

GLOBAL QUERY OPTIMIZATION BASED ON MULTISTATE COST MODELS FOR A DYNAMIC MULTIDATABASE SYSTEM*

Qiang Zhu Jaidev Haridas

*Department of Computer and Information Science
The University of Michigan - Dearborn, MI 48128, USA
Email: {qzhu,jaidev}@umich.edu*

Wen-Chi Hou

*Department of Computer Science
Southern Illinois University at Carbondale, IL 62901, USA
Email: hou@cs.siu.edu*

Keywords: Multidatabase, query optimization, multistate cost model, multiple-version plan, dynamic environment

Abstract: Global query optimization in a multidatabase system (MDBS) is a challenging issue since some local optimization information such as local cost models may not be available at the global level due to local autonomy. It becomes even more difficult when dynamic environmental factors are taken into consideration. In our previous work, a qualitative approach was suggested to build so-called multistate cost models to capture the performance behavior of a dynamic multidatabase environment. It has been shown that a multistate cost model can give a good cost estimate for a query run in any contention state in the dynamic environment. In this paper, we present a technique to perform query optimization based on multistate cost models for a dynamic MDBS. Two relevant algorithms are proposed. The first one selects a set of representative system environmental states for generating an execution plan with multiple versions for a given query at compile time, while the second one efficiently determines the best version to invoke for the query at run time. Experiments demonstrate that the proposed technique is quite promising for performing global query optimization in a dynamic MDBS. Compared with related work on dynamic query optimization, our approach has an advantage of avoiding the high overhead for modifying or re-generating an execution plan for a query based on dynamic run-time information.

1 INTRODUCTION

A multidatabase system (MDBS) integrates data from multiple local (component) databases and provides users with a uniform global view of data. A global user can issue a (global) query on an MDBS to retrieve data from multiple databases without having to know where the data is stored and how the data is retrieved. How to process such a global query efficiently is the task of global query optimization.

There are a number of challenges for global query optimization in an MDBS. They are mainly caused by heterogeneity and local autonomy of the system. One major challenge is that some necessary local optimization information such as local cost models may not be available at the global level. Several methods to derive cost models for an autonomous local database system (DBS) at the global level have been proposed in the literature, including a calibration method (Du, 1992; Gardarin, 1996), a query sampling method

(Zhu, 1994; Zhu, 1996a; Zhu, 1998), a cost vector database approach (Adali, 1996), a fuzzy approach (Zhu, 1997), and a generic model approach (Naacke, 1998; Roth, 1999). Other multidatabase query optimization issues that have been studied in the literature include: join tree parallelism (Du, 1995; Evrendilek, 1997; Subramanian, 1998), entity joins (Tsai, 1997), outerjoins (Chen, 1990; Yu, 1998), schema level optimization (Lee, 1998), semantic query optimization (Goni, 1996; Lim, 1995; Zhu, 1996b), and query modification and decomposition (Yu, 1998).

It is noted that many factors (e.g., CPU load, I/O load, and available memory space) in an MDBS may change dramatically over time. These dynamic factors make query optimization in an MDBS even more challenging. A number of researchers have studied the issues on dynamic query optimization in an MDBS recently (Amsaleg, 1996; Bouganim, 2000; Ng, 1999; Urhan, 1998). All the dynamic techniques proposed so far have a common characteristic, that is, they modify or re-generate an execution plan for a given query based on dynamic information observed at run time. One shortcoming of such an approach is that the amount of work that needs to be done to

*Research supported by the US National Science Foundation under Grant # IIS-9811980 and The University of Michigan under OVPR and UMD grants.

modify or re-generate an execution plan may be very significant, which directly affects the query response time.

We adopt an alternative approach to perform query optimization in a dynamic multidatabase environment. In our recent work (Zhu, 2000), we utilized a qualitative approach to build multistate cost models for dynamic local database systems in an MDDBS. It has been shown that a multistate cost model can give a good cost estimate for a (component) query run in any contention state at a dynamic local site in an MDDBS. In this paper, we present a technique to perform global query optimization for a dynamic MDDBS based on multistate cost models. The basic idea is to select a set of representative system environmental states for a dynamic MDDBS, generate an execution plan with multiple versions (corresponding to the representative system environmental states) for a given query at compile time, and then determine the best version to run for the query based on dynamic information at run time. Due to the limit on the number of versions allowed in an execution plan, an issue that needs to be addressed is how to choose a set of good representative system environmental states for generating versions for a query execution plan. Another issue is how to determine the best version of an execution plan for the running system environmental state at run time. Two algorithms to solve the above issues are proposed in this paper. Our simulation results demonstrate that the presented technique is quite promising in optimizing a global query in a dynamic MDDBS.

One advantage of our optimization approach is that the amount of optimization work that needs to be done at run time is minimized and in the meanwhile the variation of dynamic factors affecting query performance at run time is taken into account. This is achieved by using the multistate cost models to estimate (component) query costs in a dynamic environment and shifting significant optimization work to the compile time, which does not affect the query response time.

The rest of the paper is organized as follows. Section 2 gives an overview of multistate cost models. Section 3 discusses the potential approaches for performing query optimization based on multistate cost models in a dynamic MDDBS. Section 4 presents the details of a promising query optimization approach that employs a query execution plan with multiple versions for a dynamic multidatabase environment. We will discuss two algorithms: one is to select a set of representative system environmental states for generating multiple versions for a query execution plan at compile time, and the other is to determine the best version to invoke based on run-time information. Section 5 shows some experimental results. Section 6 summarizes the conclusions.

2 MULTISTATE COST MODEL

To incorporate the effect of dynamic factors on query performance into a cost model for an MDDBS, we introduced an effective qualitative approach recently (Zhu, 2000).

This approach considers the combined effect of all the dynamic factors on a query cost together rather than individually. Although dynamic factors may change differently in terms of changing frequency and level, they all contribute to the contention level of the underlying system environment, which represents their net effect. Notice that the cost of a query increases as the contention level. The system contention level can be divided into a number of discrete states (categories) such as “*High Contention*” (H), “*Medium Contention*” (M), “*Low Contention*” (L), and “*No Contention*” (N). A qualitative variable can be used to indicate the contention states. This qualitative variable, therefore, reflects the combined effect of the dynamic environmental factors. A cost model including such a qualitative variable can capture the dynamic factors to a certain degree.

Since, for a given query, its cost increases as the system contention level, we can use the cost of a small probing query to gage the contention level and classify the contention states for the dynamic system environment. To obtain an appropriate classification of system contention states, we first partition the range of a probing query cost in the given dynamic environment into subranges (intervals) with an equal size. Each subrange represents a contention state. If some neighbor contention states are found to have a similar effect on the derived cost model, they are merged into one state. Such a uniform partition with merging adjustment procedure for a classification of contention states has been proven to be very effective in practice (Zhu, 2000).

