QoS-based Distributed Query Processing

Haiwei Ye*[1] — Brigitte Kerhervé* — Gregor V. Bochmann**

** Département d'informatique*
Université du Québec à Montréal, CP 8888, succ. Centre-ville
Montréal Québec, Canada H3C 3P8
ye@iro.umontreal.ca ; Kerherve.Brigitte@uqam.ca

*** School of Information Technology & Engineering*
University of Ottawa, P.O. Box 450, Stn A
Ottawa Ontario, Canada K1N 6N5
bochmann@site.uottawa.ca

ABSTRACT. *Among the essential functionalities supported by distributed multimedia systems, quality of service (QoS) is of prime interest and requires the involvement of different system components. This function aims to control and guarantee the level of quality that the system is able to offer to the user. The QoS requirements may concern system performance, the quality of the information, as well as the costs of the service provision. In this paper, we propose a general framework for integrating QoS requirements into a distributed query processing environment. This framework is based on user classes, cost models, utility functions, and policy-based management. We explain how we push QoS requirements and information into the different steps of global query optimization. We present the prototype we have developed as well as the experimentation we have conducted to validate our approach.*

RÉSUMÉ. *Parmi les fonctionnalités essentielles offertes par les systèmes multimédias distribués, la qualité de service (QoS) est de première importance et requiert l'implication de différents composants du système. Cette fonction vise à contrôler et garantir le niveau de qualité que le système est capable d'offrir à l'utilisateur. Les besoins en qualité de service peuvent concerner la performance du système, la qualité de l'information comme les coûts de fourniture du service. Dans cet article, nous proposons un cadre général permettant l'intégration des besoins en qualité de service dans un environnement de traitement des requêtes distribuées. Ce cadre est fondé sur des classes d'utilisateurs, des modèles de coûts, des fonctions d'utilité et des règles d'accès. Nous expliquons comment nous répercutons les besoins en qualité de service et l'information dans les différentes étapes de l'optimisation de requêtes globales. Nous présentons le prototype que nous avons développé ainsi que l'expérimentation que nous avons menée pour valider notre approche.*

KEYWORDS: *quality of service (QoS), distributed query processing, cost model, global query optimization.*

MOTS-CLÉS : *qualité de service, traitement de requêtes distribuées, modèle de coût, optimisation de requêtes globales.*

1. Currently working at IBM Toronto Lab. DB2 Universal Database Development.
IBM Toronto Lab, 8200 Warden Avenue, Markham, ON L6G 1C7, haiweiye@ca.ibm.com

1. Introduction

Quality of Service (QoS) management has attracted a lot of research in the last decade, mainly in the fields of telecommunication networks and multimedia systems. To support QoS activities, mechanisms have been provided mainly for individual component such as operating systems, transport systems, or multimedia storage servers and integrated into QoS architectures for end-to-end QoS provision (Aurrecoechea *et al.*, 1998). None of these proposals take database systems into consideration. However, database systems are an important component of today's distributed systems. The QoS support in a distributed system requires all components to be QoS aware, the database systems should not be an exception.

Most of today's applications need to collect information from many different data sources, to access different types of network connections, and to address various user's requirements (Braumandl *et al.*, 2003). Both various user expectations and dynamic system features enhance the need for QoS in the database server. As a result, multiple QoS dimensions need to be "plugged in" a distributed query processing. Traditional database optimizer aims at minimizing query response time or disk I/O. However, the consideration of QoS means the inclusion of other dimensions such as the cost of the query, the data quality of the query, and the throughput of the database systems. Single optimization goal deployed in the traditional database optimizer cannot satisfy these new QoS dimensions. We argue that query optimization should take into account user-defined quality of service constraints (Ye *et al.*, 1999, 2003a, 2003c).

Let's consider an example. Suppose a user in Singapore wishes to access the IBM share price. This may be available in the New York Stock Exchange database (NYSEdb), it may also be available from the London Stock Exchange database (LSEdb), and alternatively, from Singapore Exchange database (SGXdb). The updates to these databases are different: the NYSEdb may be reliable to the previous second; the LSEdb may have 15 minutes delay as compared with New York Stock Exchange; the SGXdb may only provide the closing price of the previous day. Accordingly, depending on how up-to-date the user wants the stock price (and of course with different service charges associated with different update statuses of the information), the user may choose from different databases to retrieve the data.

From this example we can identify at least four QoS requirements: up-to-date status of the information, service charge, the response time of the query, and the availability of the database server. Based on the specified QoS requirements and using the QoS metadata, the query optimizer has to choose the pertinent database server that can satisfy the user's requirements. This requires (1) the description of the quality of the data and the data sources using the metadata; (2) the identification of the optimization criteria in the presence of tradeoffs (e.g. response time versus data quality) and (3) the processing of QoS-based query processing which should be transparent to the user.

The treatment of QoS requirements in our approach is reflected in the aspects of integrating multiple optimization goals and how to select a query access plan that is overall optimal. The related issues consist in identifying the possible optimization goal, the way to obtain the user's priority between different optimization goals, and how to achieve an overall optimal goal according to user's preference.

In this paper, we revisit distributed query processing. We propose an approach to integrate user-defined QoS requirements into a distributed query processing environment. We also consider the dynamic properties of the involved system components. We then propose a query optimization strategy in which multiple goals may be considered with the support of several cost models. We use utility functions and weighting factors to capture user satisfaction. Furthermore, we discuss some experimental results confirming the effectiveness of our approach.

The rest of this paper is organized as follows. Section 2 gives some related work on the integration of different data sources and points out the QoS dimension integrated in their systems. Section 3 presents the general framework for our QoS-based query processing. By analyzing the process of query processing, we propose to plug in the QoS features into different steps. Query optimization is our focus. The concept of utility function is also introduced in this section. Section 4 provides details of our QoS-based cost models. Section 5 covers three major algorithms proposed for global query processing: global query decomposition, algorithm for join ordering, and algorithm for join site selection. Section 6 describes the prototype implemented. The treatment of different user classes in this prototype is presented. We also show that different QoS requirements do affect the query access plan and therefore provide different performance. Section 7 provides the results of the experiments we conducted. The purpose of our experiments was to validate the effectiveness of our approach. Finally, section 8 presents the summary and conclusions of our work, and presents future research directions.

2. Related work

In the last decade, several approaches have been proposed for query processing over different data sources. They can be classified into two categories: 1) strategies for providing universal access over multiple information sources, and 2) dynamic and adaptive query optimization strategies. Proposals for the first category are based on mediator architectures, where different data sources are described and integrated. Different query capabilities are taken into account during the query optimization. Such approaches are deployed in Garlic (Hass *et al.*, 1997), IRO-DB (Gardarin *et al.*, 1996) and Mariposa (Stonebraker *et al.*, 1996). The query optimizer implemented in Garlic uses enumeration rules for describing query capabilities and uses dynamic programming to find a good plan. The QoS dimension considered in the Garlic project is mainly response time. IRO-DB provides federation of object-oriented and relational database systems through the ODMG model and the OQL query language. The global query processor uses services of local cost tuners and their corresponding calibrating procedure to derive the local cost parameters. In addition to the response time, the IRO-DB also considers the load of the database server in their query processing. The originality of the approach proposed in Mariposa is its economic model in the query optimization phase. The bidding mechanism allows sites to observe their environment from query to query, and autonomously restate their costs of operation for subsequent queries. Mariposa introduces a different QoS dimension: service charge, which is not considered in other projects.

The approaches proposed in the second category generally provide adaptation techniques (Ives *et al.*, 1999, Hellerstein *et al.*, 2000) and dynamic query processing (Cole *et al.*, 1994, Urhan *et al.*, 1998). They mainly address the system dynamics. For example, the author in (Urhan *et al.*, 1998) addressed the issue of minimizing response time in the context of wide-area data access, and they provided techniques for dealing with delays in data processing and transfer to remote sites. However, they did not address the issue of taking into account other user wishes concerning quality during the processing of the queries.

In our work, we propose to use QoS monitoring tools to push dynamic properties of the systems into global query optimization. The novelty of our approach lies in the fact that we take the user's QoS requirements and the system policies into consideration to support several optimization goals.

