Distrib Parallel Databases (2008) 23: 151–188
DOI 10.1007/s10619-008-7025-4

Query optimization via contention space partitioning and cost error controlling for dynamic multidatabase systems

Qiang Zhu · Jaidev Haridas · Wen-Chi Hou

Published online: 12 February 2008

Abstract A multidatabase system (MDBS) integrates information from multiple autonomous local databases. Performing global query optimization to achieve efficient query processing in such a system is challenging due to local autonomy of the data sources. Dynamic factors in the environment make the problem even more difficult. In this paper, we present two techniques, i.e., contention space partitioning and cost error controlling, to perform global query optimization in a dynamic MDBS. Both techniques generate an execution plan with multiple versions for a query in a dynamic MDBS, utilizing the multistate cost models built for the dynamic environment via our previous multistate query sampling method. The first technique partitions the contention space of a dynamic multidatabase environment into a given number of subspaces and chooses a good query execution plan version for each subspace, while the second technique selects a set of execution plan versions by using a given error tolerance to control query execution costs. Experiments demonstrate that the proposed techniques are 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 regenerating an execution plan for a query based on dynamic runtime information.

Keywords Multidatabase system · Dynamic environment · Query optimization · Multistate cost model · Execution plan · Algorithm

Communicated by Ahmed K. Elmagarmid.

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

Q. Zhu (✉) · J. Haridas
Department of Computer and Information Science, The University of Michigan, Dearborn, MI 48128, USA
e-mail: qzhu@umich.edu

W.-C. Hou
Department of Computer Science, Southern Illinois University, Carbondale, IL 62901, USA

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 efficiently process such a global query 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 techniques 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 [12, 16], a query sampling method [44, 45, 48], a cost vector database approach [1], a fuzzy approach [47], and a generic model approach [30, 36].

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. To capture dynamic factors in query cost estimation, we developed a multistate query sampling method to build multistate cost models for dynamic local database systems in an MDBS in [49]. The key idea is to divide a dynamic local database system environment into a number of contention states (such as "*High Contention*", "*Medium Contention*" and "*Low Contention*") and use the observed costs of sample queries run in the dynamic local database system to build a cost model with a qualitative variable indicating the contention states. It has been shown that such a multistate cost model can give a good cost estimate for a (component/local) query run in any contention state at a dynamic local site in an MDBS.

Establishing cost models is not the ultimate goal of query optimization. The ultimate goal is to choose an efficient execution plan for a query on the basis of the cost estimates given by the cost models. In general, there are two approaches to processing a query. The first one is called the interpretation approach. 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/embedded queries, which are usually executed repeatedly. In an MDBS 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. Hence optimization of ad hoc/interactive queries will not be further discussed in this paper.

Using multistate cost models to perform query optimization for stored/embedded queries in the compilation approach is more difficult. The main challenge is that

it is not easy to predict the runtime 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.

In this paper, we present two techniques to perform global query optimization for a dynamic MDBS based on multistate cost models in the compilation approach. Both techniques select a set of representative system environmental states for a dynamic MDBS, 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. Since multiple execution plan versions are employed to handle different dynamic situations, the query performance is expected to be better than what a traditional single-version execution plan can achieve. The difference between the above two techniques lies in the way they select the representative system environmental states. The first technique partitions the contention space into a given number of subspaces and chooses one representative execution plan version for each subspace (representing a group of system environmental states). The second technique repeatedly selects an execution plan version for a system environmental contention state in the contention space and attempts to share the plan version among as many (direct or indirect) neighboring contention states as possible if the estimated query cost is within a given tolerable range. This procedure continues until all the system environmental contention states are covered. The first technique directly controls the number of versions generated for an execution plan using a given number, while the second one directly controls the query costs using a given tolerance. Our simulation results demonstrate that the presented techniques are quite promising in optimizing a global query in a dynamic MDBS.

Global query optimization for multidatabase systems has been studied by a number of researchers in the past years [12, 13, 15, 16, 19, 20, 23, 25, 36, 37, 39, 40, 44]. The main goal of global query optimization in an MDBS is to achieve efficient query decomposition and inter-site integration during query processing. Generally speaking, there are two types of query optimization: static one and dynamic one. Static query optimization is to determine an efficient execution plan for a given query at compile time (applicable for the compilation approach), while dynamic query optimization is to improve query processing on the fly during query execution at run time (applicable for both the compilation and interpretation approaches). Some techniques such as semantic query optimization can actually be utilized at both compile time and run time. However, if a technique does not directly utilize dynamic information available at run time, we will place it in the category of static query optimization in the following discussion on related work.

As we know, the success of static query optimization relies on accurate cost models, which are difficult to obtain in an MDBS due to local autonomy of component database systems. Hence a major research effort for static query optimization in MDBSs was to explore techniques to estimate local cost parameters. As mentioned earlier, a number of such effective techniques have been proposed in the literature [1, 12, 16, 30, 34, 36, 42, 44, 45, 47, 48, 50]. Query processing architectures/models with static query optimization were suggested in [12, 22, 24, 39]. The impact of three

alternative system architectures on the performance of global query processing was studied in [9]. Semantic query optimization techniques that transform/reformulate a given global query into an equivalent but more efficient one based on semantic information available in a multidatabase were proposed in [17–19, 26, 28, 46]. Query optimization techniques that take into account the differing capabilities of local sites were presented in [10, 15, 39]. Several optimization algorithms that maximize parallelism (including independent one and pipelining one) for query processing were suggested in [13, 14, 37]. Techniques were also proposed to handle other unique optimization issues in MDBSs, which include entity join optimization [38], outerjoin optimization [8], query transformation considering schema conflicts [25], query optimization based on incomplete database concept [31], query optimization involving multiple mediators [22], and query decomposition in case of data replication [14]. The main advantage of static query optimization is that extensive optimization can be done at compile time without incurring optimization overhead during the query execution at run time. However, a major weakness of existing static query optimization techniques is its incapability of incorporating dynamic runtime factors into consideration. If the runtime environment is significantly different from the assumed one in which the execution plan is generated, query performance would suffer significantly.

A number of dynamic global query optimization techniques for MDBSs have also been reported in the literature. In [32], instead of producing an execution plan based on cost estimates at compile time, a dynamic query optimization technique was suggested to utilize a statistical decision mechanism to schedule inter-site operations in an MDBS based on partial results available at run time. In [2, 40], a query plan scrambling technique was introduced to change (based on heuristics or cost analysis) the scheduling of the operations in an active query plan determined at compile time when a delay is detected at run time. It attempts to hide an unexpected delay by performing other useful work in the hope that the cause of the delay is resolved in the meantime. After the rescheduling, the query plan needs to be restructured, typically by creating new operations that are not in the current plan. In [10, 23], techniques were discussed to improve an execution plan dynamically at run time based on partial evaluation results (e.g., pruning useless parts, tightening selection conditions). In [20, 21, 46], techniques were suggested to generate an incomplete/partial (rather than complete) execution plan at compile time (since some information may be missing) and improve the plan dynamically at run time when accurate information becomes available. New operations such as dynamic collectors and double pipelined hash join were introduced in [20] to perform adaptive/dynamic query optimization at run time. In [39], query processing and evaluation semantics were developed to process queries over unavailable data sources discovered at run time. In [5], a dynamic query processing architecture with three layers (i.e., the dynamic query optimizer, the scheduler and the query evaluator) was proposed. Each layer implements different dynamic query optimization strategies. Overall, the main advantage of dynamic query optimization is that it optimizes query processing based on more accurate dynamic information observed at run time. However, a shortcoming of dynamic query optimization is that the amount of work required to create/modify/re-generate an execution plan may be very significant, which directly affects the query response time.

Note that there is another type of dynamic/adaptive query optimization developed for continuous queries in the context of data stream management systems (DSMS)

[4, 6, 7, 11, 29, 35]. Since queries are continuously run over data streams in such a case, dynamic/adaptive query optimization is essential. Due to limited computing resources, a DSMS has to drop tuples (so-called load shedding) from the input data streams to provide approximate answers during its query processing. To maximize the answer precision, the system needs to employ adaptive query optimization strategies so as to react to changing arriving rates of input streams. Hence the objective of such adaptive query optimization is different from that of traditional one. On the other hand, some conventional dynamic query optimization techniques mentioned previously can be applied to improve query processing efficiency in a DSMS. Continuous queries over streaming data sources are not considered in this paper.

The global query optimization techniques proposed in this paper generate the execution plan for a query at compile time, which is similar to static query optimization. However, our techniques generate multiple versions for an execution plan based on multistate cost models capturing dynamic environments at run time, which essentially shifts significant runtime optimization work to the compile time. Hence 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. In other words, our techniques possess the strengths of both the static and dynamic optimization approaches and, in the meantime, overcome their shortcomings. To our knowledge, no similar work has been reported in the literature.

Nowadays, with the rapid growth of the Web/Internet, users can access a tremendous amount of information from numerous remote data sources. A Web information integration system with mediators/wrappers facilitates data accesses in the Web by providing a uniform query interface. Query optimization in such a system shares many common characteristics/issues with that in a conventional MDBS [3, 15, 21, 27, 41]. The techniques discussed in this paper can be applied to both conventional MDBSs and Web information integration systems to achieve high query performance in dynamic environments.

The rest of the paper is organized as follows. Section 2 gives an overview of multistate cost models. Section 3 discusses the contention space partitioning technique to generate an execution plan with multiple versions for a given query in dynamic multidatabase environments. The situations in which local contention levels follow uniform or non-uniform distributions are considered. Section 4 presents a cost error controlling technique to generate an execution plan with multiple versions for a given query in dynamic multidatabase environments, which does not assume any distribution. Section 5 shows some experimental results to evaluate our techniques. 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 MDBS, we introduced an effective multistate query sampling method (MQSM) in [49]. In this section, we give an overview of the development of a multistate cost model using MQSM and discuss the potential approaches to performing query optimization based on multistate cost models.

2.1 Multistate cost model development

MQSM 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 of the system increases. 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 neighboring 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 [49].

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

To develop a cost model including the indicator variables, we extend the query sampling method in [48]. More specifically, component queries that can be performed on a local DBS in an MDBS are first grouped into homogeneous classes, based on some information available at the global level in an MDBS 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:

$$Y = \underbrace{\left(B_0^0 + \sum_{j=1}^{M-1} B_0^j Z_j\right)}_{\text{intercept}} + \sum_{i=1}^{n} \underbrace{\left(B_i^0 + \sum_{j=1}^{M-1} B_i^j Z_j\right)}_{\text{slopes}} X_i, \tag{1}$$

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 intercept and slopes of (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 for a component query run in any contention state in a dynamic local database system [49].

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.,

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

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 (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 (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], namely,[1] 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 a contention state s is considered to be the same as an interval (subrange) I of contention level. They mutually represent each other. We call I the representing interval of s 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 [49].

