) and HotelGuide (`www.hotelguide.com`), which deliver tuples with information about hotels. Both hotel data providers are located in the USA.

7.1.1 *Monitoring and Adapting Plans: The movePlan Adaption.* This experiment particularly assesses the efficacy of the monitoring component in collaboration with the *movePlan* adaptation. The query asks for hotel information from HotelBook and HotelGuide for a specific city (i.e., Philadelphia). The results of HotelBook and HotelGuide are merged and returned to the user. We assume that this query is executed under cost and time constraints and that *Passau* is the client that is cheap and *Maryland* is a commercial cycle provider, which is expensive. The execution time of the sub-plan is restricted to 200 seconds.

Figure 17 and Figure 18 show the running times, executing the query at different times of the day on two different days. We studied three different plans for this query: (a) *Passau*: this plan uses the cheap client resources only; (b) *Maryland*: this plan uses the expensive cycle provider in Maryland; this cycle provider is located near the two hotel data sources; (c) *QoS enabled*: this plan uses plan adaptations and full QoS management as proposed in this article. It starts like the *Passau* plan (using cheap resources only) and adapts if necessary in order to meet the response time goals. For the other two plans, such adaptations are disabled.

In terms of running time, the *Maryland* plan is always the best; this plan uses the cycle provider in Maryland, which is located close to the data source in order to prefilter the data and thus significantly reduce communication cost between USA and Passau. However, the *Passau* plan, which has significantly

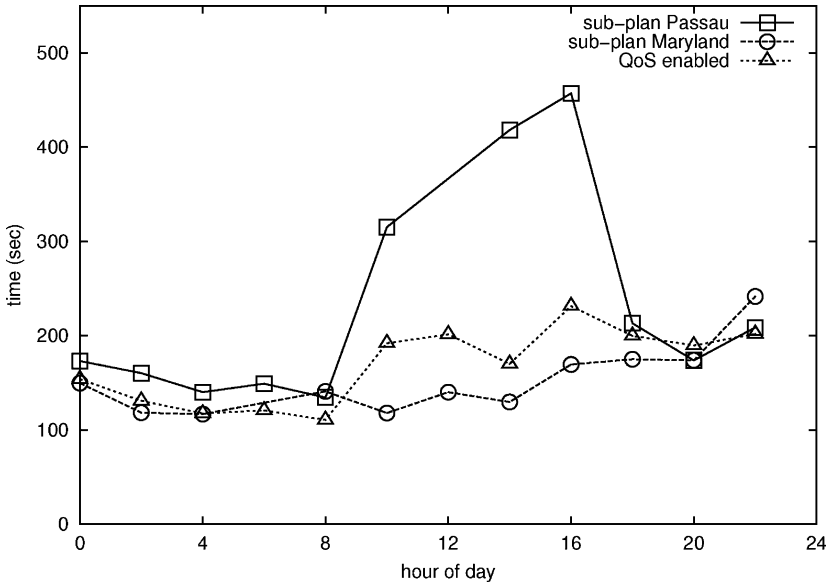

Fig. 17. Retrieving data through wrappers on day 1.

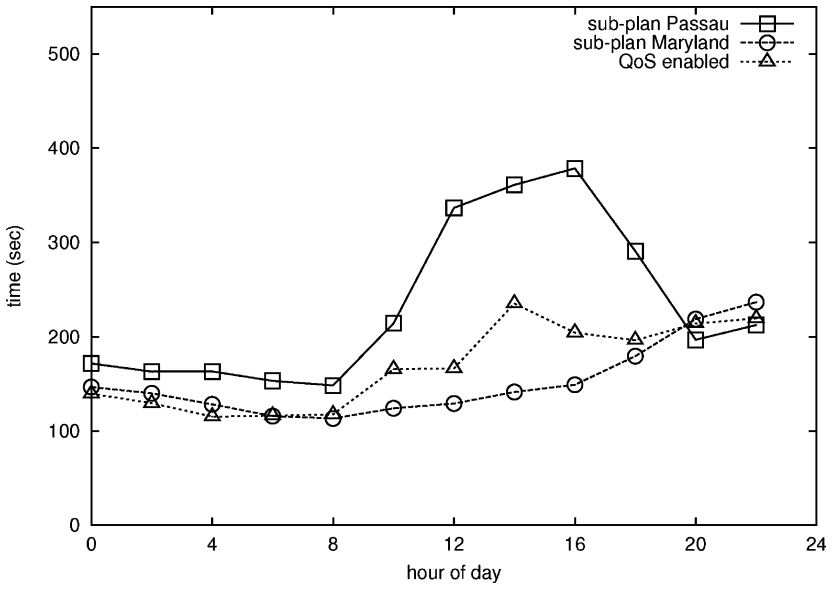

Fig. 18. Retrieving data through wrappers on day 2.

less monetary cost, can compete most of the time. During the rush hours, from 10 a.m. to 5 p.m. however, the *Maryland* plan should be used in order meet the 200 second restriction.