A qualitative variable X with M possible system contention states s_1, s_2, \dots, s_M can be represented by a set of $M - 1$ indicator (binary) variables Z_1, Z_2, \dots, Z_{M-1} . That is, $X = s_i$ ($1 \leq i \leq M - 1$) is represented by $Z_i = 1$ and $Z_j = 0$ (for any $j \neq i$); and $X = s_M$ is represented by $Z_k = 0$ (for any $1 \leq k \leq M - 1$). Including qualitative variable X in a cost model is equivalent to including indicator variables Z_1, Z_2, \dots, Z_{M-1} in the cost model.

To develop a cost model including the indicator variables, we extend the query sampling method (Zhu, 1998). More specifically, component queries that can be performed on a local DBS in an MDDBS are grouped into homogeneous classes first, based on some information available at the global level in an MDDBS such as the characteristics of queries, operand tables and the underlying local DBS. A set of sample queries are then drawn from each query class and run against the user local database in different contention

states. The observed costs of sample queries are used to derive a regression cost model with indicator variables of the following form:

$$Y = \underbrace{(B_0^0 + \sum_{j=1}^{M-1} B_j^j Z_j)}_{\text{intercepts}} + \sum_{i=1}^n \underbrace{(B_i^0 + \sum_{j=1}^{M-1} B_i^j Z_j)}_{\text{slopes}} X_i, \quad (1)$$

where Y is the query cost, X_i 's are explanatory variables (such as the operand table cardinality(ies), result table cardinality, etc), Z_j 's are the indicator variables, and B_i^j 's are the regression coefficients. The intercepts and slopes of equation (1) change from one contention state to another, indicated by the values of Z_i 's. Each query class has such a multistate cost model.

To estimate the cost of a query at run time, the query class to which the query belongs is first identified. The running system contention state is determined using the observed cost of a small probing query. The cost of the query is then estimated by using cost model (1). Studies have shown that a multistate cost model can give a good cost estimate of a query run in any contention state in a dynamic MDBS (Zhu, 2000).

To reduce the overhead of cost estimation, the running system contention state can also be determined by using an estimated cost (rather than observed cost) of a probing query Q_p . A regression equation between the probing query cost Y_{Q_p} and some major system contention parameters (such as CPU load ld_1 , I/O load io , and size of used memory space um for a dynamic environment) is built first, i.e.,

$$Y_{Q_p} = E_0 + E_1 * ld_1 + E_2 * io + E_3 * um, \quad (2)$$

where E_i ($i = 0, 1, 2, 3$) are regression coefficients. Once such an equation is in place, every time we want to determine the system contention state in which a query is executed, we need to calculate the estimated cost Y_{Q_p} of probing query Q_p by using equation (2) without actually executing Q_p . The contention state is then determined using this estimated cost. Since obtaining the parameter values (ld_1, io, um) in equation (2) usually requires much less overhead than executing a probing query, using the estimated costs of a probing query to determine system contention states is usually more efficient.

Note that different query classes may classify contention states at the same local site differently in order to obtain a good cost model specifically tuned for the underlying query class. For example, assume the cost range (i.e., the range of contention level) of a probing query at a particular local site S_1 is [0,

90], namely¹, the minimum probing cost is (approximately) 0 second and the maximum probing cost is (approximately) 90 seconds. One query class G_{11} may use three contention states: $s_1^{(11)} = [0, 30]$, $s_2^{(11)} = (30, 60]$, $s_3^{(11)} = (60, 90]$ for its cost model, while another query class G_{12} may utilize four contention states $s_1^{(12)} = [0, 15]$, $s_2^{(12)} = (15, 35]$, $s_3^{(12)} = (35, 65]$, $s_4^{(12)} = (65, 90]$ for its cost model. Note that each contention state basically represents an interval (subrange) of contention level, which is called the representing interval of the state in the following discussion. The classification of contention states for a query class in a dynamic environment is automatically determined during its cost model building (Zhu, 2000).

On the other hand, for a probing query cost Y_0 (i.e., a contention level) gaged at a local site S_i ($1 \leq i \leq N$), there is a unique corresponding contention state $s^{(ij)}(Y_0)$, i.e., the representing interval contains Y_0 , for each query class G_{ij} ($1 \leq j \leq K_i$) at the site. In other words, a unique (local) contention state vector $\vec{s}^{(i)}(Y_0) = \langle s^{(i1)}(Y_0), s^{(i2)}(Y_0), \dots, s^{(iK_i)}(Y_0) \rangle$ is determined by contention level Y_0 at site S_i . In general, when the component queries of a global query are to be performed at several sites, a (local) contention state vector can be determined by a gaged probing query cost at each site. The combination of all the (local) contention state vectors is called a (global) system environmental state in this paper, which reflects the running contention environment for the query.

3 QUERY OPTIMIZATION BASED ON MULTISTATE COST MODELS

Establishing cost models is not the ultimate goal of query optimization. The ultimate goal is to choose a good execution plan for a query on the basis of the cost estimates given by the cost models. How to make use of multistate cost models to choose a good execution plan for a given query in a dynamic environment is the issue to be studied in the rest of the paper.

3.1 Interpretation vs Compilation for Query Processing

In general, there are two approaches to processing a query. The first one is called the interpretation ap-

¹In mathematical convention, a closed end (i.e., '[' or ']') of an interval indicates that the end point is included, while an open end (i.e., '(' or ')') indicates that the end point is not included.

proach. In this approach, simple query optimization is performed on the fly while a query is being executed. This approach is suitable for ad hoc/interactive queries, which are usually executed only once. The second one is called the compilation approach. In this approach, comprehensive query optimization is performed for a given query at compile time, resulting in an execution plan. The execution plan can then be executed repeatedly at run time as needed. This approach is more suitable for stored and embedded queries, which are usually executed repeatedly. In an MDDBS environment, both stored/embedded queries and ad hoc/interactive queries are expected.

Using multistate cost models to perform query optimization in the interpretation approach is relatively easy. Since the multistate cost models give cost estimates that reflect the current running system environment, the global query optimizer can choose a good query execution plan for the current environment based on the cost estimates.

Using multistate cost models to perform query optimization in the compilation approach is more difficult. The main challenge is that it is not easy to predict the run-time system environment in which the query is to be executed when a query is optimized at compile time. Apparently, the traditional methods of using static cost models to perform query optimization are not acceptable since a system environment is dynamic rather than static. There are several potential ways to perform query optimization based on multistate cost models in the compilation approach, which are described in the following subsection.

3.2 Optimization Approaches Based on Multistate Cost Models

(A) Optimistic Approach. The idea of this approach is to simply choose an efficient execution plan using the query cost estimates given by multistate cost models for the current system environmental state at compile time. This approach works only under the assumption that the system environment changes slowly. Due to the slow change of the system environment, cost estimates given by a multistate cost model for the current system environment remain valid for a certain period of time. The execution plan chosen for a query on the basis of the cost estimates is also good for some time. Clearly, the applicability of this approach is quite restricted.

