Optimization and Execution Strategy for Multidatabase Queries

Weimin Du, Ming-Chien Shan, Jim Davis
Software Technology Laboratory
HPL-94-74
April, 1995

query optimization, heterogeneous systems

Query optimization and execution in multidatabase systems is different from that of distributed homogeneous database systems, due to not only the lack of information about cost formulae of local database systems, but also less control over local query execution and greater overhead of data access. For example, performance of nested loop join of tables in different databases degrades rapidly as the number of inner table searches increases. This is true even if a clustered index exists on the join column of the inner table. In this paper, we study the execution of multidatabase queries and its impacts on optimization. The main contributions of the paper are twofold: developing a technique which reduces the costs of multidatabase joins using partial query results and statistical data, and proposing a query optimization and execution strategy for multidatabase queries that reflects unique features of multidatabase systems. Experiments in Pegasus show that the technique and the strategy, though simple, are effective in reducing the elapsed time of multidatabase queries.

1 Introduction

A multidatabase system (MDBS) integrates pre-existing local database systems (LDBSs) and therefore allows users to access otherwise isolated and different LDBSs through a uniform interface. The MDBS provides a homogenizing layer on top of the heterogeneous LDBSs, giving users the illusion of a single system.

The MDBS hides not only discrepancies in user interface and languages, but also optimization and execution details of multidatabase queries. Like traditional distributed database systems (e.g. R* [LMH+84]), the MDBS compares and choses the execution plan that is of minimum estimated cost. Although the basic paradigm remains unchanged, query processing in MDBSs is quite different from and more difficult than that in homogeneous systems, due to local autonomy. The effects of local autonomy are manifested in both optimization and execution aspects of query processing.

First of all, the MDBS may not have enough information about LDBSs to do the optimization. For example, cost formulae of LDBSs which are the basis of multidatabase query optimization is usually not available to the MDBS. Secondly, the MDBS has neither direct control over queries executed at LDBSs, nor direct access to internal data structures and functions of LDBSs. For example, the MDBS cannot call the internal data storage system directly. SQL API (Application Programming Interface) is usually the only eligible interface to LDBSs for data access. The restrictions have great impacts on query optimization, as many commonly recognized optimizations that have been proven in homogeneous systems may not hold in MDBSs. For example, nested loop join has been considered to be the preferred join method if the join column values passed to the inner table are in sequence and the join column index of the inner table is clustered [BE77]. The statement, although holds in distributed homogeneous database systems (e.g., R*), is not true in MDBS due to the large overhead associated with each inner table retrieval, as the MDBS does not have direct access to the internal data storage.

In [DKS91], we studied the optimization problem caused by local autonomy, focusing on cost estimation of local queries. In that paper, we proposed a technique which effectively derives the logical cost formulae of LDBSs using a specially designed synthetical database and a suite of calibration queries. Based on these derived logical cost formulae, the MDBS can satisfactorily estimate cost of single table queries and multiple table joins, as well as other complex queries, executed in a single LDBS.

In this paper, we shall focus on the execution aspect of multidatabase queries and its impacts on query optimization. The main contributions of the paper are twofold: (1) a technique that improves performance of global (multidatabase) joins using partial query results and statistical data, and (2) a multidatabase query processing strategy using three phase optimization and runtime execution plan adjustment. Experiments in Pegasus show that this technique and strategy, though quite simple, are effective in reducing elapsed time of multidatabase queries.

In the next section, we briefly describe the experimental environment in which many useful experiments presented in the paper were conducted. Section 3 discusses implementations of global joins in multidatabase environments and presents a technique that improves global join performance. The overall optimization and execution strategy for multidatabase queries is presented in Section 4. Section 5 concludes the paper with a few remarks.

2 Experimental Environment

Since LDBSs behave like black boxes to the MDBS, an effective way to understand the execution environment is to experiment. In this section, we briefly describe the Pegasus execution environment in which many experiments presented in the paper have been conducted.

Pegasus is a heterogeneous information and operation management system being developed at Hewlett-Packard Laboratories [SAD+94]. It is a full-fledged database management system that integrates native and external data sources. A native database is created in and managed by Pegasus, while external databases are accessible through Pegasus but are managed independently by external DBMSs. The following commercial relational database systems have been integrated as external data sources in the current Pegasus prototype: Allbase (version F0), DB2 (version 2.2), Informix (version 5.01), Oracle (version 7) and Sybase (version 4.6.1).