3. A general framework for QoS-based query processing

In order to address the dynamically changing requirements of the user and the unpredictable performance of the underlying systems, we propose the integration of QoS within distributed query processing. Specifically, we are guided by two main goals when designing the QoS-based query processor: 1) recognition of individual user requirements, and 2) consideration of the dynamic nature of the underlying system.

3.1. *A big picture of the framework*

A logical architecture is proposed to show the relationships between QoS management and query processing, as illustrated in figure 1. In this framework, we include the typical components implemented in a query processor: *Parser and Rewriter*, *Optimizer*, and *Scheduler* (Hasan *et al.*, 1996; Ozsu *et al.*, 1999; Silberschatz *et al.*, 1997). The user's query is sent to the *Parser* to be syntactically analyzed and validated against the database schema (such as the tables and attributes that really exist in the database). The output of the parser is transformed by a set of rewriting rules in the *rewriter*. These rewriting rules are usually heuristics aiding in the transformation of the query into a semantically equivalent form that may be processed more efficiently. Then, the internal query representation is passed to the optimizer. The final generated plan is sent to the scheduler for execution. We mainly work on the optimizer to integrate the QoS factors.

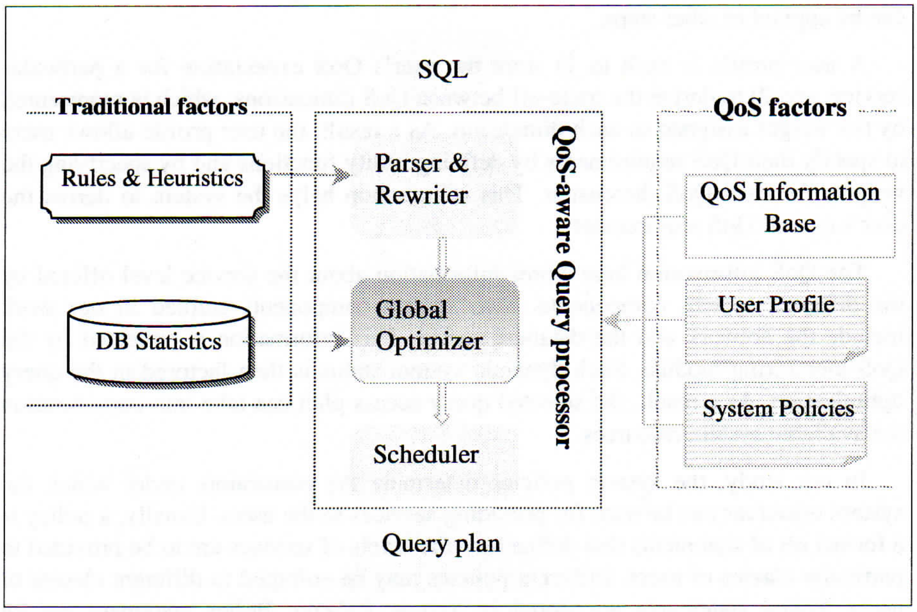


Figure 1. *A big picture of QoS-aware query processing*

As can be seen from figure 1, we keep the factors traditionally considered in the query processor (Dunkel *et al.*, 1999; Ozsu *et al.*, 1999; Kossmann, 2000), which include table statistics (such as the number of rows in a table), column statistics (such as the number of distinct values of a column), and index statistics (such as the number of leaf pages). In addition, we inject those factors called "QoS factors" in the query processor, which are information from the *QoS Information Base* (QoSIB), the *user profile,* and *system policies.* The following section elaborates these QoS factors.

3.2. *Global optimizer and QoS factors*

The main tasks of the *global optimizer* are 1) to choose an execution plan which satisfies the optimization objectives, and 2) to send it to the scheduler which coordinates the execution of the plan among the participating component DBMSs. Parallelization techniques can be applied here because concurrently executing a number of subqueries may help reducing response time, provided that it does not eventually lead to costly data transfers or context mediations. What makes our global optimizer different from the traditional query optimizer is that we build those QoS factors into the optimizer, in addition to the traditional factors. Although our focus is to push these QoS factors into the optimization phase, the similar treatment can be applied to other steps.

A user profile is built to 1) store the user's QoS expectation for a particular service, and 2) to derive the trade-off between QoS dimensions, which is represented by the weight assigned to each dimension. As a result, the user profile allows users to specify their QoS requirements by defining utility functions and by specifying the weights for each QoS dimension. This information helps the system to derive the user's overall QoS requirement.

The QoS information base stores information about the service level offered by the different system components. Two system components studied in our work include the network and the database server. This information is collected by the QoS monitoring module. Such dynamic system status is then factored in the query optimization. As a result, the selected query access plan can take into consideration the available system resources.

In our study, the system policies determine the constraints under which the system resources can be used for providing services to the users. Usually, a policy is a formal set of statements that define how the levels of services are to be provided to particular classes of users. Different policies may be enforced to different classes of users. Policy statements are stored in *System Policies.* Policy statement can be expressed in natural language such as "Give the VIP users all the authority to access all the system resources, while the normal users can only access a limited number of system resources.". The user class is usually defined in the system policies. The classification of users depends largely on the application and the business model used. In the case of web-commerce application, some example factors include the user's access pattern of the system, the user's importance and urgent degree of the requested information, and of course the service fee that the user is willing to pay for a certain level of service.

In summary, adding QoS factors into a distributed query processing environment has several impacts and requires:

– the provision of new optimization goals;

– the modification and proposition of the corresponding cost models; and

– the proposition of new algorithms for query optimization.

We discuss the possible optimization goals in the next section. The cost models and algorithms are discussed in the two subsequent sections.

3.3. *Different optimization goals*

Different user's expectations should eventually be reflected in different optimization goals (or objectives). In order to study the impact of optimization goals on query processing, the first step is to identify the possible optimization goals. The difference in optimization goals depends on various applications. It is not our intention to consider a complete set of goals. Instead, one of the major contributions of our study is to offer the possibility to express and integrate multiple optimization goals in the process of query optimization.

Optimization goals are defined by specifying how the query execution is measured. The measure of the goal attainment is given by an *objective function*. An objective function is a mathematical expression, which could be linear, that shows the relationship between the decision variables and a single goal (or objective) under consideration. Examples of such goals are total profit, total cost, response time, share of the market, and the like.

Conventional distributed/parallel database query optimization was primarily aimed at either minimizing the response time or system resource utilization. However, in the context of today's pervasive applications, such as web-commerce, this provision of a single optimization goal is not adequate. Therefore, other possible optimization goals should be proposed and integrated into the optimizer. Table 1 lists some optimization goals that are useful for our study. They are grouped into different categories. Some optimization goals are *performance oriented*, for example, the response time, and the throughput of the database system. Others are *money oriented* such as the service charge for a particular service.

Table 1. *Optimization goals and their category*

Optimization category	Optimization goal
Performance oriented	- Minimize response time - Maximize DB throughput
Money oriented	- Minimize the cost of a service - Maximize the benefit of the database system
Data quality	- Multimedia vs. Plain text - Recency of data
System oriented	- Minimize resource utilization

The example of service charge given above includes the monetary cost arising, for example, from (1) the cost of transferring data over the network, (2) the processing of data by a component database server, and (3) a combination of both. The charge from network access is easily acceptable and understandable. The pricing for the network, especially for internet, has been studied extensively in the literature (Cocchi *et al.*, 1993) and deployed by most internet services providers[2]. However, little attention was given to the study of cost incurred by local processing and this is less noticeable to the end user. Only a few papers (Bodorik *et al.*, 1988; Spiliopoulou, 1996) mentioned that the cost to access the local component database server (in a multidatabase architecture) may be charged. In our study, we believe that with the increasing need for information integration, the local processing charge also deserves careful research.

In the category of *data quality*, the optimization goal could be the representation of the query results (e.g. multimedia or plain text) or the extent to which the data is up-to-date. If the optimization goal is representation aware, some database rewriting techniques must be applied to ensure that the optimizer realizes where to retrieve which type of data. The up-to-date status of the data can be regarded as the recency of the data; the different versions of the same data are mainly caused by the charge for update frequency. The stock market example given in the introduction illustrates the situation of *recency* versus *cost.*

Most of these criteria introduced thus far are user oriented, because the majority is perceived by the user. The last category in table 1 is *system oriented.* That is, emphasis is placed on the effective and efficient utilization of the system resources. This is certainly a worthwhile measure of system performance and one that may be maximized most of the time. However, it focuses on system performance rather than the service provided to the user. Thus, it is of concern to a system administrator, but not to the user population. In this category, the system always wants to minimize system resource utilization or maximize the system capacity (in terms of number of users the database system can handle).