(B) Environment Predicting Approach. The idea of this approach is to predict/estimate the system environmental state in which a query is to be executed. The system environmental state in which a query is to be executed may be predicted/estimated by analyzing the application background. For example, system-administration-related queries are more likely to be

executed in evenings/weekends when system load is low, and business-related queries are more likely to be executed during business hours on working days when system load is high, etc. The usage (execution) pattern of user queries and the load pattern of a system environment can also be analyzed to improve the predication accuracy. In addition, users may be required to provide inputs about their queries usage to help the system to optimize the queries. Using multistate cost models based on a predicted run-time running system environmental state can usually provide better cost estimates than using traditional static cost models based on a static system environmental state or using multistate cost models simply based on a compile-time system environmental state. The degree of goodness of an execution plan based on a predicted running system environmental state depends on how accurate the prediction is. This approach works well when the user query usage and system load patterns are clear and stable. It would be difficult to deal with the situation in which a user changes his/her query usage pattern frequently.

(C) Lazy Approach. This approach generates an execution plan at compile time based on a static system environmental state or a typical system environmental state. At run time, unless the plan is found to be very inefficient, the query optimizer execute the query according to the execution plan. If the plan is found to be very inefficient, the query optimizer adaptively improves the execution plan. This approach is similar to the dynamic query optimization approaches mentioned in Section 1. As indicated, the overhead for adjusting an execution plan can be very high. An execution plan is adjusted only if the overhead is paid off by the benefits of the new execution plan. Note that it is usually very expensive and complicated to adjust an execution plan in the middle of an execution. Furthermore it is sometimes too late to find the current execution plan is very inefficient. Re-generating a brand new execution plan at run time is equivalent to the interpretation approach, which prohibits comprehensive query optimization since its overhead directly affects the query response time.

(D) Multiple Version Approach. The idea of this approach is to generate multiple versions of an execution plan for a query at compile time, one for each representative run-time system environmental state. When a user runs the query at run time, an appropriate version of the execution plan is invoked according to the actual running system environmental state. Note that the main aim for our query optimization technique is to take the dynamic behavior of the system environment into consideration when choosing an execution plan for a query and in the meantime reduce the optimization work performed at run time. This approach is very promising in achieving this aim. It is expected to provide a better plan/version than the one

generated by assuming a fixed environment (e.g., the static one), and also take much less work than that for conventional dynamic optimization at run time since most work is done at compile time. This approach makes the run-time algorithms simple, efficient and easy to implement. Since this approach is the most promising one, we will further discuss the relevant issues in the next section.

4 QUERY OPTIMIZATION VIA MULTIPLE PLAN VERSIONS

In this section, we address two key issues for the multiple version optimization approach, that is, what versions should be generated for an execution plan at compile time and which version should be invoked at run time.

4.1 Selecting Plan Versions at Compile Time

Assume that a given query Q involves N participating local sites (databases) in the MDDBS: S_1, S_2, \dots, S_N . The range of probing query cost $Y_{Q_p^i}$ (i.e., contention level) for site S_i is $[V_i, W_i]$ ($1 \leq i \leq N$). There are K_i query classes at site S_i . The number of (local) contention states used by the cost model for query class G_{ij} ($1 \leq j \leq K_i$) at site S_i is M_{ij} . Let $H_{ij} = \{s_1^{(ij)}, s_2^{(ij)}, \dots, s_{M_{ij}}^{(ij)}\}$ be the set of the contention states for query class G_{ij} at site S_i , with $s_1^{(ij)}$ representing the lowest contention state and $s_{M_{ij}}^{(ij)}$ the highest one. Let $s^{(ij)}(Y_{Q_p^i}) \in H_{ij}$ be the unique contention state determined by probing query cost $Y_{Q_p^i} \in [V_i, W_i]$ for query class G_{ij} at site S_i .

Hence the set

$$H_i = \{s^{(i1)}(Y_{Q_p^i}) = < s^{(i1)}(Y_{Q_p^i}), s^{(i2)}(Y_{Q_p^i}), \dots, s^{(iK_i)}(Y_{Q_p^i}) > \text{ where } Y_{Q_p^i} \in [V_i, W_i] \}$$

contains all contention state vectors at site S_i , and

$$H = H_1 \times H_2 \times \dots \times H_N$$

contains all possible system environmental states for query Q .

If the contention level (i.e., $Y_{Q_p^i}$) at each site is given, the contention state vector at each site is determined. The system environmental state is then also determined. In other words, there is a unique system environmental state corresponding to a point $< Y_{Q_p^1}, Y_{Q_p^2}, \dots, Y_{Q_p^N} >$ in the N -dimensional region $D_0 = [V_1, W_1] \times [V_2, W_2] \times \dots \times [V_N, W_N]$. We call $< Y_{Q_p^1}, Y_{Q_p^2}, \dots, Y_{Q_p^N} >$ a system environmental

(contention) level, and $[V_i, W_i]$ the interval of region D_0 in the i -th dimension. Note that many system environmental levels may correspond to the same system environmental state.

For any given system environmental state, the multistate cost models can give good cost estimates for component queries run at local sites in the environment. When a user issues a global query to an MDDBS, the global query optimizer needs to decide how to decompose it into component queries and where to execute the component queries. These decisions are specified in an execution plan. If the system environmental state is known, the query optimizer can generate an efficient plan for the query based on cost estimates of component queries as well as possible communication costs². Unfortunately, what run-time system environmental state in which the query is to be executed is unknown at compile time when the query optimizer optimizes the query.

One solution to this problem is to generate multiple versions of the execution plan, one for each possible system environmental state, and run the right version corresponding to the system environmental state in which the query is executed at run time. If the number of (participating) sites and the number of contention states for each query class at all sites are small, which leads to a small number of system environmental states, this solution is feasible. Otherwise, the query optimizer may have to generate too many versions for a query execution plan. In practice, there is a limit on the number of versions we could generate for a query execution plan due to the space and time constraints. Thus the number of versions that can be generated may be less than the number of possible system environmental states (with respect to the query) in an MDDBS. We therefore need a way to select an appropriate subset of system environmental states for which we generate multiple versions of a query execution plan.

Assume that each system environmental level, i.e., a point in region D_0 , has an equal chance to occur at run time. If only one version is allowed for a plan, which system environmental state we should select? In principle, the selected system environmental state should be representative in the sense that the corresponding version of the plan will minimize the performance degradation when it is invoked in another system environmental state. A good choice in this case is to select the system environmental state corresponding to the center point $\vec{p}_0 = < (V_1 + W_1)/2, (V_2 + W_2)/2, \dots, (V_N + W_N)/2 >$ in region D_0 since the sum of the distances between any

²A communication cost is usually proportional to the amount of data transferred. For simplicity, we consider a fast LAN in which the communication cost is negligible for the rest of the paper.

point in D_0 and \vec{p}_0 is minimized. In other words, the selected system environmental state is $s(\vec{p}_0) = \langle \vec{s}^{(1)}((V_1+W_1)/2), \vec{s}^{(2)}((V_2+W_2)/2), \dots, \vec{s}^{(N)}((V_N+W_N)/2) \rangle$. Figure 1 shows an example in the two-dimensional case.