The *QoS enabled* has as good or better performance as the *Passau* plan and always slightly worse performance than the *Maryland* plan. Most importantly,

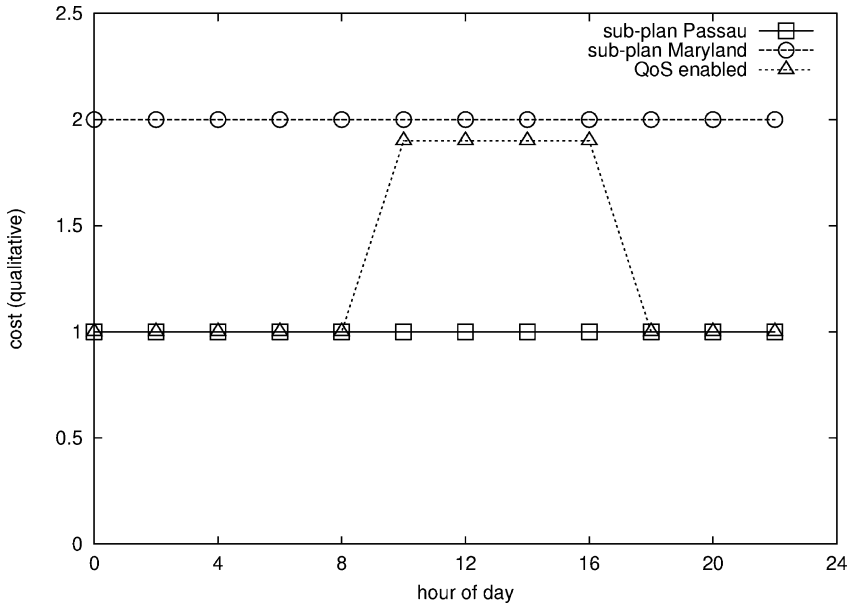

Fig. 19. Qualitative view of the wrapper experiments.

however, it always meets (or only marginally fails) the 200 second barrier and correctly adapts during the rush hours using the *movePlan* adaption. In other words, during rush hours, it forecasts response time violations caused by its initial plan and changes the plan in order to meet the response time requirements set by the user.

A qualitative analysis of this experiment can be seen in Figure 19. This figure shows that the used QoS management setup tries to find the cheapest plan that still fulfills the time constraint. The cheaper plan is favored as long as the time consumption does not exceed our limit and if this happens the execution switches to a more expensive, but faster plan. Obviously, such an adaptation cannot always save a jeopardized quality constraint.

7.1.2 *Monitoring and Adapting Plans: The useCompression Adaption.* The query used in this experiment performs a simple table scan (without any filtering) on the TPC-D *lineitem* data stored in *Mannheim*. The client machine is *Passau* and the scan operation at *Mannheim* is performed on a *lineitem* partition with a size of 10MB. Again, we assume that the query is executed under cost and time constraints and that *Mannheim* is a cycle- and also a data-provider whose services must be paid. The time constraint for the query was set to 60 seconds. The adaptation that we test here is called *useCompression*, which turns on compression for a network connection and therefore speeds up network transmission. Compression is performed with the help of the ZLIB library on blocks of the respective intermediate result.

However, compression of data increases the CPU time consumption at the receiver's and even more at the sender's site. In our tests, the CPU time consumption at the sender increased by a factor of three, when compression was