On the other hand, for a probing query cost Y_0 (i.e., a contention level) observed at a local site S_i ($1 \le i \le N$), there is a unique corresponding contention state $s^{(ij)}(Y_0)$, i.e., the representing interval containing Y_0, for each query class G_{ij} ($1 \le j \le K_i$) at the site. In other words, a unique (local) contention state vector $\vec{s}^{(i)}(Y_0) = \langle s^{(i1)}(Y_0), s^{(i2)}(Y_0), \ldots, s^{(iK_i)}(Y_0)\rangle$ 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 an observed probing query cost at each site. The combination of all the (local) contention state vectors is called a (global) system environmental (contention) state in this paper, which reflects the running contention environment for the query.

[1] In the mathematical notation, 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.

2.2 Potential query optimization approaches based on multistate cost models

One difficulty to apply multistate cost models to perform query optimization is that it is not easy to predict the runtime 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 approaches to performing query optimization based on multistate cost models, as described as follows.

(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 a certain period of 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 the system load is low, and business-related queries are more likely to be executed during business hours on working days when the system load is high, etc. The usage 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 runtime 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 executes 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 Sect. 1. As pointed out, 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 runtime 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 of 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 quite promising in realizing this aim. It is expected to provide a more efficient plan than the one generated by assuming a fixed environment (e.g., the static one), and also incur much less overhead than that for conventional dynamic optimization at run time since most work is done at compile time. This approach makes the runtime algorithms simple, efficient and easy to implement.

Clearly, the last approach is the most promising one among others. To realize this approach, the issues such as how to select representative environmental states at compile time and how to choose the best version of the execution plan to run the query at run time need to be addressed. Two techniques to solve these issues are discussed in the following sections.

3 Contention space partitioning technique

In this section, we introduce a technique to select a given number of representative environmental states for a dynamic environment. We first assume that the contention level at each local site in the MDBS follows the uniform distribution. We then discuss an extension of the technique to handle non-uniform distributions.

3.1 Selecting plan versions at compile time

Assume that a given query Q involves N participating local sites (databases) in the MDBS: $S_1, S_2, \ldots, S_N$. 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 $H_{ij} = \{s_1^{(ij)}, s_2^{(ij)}, \ldots, s_{M_{ij}}^{(ij)}\}$ 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

$$H_i = \{\vec{s}^{(i)}(Y_{Q_p^i}) = \langle s^{(i1)}(Y_{Q_p^i}), s^{(i2)}(Y_{Q_p^i}), \ldots, s^{(iK_i)}(Y_{Q_p^i})\rangle \text{ where } Y_{Q_p^i} \in [V_i, W_i]\}$$

contains all contention state vectors at site S_i, and

$$H = H_1 \times H_2 \times \cdots \times H_N \tag{3}$$

contains all possible system environmental states for query Q.

If the contention level (value), 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 $\langle Y_{Q_p^1}, Y_{Q_p^2}, \ldots, Y_{Q_p^N} \rangle$ in the N-dimensional region $D_0 = [V_1, W_1] \times [V_2, W_2] \times \cdots \times [V_N, W_N]$. We call $\langle Y_{Q_p^1}, Y_{Q_p^2}, \ldots, Y_{Q_p^N} \rangle$ a system environmental (contention) level (value), and $[V_i, W_i]$ the interval of region D_0 in the i-th dimension. We also call region D_0 the contention space for the dynamic multidatabase environment. 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 MDBS, 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 costs.[2] Unfortunately, the runtime 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 (i.e., leading to a small number of system environmental states), this solution is feasible. Otherwise, the query optimizer has to generate many versions for a query execution plan. In practice, there is a limit on the number of versions that can be generated 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 given query) in an MDBS. We therefore need a way to select an appropriate subset of system environmental states for generating the 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

[2] A 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 in the rest of the paper.

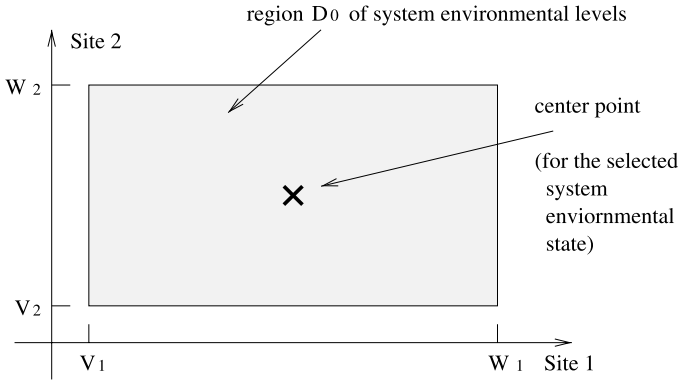

Fig. 1 Selection of one representative system environmental state

another system environmental state. A good choice in this case is to select the system environmental state corresponding to the center point $\vec{p}_0 = \langle (V_1 + W_1)/2, (V_2 + W_2)/2, \ldots, (V_N + W_N)/2 \rangle$ in region D_0 since a dramatic performance degradation is expected to be minimized. In other words, the selected system environmental state is $s(\vec{p}_0) = \langle \vec{s}^{(1)}((V_1 + W_1)/2), \vec{s}^{(2)}((V_2 + W_2)/2), \ldots, \vec{s}^{(N)}((V_N + W_N)/2) \rangle$. Figure 1 shows such an example in the two-dimensional case.

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

$$A_i(D_0) = \left(\sum_{j=1}^{K_i} |H_{ij}(D_0)| \right) / K_i \quad (1 \le i \le N) \tag{4}$$

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, $A_i(D_0) \ge 1$ is always true (assuming $D_0 \ne \emptyset$). Let $A_k(D_0) = \max\{A_1(D_0), A_2(D_0), \ldots, A_N(D_0)\}$. Unless $A_k(D_0) = 1$—no partition is needed in such a case, we split D_0 into two smaller regions. Since D_0 is related to more contention states on average in the k-th dimension, we divide D_0 into two half regions (subspaces) by partitioning it along the k-th dimension as: $D_{01} = [V_1, W_1] \times \cdots \times [V_k, W'_k] \times \cdots \times [V_N, W_N]$ and $D_{02} = [V_1, W_1] \times \cdots \times (V'_k, W_k] \times \cdots \times [V_N, W_N]$ 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 execution plan version for the query 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 gener-

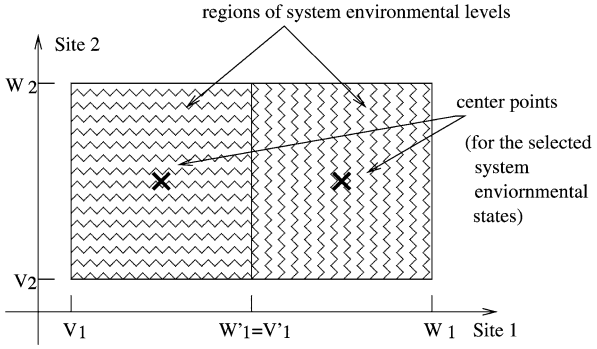

Fig. 2 Selection of two system environmental states

ating two versions[3] for the query execution plan. Figure 2 shows an example in the two-dimensional case.

In general, for any given number m, we can recursively apply this procedure to partition 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 $\langle L_1, U_1] \times \langle L_2, U_2] \times \cdots \times \langle L_N, U_N]$, where '$\langle$' 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. $\langle L_i, U_i]$ 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 determined.

Algorithm 3.1 Selecting regions for generating representative query plan versions based on contention space partitioning

Input: (1) the number m of system environmental states to be selected for generating multiple versions of the execution plan for a given query Q, (2) the contention space $D_0 = [V_1, W_1] \times [V_2, W_2] \times \cdots \times [V_N, W_N]$ for the dynamic multidatabase environment, and (3) the set H_{ij} of contention states, including their representing intervals, for each 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 (subspaces) whose center points are used to determine the system environmental states for which representative execution plan versions for Q 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 a representative execution plan version for Q at run time.

[3]Note that it is possible that the plan versions for two representative system environmental states are identical, depending on the particular MDBS. 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.

Method:

1. **begin**
2. Let $R := \{D_0\}$ where $D_0 = [V_1, W_1] \times [V_2, W_2] \times \cdots \times [V_N, W_N]$;
   /* 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 a region $x \in R$; /* every region in R has the same average number of contention states related to each individual dimension at this point */
7. Let k be a dimension such that $A_k(x) = \max\{A_1(x), A_2(x), \ldots, A_N(x)\}$ 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: $D = \langle L_1, U_1] \times \langle L_2, U_2] \times \cdots \times \langle L_N, U_N]$ in R such that $A_k(D) = a$ **do**
11. Let $L'_k := U'_k := (L_k + U_k)/2$;
12. Let $D_1 := \langle L_1, U_1] \times \langle L_2, U_2] \times \cdots \times \langle L_k, U'_k] \times \cdots \times \langle L_N, U_N]$;
13. Let $D_2 := \langle L_1, U_1] \times \langle L_2, U_2] \times \cdots \times (L'_k, U_k] \times \cdots \times \langle L_N, U_N]$;
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 3.1 repeatedly selects a region with a dimension k that has the largest average number of contention states related to the region and partitions the region along dimension k until the desired number of regions are obtained or every dimension has only one contention state related to the region. From the algorithm, we can see that most work is done in the nested loops starting at lines 5 and 10, respectively. In the worst case, the total number of iterations done in the two loops is $O(m)$. On the other hand, the most expensive step (i.e., line 7) requires $O(N * K * M)$ operations (including arithmetic operations, comparisons, etc.) in the worst case, where $K = \max_i\{K_i\}$ and $M = \max_{i,j}\{M_{ij}\}$. Therefore, the worst-case time complexity of Algorithm 3.1 is $O(m * N * K * M)$. Note that the maximum number of regions (thus the representative execution plan versions) selected by the algorithm is $O(m)$. Usually, m is much smaller than the total number of possible system environmental states.

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

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

Table 1 *SiteInfo* data structure

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
	...	...

is used to determine a representative system environmental state:

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

An optimal version of the execution plan for Q at each of such representative system environmental states is generated based on the multistate cost models for relevant sites as the execution plan versions representing the selected regions for Q. Note that, in practice, an efficient plan version, instead of the optimal version, could be used as a representative plan version.

To facilitate the search for an appropriate version of the query execution plan at run time, Algorithm 3.1 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 discussed in Sect. 3.3. *SiteInfo* requires most space (i.e., $O(m * N^2)$) in Algorithm 3.1. Hence the space complexity of the algorithm is $O(m * N^2)$.