One thing that should be noted for the optimization goals, listed in table 1, is that they are interdependent, and sometimes it is impossible to optimize all of them simultaneously. For example, providing good response time may require reducing data transmission (especially for multimedia data) over the internet by using a high compression rate algorithm. This may lead to the quality degradation of a video frame. Therefore, the design of the optimizer should also address the compromise among competing requirements.

3.4. *Utility functions and weighting factors*

When various optimization goals exist along multiple QoS dimensions, we should find an *optimal* solution that satisfies all of them, optimal either from the user perspective or the system perspective, or both. One way of combining various optimization objectives is to use *weighted combination* (for example, a weighted

2. Currently, the price offered by ISP is usually not time-dependent, but only depends on the access line capacity.

sum) of different goals. In order to solve the different measures in different goals, the concept of utility functions is introduced in section 3.4.1. The weight assigned to each goal is calculated by using the method given in section 3.4.2. Section 3.4.3 gives an example of how the utility function and the derived weighting factors help for the evaluation of different query access plans.

3.4.1. *Utility functions*

The satisfaction for each optimization goal or QoS metric can be captured by using a *utility* function. By indicating different utility functions, the user expresses his/her individual tastes. Usually the utility function maps the value of one QoS dimension to a real number, which corresponds to a satisfaction level. For example, the following formulas give the utility functions for the response time and the service charge:

$$u_t(t) = 1/e^{t},\ u_\$(x) = 1/e^{ax+b}$$

where t is the response time for a query access plan and x is the corresponding service charge for that plan. These utility functions indicate that the user's satisfaction decreases when the response time or the cost increases.

The calculation of those coefficients (*a* and *b*) is determined by the range of the QoS dimension to a particular application. For example, if we know that the service charge is less than 1 dollar, and most of the service charge falls between 0.05 dollar and 0.9 dollar, also assume that u(0.05) = 0.9 and u(0.9) = 0.1, then the coefficients for the service charge utility function can be derived, *i.e.* a = 2.585 and b = -0.1053.

Utility functions are used in our cost model to achieve an overall optimization since they are used to compare the quality of the access plans. Utility functions also provide an important link between the quality of a query plan and the user satisfaction.

3.4.2. *Deriving weights*

The above example creates an interesting issue regarding how weights should be assigned. It is less desired for a user to specify the exact weight for each optimization goal or QoS dimension. In our approach, the Analytic Hierarchy Process (AHP) (Saaty, 1992) is used to derive the weights from user's preference. This method only requires the user to provide his or her judgment about the relative importance of each criterion over another one (pairwise comparison of goals) and then specify a preference index. Based on these preference indexes, the output of the AHP is a prioritized ranking indicating the overall weights for each of the alternative decisions.

The AHP approach is composed of four main steps: development of a goal hierarchy, pairwise comparison of goals, consistency check of the comparisons, and aggregation of the comparisons. The first step in the AHP is to develop a graphical representation of the problem in terms of the overall goal, the criteria, and the decision alternatives. Such a graph depicts the hierarchy for the problem. Figure 2 shows the hierarchy for the query plan selection problem.

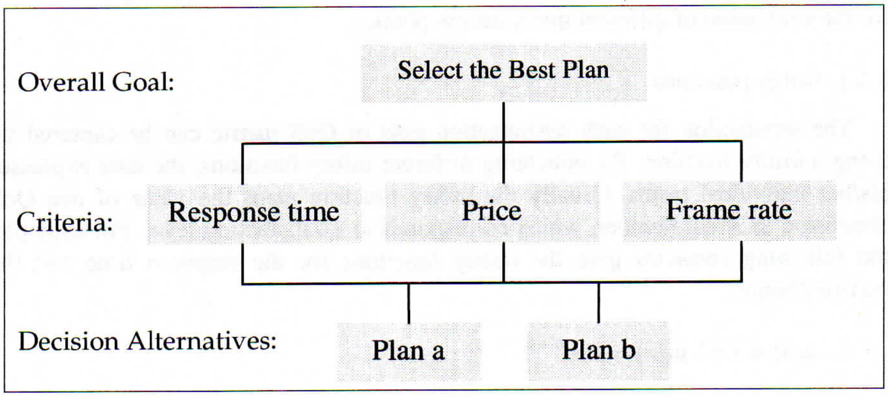


Figure 2. *AHP Hierarchy for the plan selection*

Pairwise comparisons are the fundamental building blocks of the AHP. This step requires the user to specify how important each criterion is compared to each of the other criteria. If *n* criteria are used in the plan selection, the user needs to specify *n(n-1)/2* preferences. The AHP employs an underlying scale with values from 1 (equally preferred) to 9 (extremely preferred) to rate the relative preferences for two items. For example, if the user is asked to state his or her preference for *response time* compared to *price*, he or she indicates that price was *moderately* more important than response time. The corresponding score is 3.

All the pairwise comparisons are recorded in a matrix. Using the AHP nine-point numerical rating scale (please refer to (Saaty, 1992), a value of 3 is recorded to show the higher importance of the price criterion (as compared to response time). As a result, a value of 3 is entered in the (3,1) position of the matrix shown in figure 3a. Figure 3 shows two comparison matrices for two users. In the example, the frame rate is simplified as "rate" and response time as "time".

The next step is the consistency check, which basically gives hints on which comparisons are transitively inconsistent. That is, if A is twice as desirable as B, and B is twice as desirable as C, then to be consistent A must be four times desirable as C. To handle the consistency question, the AHP provides a method for measuring the degree of consistency among the pairwise judgments. The concept of *consistency ratio (CR)* is designed in such a way that the values of the ratio exceeding 0.10 are indications of inconsistent judgment (Saaty, 1992). The CRs for User 1's and User 2's preference matrices are 0.028 and 0.075, respectively. The process continues since the values of CRs are less than 0.10 and are considered to indicate a reasonable level of consistency in the pairwise comparisons. A detailed procedure of the consistency check can be found in (Saaty, 1992).

$$\begin{array}{c|ccc} & \text{Time} & \text{Price} & \text{Rate} \\ \hline \text{Time} & 1 & 6 & 7 \\ \text{Price} & 1/6 & 1 & 2 \\ \text{Rate} & 1/7 & 1/2 & 1 \end{array} \Longrightarrow \begin{array}{c|c} & \text{weights} \\ \hline \text{Time} & 0.76 \\ \text{Price} & 0.15 \\ \text{Rate} & 0.09 \end{array}$$

(a) User 1's preference

$$\begin{array}{c|ccc} & \text{Time} & \text{Price} & \text{Rate} \\ \hline \text{Time} & 1 & 1/6 & 2 \\ \text{Price} & 6 & 1 & 5 \\ \text{Rate} & 1/2 & 1/2 & 1 \end{array} \Longrightarrow \begin{array}{c|c} & \textit{weights} \\ \hline \text{Time} & 0.17 \\ \text{Price} & 0.72 \\ \text{Rate} & 0.19 \end{array}$$

(b) User 2's preference

Figure 3. *Two pairwise comparison matrices*

The aggregation step is a mathematical procedure, which involves the computation of eigenvalues and eigenvectors. The following three-step procedure can be used for a good approximation of this step.

Step 1: sum the values in each column of the pairwise comparison matrix.

Step 2: divide each value in each column of the pairwise comparison matrix by the corresponding column sum. The resulting matrix is referred to as the *normalized pairwise comparison matrix.*

Step 3: average the values in each row of the normalized matrix.

The final AHP ranking for each user is provided in figure 3, where we give the weights associated to the three criteria for user 1 and user 2.

3.4.3. *An example of using weights and utility functions*

In order to illustrate how the utility function and weights are used in our work, we give an example. Assume we are interested in three QoS dimensions: response time, price and frame rate. Suppose that the query optimizer compares two plans, with the related information given in table 2. The weighting factors for the two users are derived in figure 3.