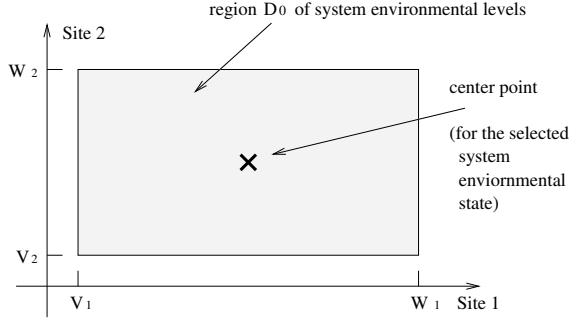


Figure 1: Selection of one representative system environmental state

If two versions are allowed for the plan, we can do the following. Let

$$A_i(D_0) = \left(\sum_{j=1}^{K_i} |H_{ij}(D_0)| \right) / K_i, \quad (1 \leq i \leq N) \quad (3)$$

where $|H_{ij}(X)|$ denotes the number of contention states for query class G_{ij} whose representing intervals have a non-empty intersection with the interval of region X in the i -th dimension. We call $A_i(D_0)$ the average number of contention states related to D_0 in the i -th dimension (site). Clearly, $A_i(D_0) \geq 1$ is always true. Let $A_k(D_0) = \max\{A_1(D_0), A_2(D_0), \dots, A_N(D_0)\}$. Since D_0 is related to more contention states on average in the k -th dimension, we divide D_0 into two half regions by splitting it along the k -th dimension as: $D_{01} = [V_1, W_1] \times \dots \times [V_k, W_k] \times \dots \times [V_N, W_N]$ and $D_{02} = [V_1, W_1] \times \dots \times (V'_k, W'_k) \times \dots \times [V_N, W_N]$ where $V'_k = W'_k = (V_k + W_k)/2$. That is, D_0 is divided into two half regions along the dimension that has the maximum number of contention states (on average for all query classes) related to D_0 . In this way, the maximum number of contention states (on average) that are covered (represented) by each sub-region (i.e., each chosen representative system environmental state) in any dimension is expected to be minimized. Hence the performance degradation caused by invoking a representative query plan version in a system environmental state other than the representative state can be reduced. If there is a tie for choosing such a dimension k , any of the tied dimensions can be used for splitting. The system environmental states corresponding to (i.e., containing) the center points of D_{01} and D_{02} are selected as the representatives for generating two versions³ for the query

³Note that it is possible that the plan versions for two

execution plan. Figure 2 shows an example in the two-dimensional case.

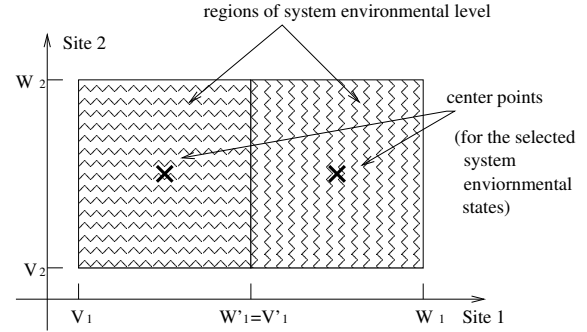


Figure 2: Selection of two system environmental states

In general, for any given number m , we can recursively apply this procedure to split a region into two until m representative system environmental states for generating m plan versions are selected. Note that the center point for a selected representative system environmental state is uniquely determined by a given region. It is sufficient to determine m regions in order to select m representative system environmental states. A general region D is denoted by $\langle L_1, U_1 \rangle \times \langle L_2, U_2 \rangle \times \dots \times \langle L_N, U_N \rangle$, where ' $\langle \rangle$ ' can be either closed '[' or open '('; L_i and U_i are the minimum and maximum contention levels at site (dimension) i for the region, respectively. $\langle L_i, U_i \rangle$ is called the interval of D in the i -th dimension.

The following algorithm, for a given number m , selects m regions from which the m representative system environmental states are extracted from.

ALGORITHM 4.1 : Selecting regions for generating query plan versions

Input: (1) the number m of system environmental states to be selected for generating multiple versions of a query execution plan, (2) the region $D_0 = [V_1, W_1] \times [V_2, M_2] \times \dots \times [V_N, M_N]$ for the system environmental level, and (3) the set H_{ij} of contention states, including their representing intervals, for every query class G_{ij} ($1 \leq j \leq K_i$) at each site S_i ($1 \leq i \leq N$).

Output: (1) a set of regions whose center points are used to determine the system environmental states for which query plan versions are to be generated, and (2) a data structure *SiteInfo* that keeps the information about the current intervals for each dimension and their associated regions to facilitate the search for

representative system environmental states are identical, depending on the particular MDDBS. Since this phenomenon does not degrade the representativeness of the versions for the regions, for simplicity, we consider plan versions as different instances regardless of their contents in this paper.

a query plan version at run time.

Method:

1. **begin**
2. Let $R := \{D_0\}$,
where $D_0 = [V_1, W_1] \times [V_2, M_2] \times \dots \times [V_N, M_N]$;
/* R keeps the current set of selected regions */
3. Initialize *SiteInfo*;
4. Let $j := 1$;
5. **while** $j < m$ **do**
/* more regions to be selected */
6. Take one region $x \in R$; /* every region in R
has one average number of contention states
related to each dimension at this point */
7. Let k be a dimension such that $A_k(x) = \max\{A_1(x), A_2(x), \dots, A_N(x)\}$ where
 $A_i(x) = (\sum_{j=1}^{K_i} |H_{ij}(x)|) / K_i$;
/* if there is a tie, choose any one of them */
8. Let $a := A_k(x)$;
9. **if** $a = 1$ **then break**; /* no need to further
split the region */
10. **while** there exists a region:
 $D = \langle L_1, U_1 \rangle \times \langle L_2, U_2 \rangle \times \dots \times \langle L_N, U_N \rangle$
in R such that $A_k(D) = a$ **do**
11. Let $L'_k := U'_k := (L_k + U_k) / 2$;
12. Let $D_1 := \langle L_1, U_1 \rangle \times \langle L_2, U_2 \rangle \times \dots$
 $\times \langle L_k, U'_k \rangle \times \dots \times \langle L_N, U_N \rangle$;
13. Let $D_2 := \langle L_1, U_1 \rangle \times \langle L_2, U_2 \rangle \times \dots$
 $\times \langle L'_k, U_k \rangle \times \dots \times \langle L_N, U_N \rangle$;
14. Replace D in R by D_1 and D_2 ;
15. $j := j + 1$;
16. Update *SiteInfo*;
17. **if** $j = m$ **then break**;
18. **end while**;
19. **end while**;
20. **return** R and *SiteInfo*;
21. **end**.

