Multidatabase Query Optimization: Issues and Solutions

Hongjun Lu *Beng-Chin Ooi* *Cheng-Hian Goh*
Department of Information Systems and Computer Science
National University of Singapore
10 Kent Ridge Crescent, Singapore 0511
Internet: {luhj,ooibc,gohch}@iscs.nus.sg

Abstract

Despite the interest in multidatabase systems, research in multidatabase query optimization (MQO) has been scarce. Many researchers perceive the difficulty to be the lack of reliable cost estimates for autonomous component databases. Consequently, some have suggested that this problem degenerates to a distributed query optimization (DQO) problem once cost model coefficients for component databases are found. In this paper, we argue that autonomy and heterogeneity of component databases give rise to a number of issues which make MQO a distinct problem from DQO. Consequently, existing solutions for DQO need to be re-evaluated in the light of these issues. The design of a multidatabase query optimizer, which accounts for the issues highlighted, is discussed. This provides a framework within which further research in MQO can be carried out.

1 Introduction

Multidatabase management systems (MDBMS) [5] enable data sharing among heterogeneous local databases (called *component databases*) and thus provide interoperability required by diverse applications. Despite the interests in multidatabase research, work in *multidatabase query optimization (MQO)* has been scarce compared to other issues (such as schema integration). Several reasons contribute to this. First, some researchers perceived MQO to be a difficult problem which may not lead to a rewarding payoff. On the other hand, there are those who feel that MQO is no different from *distributed query optimization (DQO)* once relevant information on cost functions and database statistics can be found.

In this paper, we claim that existing solutions for DQO cannot be trivially extended for MQO since issues arising from autonomy and heterogeneity of component databases make the two problems distinctly different. Having identified these issues, we outline the design of a multidatabase query optimizer which also provides a framework for further research in MQO.

2 Beyond Distributed Query Optimization

While both MQO and DQO have the same objectives of reducing processing costs and/or response time, the two problems have vastly different characteristics in many aspects. In this section, we highlight these differences which stem from autonomy and heterogeneity of component databases in a multidatabase system.

2.1 Site Autonomy

Site autonomy [1] in multidatabases refers to the situation whereby each component DBMS retains complete control over local data and processing. This has a number of implications for query optimization in a multidatabase system.

First, *communication autonomy* in multidatabase systems means that component sites independently determines what information it will share with the global system, what global request it will service, when it will participate in the multidatabase, and also when it will stop participating. This adds to the complexity of query processing and optimization since any component database system may terminate its services without any advance notice.

Second, *design autonomy* implies that component DBAs are free to optimize local access paths and query processing methods to satisfy local user requirements without any obligation to inform the MDBMS of these changes. Consequently, statistical information which is needed for effective global query optimization are not readily available and may not remain accurate as component systems evolve over time. In a recent paper [4], Du et al. suggested that estimates of the cost

0-8186-3710-2/93 $3.00

model functions and database statistics can be obtained by calibrating the component DBMSs using a synthetic database. Their approach appears viable for proprietary and conforming DBMSs but remains inadequate when component DBMSs are non-conforming.

Third, *execution autonomy* results in the situation whereby the global system interfaces with the component DBMS at the user level, and hence is not able to influence how query processing is being carred out in the component database. This means that there are now no opportunity for low-level cooperation across systems and hence primitive query processing techniques proposed for distributed databases may no longer be applicable. For example, the semijoin operation has been proposed in order that data transmission between sites can be reduced. When two relations R and S at two different sites, A and B, are joined using the semijoin method, join column values of R is filtered out and sent to site B to fetch matching tuples in relation S. The matching tuples retrieved are sent back to site A and merged with the corresponding R-tuples to complete the join. In the context of distributed database management systems (DDBMSs), this method is quite effective because of the facilities provided by the underlying system environment. In fact, the R-tuples (or just the TIDs of those tuples) retrieved in site A can be staged in memory to wait for the incoming S-tuples and the local processing time (disk access of R-tuple) can be reduced (such as implemented in System R*, the fetch-as-needed join method). In MDBMS, it is hard to implement semijoins efficiently in the same way. The semijoin becomes three subqueries: the first query is to select the join column values of R at site A and send the results to site B; the second query is to join these values with relation S at site B and send the results to site A; and the third query is to join the results with R at site A again to generate the final join results. Although the semi-join may filter out the unnecessary tuples and reduce the data transferred, the local processing time may increase dramatically because relation R has to be scanned twice.