Table 2. *The QoS information in the example*

	Plan a	Plan b	User1's weight	User2's weight
Response time	0.2s	0.1s	0.76	0.17
Price	\$0.2	\$0.3	0.15	0.72
Frame rate	20fps	25fps	0.09	0.11

Also assume that the utility functions for the response time and frame rate are decreasing, such as those defined in section 2.1, that is $u_t\ (t) = 1/e^{\ t}$ and $u_\$\ (x) = u_\$\ (x) = 1/e^{\ ax + b}$. The utility function for frame rate should be increased and the frame rate should range from 0 fps to 30 fps. It is $u_{rate}\ (fps) = 1 - e^{(a*fps + b)}$, assume that $u_{rate}\ (5) = 0.5$ and $u_{rate}\ (20) = 0.95$. Therefore, $a = -0.1535$, $b = 0.0744$.

In our approach, weighed sums of utilities are used to achieve the overall optimization. For each user, the optimizer selects the maximum utility values between two plans. For User 1, the overall utilities for Plan *a* and *b* are 0.79 and 0.83, respectively. Accordingly, Plan *b* will be chosen for User 1 since it has higher utility. Similarly, Plan *a* is optimal from User 2's perspective. From this example, we can see that corresponding to different user's QoS requirements/preferences, the optimizer should be able to choose different plans.

4. Cost models

We propose a new approach to the problem of evaluating the cost of a query plan in a multidatabase system. Our approach relies on QoS monitoring to provide dynamic system status and on user profiles. The novelty of our approach lies in the consideration of user requirements, user classes as well as the way to deal with dynamic network performance. In our work, three levels of cost models are used. The first level is the global cost model, which is used to calculate the overall utility of a query access plan. The second level is used to calculate the cost for each node in a query access plan. The last level is the local cost model, which is used to estimate locally the cost of an operator.

4.1. *Global cost model*

Global cost models are the essential parts for the global query optimizer. Each access plan is associated with several cost measures, such as response time, service charge, and the server load. The global cost model is used to calculate the overall cost of an access plan. As mentioned earlier, utility is used to unify all the cost measures. So what we really compare between different query access plans is the total utility. The plan that has the maximum utility is the winner. The global cost model is defined as follows:

$$\underset{p=1}{\overset{m}{Max}} \; \{ \sum_{i=1}^{n} \omega_i \cdot u_i(C_{i,p}) \}$$

where $C_{i,p}$ is the cost for *ith* QoS dimension of the plan p; $u_i()$ is the utility function for cost component $C_{i,p}$; ω_i is the weighting factor assigned to the cost component $C_{i,p}$, where $0 \leq \omega_i \leq 1$ and the sum of ω_i equals to 1.

4.2. *Plan cost model*

A query access plan is represented by a binary tree. Each internal node is an inter-site binary operation (such as join or union) and each leaf node is the subquery executed at one database server. Since we consider several cost components, the cost of each node is also expressed according to multiple dimensions. For example, if we select the response time, the service charge, and the availability as our cost components, then the cost information recorded in each node will include three parts: time, dollar, and availability. The cost information for leaf nodes is based on the local cost model and the QoS Information Base (*e.g.* availability). The cost information for the internal node is calculated as a combination of the cost information of its left and right child nodes. The cost formula for each QoS dimension is different.

Table 3 lists the cost functions for response time, dollar, and availability. We use join operation as our study focus. The join time (time to perform a join at a particular site) for each node is determined by the load of the server and the current TCP performance. The formula for each join is:

$$T_{join} = \text{local (site, query)} + \text{net}(\text{site}_i, \text{site}_j)$$

where local (site, query) represents the local execution time for the query at site, net (site_i, site_j) represents the data transmission time spent over the network. The calculation of each time depends on the traditional factors and the QoS factors as we introduced in section 3. Traditional factors include the statistics from the database catalog such as the cardinality, the selectivity, the size of each column and so on. The QoS parameters used to decide the local cost include the availability and the load of the database server. The QoS information used to calculate the network cost includes the available bandwidth and delay. As mentioned before, the QoS monitor is used to capture this dynamic information. The optimizer will then consult this information and plug it into the cost models.

Table 3. *Cost functions for each cost component*

Cost component	**Cost function**	**Brief description**
Response time	Join-time + max (left.respose_time, right.response_time)	The join time is the response time to perform the join between the left and the right child.
Service Charge	Join-charge + left.charge + right.charge	The join charge is the money cost to perform the join between the left and the right child.
Availability	Left.availability * right.availability	The probability that both servers are available.

4.3. *Local cost model*

The discussions in the previous subsection are based on the assumptions that the cost information for a local component DBMS is available. However, in a multidatabase environment, the local database systems usually do not expose the needed statistical information to the global level. Furthermore, cost formulas for processing an operator (*e.g.* selection, or join) vary radically depending on the implementation of the underlying (local) database system. This requires derivation or guess-work on the part of the local cost model. In this section, we illustrate how such information is obtained.

To derive the local cost model, the sampling method proposed in (Zhu *et al.*, 1998) is applied with a modification for considering server load. The idea of the query sampling method can be characterized by the following steps:

1) Classify queries ;

2) Draw a sample of queries from each class ;

3) Perform sample queries on the DBMS ;

4) Derive a cost formula for each class using multiple linear regression.

The objective of classification of queries is to group queries into homogeneous classes so that the costs of queries in each query class can be estimated by the same formula. The factors considered for classifying queries are the type of queries (unary or join), characteristics of operand tables (such as cardinality), indexed columns and characteristics of the underlying DBMS (such as supported access methods). In our work, one further factor is added: server load for the DBMS. As join operations are of the main interest (one of the assumptions indicated previously) and the join predicate is not considered, the two main criteria used for the classification of the queries are the server load and the number of joins.

After queries are classified, a cost estimation formula needs to be derived for each query class based on the observed costs of sample queries. Such a cost formula includes a set of variables that affect the costs of queries and a number of coefficients that reflect the performance behavior of the underlying DBMS. In order to estimate the coefficients of the cost formulas, the statistical procedure is applied.

4.4. *Statistical procedure to derive the local cost models*

Regression models are used to estimate the value of a variable (called *dependent variable*) as a function of several other variables (known as *independent variables*). In our problem, the dependent variable is the response time of a query. The independent variables may be the cardinality of each table involved in the query and the result cardinality.

The statistical relationship established between the cost and the independent variables can take many forms. The most commonly used relationship assumes that the dependent variable is a linear function of the independent variables. In order to decide whether our local cost model can be explained by a linear relationship, statistical tools are needed.

The residual plots may help to observe the relationship between the independent variable (X) and the dependent variable (Y). The residual is also called the error. A residual plot is the scatter plot of residuals against the explanatory variable. According to (Moore *et al.*, 1996), if each of the residual plots shows scatter in a horizontal band with no values too far from the band and no pattern such as curvature or increasing spread, it suggests that the relationship between Y and X is linear.

The residual plots were completed in our study for all the cases in the local cost model forecasting. According to the experiment results (Appendix 4 in Ye, 2003), all the plots display no patterns. Therefore, a linear relation is reasonable. This indicates that the linear regression may be applied to establish a statistical relationship between the response time of queries and the relevant independent variables.

The next question is which independent variables should be included in the linear regression model. As identified at the beginning, two types of independent variables are possible in this case: one is the cardinality of the table; the other is the cardinality of the result. Remember that a Cartesian product is assumed; therefore the cardinality of the result is directly related to the cardinalities of the tables involved in the query. This reveals that both types cannot be included in the regression model since one criterion in selecting the independent variable is that they must be independent of each other.

As a result, either the table cardinality or the result cardinality in the regression model should be included. That is, a choice must be made between two of the following cost models:

The cost model includes the table cardinality:

$$\text{Cost} = C_0 + \sum_{i=1}^{n} C_i \times N_i \qquad [1]$$

The cost model includes the result cardinality:

$$\text{Cost} = C_0 + C_1 \times RN \qquad [2]$$

where *Ci* is the coefficient to be derived, *Ni* is the table cardinality, and *RN* is the result cardinality[3].