Example 3.1 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 3.1, Fig. 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 sates 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 Fig. 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:

$$s(\vec{p}_4) = \langle \langle s_2^{(x1)}, s_2^{(x2)} \rangle, \langle s_2^{(y1)}, s_2^{(y2)} \rangle, \langle s_3^{(z1)}, s_4^{(z2)} \rangle \rangle,$$

$$s(\vec{p}_5) = \langle \langle s_2^{(x1)}, s_2^{(x2)} \rangle, \langle s_3^{(y1)}, s_3^{(y2)} \rangle, \langle s_1^{(z1)}, s_2^{(z2)} \rangle \rangle,$$

Table 2 Contention states at local sites

	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	$s_1^{(z1)} = [0, 20]$, $s_2^{(z1)} = (20, 40]$, $s_3^{(z1)} = (40, 60]$	$s_1^{(z2)} = [0, 12]$, $s_2^{(z2)} = (12, 24]$, $s_3^{(z2)} = (24, 36]$, $s_4^{(z2)} = (36, 48]$, $s_5^{(z2)} = (48, 60]$

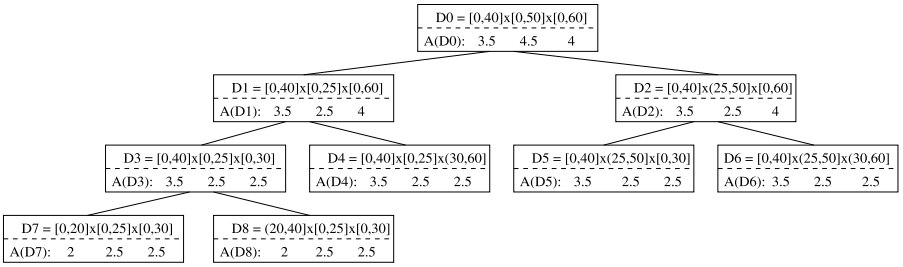

Fig. 3 Selection of regions for determining representative system environmental states

Table 3 An example of *SiteInfo*

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
Site z	[0,30]	D_5, D_7, D_8
	(30,60]	D_4, D_6

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

A version of the query execution plan is then generated for each of the selected system environmental states. Data structure *SiteInfo* for Sites x, y and z is shown in Table 3.

3.2 Handling non-uniform distributions

In the last subsection, we assume that every contention level (value) has an equal chance to occur at each local site in the multidatabase environment. However, in the real world, some ranges of the contention level may occur more often than others at a local site. For example, some very low or very high contention levels may rarely occur in a real application. In such a case, we should generate more execution versions for the area containing contention levels that have a higher chance to occur so that limited resources can be better utilized for handling real situations.

Some typical non-uniform distributions that the contention level at a local site in a multidatabase environment may follow include the normal distribution (i.e., the system load remains in a range centered around a point during most of time, as in a real company environment), the Erlang distribution (i.e., the system load remains low (or high) during most of time, as in a backup system), and the Cauchy distribution (similar to the normal distribution but with heavier tails). In general, let $f_i(x)$ be the probability density function of the distribution that the contention level $x \in [V_i, W_i]$ at site S_i $(1 \leq i \leq N)$ follows.

To calculate the mean of x for interval $\langle a_i, b_i]$ with a non-uniform distribution $f_i(x)$, we can use the following formula:

$$\overline{x}(a_i, b_i) = \frac{\int_{a_i}^{b_i} x f_i(x)\, dx}{\int_{a_i}^{b_i} f_i(x)\, dx}. \tag{5}$$

Hence, when we partition region $D = \langle L_1, U_1] \times \langle L_2, U_2] \times \cdots \times \langle L_N, U_N]$ along dimension/site k at line 11 in Algorithm 3.1, we should use the following split point:

$$L'_k = U'_k = \overline{x}(L_k, U_k)$$

based on (5). To calculate the integrals in (5), we can apply a typical numerical integration method such as the Simpson's rule [33].

Similarly, if a region $D = \langle L_1, U_1] \times \langle L_2, U_2] \times \cdots \times \langle L_N, U_N]$ is selected, its center point:

$$\vec{p} = \langle \overline{x}(L_1, U_1), \overline{x}(L_2, U_2), \ldots, \overline{x}(L_N, U_N) \rangle \in D$$

is used to determine a representative system environmental state:

$$s(\vec{p}) = \langle \vec{s}^{(1)}(\overline{x}(L_1, U_1)),\ \vec{s}^{(2)}(\overline{x}(L_2, U_2)), \ldots, \vec{s}^{(N)}(\overline{x}(L_N, U_N)) \rangle.$$

A version of the query execution plan is generated for each of such representative system environmental states for the selected regions. Clearly, the uniform distribution in Sect. 3.1 is a special case of the above discussion.

3.3 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 Sect. 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 selected states 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 3.1. 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 a sufficient number of regions have been selected. However, there is at most one such dimension along which intervals may overlap. 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 3.2 Selecting a version of the execution plan generated via contention space partitioning for a given query at run time
Input: (1) the running (global) system environmental level $\vec{Y}_{Q_p} = \langle Y_{Q_p^1}, Y_{Q_p^2}, \ldots, Y_{Q_p^N} \rangle$ 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 3.1, 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 3.2 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. Clearly, most work of the algorithm is done in the loop from line 8 to line 16 with $O(N)$ iterations. The most expensive steps (i.e., lines 14 and 15) in the loop require $O(m^2)$ operations in the worst case. Hence the worst-case time complexity of the algorithm is $O(N * m^2)$. Data structure *SiteInfo* requires most space for the algorithm. Thus the space complexity of the algorithm is $O(m * N^2)$.

Example 3.2 Let us consider the query execution plan with 5 versions generated in Example 3.1. 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 3.2, 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.

4 Cost error controlling technique

The technique introduced in the last section focuses on generating a given number m (allowed by the underlying resources) of versions of the execution plan for a user query. For an allowed m, it aims at minimizing the execution cost of a query for all system environmental states by selecting good representative execution plan versions. However, it has no direct control over the query cost. In other words, the error α between the cost of the optimal execution plan version and the cost of the representative execution plan version for a query in a given system environmental state can be large. In this section, we present an alternative method, called the cost error controlling technique, to generate multiple versions of the execution plan for a query in a dynamic multidatabase environment. This technique uses a given tolerance to control cost error α. For a given cost tolerance, this technique aims at minimizing the number of versions of the execution plan generated for a user query (to minimize resources needed) when determining the representative execution plan versions for all system environmental states. As before, two issues that need to be addressed are 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

Let Q be a query to be run in a dynamic multidatabase environment. For each system environmental state s, we need to choose a representative execution plan version for Q in such a way that the difference (relative error) α between the estimated cost of the representative version and the estimated cost of the optimal version of the execution plan is controlled within a given tolerance u (≥ 0). Note that the representative execution plan version for Q in state s is the one that is to be invoked for Q in s; while the optimal execution plan version for Q is the one that minimizes the estimated cost of Q, based on the multistate cost models, in state s. Clearly, if we choose the optimal version as the representative version, their cost error (0) meets the tolerance requirement. However, the number of selected execution plan versions can be very large. The basic idea of our technique to solve the problem is to let a selected representative execution plan version be shared by multiple system environmental states as long as the relevant cost error is controlled within the given tolerance u. In this way, the number of selected (representative) versions of the execution plan for query Q is reduced.

For the technique in the previous section, we considered the space of contention levels $D_0 = [V_1, W_1] \times [V_2, W_2] \times \cdots \times [V_N, W_N]$ and partitioned it into subspaces when determining representative execution plan versions. The (optimal) execution plan version generated for the system environmental contention state corresponding to the center of each subspace is used as the representative plan version for the subspace. In other words, if query Q is executed at a system environmental contention level belonging to a subspace S, the representative version corresponding to subspace S is invoked. In this approach, the system environmental state is determined by the system environmental contention level. Since the whole space of contention levels is covered by representative execution plan versions, all system environmental contention states are therefore covered.

Table 4 Contention states at a local site

Contention states/intervals for query class G_{11}	Contention states/intervals for query class G_{12}
$s_1^{(11)} = [0, 15]$, $s_2^{(11)} = (15, 25]$, $s_3^{(11)} = (25, 40]$	$s_1^{(12)} = [0, 10]$, $s_2^{(12)} = (10, 20]$, $s_3^{(12)} = (20, 30]$, $s_4^{(x2)} = (30, 40]$

However, for the technique presented in this section, we need to consider the system environmental states directly (rather than via the system environmental contention level) since we need to control the cost error between the representative version and the optimal version for each system environmental state. As mentioned in Sect. 2, the cost models for different query classes at each site may adopt different sets of (local) contention states, i.e., partitioning the range of contention level (probing query cost) differently. To simplify our discussion, we unify all contention states (for all query classes) at each site as follows.

We use the same notation as in Sect. 3.1. Let $E^{(ij)} = \{p_0^{(ij)}, p_1^{(ij)}, \ldots, p_{M_{ij}}^{(ij)}\}$ (assuming $V_i = p_0^{(ij)} < p_1^{(ij)} < \cdots < p_{M_{ij}}^{(ij)} = W_i$) be the end points of the representing intervals for the contention states in $H_{ij} = \{s_1^{(ij)}, s_2^{(ij)}, \ldots, s_{M_{ij}}^{(ij)}\}$ for query class G_{ij} at site S_i ($1 \le i \le N$, $1 \le j \le K_i$); that is, the representing interval for states $s_1^{(ij)}$, $s_2^{(ij)}$, …, $s_{M_{ij}}^{(ij)}$ are $[p_0^{(ij)}, p_1^{(ij)}]$, $(p_1^{(ij)}, p_2^{(ij)}]$, …, $(p_{M_{ij}-1}^{(ij)}, p_{M_{ij}}^{(ij)}]$, respectively. Let $E_i = E^{(i1)} \cup E^{(i2)} \cup \ldots \cup E^{(iK_i)}$, i.e., containing the end points of representing intervals of all contention states for all query classes at site S_i.

Using the end points in E_i, we can partition the range of contention level $[V_i, W_i]$ for site S_i into a set IS_i of refined intervals. If we consider that each refined interval represents a new (unified) contention state at site S_i, set IS_i yields a larger set $\hat{H}_i$ of contention states for site S_i. We call $\hat{H}_i$ the set of unified contention states at site S_i. If the system environment at site S_i remains in the same unified contention state, no query class will change its original contention state. However, if the system environment changes from one unified contention state to another, at least one query class has to change its (original) contention state when using its multistate cost model.