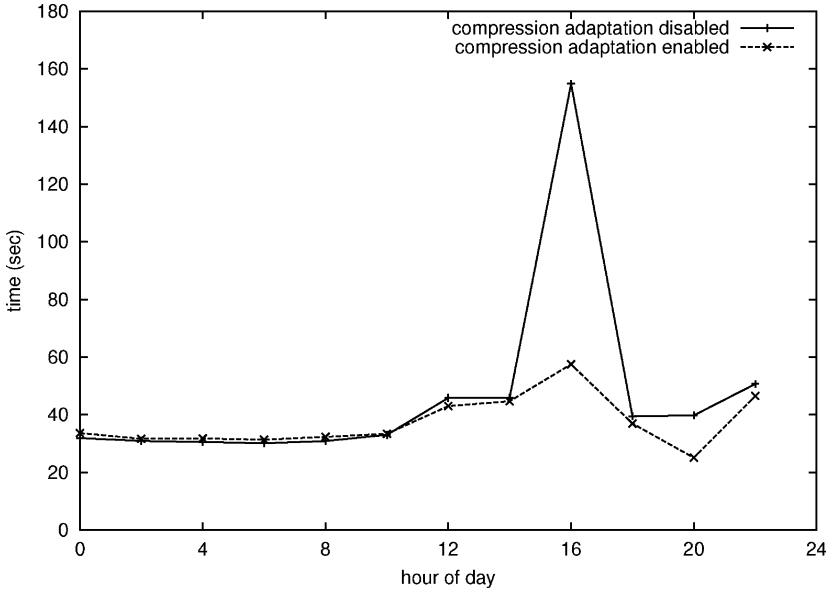

Fig. 20. Executing a remote scan operation on day 1.

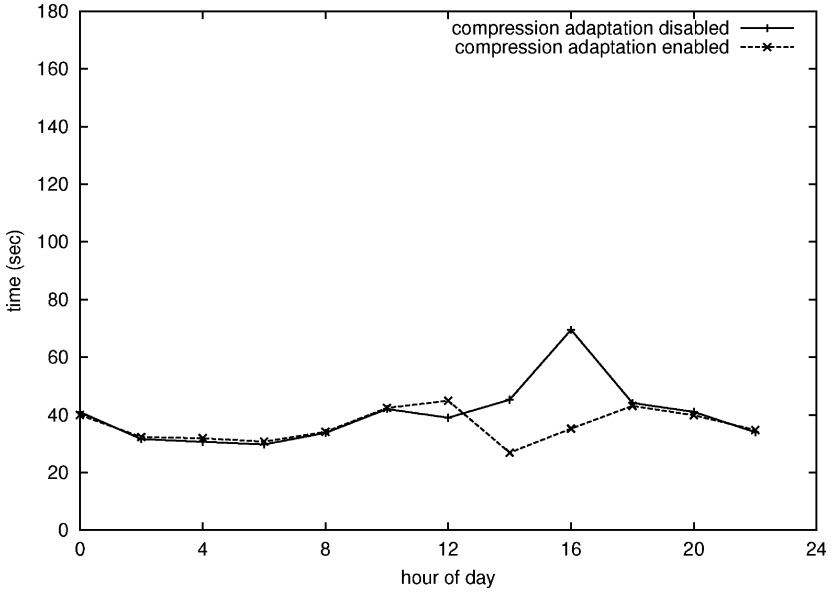

Fig. 21. Executing a remote scan operation on day 2.

turned on. Since *Mannheim* is the sender site, QoS management has to balance the time savings due to compression and the increased costs for CPU-usage at the cycle provider *Mannheim*. The running times of the experiments at two different days are shown in Figures 20 and 21. We see, that the plan without compression is able to fulfill the time constraint most of the time. Therefore,

analogous to the previous experiment, the initial plan is without compression turned on and the alternative for the resource *network* is to turn on compression.

As before, we executed the QoS plan and the one without QoS support every two hours in a 24 hour range and again used the rush hours on the Internet to simulate resource degradations on the network. In both experiments the subplan without compression missed the time constraint just once. The monitor operator in the QoS plan detected this situation and activated the compression on the network link as can be clearly seen in the graphs. The savings in time has to be paid for by an increased cost for the execution at *Mannheim*.

7.2 QoS Management in Heavily Loaded Multi-User Environments

In the previous two experiments, monitoring and making the right decisions was difficult because of the unpredictable behavior of the Internet. In the next experiment we study a local area network with a highly unpredictable multi-user behavior. In this experiment, we measure the behavior of a system during a fixed amount of time, *et*. In this time frame, each client initiates a fixed number of queries, *nq*. All clients initiate the same query, but the quality parameter for the overall execution time of the query is varied. The query scans about 2000 tuples from a hotel information data set with a total volume of 23 MB. Associated to each hotel is a picture in JPEG format that is scaled down during the query execution by a specialized operator in order to format the results for the user's desktop. The execution times of the queries are CPU-bound since this scale operation is CPU-intensive and dominates the overall running time of the query. The scale operation must be run on the server since the clients are thin.