Figure 1 shows the Pegasus system architecture. Pegasus queries are executed concurrently by the Query Evaluator (QE) and a set of Pegasus Agents (PAs). The QE is responsible for overall execution coordination according to the execution plans generated by the Query Optimizer (QO) and to perform global operations involving multiple LDBSs. A PA is superimposed on top of each LDBS and is implemented as a gateway managing dynamic SQL calls to the LDBS. PAs take native SQL queries generated by the Query Translator (QT) and pass them to the underlying LDBSs. The query results are fetched back, converted into Pegasus internal format and sent to the QE or other PAs. A PA is also capable of performing global operations (e.g. global joins involving tables of multiple LDBSs) to reduce communication costs.

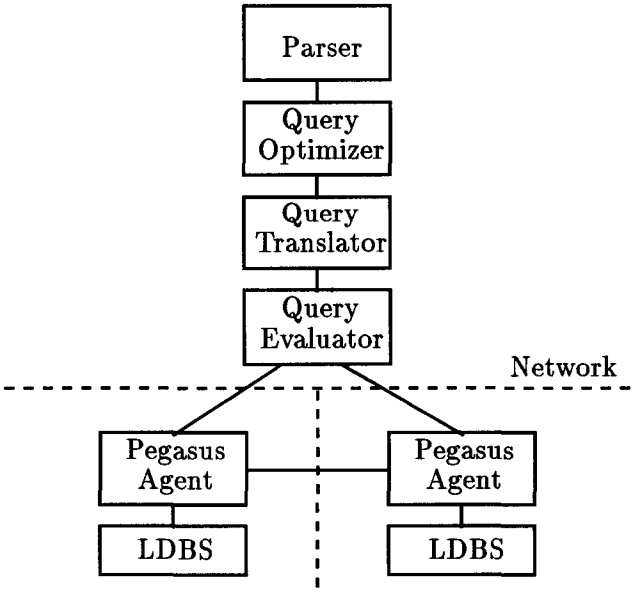

Figure 1: Pegasus system

For each LDBS, a synthetic database [DKS91] has been created. The synthetic database consists of a set of tables of cardinalities 2^n, where $10 \leq n \leq 20$. Each table has six

basic columns of INTEGER type plus an optional seventh column of CHAR_STRING type. The six basic columns represent columns of different data distributions (normal or uniform[1]), access methods (index scan or sequential scan), and in the case of index scan, data placements (clustered or unclustered). Table 1 gives basic characteristics of each columns. For more details about the synthetic database, readers are referred to [DKS91].

Table 1: Synthetic database table T_n

	Data Distribution	Access Method	Data Placement	Value Range	Uniqueness
Column C_1	Normal	Index	Clustered	0 - n	Duplicated
Column C_2	Uniform	Index	Clustered	0 - 2^n-1	Unique
Column C_3	Normal	Index	Unclustered	0 - n	Duplicated
Column C_4	Normal	No Index	N/A	0 - n	Duplicated
Column C_5	Uniform	Index	Unclustered	0 - 2^n-1	Unique
Column C_6	Uniform	No Index	N/A	0 - 2^n-1	Unique

The reason for using synthetic database in the experiments is that it not only reflects typical data requirements, but also allows us to analyze performance data in a deterministic way, as the data of synthetic database are generated not randomly but controlled by algorithms to ensure the desired characteristics.

Since the purpose of the experiments is to understand the behavior of LDBSs, possible distortion is avoided as much as possible. In the experiments, each query is posed after a buffer flushing query which sequentially scans a table whose size is greater than that of the database buffer. This eliminates possible distortion due to data buffering and the overhead of bringing up the external DBMS, The QE and PA are instrumented to collect for each query executed the elapsed time which is calculated by subtracting the start timestamp (when the query is received) from the end timestamp (when the last result is processed). Each query is issued 30 times and the average elapsed time is calculated. Except few cases, relative error between actual data and average value is within 5% with confidence coefficient of 95%.