Let us consider an example. Assume that the possible component queries at site S_1 from decomposing query Q belong to two query classes with two sets of (original) contention states as shown in Table 4. The set of end points for the refined contention intervals is: $E_1 = \{0, 10, 15, 20, 25, 30, 40\}$. From this set, the unified contention states/intervals for site S_1 are given as follows:

$$s_1^{(1)} = [0, 10], \quad s_2^{(1)} = (10, 15], \quad s_3^{(1)} = (15, 20],$$
$$s_4^{(1)} = (20, 25], \quad s_5^{(1)} = (25, 30], \quad s_6^{(1)} = (30, 40].$$

Changing a unified contention state from $s_1^{(1)}$ to $s_2^{(1)}$ does not change the original contention state for query class G_{11} but changes the original contention state for query class G_{12}. Changing a unified contention state from $s_2^{(1)}$ to $s_3^{(1)}$, on the other hand, changes the original contention states for both query classes. Assume that the

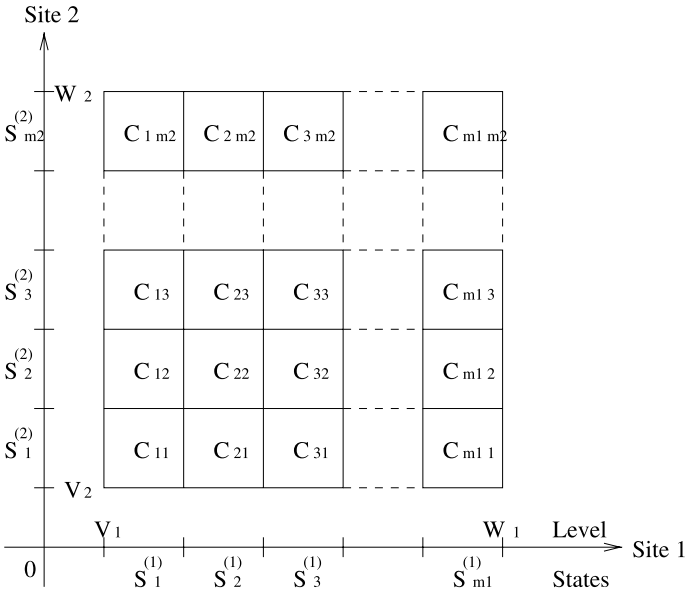

Fig. 4 Unified system environmental states in a two-site scenario

current contention level is 32. If the original contention states are used, query class G_{11} is in state $s_3^{(11)}$ and query class G_{12} is in state $s_4^{(12)}$. If the unified contention states are used, we can simply say that the system is in unified state $s_6^{(1)}$ regardless of the query class.

Clearly, the unified contention states at a local site capture all possible changes to the original contention states for all query classes at that site, and vectors in

$$\hat{H} = \hat{H}_1 \times \hat{H}_2 \times \cdots \times \hat{H}_N \tag{6}$$

capture all possible state changing scenarios in the dynamic multidatabase environment (with multiple sites). We call a vector in $\hat{H}$ a unified system environmental state. Note that each component of an original system environmental state in (3) is a vector of original contention states for different query classes at a local site, while each component of a unified system environmental state in (6) is simply a unified contention state (rather than a vector) at a local site.

It suffices to select a representative execution plan version for each unified system environmental state in $\hat{H}$ in order to run the query in the dynamic multidatabase environment. However, note that the unified contention states at each site only help us enumerate all possible state changing cases at the site. When a multistate cost model for a query class at the site is used to estimate the cost of a component query in the class, the corresponding original contention state has to be identified and applied with the model.

Figure 4 gives a visualization of unified system environmental states for two sites. Each cell in the figure represents a unified system environmental state. For example, cell C_{23} represents the unified system environmental state with a unified contention

state $s_2^{(1)}$ at site 1 and a unified contention state $s_3^{(2)}$ at site 2. Note that the boundaries of cells along each site are the end points of the representing intervals of the corresponding unified contention states at the site.

Our goal is to select a representative execution plan version for a given query Q at each unified system environmental state in the multidatabase environment so that the cost error between the representative execution plan version and the corresponding optimal one for Q is within the given tolerance. One approach to achieving this goal is to repeatedly pick up a unified system environmental state $\hat{s}$ whose representative execution plan version has not been selected (e.g., starting with the bottom-left one in a two-site scenario), generate an optimal execution plan version P for $\hat{s}$, and use P for each direct or indirect neighboring system environmental state of $\hat{s}$ if P is acceptable. Note that P is acceptable for a system environmental state $\hat{s}'$ if the cost error between P and the corresponding optimal version for $\hat{s}'$ is within the given tolerance. One disadvantage of this approach is that it has to examine every individual system environmental state. To overcome this problem, we employ the following more efficient approach.

Assume $\hat{H}_i = \{s_1^{(i)}, s_2^{(i)}, \ldots, s_{m_i}^{(i)}\}$ $(1 \le i \le N)$, with $s_1^{(i)}$ representing the lowest unified contention state and $s_{m_i}^{(i)}$ the highest one at site S_i. In other words, there are m_i unified contention states at site S_i.

Let us consider the following scenario. Assume that an execution plan version P_0 is acceptable for system environmental states $\langle s_1^{(1)}, \ldots, s_1^{(j-1)}, s_1^{(j)}, s_1^{(j+1)}, \ldots, s_1^{(N)}\rangle$ and $\langle s_1^{(1)}, \ldots, s_1^{(j-1)}, s_{10}^{(j)}, s_1^{(j+1)}, \ldots, s_1^{(N)}\rangle$; that is, only one site S_j changes its unified contention state from (lower) $s_1^{(j)}$ to (higher) $s_{10}^{(j)}$, while other sites remain in the same unified contention state $s_1^{(i)}$ $(1 \le i \le N;\ i \ne j)$. In this case, it is reasonable to assume that P_0 is also acceptable for intermediate system environmental states $\langle s_1^{(1)}, \ldots, s_1^{(j-1)}, s_n^{(j)}, s_1^{(j+1)}, \ldots, s_1^{(N)}\rangle$ for any $1 < n < 10$, which is called the one-site "sandwich" principle in this paper. Applying this principle and a binary search strategy, we can select representative execution plan versions for all system environmental states efficiently. Note that, although the cost errors for running P_0 in the intermediate system environmental states are usually within the given tolerance, it is not guaranteed.

Specifically, we use an $m_1 \times m_2 \times \cdots \times m_N$ matrix *CellInfo* to keep the information to indicate the representative execution plan versions selected for system environmental states. For example, cell (element) $CellInfo[k_1][k_2]\cdots[k_N]$ keeps the information about which execution plan version selected for system environmental state $\langle s_{k_1}^{(1)}, s_{k_2}^{(2)}, \ldots, s_{k_N}^{(N)}\rangle$. If no plan version is associated with a cell at the moment, such a cell is called a free cell. Initially, all cells are free and linked (via two linking pointers associated with each cell) in a doubly linked list, *FreeCells*, in the row-wise fashion. That is, $FreeCells: CellInfo[1][1]\cdots[1] \leftrightarrow CellInfo[2][1]\cdots[1] \leftrightarrow CellInfo[N][1]\cdots[1] \leftrightarrow CellInfo[1][2]\cdots[1] \leftrightarrow CellInfo[2][2]\cdots[1] \leftrightarrow \cdots \leftrightarrow CellInfo[N][2]\cdots[1] \leftrightarrow \cdots$.

Our algorithm picks up the first free cell C from *FreeCells*, which is $CellInfo[1][1]\cdots[1]$ at the beginning. It generates the optimal execution plan version P for the system environmental state corresponding to C, marks the cell as being selected and removes it from *FreeCells*. It then checks the direct and indirect

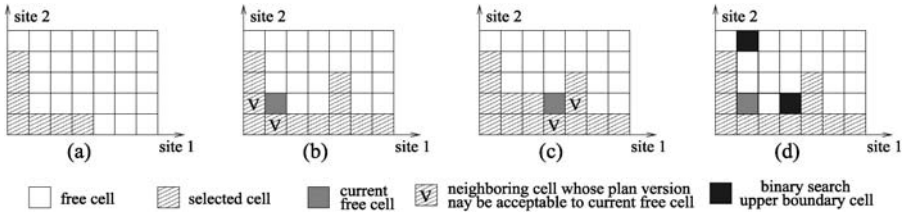

Fig. 5 Example scenarios for the cell matrix in an MDBS with two sites

neighboring cells along each dimension (fixing the other dimensions) to see if P is acceptable to them. To achieve high performance, it applies a binary search strategy to find the farthermost cell that accepts P along each dimension. It then applies the one-site sandwich principle to assign P to the relevant intermediate cells (system environmental states). Any cell whose execution plan version has been selected is removed from *FreeCells*. The adoption of a doubly linked list allows us to remove a cell from the list quickly (in $O(1)$ time). Figure 5(a) shows a resulting cell matrix after the above procedure for the first cell *CellInfo*[1][1] in an MDBS with two sites. In general, when the algorithm picks the first free cell from *FreeCells*, it also needs to check if the representative execution plan version selected for a neighboring cell is acceptable to this current free cell. Figure 5(b) and (c) show two scenarios in which the plan version selected for a neighboring cell (marked with "V") may be acceptable to the current free cell but has not been examined for it. Only if no neighboring plan version is acceptable to the current free cell, the optimal plan version for this cell is selected. For the binary search along each dimension, the search range needs to be determined before the search starts. The lower boundary of the search is the current cell, and the upper boundary of the search would be the farthermost consecutive free cell along this dimension. Figure 5(d) shows a scenario with binary search boundaries. More details of this algorithm is given as follows:

Algorithm 4.1 Selecting representative query plan versions based on cost error controlling
Input: (1) a given query Q for which an execution plan with multiple versions is to be generated, (2) the query cost error tolerance *Err* (i.e., a threshold value), and (3) the set H_{ij} of (original) contention states, including their representing intervals, for each query class G_{ij} ($1 \leq j \leq K_i$) at each site S_i ($1 \leq i \leq N$).
Output: (1) a data structure *UnifiedStates* that keeps the information about the unified contention states (including their representing intervals) for each site, and (2) an $m_1 \times m_2 \times \cdots \times m_N$ matrix *CellInfo* that keeps information about the representative execution plan versions selected for all unified system environmental states, where m_i is the number of unified contention states at site S_i.

Method:
1. **begin**
2. **for** each site S_i $(1 \leq i \leq N)$ **do**
3. Find the set $\hat{H}_i$ of unified contention states for the site and update data structure *UnifiedStates* with the relevant information;
4. **end for**
5. Create and initialize an $m_1 \times m_2 \times \cdots \times m_N$ matrix *CellInfo*, where $m_i = |\hat{H}_i|$;
6. Initialize a doubly linked list *FreeCells* to link all cells in *CellInfo*;
7. **while** *FreeCells* is not empty **do**
8. Remove the first free cell $CellInfo[k_1][k_2]\cdots[k_N]$ from *FreeCells*;
9. **if** there exists a neighboring cell whose representative execution plan P' has been selected and P' is acceptable to current $CellInfo[k_1][k_2]\cdots[k_N]$ **then** /* if there is more than one such cell, choose the one with minimum-error plan */
10. Update *CellInfo* with P' as the representative execution plan version selected for $CellInfo[k_1][k_2]\cdots[k_N]$;
11. Let $P := P'$;
12. **else** select the optimal execution plan version O for cell $CellInfo[k_1][k_2]\cdots[k_N]$ as its representative execution plan version and update *CellInfo*;
13. Let $P := O$;
14. **end if**
15. **for** each site S_i $(1 \leq i \leq N)$ **do**
16. ExpandRegion(*CellInfo*, $[k_1, k_2, \ldots, k_N]$, *Err*, S_i, P)
17. **end for**
18. **end while**
19. **return** *CellInfo* and *UnifiedStates*;
20. **end.**

The above algorithm invokes the following procedure to check if the execution plan version selected for the current cell is also acceptable to its direct and indirect neighboring cells along a given dimension, using the binary search strategy and the one-side sandwich principle.