5. Algorithm for global query optimization

Global query optimization is generally implemented by three steps (Dayal 1985; Meng *et al.*, 1995; Ozsu *et al.*, 1999). After parsing, a global query is first decomposed into query units (subqueries) such that the data needed by each subquery is available from a single local database. Second, an optimized query plan is generated based on the decomposition results. Finally, each subquery of the query plan is dispatched to the related local database server to be executed and the result for each subquery is collected to compute the final answer.

In our study, the focus is placed on the first two steps and we map them to the problems of global query decomposition, inter-site join ordering and join site selection (Cornell *et al.*, 1989; Du *et al.*, 1995; Evrendilek *et al.*, 1997; Urhan *et al.*, 1998). Therefore, the objective of our approach is to enhance the three steps of distributed query processing with pushing QoS information into each of them. In this section, we present the details of the different steps we propose for global query optimization. In addition, we also explain how QoS information is considered.

This section provides a general idea of the algorithm. First, we identify the QoS parameters integrated into the algorithm. Then, an example is given to show how the global query is processed step by step. After that, a summary of these steps for global query processing is presented. Last, section 5.4 presents the treatment of join site selection. The reason that we want to give a special treatment of join site selection is that it is usually overly simplified in the traditional query processing. We

3. Instinct suggests that the result cardinality will explain better the relationship. To verify this point, the regression model was run for both of the formulas given above and the R-Square was for each of them. The results for Formula 1 are given in Appendix 3 of (Ye, 2003). The results for Formula 2 are given in section 6.5.4 of (Ye, 2003). As can be seen from these two results, the Formula 2 has higher R-Square and therefore is adopted for the local cost model. This confirms our intuition.

have shown in (Ye, 2003b) that by considering other data transmission options, we could see a big difference in performance.

5.1. *QoS information integrated in the algorithm*

To be QoS-centric, one of the feasible solutions is to study how to integrate QoS information into the three steps discussed above. Table 4 identifies the related QoS parameters that could be factored into each step.

Table 4. *QoS information relevant to global query optimization*

Global query optimization	**Relevant QoS parameters**
Decomposition	Server availability Server load
Inter-site join ordering	Server load TCP throughput TCP delay
Join site selection	Server load TCP throughput TCP delay

5.2. *Step-wise illustration of the algorithm*

The idea is illustrated using an example, as given in figure 4. The original query, given in table 5, is represented by a list of tables to be joined, a list of join predicates, and the location information.

Table 5. *The original query of the example*

Tables:	A, B, C, D, E, F
Join Predicates:	A.a = B.a, B.b = C.b, C.c = D.c, D.d = E.d, E.e = F.e
Locations:	Site1: A, B, Site2: C, D, Site3: E , Site4: F

The main task of global query decomposition is to break down a global query into several subqueries such that the tables involved in each subquery are located in one site. One such example of decomposition is shown in figure 4a. Figure 4b gives the result of the decomposition which contains 4 subqueries. Subquery 1 and 2 are joins performed at the local database server, while subquery 3 and 4 are the single table access (and possible some selection operations, if required in the query).

Next, in the join ordering step, the optimizer tries to develop a good ordering to combine inter-site joins between the results of the local subqueries. The join ordering can also be represented as a binary tree, where leaf nodes are the subqueries and internal nodes are inter-site join operations.

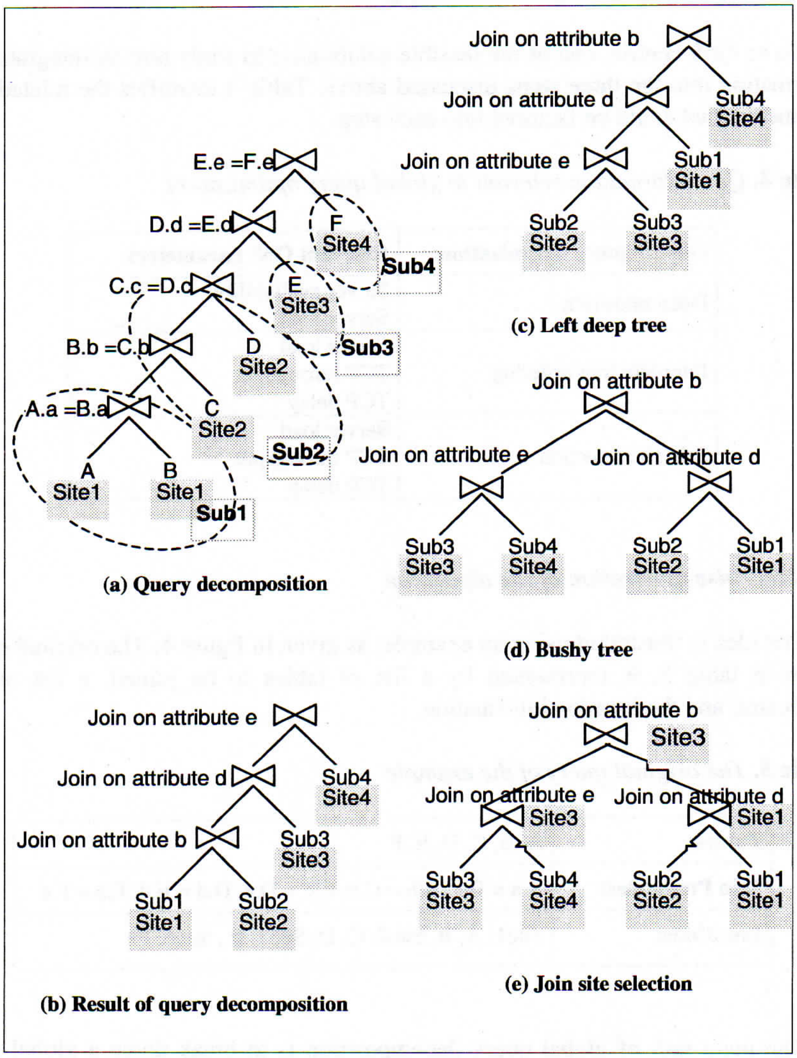


Figure 4. *Illustration of different steps for global query processing*

This phase can be further implemented by a two sub-step procedure: left deep tree generation and bushy tree generation. An example of left deep tree is shown in figure 4c. It should be noted that the join ordering of figure 4c is different from that of figure 4b. Through utilization of the distributed nature of the multidatabase system, an attempt is made to create this tree as low as possible by building the join in parallel as much as possible. This is accomplished by balancing the left deep tree to a bushy tree, as illustrated in figure 4d. The join ordering is used as an example to study the binary operation ordering. The same method can be generalized to other binary operations such as *union*.

Last is the decision of where to perform the inter-site joins – this is referred to as *join site selection*. In our approach, this step is implemented by annotating the binary tree generated from the join ordering step. Each node is annotated by the location where this join should be performed, as demonstrated in figure 4e. In this step, the information of network data transfer is also marked. Those "zigzag" lines embedded in a straight line are used for this purpose. For example, the result of subquery 4 needs to be shipped from site 4 to site 3 to perform the join. The straight lines used in this step denote that data transfer is not needed.

5.3. *Algorithm summary*

In table 6, the big picture of the algorithms is provided according to the different steps in global query processing. We cover the objective of each step, the QoS information plugged into each step, the assumptions used, and the overall idea. Due to the space limit, we only give the algorithm of the join site selection in section 5.4. For more information on the other algorithms, please refer to (Ye, 2003).