The client *c1* issues queries with an overall response time limit of 185 seconds. Since the scale operation requires 130 seconds CPU time at the server, *c1* has an average server CPU utilization of 70% if the query completely exhausts its time limit. The clients *c2* and *c3* restrict the response time of the query to 370 seconds (twice the value of *c1*) and 555 seconds (three times the value of *c1*), respectively. The corresponding CPU utilizations are 35% and 23%, respectively. During our experiments we use an *nq* value of 20, which means that each client issues 20 queries, and we used *et* values of 1, 2, 2.5, and 3 hours. At the beginning of an experiment, each client randomly chooses the *nq* starting times in the time frame given by *et*. This way, the resulting arrival process for each client approximates a Poisson process with parameter $\lambda = nq/et$. A client starts its queries at the randomly chosen points in time and thus it is possible that two query executions of the same client overlap[6] and of course, queries from different clients overlap with a very high probability in this scenario. If we tried to execute all queries, the average CPU utilization on the server would be 217% ($et = 1$), 108% ($et = 2$), 87% ($et = 2.5$) and 72% ($et = 3$). Consequently, only for $et = 2.5$ and $et = 3$, there is a theoretical chance to execute all queries and meet the response time requirements of the users. In all situations, there will be heavy contention on the server's CPU.

[6]Each query is issued by its own client process, so this overlap is possible.

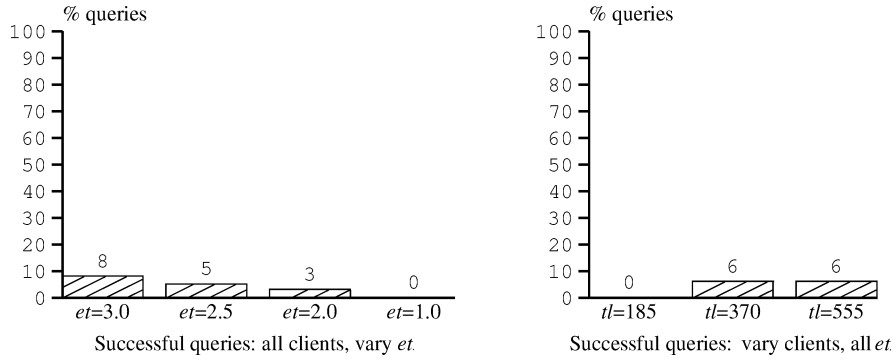

Fig. 22. The success of queries with no QoS techniques applied.

7.2.1 *Experimental Results without QoS Management.* Figure 22 shows the number of queries that could be executed successfully (not violating the response time restrictions of the individual clients) if no QoS management is carried out; if no admission control is used, all queries have the same priority, and plans are neither adapted nor stopped if they miss their quality constraints. This is the typical best-effort configuration as found in most distributed information systems today, for example, the WWW.

Figure 22a shows the percentage of successful queries for different *et* values. Figure 22b shows the results from the perspective of each client. Both charts show that only a very small fraction of queries fulfill their QoS requirements. Naturally, queries with lower demands have a better chance to meet their constraints, thus clients *c2* and *c3* look better than *c1* in Figure 22b. As expected, the main problem is the high load on the CPU of the server that results in a high backlog of queries in the system. Ultimately no query can be executed within its limits.

7.2.2 *Experimental Results with Admission Control Activated.* In order to study the effectiveness of the different QoS management components, we first enabled admission control, but we did not activate plan adaptations during the execution of queries. We studied the following metrics in this scenario.

—the fraction of queries that succeeded within their time limits;

—the fraction of queries that were rejected by admission control;

—and the fraction of queries that were admitted but failed to meet their time limits. (All fractions sum up to 100%.)

Figure 23 shows the results for varying *et* values. Figure 24 shows the results from the perspective of each of the three clients with different *tl* values. It is clear that admission control has improved the results significantly. Rejection of queries effectively takes load from the CPU of the server. For client *c1*, 24% of the queries met the QoS requirements; whereas not a single query met the QoS requirements without admission control.

The percentages of admitted but unsuccessful queries show that there is still a great deal of potential for improvement. These percentages are rather high; especially the queries with more demanding time limits (client *c1* with $tl = 185$)