As a final note, we would like to point out that the purpose of the experiments is not to evaluate the performance of specific DBMSs, but to identify the best way of accessing data stored in external databases. Therefore, the differences in hardware platforms and software configurations, including operating systems and networking, will not be singled out in the experiments. What we are really interested in are comparisons of elapsed times of different but semantically equivalent queries executed under each integrated DBMS running in its given configuration.

3 Global Join Strategies

Nested Loop Join (NLJ), Sort Merge Join (SMJ), and Hash Join (HJ) are the three most commonly used join methods in commercial relational database systems. In this section, we review possible implementations of these join methods in the MDBS environments and propose a technique to improve their performance.

[1]Note that columns in the synthetic tables are not really in normal or uniform distribution as defined in statistics. They are in absolute normal or uniform distribution. For example, values of C_2 are unique and cover the entire range from 0 to $2^n - 1$.

3.1 Nested Loop Join

In the NLJ method joining two tables, one table is chosen as the outer table and the other as inner table [2]. For each tuple in the outer table that satisfies the local predicate, the inner table is searched for the matching tuples. Therefore, NLJ is preferred when either the number of qualified tuples from the outer table is small or the inner table can be searched efficiently. For example, NLJ performs best when the outer table tuples are retrieved in sequence of join column values and a clustered index exists on the join column of the inner table.

3.1.1 Dynamic SQL Overhead

As we mentioned, the MDBS does not have direct access to LDBS' internal functions and data structures. The only interface to LDBSs is SQL API. Given the restriction, a straightforward implementation of NLJ in MDBSs is for each tuple retrieved from the outer table, the QE issues a dynamic SQL query to search for matching tuples in the inner table.

Algorithm 1 (NLJ by Single Value Predicate Query)

Step 1 *Form a nonparameterized dynamic SQL query to scan the outer table and prepare it.*

Step 2 *Form a parametrized dynamic SQL query to search the inner table and prepare it.*

Step 3 *Open a cursor for the outer table scan query and for each tuple fetched do the following:*

1. *Open a cursor for the inner table search query with the join column value of the tuple as the binding argument.*
2. *Fetch all matching tuples from the inner table.*
3. *Compute the join results and retun them.*

Clearly, there is an overhead associated with each inner table search. The issues are how much the overhead is, and how much does it affect the performance of global NLJ in the MDBS environment. Table 2 shows the results of an experiment we have conducted with Oracle DBMS for the purpose.

In the experiment, the following two types of queries have been issued against the synthetic database: single value query (SVQ) and range predicate query (RPQ).

SVQ: **select** C_2 **from** T_n **where** $C_2 = arg$;

RPQ: **select** C_2 **from** T_n **where** $C_2 < arg$;

Recall that values of column C_2 are unique, uniformly distributed and range from 0 to $2^n - 1$. Therefore, the value of arg determines the number of tuples a RPQ will retrieve from

[2] Actually, NLJ can also be used in n-way joins if one table joins with all other tables on the same join column. Without loss of generality, we consider 2-way NLJ only in the paper.

table T_n. The SVQ always retrieves a single tuple, and the value of *arg* determines which tuple it retrieves. To retrieve tuples from the inner table with values of the join column in the range of 0 to a given number $n-1$, the SVQ is compiled once (i.e., PREPAREd and DESCRIBEd) and repeatedly executed n times (i.e., OPENED and FETCHED with arguments of consecutive values from 0 to $n-1$). Clearly, a RPQ with binding argument of value n is semantically equivalent to an SVQ with binding argument of n values from 0 to $n-1$. The difference is that the RPQ is executed only once, while the SVQ is OPENed and FETCHed n times although both are compiled only once.

SVQs are meant to be used in Step 2 of Algorithm 1. They are chosen because they represent the ideal case of inner table search of NLJ in traditional DBMS when the argument values are given in the same order as the clustering order of the search column which is indexed.

Table 2 gives the elapsed time (in seconds) of both types of queries against Oracle synthetic tables. For SVQs, the number is the elapsed time of one parameterized query being compiled once but executed many times with consecutive binding argument values. Note that, in this experiment, results of SVQ or RPQ are fetched into PA's buffer without further processing. The table also shows the cost difference between two types of equivalent queries. As we can see, the difference increases rapidly as the number of tuples returned increases.