Table 6. *Algorithm summary*

	Step 1: Global query decomposition	**Step 2: Inter-site join ordering**	**Step 3: Join site selection**
Objective	To decompose a global query into subqueries that target one location	Select join ordering and join site	Assign a join site to each internal node with consideration of a large number of possible candidates sites
Input	• Global query graph (linear) • Involved tables with location information	A set of subqueries with location information	Bushy tree
Output	A set of subqueries with location information	Query plan tree (bushy tree) labelled with the location for performing each join and data transfer operations	Query plan tree labelled with the location for performing each join and data transfer operation

Algorithm idea	Consider all the possible decompositions and select the one with minimum cost	• Left linear tree generation with consideration of join site selection • Tree balancing by applying transformation rules	• A threshold is used to decide the candidate set for the third site • Post order tree traversal is used to label the location of each internal node
QoS information	• Server load • Server availability	• Communication cost • Server load	• Communication cost • Server load
Assumptions / heuristics	• Linear join graph is assumed • Subqueries are not overlapping	The inter-site joins are done on one of the component DBMSs (*i.e.* there is no central site)	Only consider the node that bottlenecks the cost of its parent node

5.4. *Join site selection*

In a complete survey (Kossmann, 2000), Kossmann summarized three site selection strategies for client-server architectures. Depending on whether to move the query to the data (execution at servers) or to move the data to the query (execution at clients), the strategy is called *query-shipping* or *data-shipping*. A *hybrid* approach of these two solutions is also possible. Traditionally (Cornell *et al.*, 1988; Selinger *et. al.*, 1980), two types of strategies were proposed: *Move-Small* and *Query-Site*. In the query-site strategy, all joins are performed at the site where the query was submitted. In the move-small strategy, for each join operation, the smaller (perhaps temporary) relation participating in the join is always sent to the site of the larger relation. Selinger and Adiba suggested another option in (Selinger *et al.*, 1980). They mentioned that for a join of two given tables at different sites, they could move both tables to a "third" site yet to be specified. However, this "third" site strategy has not been completely studied and none of the commercial database systems adopts this strategy either.

The above strategies for join site selection in a distributed environment are inadequate in the sense that they are only suitable in a static environment where network performance and load of the database server are fixed and predictable. This is obviously not the case for today's highly distributed and dynamic systems. In this paper, we address the issue of join site selection in a distributed multidatabase system deployed for e-commerce applications. The objective is to optimize the performance according to current system status. Dynamic system properties include the performance of the internet and the load of the server. The decisions are cost based. No effort has previously been made to integrate the dynamic system properties and consider a "third" candidate site.

5.4.1. *Selecting candidate site*

The key issue in site selection is to decide which site is the best (optimal) for each binary operator. The site selection process becomes complicated when several candidate sites are capable of handling the operation. In fact, the crucial question is how many candidates should be considered for the third site. There are three possible approaches to determine candidate sets:

– consider all the available sites in the system. This is simple but this will usually incur too much overhead for the optimizer;

– we can shrink the above set to all the sites involved in this particular query. By considering these sites, we may benefit from the situation where the result of this join needs to be shipped to the selected third site for further calculation. However, if the number of locations involved in the query is larger, we may have the same problem as above: too much optimization overhead;

– we can apply some heuristics to further decrease the size of the candidate set. For example, we can restrict the third site for a particular join operator to its "close relatives", such as niece/nephew sites in the join tree.

In our consideration of candidate set, we combine the option 2 and the option 3. A *threshold* for the number of sites is therefore used to describe the situation where option 3 should be used. That is,

$$\textbf{Candidate sites for a join} = \begin{cases} \text{All the sites in the tree,} & \text{if N < =}\textit{threshold} \\ \text{Close relative (children and niece),} & \text{otherwise} \end{cases}$$

where N is the total number of sites involved in the query. The value of threshold should be derived from the experiment. In the following algorithm, we use a procedure *CandidateSiteSelection*() to represent this procedure.

5.4.2. *The algorithm for join site selection*

The procedure of join site selection can be regarded as marking the site for each join node in the tree. And this process is usually done in a bottom-up fashion. Thus we can employ one of the standard bottom-up tree traversal algorithms for this purpose. In our algorithm, we use *post order* tree traversal to visit the internal nodes of the tree. We ignore the post order algorithm and only give the algorithm (SiteSelection()) to visit each node.

```
Algorithm: SiteSelection (treenode)
{
if (hasChild(treenode) == true ) {
  candidate_set[] = CandidateSiteSelection (treenode);
  for each site_s in candidate_set
        cost [s] = Cost-node (treenode, site_s); //Using the cost
        model to compute the cost if the join is performed on this site
  min_site = select-min (cost[]);
7.        treenode.join_site = min_site;
8.        if (sites is also marked as join site for treenode m)
9.          cost-node (m, site_s); //recalculate cost for node m under the new
            added load introduced by site s
10.     }   // end if
11.     }
```

The procedure of *SiteSelection*() returns nothing, it only picks up the join site based on the cost model and records the join site in the root node of the input tree. The *Cost-node*() implements the cost model, which was explained section 4.

6. Prototype implementation

In order to validate our approach, we implemented a prototype where we concentrated on those aspects that are representative for the QoS-based distributed query processing we propose. Based on the experience we learned from this prototype, we could apply it toward the construction of larger and more realistic system. For simplicity, we integrate two QoS dimensions in the prototype. However, the implementation is not limited to these two dimensions, the modules implementing other dimensions can be easily plugged into our prototype. Highlights of the implementation are given below.

User classes: In order to show the differentiated services in our prototype, we have adopted the priority-based user classification and considered two user classes, namely *VIP user* and *normal user*.

Optimization goal: For our prototype implementation, we focus on two optimization goals: minimize the response time and/or the service charge. Basically, we want to demonstrate the integration of the criteria of *time* and *money* into our prototype. Accordingly the overall optimization goal is calculated by the following formula:

$$\text{Min } \{ \omega_t \, u_t \text{ (response\_time)} + \omega_\$ \, u_\$ \text{ (service\_charge)} \}$$

where ω_t and $\omega_\$$ are the weights specified by the users for the response time and service charge, respectively; u_t and $u_\$$ are utility functions used for the response time and service charge respectively. For the purpose of simplicity, we use the utility functions $u_t(t) = 1/t$ and $u_\$ (x) = 1/x$ for the response time and the cost, respectively.

Global cost models: The general cost model contains two cost components: response time and service charge. Depending on the optimization goals, three cost models can be selected:

C_{time} = response_time;

C_{dollar} = service_charge;

$C_{overall} = W_{time} * u_t(\text{response\_time}) + W_{dollar} * u_\$(\text{service\_charge})$

The calculation of the response time is straightforward. The total response time of a query plan (represented as a tree structure) is the sum of the response time on each node along the critical path in the query access tree.

For the service charge, we are dealing with a pricing issue. Typically, two types of charging schemes are popular today. They are *flat-rate* and *usage-based* (Odlyzko, 2001). We adopt the usage-based pricing policy for our prototype implementation. We concentrate on network bandwidth utilization. A complete pricing schema, however, should consider all the resources including both the network and the server. The reason for only considering the network resource is not only because we want to simplify the implementation, but also because there have already been many studies for the pricing for the internet. We assume the service charge of a query plan is proportional to the network resource consumed. Accordingly, this second optimization goal is eventually simplified as the problem of minimizing the network bandwidth utilization.

Local cost model: In the prototype, four levels of server load (no load, low, medium, and high) are considered. Two-way, three-way and four-way joins are considered in the prototype. The cost formula is derived for each level of server load. Thus, 12 types of queries for the "sample" database were used.

Prototype architecture

The functional modules of the prototype include the *user interface* part for SQL input and QoS schema selection, the *optimization part* based on the algorithms proposed, the *visualization part* for the query plan and QoS information and the *result display* part. A big picture of the prototype is shown in figure 5. The *optimizer* takes several elements as input. All the input information is stored as XML files. Our prototype offers a simplified GUI for SQL input[4]. This component allows a user to specify a query by selecting the desired attributes and tables as well as join and restriction predicates.

The user can also choose to view the XML representation of the specified query that will be forwarded to the optimizer by clicking the "show query (XML)" button. The other component integrated with the SQL Input GUI is the User Preference manager, shown in the lower part in figure 6. In this part, the user can select his trade-off between the response time and the service charge. The sliding bars are used for this purpose and this ratio is further integrated in the optimizer to derive the overall optimization goal. To simplify the implementation, we did not use the AHP method introduced in section 3.4.2 in our prototype. Since we only deal with two QoS dimensions here, we assume that the user is able to provide the weights among two things, which does not seem to be a hard choice.

4. Please note that we did not intend to implement the whole database engine in our prototype. Thus we bypassed the parser/semantics and rewrites steps. We also assume that the user always provides us a valid (syntactically and semantically) query.