2.2 System Heterogeneity

Unlike distributed database systems, component sites in a multidatabase system are not homogeneous. *System heterogeneity* may occur at several levels: component databases may reside on computer systems with different architectures, they may be connected via different types of network supporting different communication protocols, or they may support different data models.

In query optimization for distributed database systems, it is assumed that component sites are equal in terms of their processing capability. This assumption no longer seems reasonable in the context of multidatabase systems since component sites may vary drastically in terms of their availability and processing costs. For instance, a component DBMS may not even be a first class DBMS and thus lack important DBMS features. Some systems also will not support intermediate processing due to the constraints placed on utilization of system resources. Obviously, these constraints need to be conveyed to the multidatabase query processor since they have a significant impact on the generation of the query processing plan.

2.3 Semantic Heterogeneity

In a distributed database system, the same real world object may be represented in more than one component sites but these representations are always structurally compatible. On the other hand, semantic heterogeneity in a multidatabase system results in the same data being represented differently in different component databases. Until recently, this problem has been mostly addressed from a schema integration perspective. We suggest that semantic heterogeneity has an important impact on query processing and optimization.

Consider the *inter-database instance identification problem* which is first highlighted in [7]. In order to match the same entity instance stored in two relations at distinct component sites, some common identifier must exist between them. In many instances, this may not be the case. A brute force solution (which is impractical for large databases) is to store the synonyms of all identifiers in a table and use this for conflict resolution. A more promising approach, described in [7], is to make use of entity properties common to both databases in deciding whether or not two tuples actually refer to the same real world entity. From a query processing perspective, this implies that new query processing methods need to be introduced and it is likely that these will not be supported by the component DBMS. For instance, Chatterjee and Segev [2] have described a new join operator, called *Entity Join (E-join)*, which accomplishes the join of two relations by comparing *useful* attributes which are not necessarily structurally identical.

Recently, it has been pointed out that comparing data originating from different databases is only meaningful when their *contexts* are being taken into con-

sideration [6]. For example, it is not meaningful to compare share prices from New York Stock Exchange and the European markets since the currencies used in reporting their prices are different. Hence, data interchange must be preceded by *context mediation* during which data from a different context are converted to an equivalent representation in the current context. Obviously, context mediation can be an expensive operation. The cost incurred as a result of context mediations must be taken into consideration during the generation of the query plan since different orderings of subqueries can result in drastic variations in the cost of context mediations.

3 Design of a Multidatabase Query Optimizer

Following Dayal [3], we identify four subtasks in processing a global multidatabase query Q. First, Q (which is defined on the global schema) is mapped to an equivalent query Q' defined over the component schemas. Second, a *query execution plan (QEP)* P comprising of single-site queries is generated from Q'. Third, the single site queries from step 2 is sent to each local site in accordance to the QEP and each is subjected to local access path optimization. Finally, the optimized query at each site is translated into an equivalent query over the local schema in the data language of the component DBMS.

In this paper, we define *multidatabase query optimization (MQO)* to mean the generation of a query exection plan P for a given query Q' defined over the collection of local schemas. The QEP P defines

- the single-site queries which will be generated;
- the shipment of intermediate results between sites; and
- the postprocessing steps which combines the results from local processing steps.

The architecture of a proposed multidatabase query optimizer is shown in Figure 1. A global query is first parsed and then decomposed (by the *query decomposer*) into query units which are represented in the form of a *query unit graph*. The *plan generator* constructs subqueries from the query unit graph and estimates their execution costs. The query plan with the minimum estimated cost will be sent to the *dispatcher* who will coordinate the execution of the query. The *execution monitor* collects the statistics about subquery execution and send to the *statistics manager*,

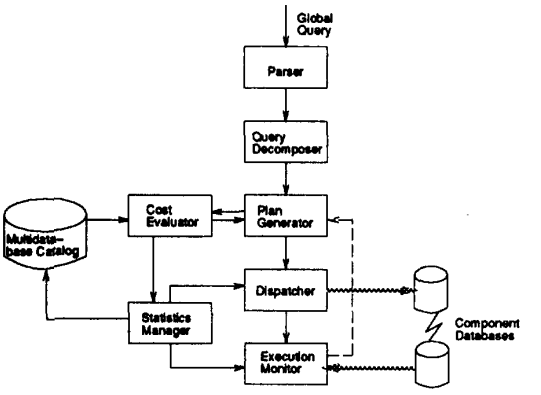

Figure 1: Architecture for a multidatabase query optimizer.

which will update the *multidatabase catalog* if necessary.