Procedure ExpandRegion(*CellInfo*, $[k_1, k_2, \ldots, k_N]$, *Err*, S_i, P): Expand the selected region using the binary search and the one-site sandwich principle
Input: (1) *CellInfo*: the matrix containing information about representative execution plan versions selected for (some) system environmental states, (2) $[k_1, k_2, \ldots, k_N]$: the indexes of the current cell whose representative execution plan version has just been selected, (3) P: the execution plan version selected for current cell $CellInfo[k_1][k_2]\cdots[k_N]$, (4) *Err*: the acceptable cost error tolerance, and (5) S_i: the site/dimension to be examined so as to extend the region where plan P is acceptable.
Output: *CellInfo* with updated information for the new cells that accept execution plan version P.

Method :

1. **begin**
2. $L := k_i$;
3. Let U be the i-th index for the farthermost consecutive free cell *CellInfo*$[k_1]\cdots[k_{i-1}][U][k_{i+1}]\cdots[k_N]$ from the current cell along the dimension for S_i;
4. **if** P is acceptable to cell *CellInfo*$[k_1]\cdots[k_{i-1}][U][k_{i+1}]\cdots[k_N]$ **then**
5. Assign P to each cell *CellInfo*$[k_1]\cdots[k_{i-1}][j][k_{i+1}]\cdots[k_N]$ for $L < j \leq U$
6. and update *CellInfo* (and *FreeCells*) to reflect the changes;
7. **else**
8. **while** L != U **do**
9. $X := \lceil (L+U)/2 \rceil$;
10. **if** P is acceptable to cell *CellInfo*$[k_1]\cdots[k_{i-1}][X][k_{i+1}]\cdots[k_N]$ **then**
11. Assign P to each cell *CellInfo*$[k_1]\cdots[k_{i-1}][j][k_{i+1}]\cdots[k_N]$ for $L < j \leq X$
12. and update *CellInfo* (and *FreeCells*) to reflect the changes;
13. $L := X$;
14. **else**
15. $U := X$;
16. **end if**
17. **end while**
18. **end if**
19. **end.**

Most work of Algorithm 4.1 is done in lines 7–18. There are $O(M^N)$ cells in *CellInfo*, where $M = \max\{m_1, m_2, \ldots, m_N\}$. The algorithm determines a representative execution plan version for each cell either directly in lines 8–14 or indirectly in line 16 via *ExpandRegion*(). *ExpandRegion*() applies the binary search strategy and the one-side sandwich principle to share a representative execution plan version among as many neighboring cells as possible. However, in the worst case, *ExpandRegion*() spends $O(\log M)$ time in the binary search for each site but fails to share any plan version among neighboring cells. The representative execution plan version for each cell has to be determined in lines 8–14 in such a case. Thus the worst-case time complexity of Algorithm 4.1 is $O(M^N * N * \log M)$. Matrix *CellInfo* requires most space for the algorithm. Hence the space complexity of the algorithm is $O(M^N)$. Clearly, this algorithm is suitable for queries involving a small number of sites (i.e., small N). On the other hand, although the number of (unsharable) representative execution plan versions selected by the algorithm is $O(M^N)$ in the worst case, the actual number of selected plan versions is usually much smaller due to the adoption of the neighbor sharing strategy and the one-site sandwich principle in the algorithm. Furthermore, the larger the given cost error tolerance, the smaller the number of representative execution plan versions that would be selected by the algorithm. In the extreme, it is possible that only one representative execution plan is selected/acceptable for all the cells (unified system environmental states).

4.2 Determining an appropriate version at run time

For a given query Q, once *CellInfo* is filled, we have all the execution plan versions that are necessary to cover all the unified system environmental states. When we run Q at run time, we first determine the current running contention level at each local site based on a probing query cost. Using the running contention level and data structure *UnifiedStates*, we can determine the corresponding (running) unified contention state at the relevant site. *UnifiedStates* basically contains an array of the unified contention states (with their representing contention level intervals) for each site. To efficiently search for a unified contention state for a given contention level at a site, a binary search strategy can be applied. Using the running unified contention states at all sites and *CellInfo*, we can easily identify the representative execution plan selected for the corresponding cell (i.e., the running unified system environmental state). More details are given in the following algorithm.

Algorithm 4.2 Finding the execution plan version determined via cost error controlling for a given query Q at run time.
Input: (1) the running (global) system environmental level $\vec{Y}_{Q_p} = \langle Y_{Q_p^1}, Y_{Q_p^2}, \ldots, Y_{Q_p^N} \rangle$ at run time, where $Y_{Q_p^i}$ is a running (local) contention level at site S_i $(1 \leq i \leq N)$, (2) data structure *UnifiedStates* containing the information about the unified contention states for each site, and (3) matrix *CellInfo* containing the formation about the representative execution plan versions selected for all unified system environmental states.
Output: The version of the execution plan for query Q that is selected for the given running unified system environmental state corresponding to given $\vec{Y}_{Q_p}$.
Method:

1. **begin**
2. Initialize array A of indices;
   /* $A[i]$ will store the index for the running unified contention state at site S_i */
3. **for** $i = 1$ **to** N **do** /* For each site, retrieve the unified contention state corresponding to the running contention level */
4. Let $A[i]$ be the index for the running unified contention state that corresponds to the given running contention level $Y_{Q_p^i}$ at site S_i, based on *UnifiedStates*;
   /* The running unified contention state can be found using a binary search algorithm */
5. **end for**
6. $C :=$ *CellInfo*$[A[1]][A[2]] \cdots [A[i]] \cdots .[A[N]]$;
   /* Return the version for the cell C */
7. **return** the plan version selected for cell C;
8. **end.**

Most work of the algorithm is done in lines 3–5. Hence, the time complexity of the algorithm is $O(N)$. Most space required by the algorithm is for its inputs *UnifiedStates* and *CellInfo*. Therefore, the space complexity of the algorithm is $O(M^N)$.

The contention space partitioning technique in Sect. 3 and the cost error controlling technique in this section both have advantages and shortcomings. The main advantage of the former is that it controls the resources (space and time) to generate execution plan versions for a given query in a dynamic multidatabase environment. Although it aims at minimizing the execution cost of a query in each system environmental state, it has no direct control over the difference/error between the cost of the selected execution plan version and the cost of the optimal execution plan version. However, the cost error can be indirectly controlled by allowing a larger number of versions for an execution plan. The number m of execution plan versions to be selected for a given query can be determined based on several factors including time spent on query optimization, space used to keep execution plan versions and query cost error tolerated. A trade-off between the required resources (time and space) and the achieved efficiency (cost error) needs to be considered when determining m. The main advantage of the cost error controlling technique is that it can directly control the error/difference between the cost of the selected execution plan version and the cost of the optimal execution plan version with a given tolerance. It tries to minimize the number of versions generated for an execution plan by sharing a version among as many system environmental states as possible and reduce optimization time by avoiding checking some system environmental states. However, it has no control over the number of versions generated for an execution plan. It can only indirectly reduce the number of generated versions by allowing a larger cost error tolerance. Therefore, if the system resources used for generating and maintaining an execution plan are limited, the former technique should be employed. On the other hand, if the query performance needs to be ensured, the latter technique is preferred. Note that, although the optimal execution plan versions are considered in both techniques when determining representative execution plan versions, other efficient (rather than optimal) execution plan versions could be used in practice. In such a case, the cost error controlling technique would accept a neighboring execution plan version P for the current cell if the cost of P is smaller than that of the efficient execution plan version P' known to the current cell or the cost of P is within the error tolerance relative to the cost of P'.

5 Experiments

To evaluate the effectiveness of our query optimization techniques, we conducted some simulation experiments. In the experiments, global queries for an MDBS with four participating local database systems (sites) were considered. Two sites (sites a and b) ran Oracle 8.0 and the other two sites (sites c and d) ran DB2 5.0. Each site was equipped with a SUN UltraSparc 2 workstation running Solaris 5.1. A synthetic load builder was employed to generate dynamic loads to simulate dynamic application environments.

An experimental database was created at each local site, with the same set of tables as those used in previous work [49]. More specifically, each local database has twelve tables $R_i(a_1, a_2, \ldots, a_j)$ $(i = 1, 2, \ldots, 12;\ j \in \{3, 5, 7, 9, 11, 13\})$ with cardinalities ranging from 3,000–250,000. The data in the tables consists of randomly-generated

integers. 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 (i.e., query classes G_{15} and G_{24} in [48]). A multistate cost model was developed for each query class at each local site by applying the multistate query sampling method described in Sect. 2 based on the observed costs of sample queries run on the dynamic local database systems. Due to the limitations of our available computing resources (e.g., memory size and CPU speed), the cost models developed directly for our environment had only 4–6 contention states, which cannot reflect a more dynamic environment that may occur in the real world. To simulate more dynamic environments in some experiments, we extrapolate the directly-obtained cost models to include more (up to 11) contention states. 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 performance, 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 [49], this simulation is reasonable.

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

$$(\pi_{\alpha^a}(\sigma_{F^a}(R^a))) \underset{F^{ab}}{\bowtie} (\pi_{\alpha^b}(\sigma_{F^b}(R^b))) \underset{F^{bc}}{\bowtie} (\pi_{\alpha^c}(\sigma_{F^c}(R^c))) \underset{F^{cd}}{\bowtie} (\pi_{\alpha^d}(\sigma_{F^d}(R^d))) \tag{7}$$

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 $x, y \in \{a, b, c, d\}$. Each qualification condition is in the conjunctive normal form. A typical query example is given as follows:

$$\begin{aligned}
&(\pi_{R_2^a.a_2,\ R_2^a.a_5}(\sigma_{R_2^a.a_3>5\ \wedge\ R_2^a.a_5=2}(R_2^a))) \\
&\quad \underset{R_2^a.a_2=R_8^b.a_9}{\bowtie} (\pi_{R_8^b.a_4,\ R_8^b.a_9}(\sigma_{R_8^b.a_1\neq 3}(R_8^b))) \\
&\quad \underset{R_8^b.a_4>R_9^c.a_7}{\bowtie} (\pi_{R_9^c.a_1,\ R_9^c.a_3,\ R_9^c.a_6,\ R_9^c.a_7}(\sigma_{R_9^c.a_1=8\ \wedge\ (R_9^c.a_4\leq 101\ \vee\ R_9^c.a_2=23)}(R_9^c))) \\
&\quad \underset{R_9^c.a_6\leq R_5^d.a_4}{\bowtie} (\pi_{R_5^d.a_2,\ R_5^d.a_4,\ R_5^d.a_6}(\sigma_{R_5^d.a_3\geq 54}(R_5^d))).
\end{aligned}$$

The test queries in the experiments were generated by randomly choosing a table at each site, randomly choosing a set of columns to project from each table, randomly choosing the types of relevant selection and join conditions, and randomly choosing the columns and constants in the qualification conditions from their respective allowed domains.