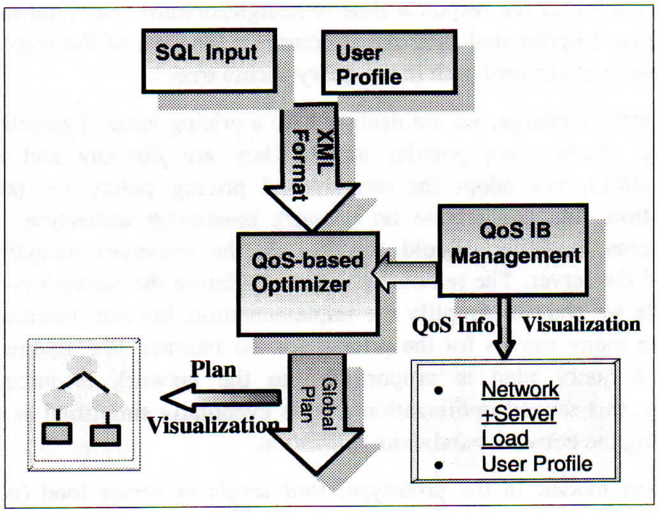
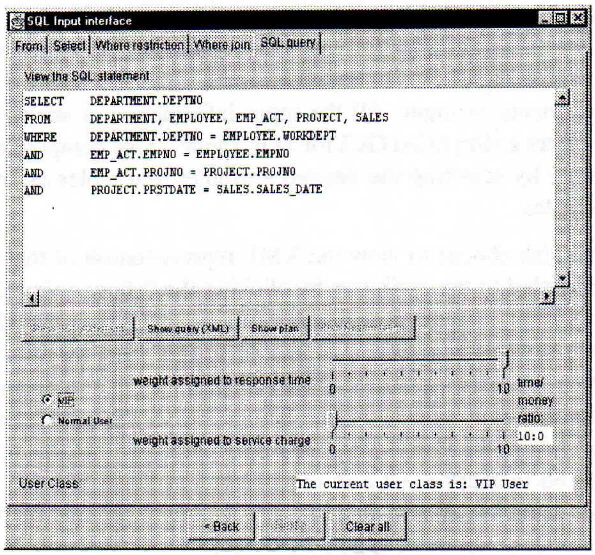


Figure 5. *Logical components of the prototype*

When an SQL query and the QoS preferences are specified by the user, he/she can see the generated query access plan. For the query specified as in figure 6, the query plan shown to the user will look like the one shown in figure 7.



View the SQL statement

```
SELECT  DEPARTMENT.DEPTNO
FROM    DEPARTMENT, EMPLOYEE, EMP_ACT, PROJECT, SALES
WHERE   DEPARTMENT.DEPTNO = EMPLOYEE.WORKDEPT
AND     EMP_ACT.EMPNO = EMPLOYEE.EMPNO
AND     EMP_ACT.PROJNO = PROJECT.PROJNO
AND     PROJECT.PRSTDATE = SALES.SALES_DATE
```

weight assigned to response time 0 10

[✓ VIP] Normal User

weight assigned to service charge 0 10

time/money ratio: 10:0

User Class: The current user class is: VIP User

Figure 6. *An example of SQL input interface and selection of query preference*

In short, through the implementation of the prototype, we have demonstrated the following points:

– different user classes are provided in the prototype. Users are classified based on priority and a system policy is made for each user class;

– two optimization goals are supported in the current prototype, according to two QoS dimensions: response time and service charge. The overall optimization goal is achieved by using the weighted sum of the resulting utility functions applied for different goals;

– different query access plans can be generated for different user classes;

– dynamic QoS conditions for systems may affect the decision. The system parameters include both the network information and server characteristics.

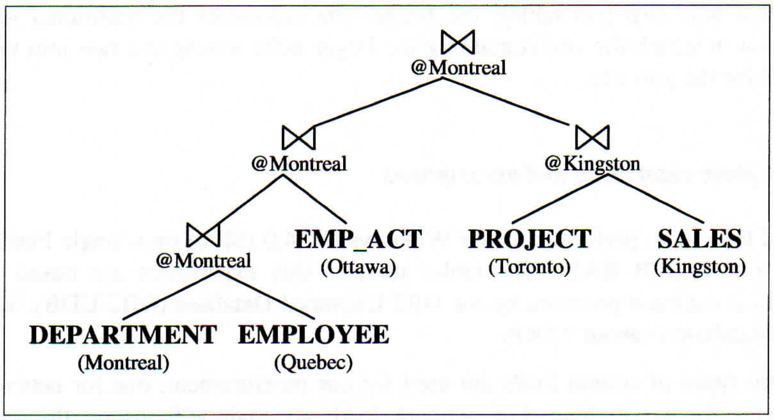


Figure 7. *An example of generated query access plan*

7. Experiment

In this section, we evaluate the performance of our QoS-based query processing strategy according to the framework proposed in the previous sections. The objective of our experiment is to show that our query optimizer can adapt itself to workload changes (both server load and network load) and always chooses the best plan for different user classes. In the experiment we simulate two classes of users: *VIP* user and *normal* user.

We have three goals for our experiments: (i) how our estimated plan cost (in terms of response time) is close to the real execution cost; (ii) what are the quality of service for VIP user and normal user under different workloads (we focus on response time in the experiment); and (iii) to design a scenario where the traditional join site selection strategy is no longer optimal. Corresponding to these goals, three sets of experiments were set up.

In the first experiment, named *estimated vs executed*, we take the query plan generated by the prototype and execute it under different server loads. The network bandwidth used for the plan estimation is 5Mbps since this is the most representative maximum bandwidth during the daytime according to our observation between University of Montreal and University of Ottawa.

The second experiment, named *VIP vs normal* is designed to measure the response times for VIP users and normal users under different server loads and network congestion levels. The 3-way join with different resulting cardinalities is used for the second experiment.

The third experiment, called *Third-site vs Larger-site*, is carried out to compare the response times for the two join site selection algorithms. The *Third-site* represents our algorithm that considers a third candidate as join site except for the sites that hold two join tables; the *Larger-site* represents the traditional join site selection in which the site containing the larger table among the two join tables is chosen for the join site.

7.1. *Experimental setup and assumptions*

All tests were performed under Windows NT 4.0 (SP 6) on a single Pentium III CPU and 192MB RAM. The tables used in this experiment are based on the SAMPLE database provided by the DB2 Universal Database (DB2 UDB). The size of the database is about 7.5KB.

Two types of system loads are used for our measurement, one for network and the other for server load. For network load, we mainly focus on the available bandwidth as the indication of network congestion level. For server load, we concentrate on the CPU utilization as the indication of server load.

Concerning the server load, in our experiments we degrade the performance of one server by loading it with additional processes. We categorized the server load into 4 levels: *no load*, *low load*, *medium load*, and *high load.*

As for the network load, we consider the TCP congestion level. In our global database schema, we assume the data are distributed among different cities in Canada. For this purpose, we observed the TCP traffic using IPERF [IPERF] between UdeM (University of Montreal) and UO (University of Ottawa). Based on the observation, we find the maximum bandwidth ranging from 0.2Mbps to 10 Mbps depending on the time of the day. We group those throughputs into 6 congestion levels of within the range between 0.1Mbps and 8Mbps (Ye 2003; Ye *et al.*, 2003a).

7.2. *Summary of experiment results*

We conducted a number of experiments and performance data are collected for the two sets of experiments identified previously. Detailed results can be found in (Ye, 2003; Ye *et al.*, 2003a). Here we explain what we got from our experimentations.

Estimated versus execution time. In the first set of experiments, *estimated vs. executed*, we first vary the workload of the server. Then under different loads, a plan is generated with an estimated time. This plan is then executed and the observed execution time is recorded for the purpose of comparison. The network congestion level for all links is *level 1* (which is equivalent to 5Mbps). From the experiments, the collected execution times are very close to the estimated response time. We also analyze the result statistically by constructing a linear regression model of these two times. The regression results indicate that the estimated times can explain about 95% of the real execution times (detailed in Ye, 2003).

VIP versus Normal users. We compare the execution times for VIP users and normal users under different server loads. Concerning the server load, we observed (Ye, 2003; Ye *et al.*, 2003a) that, under no load, all the users would get the same performance. With the increasing load, the VIP user always stays at the same curve (the same performance), while the normal user will get higher response time (the curves marked with square sign). And the advantage of performance for VIP users increases with increasing server load. In short, this set of experiments shows that the VIP users always get best performance while the normal user will suffer the slow response when the load increases.