Algorithm 4.1 repeatedly splits a region along the dimension with the largest average number of contention states related to the region until the desired number of regions are obtained or every dimension has only one contention state related to the region. Thus the number of loops to be done is $O(m)$. Usually, m is much less than the total number of possible system environmental states.

If a region $D = \langle L_1, U_1 \rangle \times \langle L_2, U_2 \rangle \times \dots \times \langle L_N, U_N \rangle$ is selected, its center point:

$$\vec{p} = \langle (L_1 + U_1) / 2, (L_2 + U_2) / 2, \dots, (L_N + U_N) / 2 \rangle \in D$$

is used to determine a representative system environmental state:

$$s(\vec{p}) = \langle \bar{s}^{(1)}((L_1 + U_1) / 2), \bar{s}^{(2)}((L_2 + U_2) / 2), \dots, \bar{s}^{(N)}((L_N + U_N) / 2) \rangle.$$

A version of the query execution plan is generated for each of such representative system environmental states for the selected regions.

To facilitate the search for an appropriate version of the query execution plan at run time, Algorithm 4.1 also maintains a data structure *SiteInfo*, which contains a substructure for each site (dimension) (see Table 1). The use of this data structure will be dis-

Site	Interval	Regions containing interval
Site 1	interval 1	regions containing interval 1
	interval 2	regions containing interval 2

Site 2	interval 1	regions containing interval 1
	interval 2	regions containing interval 2

...
Site N	interval 1	regions containing interval 1
	interval 2	regions containing interval 2

Table 1: *SiteInfo* data structure

cussed in the next subsection.

Example 4.1 Suppose we have 3 participating sites (dimensions) x, y, z for a given query. The entire region of contention levels is $D_0 = [0, 40] \times [0, 50] \times [0, 60]$. Each site has two query classes, whose contention states and their representing intervals are shown in Table 2. Using Algorithm 4.1, Figure 3 shows the regions selected when 5 versions are to be generated for a query execution plan. In the first iteration, region D_0 is split into two regions D_1 and D_2 along the y dimension (since it has the largest average number of contention states related to D_0). In the next iteration, region D_1 is split into two regions D_3 and D_4 along z dimension and similarly region D_2 is split into two regions D_5 and D_6 . In the final iteration, region D_3 is split along the x dimension to get regions D_7 and D_8 . The final selected regions are D_4, D_5, D_6, D_7 and D_8 (i.e., the leave nodes of the tree in Figure 3). The center points of the selected regions are: $\vec{p}_4 = \langle 20, 12.5, 45 \rangle$, $\vec{p}_5 = \langle 20, 37.5, 15 \rangle$, $\vec{p}_6 = \langle 20, 37.5, 45 \rangle$, $\vec{p}_7 = \langle 10, 12.5, 15 \rangle$, and $\vec{p}_8 = \langle 30, 12.5, 15 \rangle$. Then the selected representative system environmental states are:

$$\begin{aligned} s(\vec{p}_4) &= \langle \langle s_2^{(x1)}, s_2^{(x2)} \rangle, \langle s_2^{(y1)}, s_2^{(y2)} \rangle, \langle s_3^{(z1)}, s_4^{(z2)} \rangle \rangle, \\ s(\vec{p}_5) &= \langle \langle s_2^{(x1)}, s_2^{(x2)} \rangle, \langle s_3^{(y1)}, s_3^{(y2)} \rangle, \langle s_1^{(z1)}, s_2^{(z2)} \rangle \rangle, \\ s(\vec{p}_6) &= \langle \langle s_2^{(x1)}, s_2^{(x2)} \rangle, \langle s_3^{(y1)}, s_3^{(y2)} \rangle, \langle s_3^{(z1)}, s_4^{(z2)} \rangle \rangle, \\ s(\vec{p}_7) &= \langle \langle s_1^{(x1)}, s_1^{(x2)} \rangle, \langle s_2^{(y1)}, s_2^{(y2)} \rangle, \langle s_1^{(z1)}, s_2^{(z2)} \rangle \rangle, \\ s(\vec{p}_8) &= \langle \langle s_3^{(x1)}, s_3^{(x2)} \rangle, \langle s_2^{(y1)}, s_2^{(y2)} \rangle, \langle s_1^{(z1)}, s_2^{(z2)} \rangle \rangle. \end{aligned}$$

A version of the query execution plan is then generated for each of the selected system environmental

	States for query class 1	States for query class 2
Site x	$s_1^{(x1)} = [0, 15], s_2^{(x1)} = (15, 25], s_3^{(x1)} = (25, 40]$	$s_1^{(x2)} = [0, 10], s_2^{(x2)} = (10, 20], s_3^{(x2)} = (20, 30], s_4^{(x2)} = (30, 40]$
Site y	$s_1^{(y1)} = [0, 8], s_2^{(y1)} = (8, 25], s_3^{(y1)} = (25, 40], s_4^{(y1)} = (40, 50]$	$s_1^{(y2)} = [0, 10], s_2^{(y2)} = (10, 20], s_3^{(y2)} = (20, 30], s_4^{(y2)} = (30, 40], s_5^{(y2)} = (40, 50]$
Site z	$s_1^{(z1)} = [0, 20], s_2^{(z1)} = (20, 40], s_3^{(z1)} = (40, 60]$	$s_1^{(z2)} = [0, 12], s_2^{(z2)} = (12, 24], s_3^{(z2)} = (24, 36], s_4^{(z2)} = (36, 48], s_5^{(z2)} = (48, 60]$

Table 2: Contention states at local sites

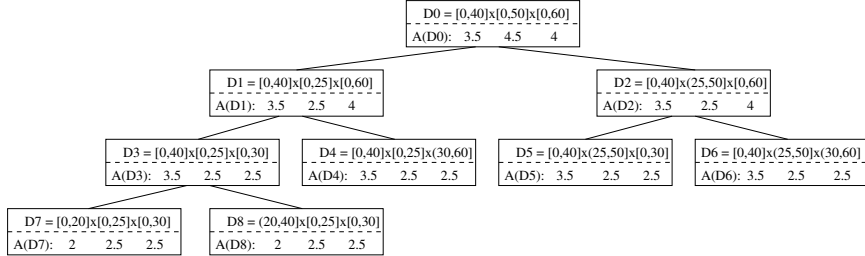


Figure 3: Selection of one system environmental state

states. Data structure *SiteInfo* for Sites x, y and z is shown in Table 3.

Site	Interval	Regions containing interval
Site x	[0,20]	D_7
	(20,40]	D_8
	[0,40]	D_4, D_5, D_6
Site y	[0,25]	D_4, D_7, D_8
	(25,50]	D_5, D_6
	[0,30]	D_5, D_7, D_8
Site z	(30,60]	D_4, D_6

 Table 3: An Example of *SiteInfo*

Note that if the assumption that every system environmental level has an equal chance to occur is not true, the actual distribution of system environmental level can be analyzed. Based on the analysis, more regions could be selected for the space area in which a system environmental level has a higher chance to occur. Due to the length limitation of the paper, the details of such a distribution-dependent partition of the space will be discussed in a separate paper.