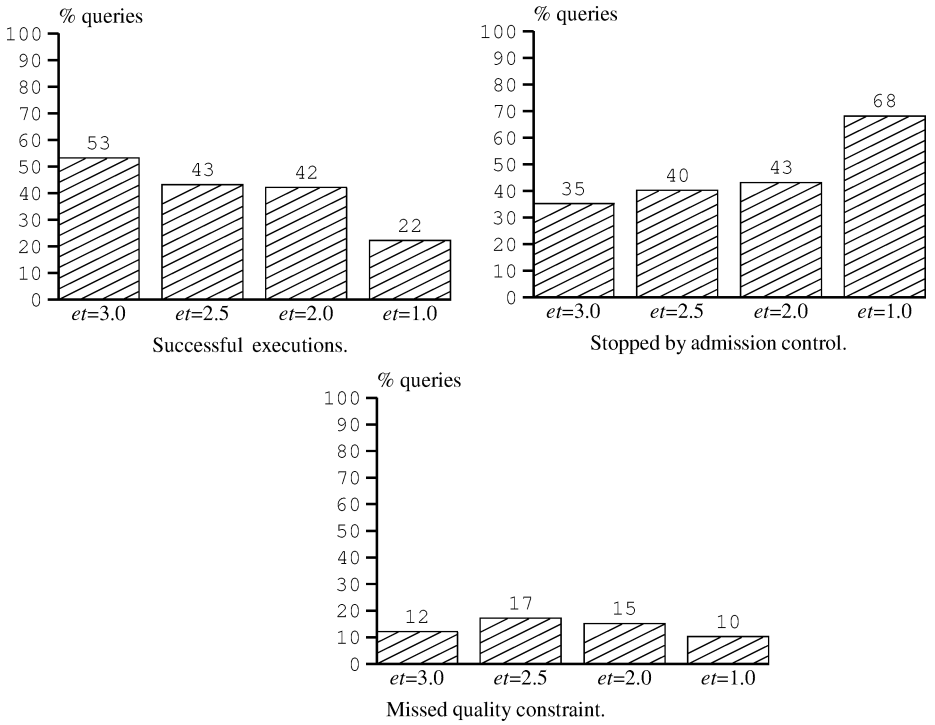

Fig. 23. The success of queries for different *et* values with admission control activated.

suffer from a relatively high probability of missing their target response time. The problem here is that the queries are not prioritized according to their demand. As in earliest deadline scheduling in real-time systems, queries with a shorter execution period should get a higher execution priority than those with a more relaxed time limit. The results of the experiments that use run-time adaptations to adjust such priorities are discussed in the following.

7.2.3 *Experimental Results with Full QoS Management Support.* The task of our QoS management in this experiment is to use the adaptations *increasePriority* and *decreasePriority* such that as many queries as possible are executed within their time limit. Furthermore, admission control and the usage of the *abort* adaptation must be used to stop queries that will obviously miss their time limit. This means, that the affected components of our QoS management are admission control, monitoring, and adaptation control, where the latter is performed by the fuzzy controller described in Section 6.

Figure 25 shows the corresponding results with full-fledged QoS management. We report on the fraction of queries that were executed successfully, stopped by admission control, and stopped during execution due to a predicted response time violation. Again, we show the results varying *et* (Figure 25) and for each client, varying *tl* (Figure 26). We observe a significant improvement compared to the situation when only admission control was in effect and a dramatic improvement compared to the situation when no QoS management

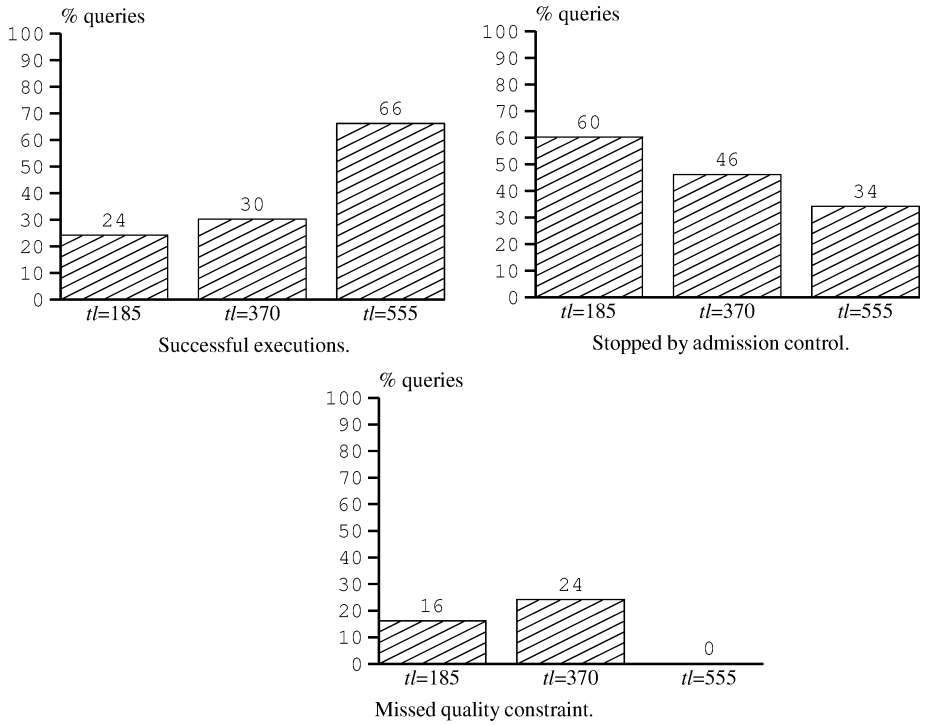

Fig. 24. The success of queries for different *tl* values with admission control activated.

was carried out. The reason is that admission control (just as initial query optimization) cannot be perfect in this complex workload. Given the complicated and unpredictable pattern in which queries with different requirements arrive, only adaptations during the execution time of queries, after resource contention has been detected, will really help.