Table 2: Elapsed time: SVQ vs. RPQ

# of Tuples	1	16	64	128	1024	8192	16384	32768	131072
RPQ	0.17	0.17	0.17	0.18	0.23	0.45	0.58	0.73	1.28
SVQ	0.17	0.25	0.48	0.80	5.31	41.09	80.21	160.4	623.92
Ratio	1	1.5	2.8	4.4	23	92	134	224	487

It is also worth noting that when the number of tuples an SVQ retrieved reaches a certain level (e.g., 64 for T_{13}), the cost (e.g., 0.48 second for T_{13}) becomes greater than that of a RPQ which retrieves all tuples (e.g., 8192 tuples in T_{13}) of the table (e.g., 0.45 second for T_{13}). In other words, the overall cost of NLJ in the algorithm 1 could be greater than those of SMJ or HJ if the number of tuples retrieved from outer table exceeds ceratin limit, since SMH and HJ require only one scan of the inner table but retrieve all tuples.

3.1.2 Improving NLJ

There are two major unnecessary costs associated with the above straightforward implementation of NLJ. The first part is the overhead of dynamic SQL execution as shown in the previous subsection and the other is possible network overhead as the inner table PA is invoked at least once for each outer table tuple.

The network overhead can be reduced by block NLJ implementation. More specifically, the inner table PA is invoked once for a block of outer table tuples while the underline DBMS is still invoked once for each outer table tuple. The network cost is reduced as a block of outer table tuples is sent to the inner table PA in one message, rather than many different messages.

It is also possible to reduce both dynamic SQL and network overheads by reducing the number of DBMS invocations for the inner table search. This can be done in two ways: (a)

to use set predicate queries (SPQ) for the inner table search, or (b) to use range predicate queries as suggested in the above experiment.

SPQ: **select** C_2 **from** T_n **where** C_2 **in** (*value_list*);

The following algorithm illustrates the implementation using SPQ.

Algorithm 2 (NLJ by Set Predicate Query)
Repeat the following steps until the outer table is exhausted.

Step 1 *Retrieve from the outer table n tuples that satisfy the local predicates, where n is smaller than the max number of set elements allowed by the underlying DBMS.*

Step 2 *Modify the original inner table search query by replacing the original join predicate (C_i = arg) with a new set predicate (C_i in ($arg_1, arg_2, ..., arg_k$)), where arg_i are the join column values of tuples returned from the outer table.*

Step 3 *Execute the modified query. For each returned tuple, form the join result and return it.*

There are, however, several problems with this implementation. First, the number of outer table tuples that can be packed into a single set is limited, due to limitations of both the size of query statement buffer and the number of elements allowed in a set constant. For example, Oracle allows at most 254 set elements. This is the reason why in Step 1 of Algorithm 2 only a number of (not all) outer table tuples are retrieved.

Another problem with the SPQ approach is that the cost of SPQ also degrades quite rapidly as the size of the set increases. Figure 2 shows a comparison of performance of the three types of queries in Informix.

The range predicate query implementation tries to find all matching tuples from inner table with a single RPQ. Unlike the other two NLJ implementations, the tuples returned from the inner table may or may not have matching tuples from the outer table. An additional step is therefore needed to filter out unmatched inner table tuples.

Algorithm 3 (NLJ by Range Predicate Query)

Step 1 *Retrieve from the outer table all tuples that satisfy the local predicates and remember the min and max values of the join columns of these tuples.*

Step 2 *Modify the original inner table search query by conjuncting the original predicate with additional predicates that limits the range of join column values according to the min and max values from the outer table.*

Step 3 *For each tuple retrieved from the inner table, check to see if it has a matching tuple from the outer table. If yes, form the join result tuple and return it.*