To study the affect of the network congestion levels, we assume that the links among the nodes involved in the join are congested while other links have the normal throughput (5Mbps). In addition, there is no load of the server during the experimental periods. Again, estimated times are used for the comparison of this experiment. We observed the same trend as in the load test, whenever the links are congested to a certain level (usually at level 3, *i.e.* 1Mbps), the plan for the VIP user can choose another smooth route for data transformation and maintain the fast response time. Since doing so may incur extra data transmission, and this is regarded as "expensive" for normal users, the normal user will experience a slower query response in these cases.

Third-site versus Larger-site. In the last experiment, we evaluate our "third-site" algorithm as opposed to traditional site selection. The purpose for this experiment is to find a scenario where traditional site selection (the larger-site selection strategy) is not always optimal in terms of response time. The results shown in (Ye *et al.*, 2003b) demonstrate the superiority of our algorithm over the traditional algorithm whenever the larger site of the two-way join is loaded. In the case that the network link between the two operand sites is congested to a certain level, our algorithm can also avoid the congested link by selecting a third site as the join site.

8. Conclusion and future work

In this paper, we have proposed a general framework for integrating QoS requirements in a distributed query processing environment. This framework is based on user classes, cost models, utility functions, and policy-based management. Our approach allows us to offer differentiated services to different classes of users according to their expectations in terms of QoS. We have presented our QoS-based distributed query processing strategy where we push QoS requirements and information into the different steps of global query optimization: global query decomposition, join ordering and join site selection. We presented the prototype we have developed as well as experimentation we have conducted to validate our approach. The current prototype considers two classes of users as well as two different optimization goals. In the future, we will consider other QoS dimensions to be specified by the user, such as data quality or freshness and will work on rewriting rules to transform specifications on these dimensions into optimization goals and corresponding cost models. To test the feasibility of our method, we designed a very simple scenario. To test our algorithm in a more general case, further experiments should be conducted on a larger and real database system.

9. References

Aurrecoechea C., Campbell A., Hauw L, "A Survey of QoS Architectures", *ACM Multimedia Journal*, 6, May 1998, p. 138-151.

Bodorik P., Riordon J. S., "Distributed query processing optimization objectives", in *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, CA, IEEE Computer Society, February 1988, p. 320-329.

Braumandl R., Kemper A., Kossmann D., "Quality of Service In an Information Economy", *ACM Transactions on Internet Technology (TOIT)*, Vol. 3, No. 4, November 2003, p. 291-333

Cocchi R., Estrin D., Shenker S., Zhang L., "Pricing in Computer Networks: Motivation, Formulation, and Example", *IEEE/ACM Transactions on Networking*, Vol. 1, No. 6, December 1993, p. 614-627.

Cole R., Graefe G., "Optimization of Dynamic Query Evaluation Plans", *SIGMOD Conference*, 1994, p. 150-160.

Cornell D. W., Yu P. S., "On Optimal Site Assignment for Relations in the Distributed Database Environment", *IEEE Transactions on Software Engineering*, Vol. 15, No. 8, August 1989, p. 1004-1009.

Dayal U., "Query Processing in Multidatabase System", in W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, Springer Verlag, 1985.

DB2 UDB Administration Guide V7.2,
http://www-4.ibm.com/cgi-
bin/db2www/data/db2/udb/winos2unix/support/v7pubs.d2w/en_main.

Du W., Shan M.-C., Dayal U., "Reducing Multidatabase Query Response Time by Tree Balancing", *SIGMOD Conference*, 1995, p. 293-303.

Dunkel B., Zhu Q., Lau W., Chen S., "Multiple-Granularity Interleaving for Piggyback Query Processing", in *Proceedings of CASCON*, 1999, p. 24-39.

Evrendilek C., Dogac A., Nural S., Ozcan F., "Multidatabase Query Optimization", *Distributed and Parallel Databases*, Vol. 5, 1997, p. 77-114.

Gardarin G., Sha F., Tang Z., "Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System", *Proceedings of the 22nd VLDB*, Mumbai, India, 1996, p. 378-389.

Hasan W., Florescu D., Valduriez P., "Open Issues in Parallel Query Optimization", *SIGMOD Record*, Vol. 25, No. 3, September 1996, p. 28-33.

Hass L., Kossmann D., Wimmers E., Yang J., "Optimizing queries across diverse data sources", in *proceedings of the Conference on Very Large Data Bases* (VLDB), Greece, Aug. 1997, p. 276-285.

Hellerstein J., Franklin M., Chandrasekaran S., Deshpande A., Hildrum K., Madden S., Raman V., Shah M, "Adaptive Query Processing: Technology in Evolution", in *IEEE Bulletin on Data Engineering*, Vol. 23, No. 2, 2000, p. 7-18.

IPERF, http://dast.nlanr.net/Projects/Iperf/

Ives Z., Florescu D., Friedman M., Levy A., Weld D., An Adaptive Query Execution Engine for Data Integration, *Proc. of ACM SIGMOD Conf. on Management of Data*, 1999.

Kossmann D., "*The state of the art in distributed query processing*", ACM Computing Surveys (CSUR), Vol. 32, Issue 4, December 2000, p. 422-469.

Meng W., Yu C., "*Query Processing in Multidatabase Systems*", Modern Database Systems: The Object Model, Interoperability, and Beyond, edited by W. Kim, Addison-Wesley/ACM Press, 1995, p. 551-572.

Moore D. S., McCabe G. P., *Introduction to the Practice of Statistics*, 2nd edition, WH Freeman and Company, 1996.

Odlyzko A.M., "Internet pricing and the history of communications", *Computer Networks*, Vol. 36, 2001, p. 493-517.

Ozsu M. T., Valduriez P., *Principles of Distributed Database System*, 2nd edition, Prentice Hall, 1999.

Saaty T., Multicriteria Decision Making-The Analytic Hierarchy Process, Technical report, University of Pittsburgh, RWS Publications, 1992.

Selinger, P. G., Adiba M., "Access path selection in distributed data base management systems", in *proceedings of the International Conference on Data Bases*, 1980, p. 204-215.

Silberschatz, H.F. Korth, S. Sudarshan, *Database System Concepts*, Third Edition, McGraw-Hill, 1997.

Spiliopoulou M., Identifying the Optimization Principles of a DBMS Participating in a Multidatabase, Technical Report ISS-23, Institut für Wirtschaftsinformatik, Humboldt-Universität zu Berlin, January 1996.

Stonebraker M. *et al.*, "Mariposa: A Wide-Area Distributed Database System", *VLDB Journal*, 5, 1, January 1996, p. 48-63.

Urhan T., Franklin M.J., Amsaleg L., "Cost-based Query Scrambling for Initial Delays", *SIGMOD'98*, Vol. 27, No. 2, Seattle, June 1998, p. 130-141.

Ye H., Kerhervé B., Bochmann G. V., "QoS-aware distributed query processing", *DEXA Workshop on Query Processing in Multimedia Information Systems (QPMIDS)*, Florence, Italy, September 1-3, 1999, p. 923-927.

Ye H., Kerhervé B., Bochmann G. V., Oria V., "Pushing Quality of Service Information and Requirements into Global Query Optimization", *the Seventh International Database Engineering and Applications Symposium (IDEAS 2003)*, Hong Kong, China, July 16-18, 2003, p. 170-179.

Ye H., Kerhervé B., Bochmann G. V., "Revisiting Join Site Selection in Distributed database Systems", *International Conference on Parallel and Distributed Computing, EURO-PAR 2003*, August 26th-29th, 2003, Klagenfurt, Austria, p. 342-347.

Ye H., Kerhervé B., Bochmann G. V., (2003c) "Integrating Quality of Service into Database Systems", *14th IEEE International Conference and Workshop on Database and Expert Systems Applications (DEXA03)*, September 1-5 2003c, Prague, Czech Republic, p. 803-812.

Ye H., Integrating Quality of Service Requirements into a Distributed Query Processing Environment, Ph.D. Thesis, University of Montreal, 2003.

Zhu Q., Larson P., "Solving local cost estimation problem for global query optimization in Multidatabase systems", *Distributed and parallel Database*, Vol. 6, No. 4, 1998, p. 373-420.