Admission control assesses the future development of already running queries in order to decide if a given query can accumulate enough resources to fulfill its quality constraints. This means that admission control states that theoretically resource availability will be sufficient in order to meet the quality constraints of all active queries on the respective server. The fuzzy controller's task now is to try to 'prove' this statement:

—The fuzzy controller uses the *increasePriority* and *decreasePriority* adaptations in order to balance the priorities of queries in such a way that the monitoring component for each query signals compliance with the constraints and no query excessively consumes resources while other queries are in danger of missing their constraints. For example, the priority of a query with a *tl* value of 185 will be increased until the monitoring component signals that this time constraint will probably be met. On the other hand, if a query with a *tl* value of 555 seems to meet this time limit rather easily and other queries seem to lack resources, the priority of this query will be decreased to free resources for the queries with higher needs.

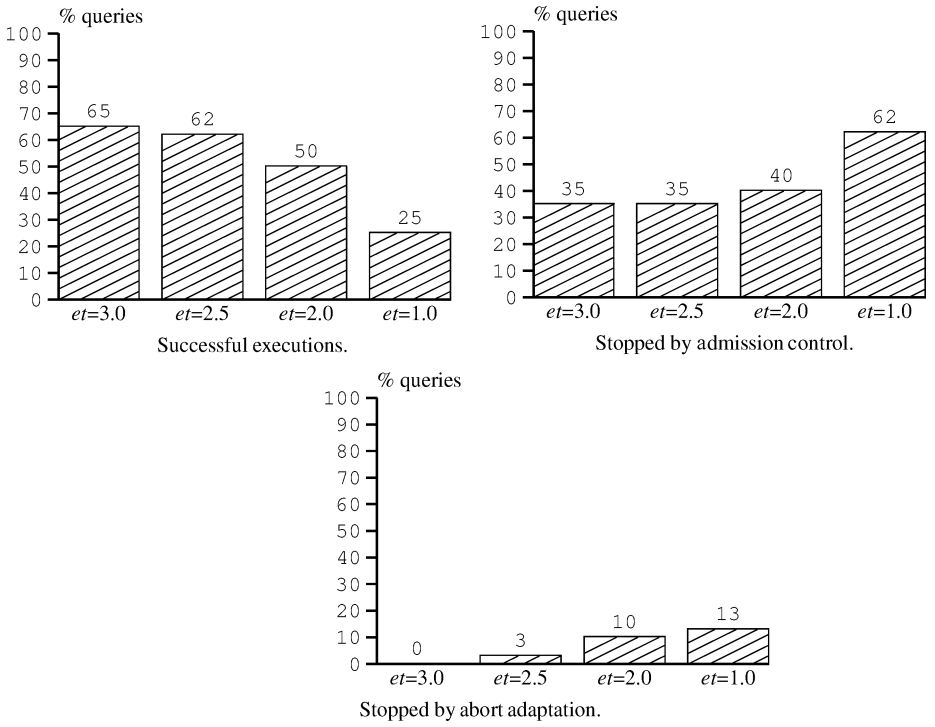

Fig. 25. The success of queries with different *et* values.

—If due to resource fluctuations and estimation errors a query will not meet its time constraint with a high probability, the *abort* adaptation is used by the fuzzy controller in order to free resources for other queries.

As a positive result of this balancing between the selfishness of queries and the cooperation between them, we can see in Figure 26 that QoS management at a cycle provider does not overly prefer a specific class of queries. Although queries with smaller demands will meet their quality constraints much more easily than the others, in our experiments the queries with a time limit of 185 seconds also have a good chance to meet their quality constraint.

A little bit astonishing is the observation that the percentage of queries that have to be stopped by the *abort* adaptation is much higher for the class of queries that have a time limit of 555 seconds than for the classes of queries with time limits of 185 or 370 seconds. This effect can be explained by the time a fuzzy controller needs to adapt the priorities of queries to new load situations. Queries with a longer running time are more exposed to these transition phases of the fuzzy controller and so, when resources are scarce, these queries are more prone to quality misses regarding the overall execution time.