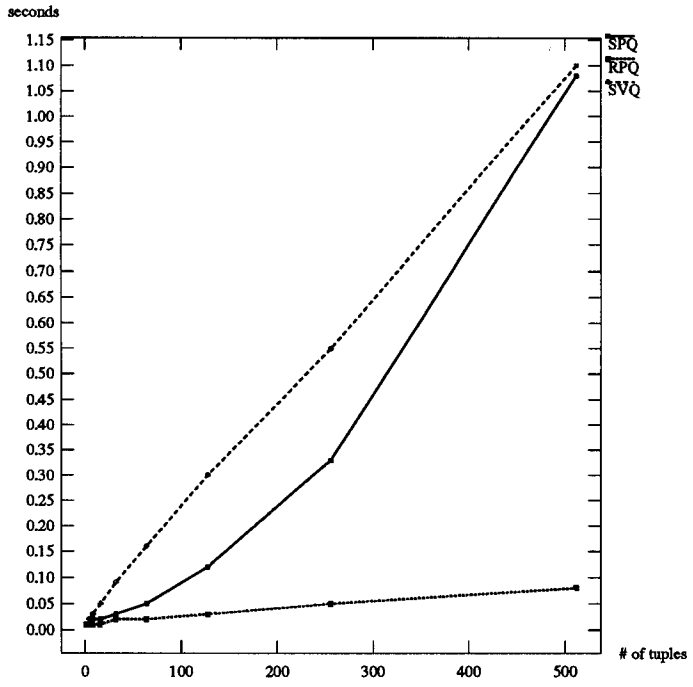

Figure 2: Comparison of three types of queries

Like other NLJ approaches, the approach is preferred only when the number of tuples returned from the outer table is small (e.g., can fit into the MDBS's buffer). Note that Step 3 of the algorithm can usually be performed efficiently if we sort the tuples returned from the outer table. In addition, the range of join column values from the outer table should be small comparing to that of inner table. Otherwise, the second step would retrieve most of inner table tuples including many unneeded ones.

In summary, the single value predicate approach (Algorithm 1) is easy to implement but is most expensive. The set predicate approach (Algorithm 2) performs better, but only works when the number of tuples returned from the outer table is small. The block NLJ implementation is comparable to the set predicate approach, and also scales linearly for a large number of outer table tuples. The range predicate approach (algorithm 3) could be the most efficient implementation under the condition that one table (the outer table) is much smaller than the other (the inner table) in terms of join column value range. Table 3 gives the overall elapsed time of the four NLJ implementations for the following query which joins an Oracle table ($D_1.T_n$) with an informix table ($D_2.T_m$)[3].

select $T_n.C_2$, $T_m.C_2$ **from** $D_1.T_n$, $D_2.T_m$ **where** $T_n.C_2 = T_m.C_2$ **and** $T_n.C_2 <$ *arg*;

Note that the numbers in the table are much larger than those in Figure 2, due to overhead of result processing, including data format conversion, join computation, and LDBS connection. Also note that the elapsed times for Algorithm 1 implementation are much larger than the others, due to network overhead. The elapsed times of block NLJ implementation are shown

[3]The value of n and m are not important here as there exists an clustered index on C_2.

Table 3: Comparison of NLJ implementations

Arg	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 1 (Block)
1	0.80	0.80	0.80	0.80
2	0.90	0.80	0.80	0.80
4	1.00	0.80	0.80	0.81
8	1.20	0.80	0.80	0.82
16	1.48	0.80	0.80	0.84
32	1.94	0.80	0.81	0.87
64	3.86	0.80	0.81	0.93
128	7.70	0.90	0.82	1.07
256	15.02	1.10	0.84	1.35
512	30.74	1.80	0.85	1.90

for the block size of 1024 tuples.

3.2 Hash Join and Sort Merge Join

The idea of using RPQs to reduce the number of tuples returned from external tables can also be used to implement HJ and SMJ in MDBSs. The implementation of simple HJ [BRAT84, DG85] is quite straightforward: using the min and max values of the join columns of table R (the table that is used to build the hash table) to modify the table scan query of table S (the probing table) to return only those tuples whose join column values are within the range of min and max of table R.

The modifications of SMJ and more complicated HJs (e.g., GRACE and Hybrid) however are not that straightforward. Since the two table scans are performed concurrently, it is generally impossible to use the actual join column value range of one table to reduce the number of tuples returned from the other table, as we did in NLJ and simple HJ.