4.2 Determining an Appropriate Version at Run Time

When a user requests to execute a query at run time, the best version of the relevant query execution plan should be invoked. How to determine the best version of the plan at run time is the issue to be discussed in this subsection.

First of all, the (current) running contention level at each participating local site can be gaged by the (observed or estimated) cost of a small probing query at run time as described in Section 2. Hence we have the running system environmental (contention) level. The corresponding running system environmental state can also be determined.

If the system environmental state in which the query is running is one of the selected representative system environmental states for which the versions of the query execution plan are generated, the corresponding version should be invoked. However, in general, the running system environmental state may not be one of the selected states since the number of the latter is limited. In this case, the selected region that contains the running system environmental level needs to be identified. Since the set of selected regions form a partition of initial region D_0 , there exists only one selected region containing the running system environmental level. The version of the execution plan generated for the representative system environmental state corresponding to the center point of the region should be invoked for the given query. Although this version was not generated exactly for the running system environmental state, it can usually yield a fair performance, compared with the single version (for a fixed static system environmental state) provided by a traditional static optimization approach. The more versions the query execution plan has, the better the query performance is expected.

Clearly, we need an efficient technique to search for the region that contains the running system environmental level. To do that, we make use of the data structure *SiteInfo* maintained by Algorithm 4.1. In fact, *SiteInfo* provides an index for relevant regions along each participating site/dimension. Note that it is possible that a running (local) contention level at a participating local site belongs to two region intervals. One is a sub-interval of the other (see intervals [0,20] and [0,40] in Table 3). The cause for this phenomenon is that the split of regions along one dimension may not be completely done before the sufficient number of regions have been selected. However, there is at most one such dimension along which intervals

with different lengths may exist. Although the region to which the running (global) system environmental level belongs is unique, we may have to search two lists for a desired region at one site/dimension.

The following algorithm makes use of *SiteInfo* to efficiently search for the relevant region to which a running system environmental level belongs and then returns the version of the execution plan for a query to be executed in the environment.

ALGORITHM 4.2 : Selecting a version of the execution plan for a given query at run time

Input: (1) the running (global) system environmental level $\vec{Y}_{Q_p} = \langle Y_{Q_p^1}, Y_{Q_p^2}, \dots, Y_{Q_p^N} \rangle$ at run time, where $Y_{Q_p^i}$ is a running (local) contention level at site i ($1 \leq i \leq N$), (2) data structure *SiteInfo* maintained by Algorithm 4.1, and (3) the execution plan with multiple versions, one for each selected region, for a given query Q .

Output: The best version of the execution plan for query Q in the running system environmental state corresponding to \vec{Y}_{Q_p} .

Method:

1. **begin**
2. Find an interval I_1 along dimension (site) 1 that contains local contention level $Y_{Q_p^1}$;
3. Use *SiteInfo* to get set R_{match} of regions containing interval I_1 ;
4. **if** there exists another interval I'_1 along dimension 1 that contains $Y_{Q_p^1}$
5. **then** use *SiteInfo* to get set R'_{match} of regions containing interval I'_1 ;
6. **else** let $R'_{match} := \emptyset$;
7. Let $R_{sel} := R_{match} \cup R'_{match}$;
8. **for** $i = 2$ to N **do**
9. Find an interval I_i along dimension i that contains local contention level $Y_{Q_p^i}$;
10. Use *SiteInfo* to get set R_{match} of regions containing interval I_i ;
11. **if** there exists another interval I'_i along dimension i that contains s_i
12. **then** use *SiteInfo* to get set R'_{match} of regions containing interval I'_i ;
13. **else** let $R'_{match} := \emptyset$;
14. Let $R_{match} := R_{match} \cup R'_{match}$;
15. Let $R_{sel} := R_{sel} \cap R_{match}$;
16. **end for**
17. Find the center point \vec{p} of the final selected region in R_{sel} ;
18. Find the representative system environmental state $s(\vec{p})$ for \vec{p} ;
19. **return** the plan version generated for $s(\vec{p})$;
20. **end.**

Algorithm 4.2 essentially utilizes indexes to locate the relevant regions without exhaustively checking all regions, when searching for a representative region for

a given running system environmental level.

Example 4.2 Let us consider the query execution plan with 5 versions generated in Example 4.1. Let the running system environmental level at which the query is executed at run time be $\langle 25, 45, 50 \rangle$, that is, the probing query costs at sites x, y and z are 25 sec., 45 sec. and 50 sec., respectively. Now, we need to find a selected region that contains point $\langle 25, 45, 50 \rangle$. Applying Algorithm 4.2, we first use *SiteInfo* to find the interval(s) that contains 25 along dimension x . Once such intervals $(20,40]$ and $[0,40]$ are found, the regions associated with the intervals are saved in R_{sel} , namely, $R_{sel} = \{D_4, D_5, D_6, D_8\}$. We then use *SiteInfo* to find the interval containing 45 along dimension y . Once such an interval $[25, 50]$ is found, the regions associated with the interval is saved in R_{match} , namely, $R_{match} = \{D_5, D_6\}$. Then $R_{sel} = R_{sel} \cap R_{match} = \{D_5, D_6\}$. We finally use *SiteInfo* to find the interval containing 50 along dimension z . Once such an interval $(30, 60]$ is found, the regions associated with the interval is saved in R_{match} , namely, $R_{match} = \{D_4, D_6\}$. Calculating $R_{sel} = R_{sel} \cap R_{match}$, we get $R_{sel} = \{D_6\}$. Hence the desired region is D_6 . The center point for D_6 is $\vec{p}_6 = \langle 20, 37.5, 45 \rangle$. Its corresponding representative system environmental state is $s(\vec{p}_6) = \langle \langle s_2^{(x1)}, s_2^{(x2)} \rangle, \langle s_3^{(y1)}, s_3^{(y2)} \rangle, \langle s_3^{(z1)}, s_4^{(z2)} \rangle \rangle$. Therefore, the version generated for $s(\vec{p}_6)$ is selected for executing the query.

5 EXPERIMENTS

To verify the effectiveness of our technique, we conducted some simulation experiments. In the experiments, global queries with 4 participating local database systems (sites) in an MDBS were considered. Two sites (Sites a and b) run Oracle 8.0 and the other two sites (Sites c and d) run DB2 5.0. Each site uses a SUN UltraSparc 2 workstation running Solaris 5.1.

An experimental database was created at each local site, with the same set of tables as those used in previous work (Zhu, 2000). More specifically, each local database has 12 tables $R_i(a_1, a_2, \dots, a_j)$ ($i = 1, 2, \dots, 12; j \in \{3, 5, 7, 9, 11, 13\}$) with cardinalities ranging from 3,000 ~ 250,000. The data in the tables is randomly generated. Each table has a number of indexed columns and various selectivities for different columns.