3.1 The Query Decomposer

The function of the query decomposer is to decompose a multidatabase query into *query units*. A query unit corresponds to primitive operations needed to process a query, such as selection, projection, or join on available data at a single database site. The decomposition can be accomplished according to the following heuristics:

1. Selections and projections on single relations form query units by themselves.
2. Joins and other operations involving only relations stored at the same component database also form query units.
3. When a relation is the union of relations at different component databases, query units are formed for each site.
4. For a join (or other operation) involving two different databases, the relations in the join condition are replaced with query units resulting from the query units that retrieve the relation (or part of it) from the original database.

The basic principle here is to decompose a query to the finest level in order to explore all possible execution plans.

Example 1 Consider the query

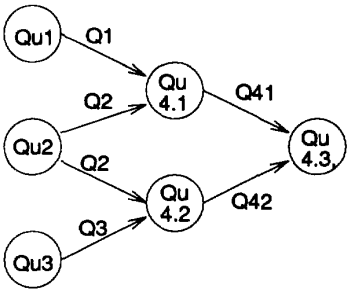

Figure 2: Query graph corresponding to Example 1.

```
select *
from T1, T2, T3
where T1.c1 = T2.c2
and T2.c3 = T3.c4
and T1.cx = 5
and T2.cy = 7
and T3.cz = 6;
```

where T1, T2 and T3 are stored in three different database systems D1, D2, and D3 respectively.

This query can be decomposed into the following set of query units.

```
Qu1:   select * into Q1
       from D1.T1
       where D1.T1.cx = 5;
Qu2:   select * into Q2
       from D2.T2
       where D2.T2.cy = 7;
Qu3:   select * into Q3
       from D3.T3
       where D3.T3.cz = 6;
Qu4.1: select * into Q41
       from D1.Q1, D2.Q2
       where Q1.c1 = Q2.c2;
Qu4.2: select * into Q42
       from D3.Q1, D2.Q2
       where Q2.c3 = Q3.c4;
Qu4.3: select * from Q41, Q42
       where Q41.c2 = Q42.c2
       and Q41.c2 = Q42.c2
       and Q41.c3 = Q42.c3;
```

These six query units can be represented by the query unit graph in Figure 2. □

3.2 The Plan Generator

Given a query unit graph, the plan generator constructs the possible query execution plans (QEPs) and the expected response time and/or processing cost. Subqueries for each QEP are formed by grouping adjacent query units in the query unit graph together. This grouping process is guided by the cost functions as well as heuristics which help to reduce the search space.

Example 2 Based on the query execution graph in Figure 2, a number of QEPs can be identified. The simplest QEP is as follows: (i) broadcast Q2 to the database D1 and D3, (ii) execute $Q4.1$ and $Q4.2$ at D1 and D2 respectively, and (iii) send the result of $Q4.2$ to D1 where $Q4.3$ is executed. Alternatively, the last three query units $Q4.1$, $Q4.2$ and $Q4.3$ can be executed as a single database query by transfering Q2 and Q3 to database D2 and execute $Qu4$ as a single database query at D2 since all Q1, Q2 and Q3 are available:

```
Qu4:   select * from Q1, Q2, Q3
       where Q1.c1 = Q2.c2
       and Q2.c3 = Q3.c4;
```

Note that the first three queries can be processed at the three different DBMS in parallel but the last one has to wait for the results from the first three subqueries.

In fact, more alternatives are available. For example, we can group $Qu2$ and $Qu4.1$ together so that the result of $Qu1$ is sent to D2 after it is being evaluated. The query to be executed at D2 now becomes

```
Qu2': select * into Q12
      from D2.Q1, D2.T1
      where Q1.c1 = T2.c1
      and T2.cy = 7;
```

The result of $Qu2'$ can now be sent to D3 and the execution of the subquery

```
Qu3': select * from Q12, T3
      where Q12.c3 = T3.c4
      and T3.cz = 6;
```

yields the final result. □

3.3 The Cost Evaluator

The cost evaluator interacts with the plan generator during the plan generation process to identify those QEPs which meets the optimization objective specified by the user issuing the query. In most instances,

this means the minimization of either the *response time* or the *processing cost*. When an execution plan is generated, it is passed to the cost evaluator which provides an estimated cost based on a *cost model* and available statistics of component databases. A plan which satisfies the optimization objective is finally chosen as the execution plan for the query and sent to the dispatcher who coordinates the execution of the plan among the participating component DBMS.