One way to address the problem is to use semantic information of the query such as range predicates on join columns. Let us consider, for example, the following query:

select * **from** T_1, T_2 **where** $T_1.C_1 = T_2.C_2$ **and** $T_1.C_1 < 100$;

The range predicate "$T_1.C_1 < 100$" suggest that we only need to retrieve tuples from both T_1 and T_2 whose join colum values are less than 100. There are two problems with the approach. It is generally expensive to analyze and understand the query predicate to identify useful predicates and there may not exist such a predicate for some cases.

The problem can also be overcome by using the database statistical data. Pegasus keeps in its own catalog statistical data about LDBSs, such as cardinality of each table, min and max values of each column, etc. Therefore, we can modify the table scan queries by conjuncting the original predicate with predicate "$(C_i \leq max_{stat})$ and $(C_i \geq min_{stat})$", where C_i is the join column and min_{stat} and max_{stat} are the min and max values of the join column of the other join table in Pegasus system catalog.

There is however a severe problem with the implementation: the modification may result in a query which is semantically not equivalent to the original one. This is possible as the statistical data may not consistent with the actual data. Let us consider, for example, the following join of synthetic tables T_{13} and T_{15} of LDBSs D_1 and D_2:

select * **from** $D_1.T_{13}$, $D_2.T_{15}$ **where** $T_{13}.C_2 = T_{15}.C_2$;

The following query retrieves inner table tuples for the SMJ.

OQ: **select** * **from** $D_2.T_{15}$;

Suppose the statistical data of table T_{13} is not up to date and shows that the min and max of $T_{13}.C_2$ are 100 and 8192, respectively. The following modified query (MQ) is not semantically equivalent to the original query OQ, as the first 100 results are missing.

MQ: **select** * **from** $D_2.T_{15}$ **where** $(T_{15}.C_2 \geq 100)$ **and** $(T_{15}.C_2 \leq 8192)$;

The solution to the problem is runtime checking and adjustment. More specifically, statistical data being used in query modification will be validated at run time when the corresponding table scan query is executed. When an error is detected, an adjustment query (AQ) will be automatically generated for the modified queries to retrieve the missing tuples.

AQ: **select** * **from** $D_2.T_{15}$ **where** $(T_{15}.C_2 < 100)$;

The following is a modification of the SMJ algorithm. HJ algorithms can be modified similarly.

Algorithm 4 (Improved SMJ)

Step 1 *Retrieve and sort data from both join tables using the modified table scan queries.*

Step 2 *Check the* min_{stat} *and* max_{stat} *values used in query modification against the real data. If they are consistent (i.e.,* $min_{real} \geq min_{stat}$ *and* $max_{real} \leq max_{stat}$*), go to Step 4.*

Step 3 *Use* min_{stat} *and* max_{stat} *to form the adjustment query and execute it. Merge the result with the one returned in Step 1.*

Step 4 . *Merge the two table scan results and form the join results.*

There are a few things about this implementation worth mentioning. First, the basic assumption is that the ranges of join column values of the join tables are usually different. The larger the difference, the more improvements in performance we will get. The assumption is reasonable, especially in MDBSs, when SMJ or HJ is used in the cases where NLJ is usually preferred (i.e., the number of tuples returned from the outer table is not very large). In many cases, small number of tuples implies small range of join values. There is however a potential runtime overhead: that of an adjustment query. Since the tuples returned by AQ are needed anyway, the real overhead is to access the join table twice, instead of just once. The overhead is usually not very large, but not negligible. The query modification is therefore worthwhile only when the improvement is significant.

Another thing worth noting is that only one table scan query can be modified. We cannot use the statistical data of both join tables to reduce the number of tuples returned from other tables, as this will not provide the information needed to verify the database statistical data. The problem however is not that severe, as the technique is supposed to be used when the range difference between join column values of two join tables is large. Therefore, we can always use the statistical data of the smaller table to reduce number of tuples returned from the larger one.