Two query classes were considered at each local site: one for unary queries and the other for join queries. A multistate cost model was developed for each query class by applying the qualitative approach described in Section 2 based on the observed costs of

sample queries run on the local database systems. Using the multistate cost models, we can estimate the cost of a (component) query run in any contention state at a local site. Since we focus on studying the effect of dynamic factors at autonomous local sites on query processing, we assume that the communication costs are negligible by using a high-performance local area network. For our simulation experiments, the costs of component queries obtained from decomposing a global test query are simulated by using the corresponding cost estimates given by the multistate cost models with a random error within 30%. From our previous empirical studies (Zhu, 2000), this simulation is reasonable.

The global queries tested in our experiments are of the following form:

$$\begin{aligned}
 & (\pi_{\alpha_a}(\sigma_{F^a}(R^a))) \bowtie_{F^{ab}} (\pi_{\alpha_b}(\sigma_{F^b}(R^b))) \\
 & \bowtie_{F^{bc}} (\pi_{\alpha_c}(\sigma_{F^c}(R^c))) \bowtie_{F^{cd}} (\pi_{\alpha_d}(\sigma_{F^d}(R^d))) \quad (4)
 \end{aligned}$$

where R^x is a table at local site x , α^x is a list of columns in R^x , F^x is a qualification condition on R^x , F^{xy} is a qualification condition on R^x and R^y , and $x, y \in \{a, b, c, d\}$.

There are a number of strategies to perform such a query. Figure 4 shows two of them. Given a system environmental level, we can determine the corresponding system environmental state. We can then use the cost estimates given by the multistate cost models in the system environmental state to determine a best execution strategy from many alternatives. Each chosen execution strategy for a given representative system environmental state yields a version in the corresponding query execution plan. If several representative system environmental states are considered, we have a query execution plan with multiple versions. At run time, the best version is chosen to run by the technique discussed in Section 4.2, based on the given running system environmental level.

Figure 5 shows the comparison of costs for a set of random global queries of form (4) run in system environmental states determined by some random system environmental levels. The following costs for each query were compared: (i) the cost following the best version of the execution plan with (a random number between 2 and 8) multiple versions generated by the technique discussed in Section 4.1; (ii) the cost following the execution plan chosen by the static cost models (i.e., the ones assuming a static environment without considering dynamic factors); and (iii) the cost following the optimal execution plan for the running system environmental state determined by a given system environmental level. The figure shows that the execution plans with multiple versions are more efficient than the static execution plans. The performance of the execution plans with multiple versions well approximates the performance of optimal

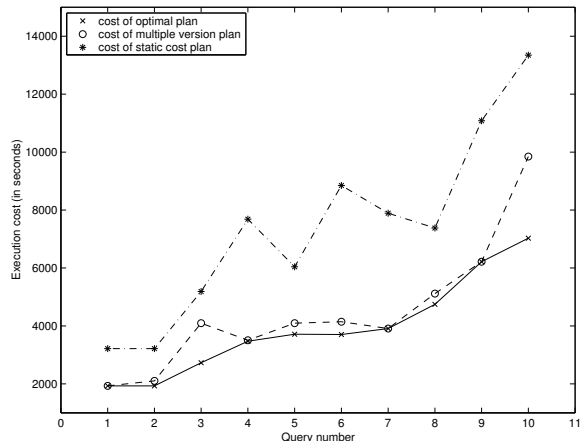


Figure 5: Costs of query execution plans

execution plans for the queries in the running system environmental states.

Figure 6 shows that the performance of a query execution plan with multiple versions is usually getting

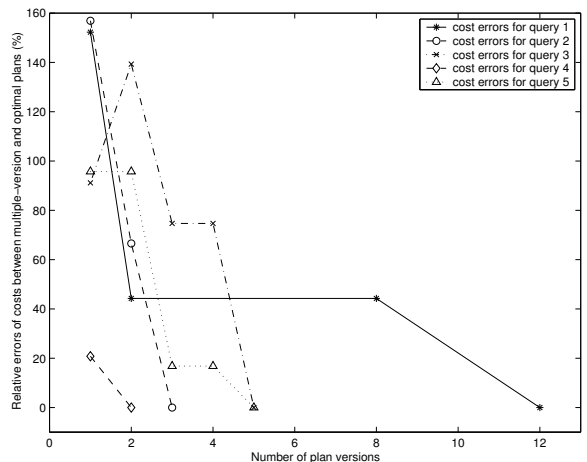


Figure 6: Performance improvement as number of versions increases

better and better as the number of versions allowed in the plan increases. In the experiment, we considered a number of global queries run in the system environmental states determined by some random running system environmental levels. As we increase the number of versions allowed in the execution plan for a query, the performance is usually getting better since a better version could be chosen to run the query. It is observed that the performance of the optimal plans for most queries in the experiment can be achieved without having to use a large number of versions. Besides, although in some cases the query performance

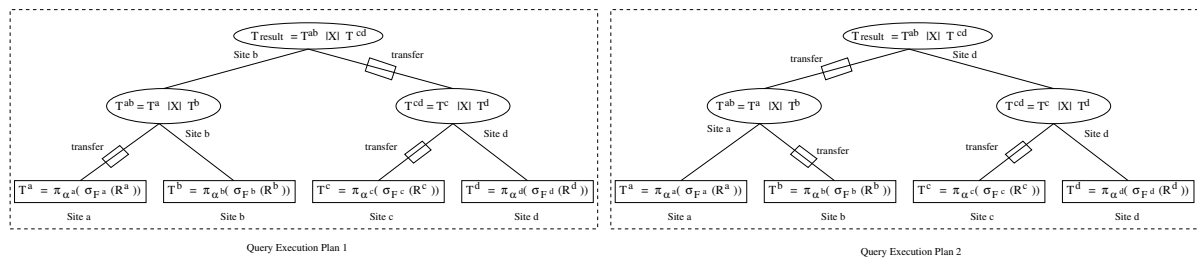


Figure 4: Examples of execution plan versions

may stay the same or even temporarily degrade, it always improves eventually as the number of versions in the plan increases. The figure demonstrates some typical types of query performance behavior in the experiment.