There are a number of strategies to perform such a query. Figure 6 shows two of them. Given a system environmental state, we can use the cost estimates given by the multistate cost models for the system environmental state to determine the optimal

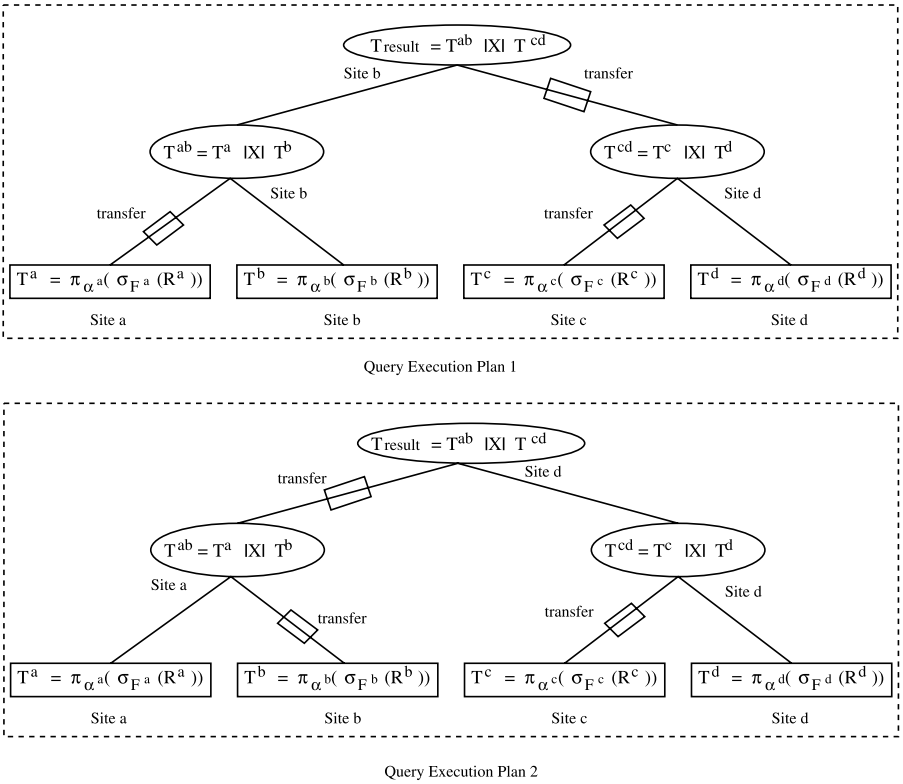

Fig. 6 Examples of execution plan versions

execution strategy from many alternatives. If the contention space partitioning technique is applied, the optimal execution strategy chosen for a representative system environmental state yields a version in the corresponding query execution plan. The versions for all representative system environmental states comprise the query execution plan with multiple versions for the given query. If the cost error controlling technique is used, the optimal execution strategy is chosen for a system environment state and examined to see if it is also acceptable to the neighboring system environmental states. The set of selected optimal execution strategies acceptable to all the system environmental states comprise the query execution plan with multiple versions for the given query. At run time, the current system environmental (contention) level can be determined by the measured/estimated costs of the small probing queries run at participating local sites. From the current system environmental level, we can determine the corresponding system environmental state. The execution plan version covering the given system environmental state is invoked to run the given query.

Figure 7 shows the comparison of costs for a set of random global queries of form (7) run in system environmental states determined by random system environmental levels. The following costs for each query were compared: (i) the cost using the best version of the execution plan with multiple (a random number $2 \leq n \leq 8$) versions generated by the contention space partitioning technique discussed in Sect. 3.1,

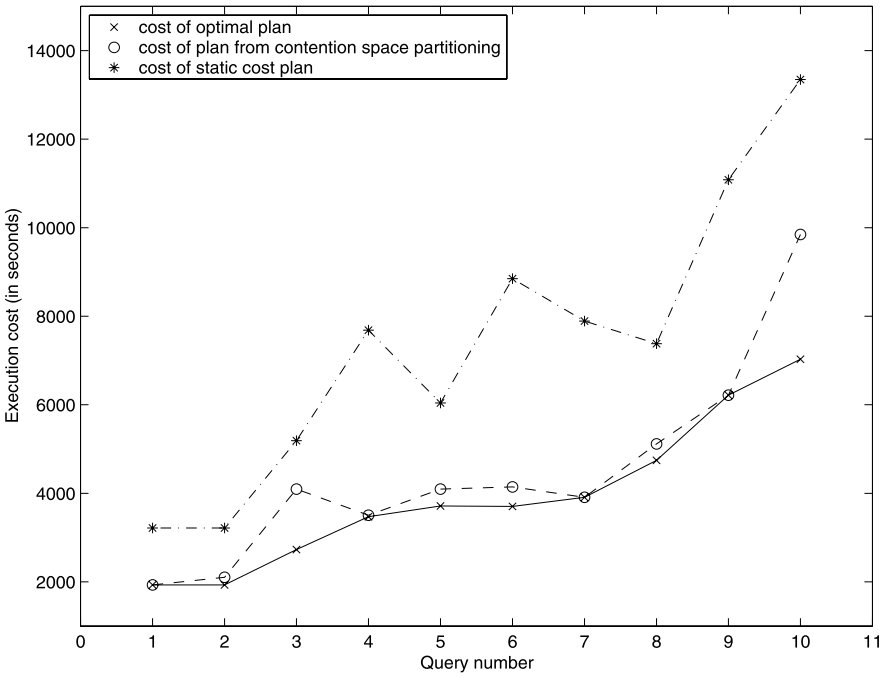

Fig. 7 Query performance comparison for the contention space partitioning technique

(ii) the cost using the execution plan generated from the static cost models (i.e., the ones assuming a static environment without considering dynamic factors), and (iii) the cost using 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 corresponding static execution plans. The performance of the execution plans with multiple versions generally well approximates the performance of the corresponding optimal execution plans for the queries in the running system environmental states. Note that the cost of a query depends on a number of factors such as the size(s) of input table(s), the sizes of intermediate and result tables, and the system environmental contention level. Hence the execution of various queries in different environments may incur different costs. An efficient execution plan attempts to minimize the cost of a query under a given condition/environment.

Figure 8 shows that the performance of a query execution plan with multiple versions is usually getting increasingly 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 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 evaluate 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 may remain

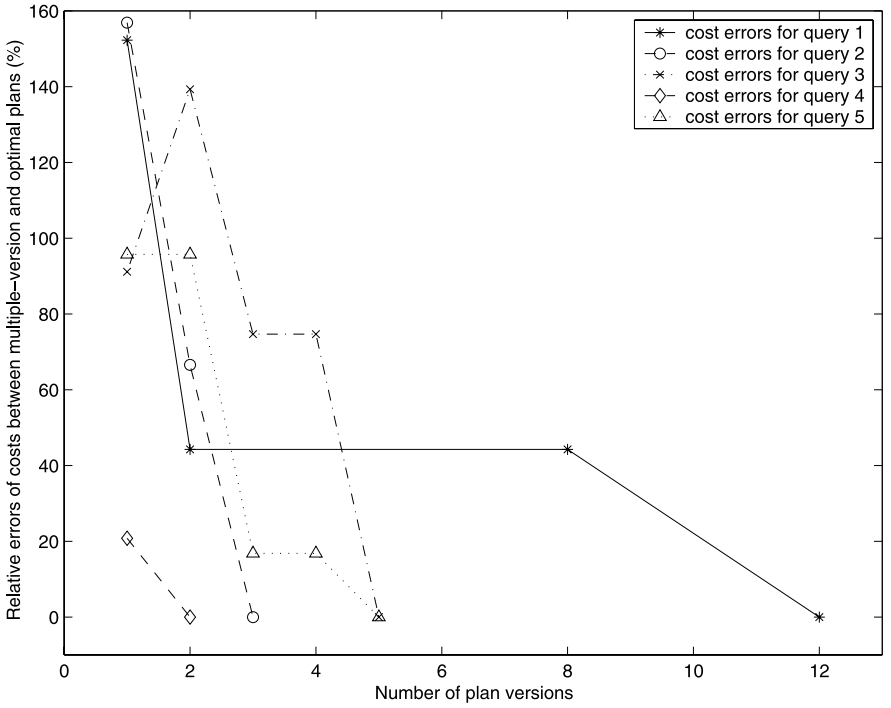

Fig. 8 Query performance improvement as the number of versions in a plan increases

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

Figure 9 shows the result from another experiment, in which we assumed that only one version (i.e., the 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 become smaller (i.e., the maximum distance between the center and any point in the region is smaller), a better performance is expected for the execution plan with multiple versions.