Figure 3 is the result of an experiment we have conducted on the performance comparison of Algorithm 4, the original SMJ implementation and the block NLJ implementation. In the experiment, the following query is executed with different *arg* values (from 1 to 8192).

select * **from** $D_1.T_{13}$, $D_2.T_{15}$ **where** $(T_{13}.C_2 = T_{15}.C_2)$ **and** $(T_{13}.C_5 < arg)$;

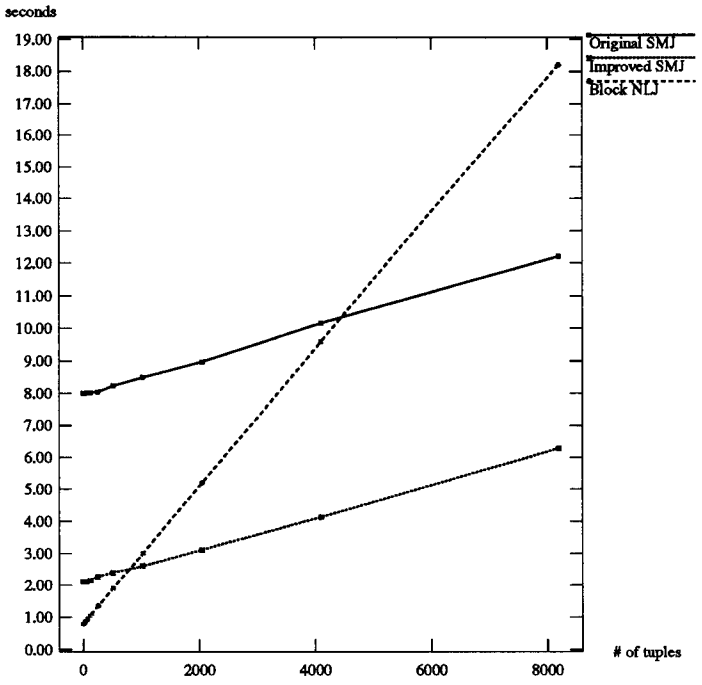

Figure 3: Comparison of join methods

The picture shows that the improvement of Algorithm 4 is quite significant over the original SMJ implementation, especially when join selectivity is low. One reason for the improvement, as we expected, is the reduction of number of tuples (8192 instead of 32678) returned from T_{15}. There is another reason why the improved implementation performed especially well when join selectivity is low. The reason is that the modified query suggested that the Informix optimizer use index scan. Without the added range predicate, the entire T_{15} will be returned and a sequential scan is used even if there is an index on the column. This property of the improved implementation makes it a good candidate for cases where both original NLJ, SMJ, and HJ do not perform well.

The picture also shows that NLJ out-performs SMJ when the number of tuples from the outer table is very small (e.g., < 200). Although the number of outer table tuples returned

is the same for both NLJ and SMJ, the two join strategies retrieve different numbers of inner table tuples. NLJ (except Algorithm 3) retrieves only needed inner table tuples that match with qualified outer table tuples. SMJ (including the improved implementation) usually retrieves more than needed. In the experiment, NLJ retrieves *arg* inner table tuples, while SMJ (improved implementation) retrieves 8192 tuples from the inner table.

4 Query Optimization and Execution Strategy

In the previous section, we have discussed implementations of global joins in the multi-database environments. In this section we focus on overall query optimization and execution strategy in the context of Pegasus prototype.

4.1 Overview

Base on the result of previous section and result of other research [GRAE94], the MDBS execution environment can be summerized as follows. NLJ is preferred when the number of tuples returned from the outer table is very small. When both join tables (after local predicates) are large and clustered indexes exist on join columns, SMJ is clearly preferred, because of not only concurrent execution of two table scans, but also smaller overhead of finding matching tuples. SMJ is also preferred when the order of join result can be reused in later operations (e.g., join and group by). In other cases, HJ is generally preferred.

There are two things worth noting in this new execution environment. First, the execution plans generated by QO tend to be SMJ/HJ dominated, as NLJ is preferred only in very limited cases. As pointed out in [MS91], SMJ and HJ provide poor inter-operator parallelism in a left deep execution tree, as the first result will not be generated until the entire input table has been processed. The same statement is also true for improved NLJ implementations. Therefore, traditional way of query processing (e.g., in System R and R*) needs to be revised to generate elapsed time efficient execution plans.

Second, accuracy of statistical data is very important in the new environment. This is true not only because the inaccurate statistic data may cause run-time overhead for the improved SMJ/HJ implementation, but also because of the wrong choice of NLJ can be very expensive. On the other hand, keeping statistical data accurate is very difficult, as they may be outdated, due to updates by local users to the underlying data.