Example 2 in the previous section illustrates that there could be a number of QEPs for even a simple multidatabase query. Moreover, these plans may have different number of subqueries and hence different number of invocations of component DBMS. The parallelism and the size of data to be transferred among the participating DBMS may also differ. Furthermore, the frequency and cost of context mediations will certainly be different as well. All of these differences contribute to different performances of QEPs which can be generated. In the given example, subquery $Qu1$, $Qu2$ and $Qu3$ can be executed concurrently at three different database sites, and so can $Qu4.1$ and $Qu4.2$. Concurrently executing a number of subqueries can help reduce the response time, provided that they do not eventually lead to costly data transfers or context mediations. On the other hand, sequential execution of the subqueries may lead to better total processing time. For example, subquery $Qu2'$ combines the selection on T2 (T2.cy > 7) and the join with Q1 into one query. The selection can therefore be performed during the join. The cost of scanning over T2, and hence the total processing cost, is reduced. Minimizing the total processing time may be important in those instances in which a user does not mind waiting a little longer in order that he can be charged less for accessing the databases.

3.4 The Subquery Execution Monitor

To overcome the difficulty arising from the autonomy of component DBMS, subquery executions must be monitored closely. This is accomplished by adopting a proactive approach to the generation of QEP: i.e., instead of generating the whole QEP before query execution commences, the query optimizer determines the next step in the query execution sequence only after the previous step has been completed. This strategy has the advantage of providing the query optimizer with more accurate information about the data size of the intermediate results (which is one important parameter used in estimating the processing costs), and hence seems more attractive in a multidatabase environment.

We introduce a new function to the multidatabase query optimizer: subquery execution monitoring. Basically, the function requires the query optimizer to collect the completion time for subqueries sent to the component DBMS. A subquery has an estimated completion time that forms the basis for global query optimization. When a component DBMS completes the execution of a subquery, the monitor collects information about the executed subquery and sent to the statistics manager. This information includes the subquery type, the component DBMS involved, the expected response time and the actual response time. The related statistics which are present in the multidatabase catalog is then modified accordingly. If the execution of subqueries are much more costly than expected, the execution monitor may communicate with the plan generator and the dispatcher and the unexecuted portions of the plan may be modified. If possible, subqueries to be executed at the slow site could be moved to other sites. The processing plan can also be changed so that the more responsive sites may be assigned more tasks.

4 Cost Estimation for MQO

The efficacy of a query optimizer, whether for centralized, distributed, or multidatabase systems, depends largely on

1. the optimization strategy which is adopted;
2. the extend to which the cost model mirrors the actual response time or processing costs; and
3. accuracy of the statistics used for arriving at the costs of alternative QEPs while adopting the said cost model.

Contrary to what is commonly assumed, existing algorithms and optimization strategies employed for DQO cannot be readily applied for MQO. For instance, a common optimization strategy for distributed database systems is to make extensive use of semijoins to reduce the amount of data transmissions between component sites. This no longer seems attractive in a multidatabase context due to reasons we have explained earlier in section 2.1. Furthermore, these strategies invariably do not take into consideration the cost incurred by context mediations, nor the effect of system heterogeneity in generating the query plan. Similarly, the non-availability of reliable estimates for subquery execution cost and intermediate

data size places severe constraints on the effectiveness of the optimization strategies. It is therefore essential for the multidatabase query optimizer to gather vital statistics of component databases while incurring minimal overhead, and to develop query optimization strategies which are more tolerant of fuzzy input information. We address some of these issues in greater detail in the rest of this section.

4.1 The Cost Model

The cost of executing a global multidatabase query comprises many components, arising from

- generation of a query plan;
- invocation of component DBMS;
- processing of subqueries;
- transferring intermediate results among participating component DBMS;
- context mediations; and
- assembly of global query results.

Unlike the case in distributed query optimization, the cost of subqueries cannot be easily determined since the multidatabase query optimizer has no information on the profile of the database, the access paths, or the access methods which are supported by the component DBMS. Similarly, the size of intermediate results are generally unknown and hence communication and context mediation costs are also hard to determine.