8. CONCLUSION

In this work, we presented extensions for the different phases of query processing: query optimization, query plan instantiation and query plan execution. The

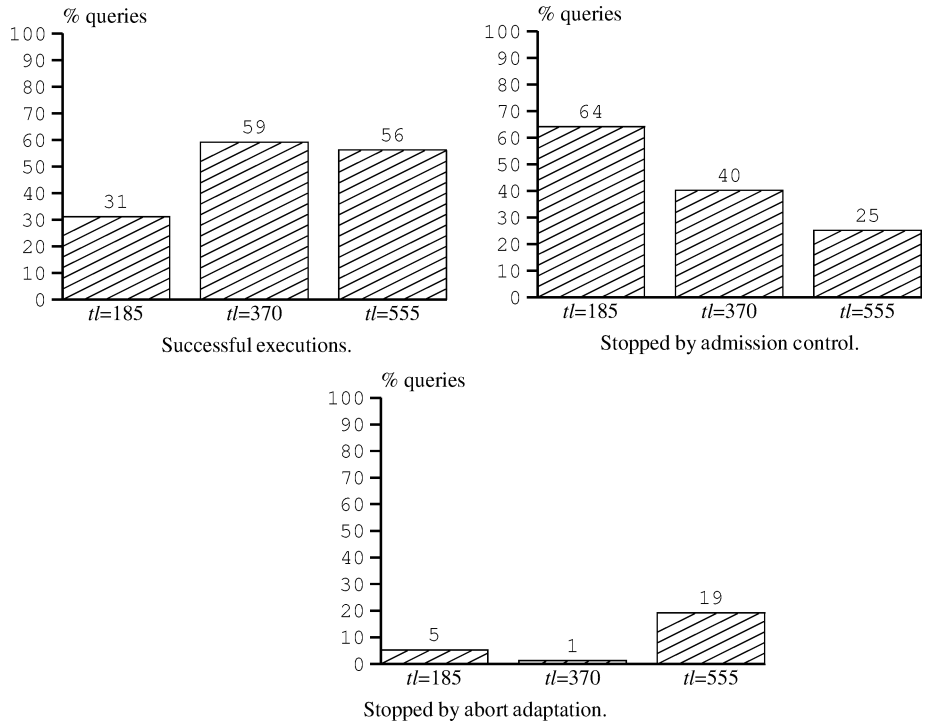

Fig. 26. The success of queries with different time limits.

goal was to support user-defined QoS constraints. One of the main challenges is that the query processor works in an open information economy, where a query uses resources and services of independent and autonomous providers for CPU cycles, data and functions. Based on statistics about providers, the optimizer constructs a plan that combines the services of selected providers in such a way that its quality estimates are compliant with the user-defined quality constraints. For each requested service, necessary resource requirements and quality constraints are annotated in the plan and these are monitored during query plan instantiation and execution. If a forecast detects a potential quality violation, a fuzzy controller tries to adapt a plan based on the current situation of the system. In many cases, an adaptation claims additional resources that are not used by other queries at that time. Effectively, a plan adaption trades one performance target (e.g., response time) for another (e.g., monetary cost or completeness and freshness of results). If all adaptations violate at least one QoS goal, the query is aborted. We carried out initial performance experiments which are very encouraging and demonstrate that our techniques for QoS management are very effective in many situations.

ACKNOWLEDGMENTS

We would like to thank all the members of the ObjectGlobe team for their work within that project: Markus Keidl, Alexander Kreutz, Stefan Pröls, Stefan Seltzsam, and Konrad Stocker.

REFERENCES

ABOULNAGA, A. AND CHAUDHURI, S. 1999. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*. Seattle, WA, 181–192.

ANTOSHENKOV, G. AND ZIAUDDIN, M. 1996. Query processing and optimization in Oracle RDB. *VLDB Journal 5*, 4, 229–237.

AURRECOECHEA, C., CAMPBELL, A., AND HAUW, L. 1998. A survey of QoS architectures. *Multimedia Systems Journal 6*, 3, 138–151.

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Dallas, 261–272.

BEYER, K. S., LIVNY, M., AND RAMAKRISHNAN, R. 1998. Protecting the quality of service of existing information systems. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems*. New York.

BRAUMANDL, R., KEIDL, M., KEMPER, A., KOSSMANN, D., KREUTZ, A., SELTZSAM, S., AND STOCKER, K. 2001. ObjectGlobe: Ubiquitous query processing on the Internet. *The VLDB Journal: Special Issue on E-Services 10*, 3 (Aug.), 48–71.