The previous experiments were conducted in a dynamic multidatabase environment where every local contention level has an equal chance to occur at its corresponding site; that is, the contention level at each local site follows the uniform distribution. We also examined the effectiveness of the contention space partitioning technique with the extension suggested in Sect. 3.2 for non-uniform environments. In the experiment, we considered dynamic multidatabase environments with four local sites whose contention levels follow the normal distribution, the Erlang distribution, the Cauchy distribution, and the uniform distribution, respectively. A set of random

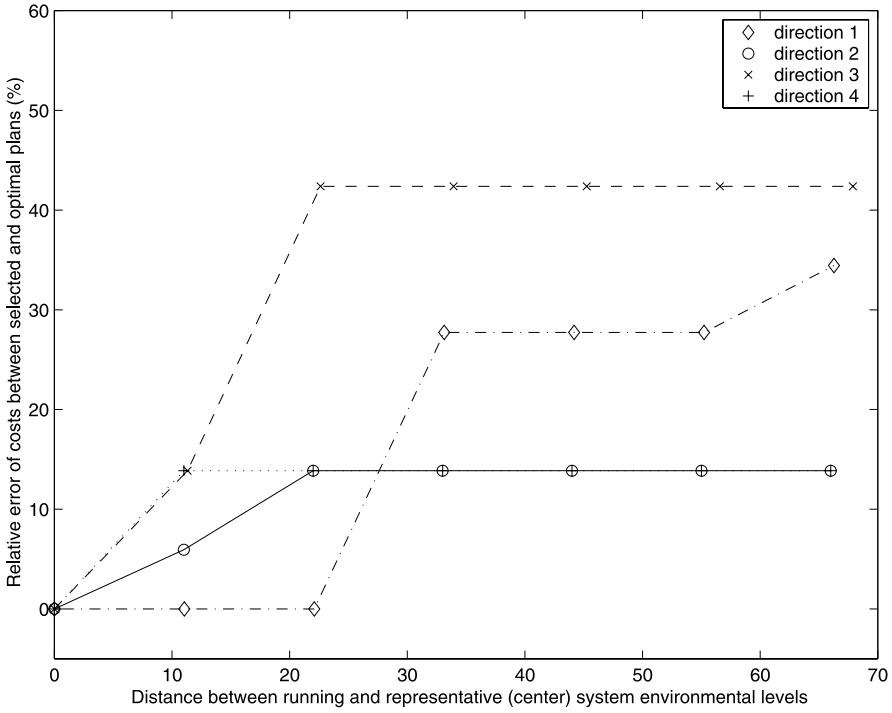

Fig. 9 Effect of running contention level departed from the region center on query performance

global queries of form (7) was used in the experiment. The following costs for each query were compared: (i) the cost using the best version of the execution plan with multiple (a random number $2 \leq n \leq 8$) versions generated by the contention space partitioning technique extended for non-uniform distributions in Sect. 3.2, (ii) the cost using the best version of the execution plan with multiple (the same number n) versions generated by the original contention space partitioning technique (i.e., assuming the uniform distribution for all local sites), and (iii) the cost using the optimal execution plan for the running system environmental state determined by a given running system environmental level. Figure 10 shows the comparison result. From the figure we can see that the extension suggested in Sect. 3.2 is quite effective for multidatabase environments exhibiting non-uniform distributions of contention levels at local sites. The extended technique is usually more efficient than the original technique in such environments. In fact, the performance of an execution plan generated by the former well approximates the performance of the optimal execution plan in most cases.

The above non-uniform multidatabase environments were also used to test the effectiveness of the cost error controlling technique. Note that the cost error controlling technique works for a dynamic environment with any distribution. Testing it in non-uniform environments can demonstrate its robustness. A set of random global queries of form (7) was used in the experiments. The following costs for each query were compared: (i) the cost using the best version of the execution plan with multi-

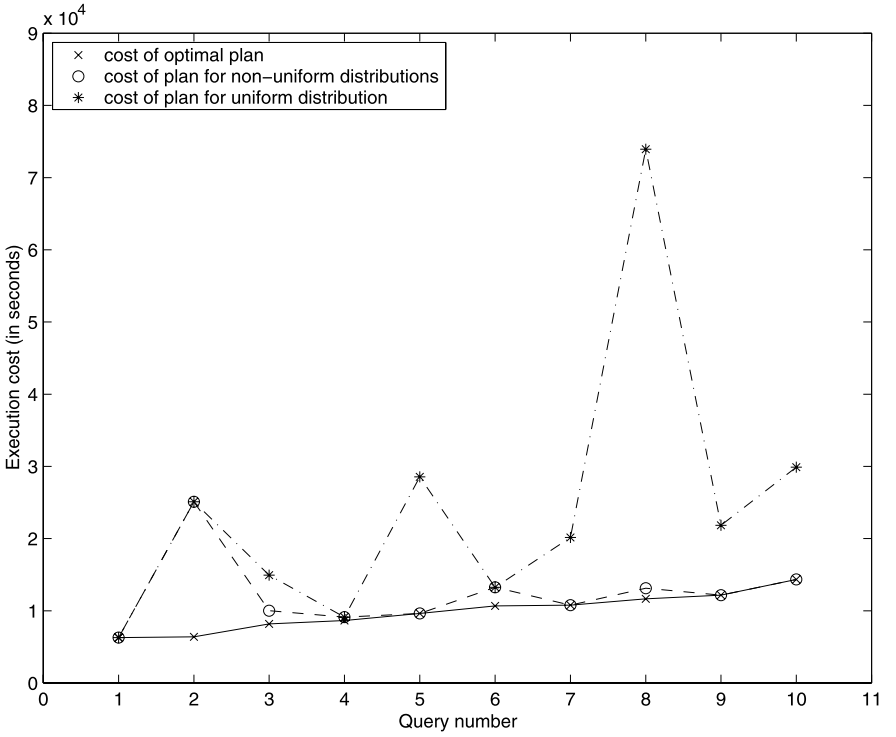

Fig. 10 Query performance comparison for the contention space partitioning technique in non-uniform environments

ple versions generated by the cost error controlling technique (with random threshold values between 0.0 and 0.3, i.e., up to 30% error tolerance), (ii) the cost using the best version of the execution plan with multiple versions generated by the contention space partitioning technique (for non-uniform distributions), (iii) the cost using the execution plan generated from the static cost models, and (iv) the cost using the optimal execution plan for the running system environmental state determined by a given system environmental level. To have a fair comparison between (i) and (ii), we let both techniques generate the same number of versions for an execution plan for the same test query. Specifically, for a given test query, we applied the cost error controlling technique with a random threshold value between 0.0 and 0.3 to generate an execution plan with multiple versions. Once we knew the number m of versions in the generated plan, we employed the contention space partitioning technique to generate a plan with m versions for the test query. Figure 11 shows the comparison result. From the figure we can see that the execution plans generated by the cost error controlling technique are usually more efficient than those generated by the contention space partitioning technique when the numbers of versions in the corresponding execution plans are the same. The performance of queries using the former plans are predictable in the sense that the error between the cost of the selected plan version and the cost of the optimal version is controlled using a given tolerance. The figure

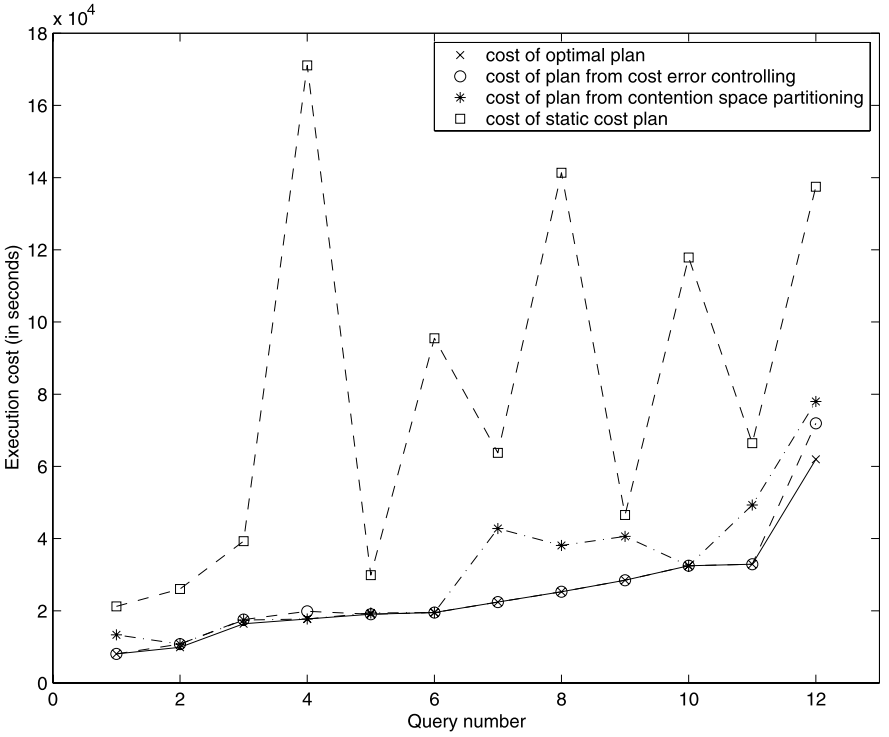

Fig. 11 Query performance comparison for the cost error controlling technique

also shows that the execution plans generated by the previous two techniques are significantly more efficient than the corresponding static execution plans.

Although the cost error controlling technique uses a given tolerance to directly control the cost error, it has no direct control over the number of versions for an execution plan. In general, the smaller the error tolerance, the more the number of versions needed for an execution plan to cover all system environmental states in the environment. Figure 12 shows such a relationship between the error tolerance and the number of versions of an execution plan for a typical test query. The number of versions in an execution plan is indirectly controlled by the error tolerance. In contrast, the contention space partitioning technique directly controls the number of versions in an execution plan, which in term controls the query performance as shown in Fig. 8. Hence, the cost error controlling technique is preferred when query performance needs to be ensured, while the contention space partitioning technique is preferred when space (for keeping plan versions) presents a constraint.

Our experimental results demonstrate that the techniques proposed in this paper are quite promising in performing global query optimization for dynamic multidatabase environments.

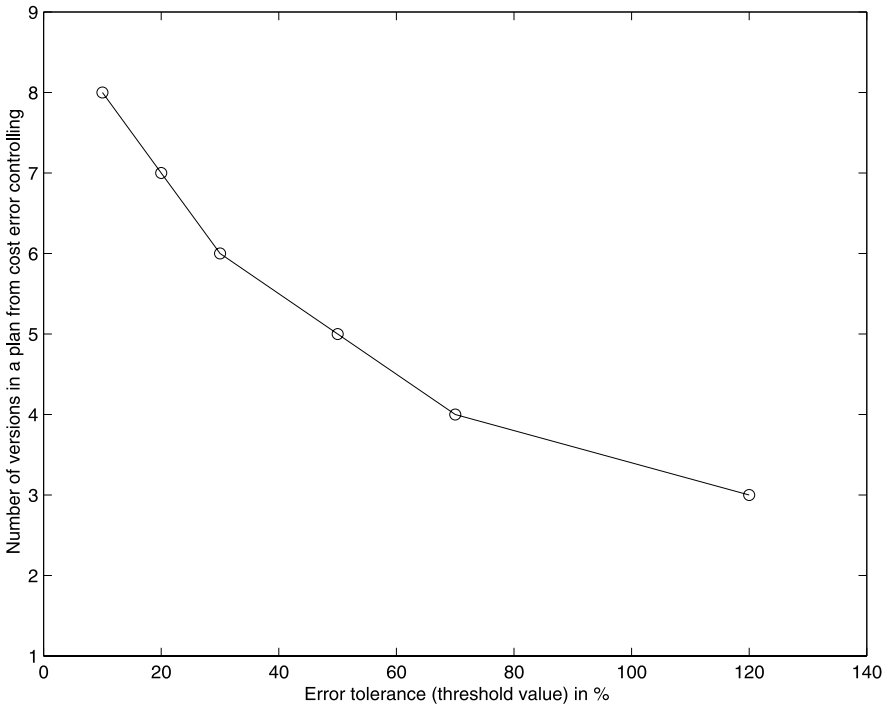

Fig. 12 Effect of error tolerance on query performance

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 query performance 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 optimization work performed at run time in this case is negligible if not nothing at all since 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 generated 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 techniques proposed in this paper aim to overcome the shortcomings of existing static and dynamic query techniques in MDBSs. They take 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 its dynamic local site.