We propose the adoption of a *fuzzy* approach in circumventing the above difficulties. Under this approach, subqueries to a component databases are classified according to their complexity into various categories which fall on a linear scale. The execution of queries in each category is sampled againest the component database, and the costs are recorded to form the cost estimates for queries in this category. Note that, with proper sampling technique, this cost estimation takes the system characteristic of the component DBMS into consideration. In a similar manner, the component DBMSs are also classified. The optimizer can make use of cost estimates for similar systems in making estimates for component DBMS whose cost information are not available in the catalog. We also make estimates of intermediate result size and context conversion cost for each such category. These estimates are used as inputs to the query optimizer and will be updated as the component databases evolve.

4.2 The Multidatabase Catalog

In order to generate an execution plan for a given query, the multidatabase query optimizer requires information on

- the locations of data which are referenced by the query;
- the names of the required data as understood by the component DBMS;
- the database profile for the respective component databases which are needed for estimating the cost of subqueries;
- system-related statistics of components sites, including
  - accessibility of individual component systems (some systems provide unlimited access to foreign users but others may only provide limited access to their resources);
  - workload characteristics measured in terms of CPU, I/O and communication line utilization; and
  - computing power of the host computing system in terms of its processing speed.

We assume that these information are kept in a *multidatabase catalog* which is accessible to the query optimizer.

It is worthwhile pointing out that the multidatabase catalog is not merely a repository for the global schema. For instance, in order to support new processing methods such as the E-join, additional semantic information (such as those properties of an entity which are deemed to be 'useful' in identifying an entity) has to be kept. Moreover, it is necessary to identify what statistics should be kept, who should update these statistics, and when the update should be performed. One solution to this is a lazy evaluation approach: statistical information of the component databases are collected and updated when they needed. Hence, when a global query is to be processed, the global query optimizer will sent a query to the component database system to get the most updated statistics. The information obtained is used both in the subsequent optimization process and to update the catalog. This approach is justified on the basis that component databases has no obligation to inform the multidatabase of any changes in its system. Updating the multidatabase catalog according to every change in component database systems is usually expensive

and may not be realistic. The cost of maintaining the multidatabase catalog should be charged to the multidatabase users who issue the global queries. Furthermore, requesting the statistics when query is to be processed ensures that the most updated statistics is being used. To reduce the optimization overhead for the global queries incurred by gathering statistics during optimization, two mechanisms can be incorporated. First, the query optimizer may occasionally issue queries to collect and update the statistic for those frequently accessed component databases during off-peak hours. Second, the query optimizer can also use the information currently available in the catalog without issuing sampling queries to reduce the overhead, whenever it is felt that the available statistics are sufficiently current.

5 Conclusion

In this paper, we suggested that the multidatabase query optimization problem is fundamentally different from distributed query optimization. These differences are shown to be direct consequences of autonomy and heterogeneity of component databases in a multidatabase system. The existing query processing and optimization technologies must therefore be re-examined in the light of these findings.

Following this observation, we present the architectural design of a multidatabase query optimizer which circumvents the problems highlighted earlier. Our design objective is to have a query optimizer which is (i) robust to the heterogeneous environment, and (ii) adaptive to the evolution of component databases. Performance of the proposed optimization approach is also being studied currently.

Acknowledgment

The first author would like to thank M.-C. Shan of Hewlett-Packard Laboratories who initiated the work reported here and provided stimulating suggestions.

References

[1] M. Bright, A. Hurson, and S. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3):50–60, 1992.

[2] A. Chatterjee and A. Segev. Data manipulation in heterogeneous databases. *SIGMOD RECORD*, 20(4):64–68, 1991.

[3] U. Dayal. Query processing in multidatabase system. In W. Kim, D. Reiner, and S. Batory, editors, *Query Processing In Database System*. Springer Verlag, 1985.

[4] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous dbms. In *Proc. of the VLDB*, 1992. To appear.

[5] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[6] M. Siegel and S. Madnick. A metadata approach to resolving semantic conflicts. In *Proc. of the 17th International Conference on Very Large Data Bases*, 1991.

[7] R. Wang and S. Madnick. The inter-database instance identification problem in integrating autonomous systems. In *Proc. of the International Conference on Data Engineering*, pages 46–55, 1989.