BROWN, K. P., MEHTA, M., CAREY, M. J., AND LIVNY, M. 1994. Towards automated performance tuning for complex workloads. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Santiago, Chile, 72–84.

CAREY, M. AND KOSSMANN, D. 1998. Reducing the braking distance of an SQL query engine. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. New York, 158–169.

CHU, F., HALPERN, J. Y., AND SESHADRI, P. 1999. Least expected cost query optimization: An exercise in utility. In *Proceedings of the ACM SIGMOD/SIGACT Conference on Principle of Database System (PODS)*. ACM Press, Philadelphia, Pennsylvania, 138–147.

FLORESCU, D., KOLLER, D., AND LEVY, A. Y. 1997. Using probabilistic information in data integration. See VLDB [1997], 216–225.

GANGULY, S., HASAN, W., AND KRISHNAMURTHY, R. 1992. Query optimization for parallel execution. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. San Diego, CA, USA, 9–18.

GAROFALAKIS, M. AND IOANNIDIS, Y. 1997. Parallel query scheduling and optimization with time- and space-shared resources. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Athens Greece, 296–305.

GRAEFE, G. AND WARD, K. 1989. Dynamic query evaluation plans. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Portland, OR, 358–366.

GRUSER, J. R., RASCHID, L., ZADOROZHNY, V., AND ZHAN, T. 2000. Learning response time for websources using query feedback and application in query optimization. *The VLDB Journal 9*, 1 (May), 18–37.

IOANNIDIS, Y., NG, R., SHIM, K., AND SELLIS, T. 1992. Parametric query optimization. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Vancouver, Canada, 103–114.

IVES, Z., FLORESCU, D., FRIEDMAN, M., LEVY, A., AND WELD, D. 1999. An Adaptive Query Execution Engine for Data Integration. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Philadelphia, PA, 299–310.

KABRA, N. AND DEWITT, D. 1998. Efficient mid-query re-optimization for sub-optimal query execution plans. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Seattle, WA, 106–117.

KLIR, G. J. AND YUAN, B. 1995. *Fuzzy Sets and Fuzzy Logic*. Prentice Hall.

LEVY, A., RAJARAMAN, A., AND ORDILLE, J. 1996. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Bombay, India, 251–262.

LI, B. AND NAHRSTEDT, K. 1999. A control-based middleware framework for quality of service adaptations. *IEEE J. Sel. Areas Comm. 17*, 9, 1632–1650.

PANG, H., CAREY, M. J., AND LIVNY, M. 1995. Multiclass query scheduling in real-time database systems. *IEEE Trans. Knowl. Data Eng. 7*, 4 (Aug.).

PAPADIMITRIOU, C. H. AND YANNAKAKIS, M. 2000. On the approximability of trade-offs and optimal access of web sources. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*.

PETIT, J. 1999. Real Estate DTD. `http://www.4thworldtele.com`.

POOSALA, V., IOANNIDIS, Y., HAAS, P., AND SHEKITA, E. 1996. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Montreal, Canada, 294–305.

ROTH, M. T., OZCAN, F., AND HAAS, L. 1999. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Edinburgh, UK, 599–610.

SISTLA, A. P., WOLFSON, O., YESHA, Y., AND SLOAN, R. 1998. Towards a theory of cost management for digital libraries. *TODS 23*, 4 (Dec.), 411–452.

STONEBRAKER, M., AOKI, P., LITWIN, W., PFEFFER, A., SAH, A., SIDELL, J., STAELIN, C., AND YU, A. 1996. Mariposa: A Wide-Area Distributed Database System. *The VLDB Journal 5*, 1 (Jan.), 48–63.

URHAN, T., FRANKLIN, M., AND AMSALEG, L. 1998. Cost based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Seattle, WA, 130–141.

VLDB 1997. *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Athens, Greece.

WEIKUM, G. 1999. Towards guaranteed quality and dependability of information services. In *Proceedings of the GI Conference on Database Systems for Office, Engineering, and Scientific Applications (BTW)*. Informatik aktuell. Springer-Verlag, New York, Berlin.

WILSCHUT, A. AND APERS, P. 1991. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems*. Miami, Fl, 68–77.

WOLSKI, R., SPRING, N., AND PETERSON, C. 1997. Implementing a performance forecasting system for metacomputing: The network weather service. In *Proceedings of Supercomputing'97 (CD-ROM)*. ACM SIGARCH and IEEE, San Jose, CA.

Received September 2001; revised January 2003; accepted March 2003