Based on the cost estimates, our techniques generate an execution plan with multiple versions at compile time, one for each selected representative system environmental state. The two presented techniques, i.e., the contention space partitioning one and the cost error controlling one, differ in the way for generating the multiple versions of an execution plan. The former partitions the space of system environmental levels in a dynamic multidatabase environment into a given number of smaller regions and generate a representative execution plan version for each region. The latter systematically generates execution plan versions for a dynamic multidatabase environment so that the error between the cost of each generated execution plan version and the cost of the corresponding optimal execution plan for the underlying system environmental state is controlled using a given tolerance. When optimization time and space are highly restricted, the former technique is preferred. When query performance needs to be ensured, the latter is more favorable. Our experiments demonstrate that the proposed optimization techniques are 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.

Acknowledgements The authors would like to thank Per-Åke (Paul) Larson, Tamer M. Ozsu, Guy M. Lohman, Yuqing Song, Roberto Kampfner, Yu Sun and Satyanarayana Motheramgari for their valuable comments and suggestions for some work reported in this paper. We are also grateful to the anonymous reviewers for their careful reviews and constructive suggestions for improving the paper. Some preliminary work of this paper was presented at ICEIS'03 [43].

References

1. Adali, S., et al.: Query caching and optimization in distributed mediator systems. In: Proc. of ACM SIGMOD Conf., pp. 137–148 (1996)
2. Amsaleg, L., Franklin, M.J., Tomasic, A., Urhan, T.: Scrambling query plans to cope with unexpected delays. In: Proc. of Int. Conf. on Paral. and Distr. Inf. Syst., pp. 208–219 (1996)
3. Amsaleg, L., et al.: Scrambling query plans to cope with unexpected delays. In: Proc. of Int. Conf. on Paral. and Distr. Inf. Syst., pp. 208–219 (1996)
4. Arasu, A., Babcock, B., et al.: STREAM: the Stanford stream data manager. IEEE Data Eng. Bull. **26**(1), 19–26 (2003)
5. Bouganim, L., et al.: Dynamic query scheduling in data integration systems. In: Proc. of IEEE Int. Conf. on Data Eng., pp. 425–434 (2000)
6. Chandrasekaran, S., Cooper, O., et al.: TelegraphCQ: continuous dataflow processing for an uncertain world. In: Proc. of CIDR Conf., pp. 1–12 (2003)
7. Chandrasekaran, S., Cooper, O., et al.: TelegraphCQ: continuous dataflow processing. In: Proc. of ACM SIGMOD Conf., pp. 668 (2003)
8. Chen, A.L.P.: Outerjoin optimization in multidatabase systems. In: Proc. of Int. Symp. on DB in Paral. and Distr. Syst., pp. 211–218 (1990)
9. Chen, C.-M., Sun, W., Rishe, N.: Performance comparison of three alternatives of distributed multidatabase systems: a global query perspective.. In: Proc. of Int. Conf. on Performance, Computing and Communications, pp. 53–59 (1998)
10. Cheng, X., Dong, G., Lau, T., Su, J.: Data integration by describing sources with constraint databases. In: Proc. of IEEE Int. Conf. on Data Eng., pp. 374–381 (1999)
11. Reiss, F., Hellerstein, J.M.: Lifting the burden of history from adaptive query processing. In: Proc. of VLDB Conf., pp. 948–959 (2004)
12. Du, W., et al.: Query optimization in heterogeneous DBMS. In: Proc. of VLDB Conf., pp. 277–291 (1992)

13. Du, W., Shan, M.C., Dayal, U.: Reducing multidatabase query response time by tree balancing. In: Proc. of ACM SIGMOD Conf., pp. 293–303 (1995)
14. Evrendilek, C., Dogac, A., Nural, S., Ozcan, F.: Multidatabase query optimization. Distrib. Parallel Databases **5**(1), 77–113 (1997)
15. Garcia-Molina, H., Labio, W., Yerneni, R.: Capability-sensitive query processing on Internet sources. In: Proc. of IEEE Int. Conf. on Data Eng., pp. 50–59 (1999)
16. Gardarin, G., et al.: Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In: Proc. of VLDB Conf., pp. 378–389 (1996)
17. Goni, A., Bermudez, J., Blanco, J.M., Illarramendi, A.: Using reasoning of description logics for query processing in multidatabase systems. In: Proc. of the 3rd Workshop on Knowl. Repres. Meets DB, pp. 1–6 (1996)
18. Hsu, C.-N., Knoblock, C.A.: Reformulating query plans for multidatabase systems. In: Proc. of ACM CIKM Conf., pp. 423–432 (1993)
19. Hsu, C.-N., Knoblock, C.A.: Semantic query optimization for query plans of heterogeneous multidatabase systems. IEEE Trans. Knowl. Data Eng. **12**(6), 959–978 (2000)
20. Ives, Z.G., Florescu, D., Friedman, M.: An adaptive query execution system for data integration. In: Proc. of ACM SIGMOD Conf., pp. 299–310 (1999)
21. Ives, Z.G., Levy, A.Y., Weld, D.S.: Adaptive query processing for Internet applications. IEEE Data Eng. Bull. **23**(2), 19–26 (2000)
22. Josifovski, V., Katchaounov, T., Risch, T.: Optimizing queries in distributed and composable mediators. In: Proc. of Int. Conf. CoopIS, pp. 291–302 (1999)
23. Josinski, H.: Dynamic query optimization and query processing in multidatabase systems. In: Int. Conf. on Extending DB Tech. Ph.D. Workshop, pp. 1–4 (2000)
24. Kang, S., Moon, S.: Global query management in heterogeneous distributed database systems. Microproces. Microprogram. **38**, 377–384 (1993)
25. Lee, C., Chen, C.J.: Query optimization in multidatabase systems considering schema conflicts. IEEE Trans. Know. Data Eng. **9**(6), 941–955 (1997)
26. Lee, J.-O., Baik, D.-K.: SemQL: a semantic query language for multidatabase systems. In: Proc. of ACM CIKM Conf., pp. 259–266 (1999)
27. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: Proc. of VLDB Conf., pp. 226–251
28. Lim, E.-P., et al.: An algebraic transformation framework for multidatabase queries. Distrib. Parallel Databases **3**, 273–307 (1995)
29. Motwani, R., Widom, J., et al.: Query processing, resource management, and approximation in a data stream management system. In: Proc. of CIDR Conf., pp. 1–12 (2003)
30. Naacke, H., Gardarin, G., Tomasic, A.: Leveraging mediator cost models with heterogeneous data sources. In: Proc. of IEEE Int. Conf. on Data Eng., pp. 351–360 (1998)
31. Otsuka, S., Miyazaki, N.: An incomplete database approach to global query processing. In: Proc. of the 12th Int. Conf. on Inf. Networking, pp. 337–342 (1998)
32. Ozcan, F., Nural, S., Koksal, P., Evrendilek, C.: Dynamic query optimization in multidatabases. IEEE Data Eng. Bull. **20**(3), 38–44 (1997)
33. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C: the Art of Scientific Computing, 2nd edn. Cambridge University Press, Cambridge (1992)
34. Rahal, A., Zhu, Q., Larson, P.-Å.: Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. VLDB J. **13**(2), 162–176 (2004)
35. Reiss, F., Hellerstein, J.M.: Data Triage: an adaptive architecture for load shedding in TelegraphCQ. In: Proc. of IEEE Int. Conf. on Data Eng., pp. 155–156 (2005)
36. Roth, M.T. et al.: Cost models DO matter: providing cost information for diverse data sources in a federated system. In: Proc. of VLDB Conf., pp. 599–610 (1999)
37. Subramanian, D.K., Subramanian, K.: Query optimization in multidatabase systems. Distrib. Parallel Databases **6**(3), 183–210 (1998)
38. Tsai, P.S.M., Chen, A.L.P.: Optimizing entity join queries when data transmission cost dominates. Data Knowl. Eng. **22**, 283–308 (1997)
39. Tomasic, A., Raschid, L.: Scaling access to heterogeneous data sources with DISCO. IEEE Trans. Knowl. Data Eng. **10**(5), 808–823 (1998)
40. Urhan, T., Franklin, M.J., Amsaleg, L.: Cost-based query scrambling for initial delays. In: Proc. of ACM SIGMOD Conf., pp. 130–141 (1998)
41. Vassalos, V., Papakonstantinou, Y.: Describing and using query capabilities of heterogeneous sources. In: Proc. of VLDB Conf., pp. 256–265 (1997)

42. Wei, C.-P., Sheng, O.R.L., Hu, P.J.-H.: Fuzzy statistics estimation in supporting multidatabase query optimization. Electron. Commer. Res. **2**(3), 287–316 (2002)
43. Zhu, Q., Haridas, J., Hou, W.-C.: Global query optimization based on multistate cost models for a dynamic multidatabase system. In: Proc. of Int. Conf. on Enterprise Infor. Syst., pp. 144–155 (2003)
44. Zhu, Q., Larson, P.-Å.: A query sampling method for estimating local cost parameters in a multidatabase system. In: Proc. of IEEE Int. Conf. on Data Eng., pp. 144–153 (1994)
45. Zhu, Q., Larson, P.-Å.: Building regression cost models for multidatabase systems. In: Proc. of Int. Conf. on Paral. and Distr. Inf. Syst., pp. 220–231 (1996)
46. Zhu, Q., Larson, P.-Å.: Global query processing and optimization in the CORDS multidatabase system. In: Proc. of 9th Int. Conf. on Paral. and Distr. Comp. Syst., pp. 640–646 (1996)
47. Zhu, Q., Larson, P.-Å.: A fuzzy query optimization approach for multidatabase systems. Int. J. Uncertain. Fuzziness Knowl. Based Syst. **5**(6), 701–722 (1997)
48. Zhu, Q., Larson, P.-Å.: Solving local cost estimation problem for global query optimization in multidatabase systems. Distrib. Parallel Databases **6**(4), 373–420 (1998)
49. Zhu, Q., Sun, Y., Motheramgari, S.: Developing cost models with qualitative variables for dynamic multidatabase environments. In: Proc. of IEEE Int. Conf. on Data Eng., pp. 413–424 (2000)
50. Zhu, Q., Larson, P.-Å.: Classifying local queries for global query optimization in multidatabase systems. Int. J. Cooperative Inf. Syst. **9**(3), 315–355 (2000)