Figure 7 shows the results of another experiment, in which we assumed that only one version (i.e., the

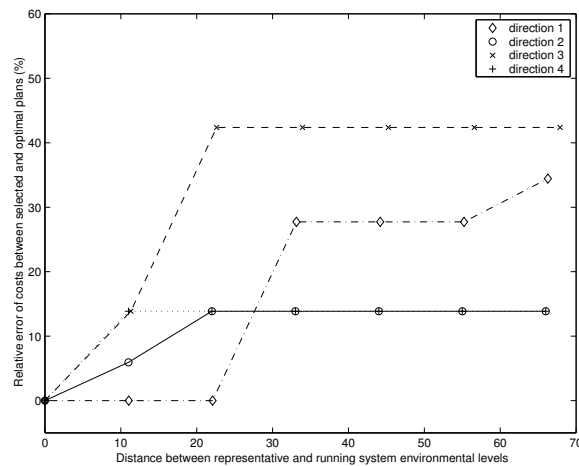


Figure 7: Effect of level distance on performance

one corresponding to the center point of the initial region) was allowed for the execution plan of a given query. We considered a number of running system environmental levels with various distances from the center point along different directions. The figure shows that the closer the running level is to the representative level (i.e., the center point), the closer the performance of the selected representative version is to the performance of the optimal plan. Therefore, when the representative regions are getting smaller and smaller (i.e., the maximum distance between the center and any point in the region is smaller), the better and better performance is expected for the execution plan with multiple versions.

Our experimental results demonstrate that the technique proposed in this paper is quite promising in performing global query optimization for a dynamic mul-

tidatabase environment.

6 CONCLUSIONS

The techniques proposed so far in the literature for global query optimization in multidatabase systems can be classified into static ones and dynamic ones. A static technique optimizes a query at compile time and does not consider the dynamically-changing environmental factors that may have a significant effect on the query cost at run time. Hence, the query execution plan generated with such a technique is often sub-optimal in a dynamic environment. However, the amount of work performed at run time for optimization in this case is negligible if not nothing as all the work for generating plans is done at compile time. A dynamic optimization technique on the other hand takes into consideration the dynamically-changing environmental factors and modifies or re-generates query execution plans at run time. Hence, the modified/re-generated plans are usually more efficient than the ones produced at compile time. However, the amount of work performed for such optimization is significant at run time, which directly affects the query response time and thus greatly reduces the benefits of an improved query execution plan.

The query optimization technique proposed in this paper overcomes the problems with the above two types of techniques. It takes into account the dynamically-changing environmental factors by adopting so-called multistate cost models for dynamic local sites. A multistate cost model can give a good cost estimate of a query run in any contention state at a dynamic local site. Based on the cost estimates, the technique generates an execution plan with multiple versions at compile time, one for each selected representative system environmental state. The algorithms to select a set of good representative system environmental states at compile time and to determine the best query execution plan version at run time are presented. Our experiments demonstrate that the proposed optimization technique is quite promising in

optimizing global queries in a dynamic multidatabase environment. However, our work is just the beginning of further research that needs to be done in the future in order to completely solve all relevant issues.

REFERENCES

- Adali, S., *et al.* (1996). Query caching and optimization in distributed mediator systems. In *Proc. of ACM SIGMOD Conf.*, pp 137–48.
- Amsaleg, L., *et al.* (1996). Scrambling Query Plans to Cope With Unexpected Delays. In *Proc. of Int'l Conf. on Paral. and Distr. Inf. Syst.*, pp 208–19.
- Bouganim, L., *et al.* (2000). Dynamic Query Scheduling in Data Integration Systems. In *Proc. of IEEE Int'l Conf. on Data Eng.*, pp 425–34.
- Chen, A. L. P. (1990). Outerjoin optimization in multidatabase systems. In *Proc. of Int'l Symp. on DB in Paral. and Distr. Syst.*, pp 211–18.
- Du, W., *et al.* (1992). Query optimization in heterogeneous DBMS. In *Proc. of VLDB Conf.*, pp 277–91.
- Du, W., M. C. Shan, and U. Dayal. (1995). Reducing Multidatabase Query Response Time by Tree Balancing. In *Proc. of ACM SIGMOD Conf.*, pp 293 – 303.
- Evrendilek, C., *et al.* (1997). Multidatabase Query Optimization. In *Distributed and Parallel Databases*, 5(1): 77–113.
- Gardarin, G., *et al.* (1996). Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *Proc. of VLDB Conf.*, pp 378–89.
- Goni, A., *et al.* (1996). Using Reasoning of Description Logics for Query Processing in Multidatabase Systems. In *Proc. of 3rd Workshop on Knowl. Repres. Meets DB*, pp 1–6.
- Lee, C. and C. J. Chen. (1998). Query Optimization in Multidatabase Systems Considering Schema Conflicts. In *IEEE Trans. on Knowledge and Data Eng.*, 9(6): 941–55.
- Lim, E.-P., *et al.* (1995). An Algebraic Transformation Framework for Multidatabase Queries. In *Distributed and Parallel Databases*, 3: 273–307.
- Yu, C.T. and W. Meng. (1998). Principles of Database Query Processing for Advanced Applications. *Morgan Kaufmann Publishers, Inc.*
- Naacke, H., G. Gardarin, and A. Tomasic. (1998). Leveraging mediator cost models with heterogeneous data sources. In *Proc. of IEEE Int'l Conf. on Data Eng.*, pp 351–60.
- Ng, K. W., *et al.* (1999). Dynamic Query Re-Optimization. In *Proc. of Int'l Conf. on Sci. and Stat. DB Manag.*, pp 264–273.
- Roth, M. T., *et al.* (1999). Cost models DO matter: providing cost information for diverse data sources in a federated system. In *Proc. of VLDB Conf.*, pp 599–610.
- Subramanian, D. K., and K. Subramanian. (1998). Query optimization in multidatabase systems. *Distributed and Parallel Databases*, 6(3): 183 – 210.
- Urhan, T., M. J. Franklin and L. Amsaleg. Cost-based Query Scrambling for Initial Delays. In *Proc. of ACM SIGMOD Conf.*, pp 130–141.
- Tsai, P. S. M. and A. L. P. Chen. (1997). Optimizing entity join queries when data transmission cost dominates. In *Data & Knowledge Engineering*, 22, pp 283–308.
- Zhu, Q. and P.-Å. Larson. (1994). A query sampling method for estimating local cost parameters in a multidatabase system. In *Proc. of IEEE Int'l Conf. on Data Eng.*, pp 144–53.
- Zhu, Q. and P.-Å. Larson. (1996). Building regression cost models for multidatabase systems. In *Proc. of Int'l Conf. on Paral. and Distr. Inf. Syst.*, pp 220–31, 1996.
- Zhu, Q. and P.-Å. Larson. (1996). Global Query Processing and Optimization in the CORDS Multidatabase System. In *Proc. of 9th Int'l Conf. on Paral. and Distr. Comp. Syst.*, pp 640–6.
- Zhu, Q. and P.-Å. Larson. (1997). A fuzzy query optimization approach for multidatabase systems. *Int'l J. of Uncertainty, Fuzziness and Knowledge-Based Sys.*, 5(6):701 – 22.
- Zhu, Q. and P.-Å. Larson. (1998). Solving local cost estimation problem for global query optimization in multidatabase systems. *Distributed and Parallel Databases*, 6(4): 373 – 420, 1998.
- Zhu, Q., Y. Sun and S Motheramgari. (2000). Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environments. In *Proc. of IEEE Int'l Conf. on Data Eng.*, pp 413-24.