Pegasus employs a three phase query optimization and a run-time execution plan adjustment strategy to address the above problems. In the first phase, queries are simplified using heuristic rules. The purpose is to simplify operations that are expensive to evaluate and difficult to optimize in traditional way. For example, nested queries are flattened and outerjoins are simplified to regular joins whenever possible.

The second phase does conventional cost based optimization which generates left-deep execution plans only. The key to the success of this phase is a good estimate of execution costs for various operations performed in LDBSs. In [DKS91], a new cost model and a calibration process have been developed for the purpose. The technique makes it possible to estimate the cost of queries executed at an LDBS even if no knowledge is available about its optimizer.

The execution plans generated in the second phase are the cheapest in terms of total resources consumed. These however may not be good plans in terms of elapsed time, as parallelism

was not taken into account. The problem is addressed in this last phase (post-optimization phase) which takes as input an optimized execution plan and generates as output another execution plan of similar total cost but smaller elapsed time.

In the case where an inefficient or even a wrong execution plan is generated due to inaccurate statistical data by the QO, the QE and PA will detect it using actual query result at execution time and adjustment is made automatically whenever necessary.

In the next two subsections, we shall discuss in detail issues of post-optimization and run-time plan adjustment.

4.2 Post-optimization

As in traditional distributed database systems, both total resources consumed and elapsed time are important optimization metrics in MDBSs. Optimization of both metrics however is very difficult [GHK92], as search space has to be extended from only left-deep execution trees to both left-deep and bushy execution trees. Search over bushy trees is much more costly than over left-deep trees.

The purpose of post-optimization is not to generate an execution plan which is optimal in elapsed time. Instead it tries to improve the elapsed time for a given (left-deep) execution plan generated by the previous phase of QO. This is possible, as improvement in elapse time does not necessarily imply increase in total resource usage.

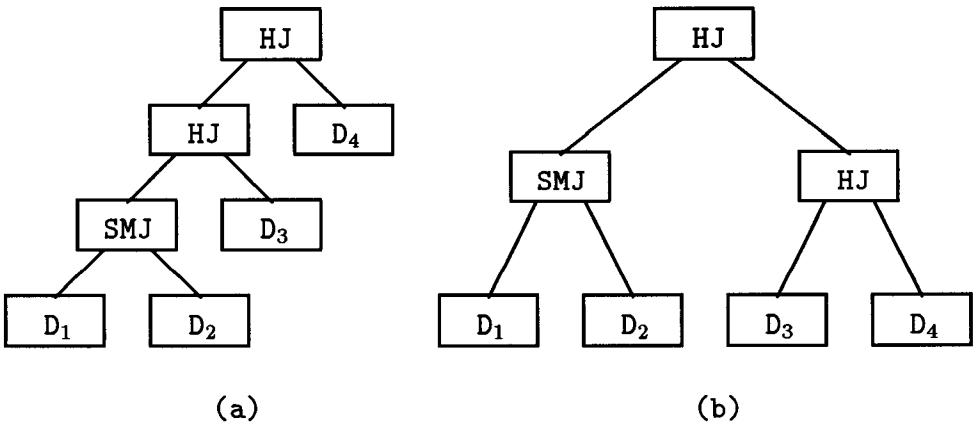

Figure 4: Improving elapsed time

Example 1 *Consider the following multidatabase query, where* D_1, D_2, D_3 *and* D_4 *are synthetic databases described in the Section 2.*

select * **from** $D_1.T_{13}$, $D_2.T_{13}$, $D_3.T_{13}$, $D_4.T_{13}$
where $D_1.T_{13}.C_2 = D_2.T_{13}.C_2$ **and** $D_3.T_{13}.C_6 = D_4.T_{13}.C_6$ **and** $D_2.T_{13}.C_2 = D_3.T_{13}.C_6$;

Figure 4.(a) shows a possible left-deep execution tree for the query. Figure 4.(b) shows an improved execution tree for the same query. The two trees evaluate the same query expression

and also have the same total resource usage. But the tree in Figure 4.(b) is better than that in Figure 4.(a) with respect to elapsed time.

Transformation from a left-deep query tree into an improved bushy query tree can be done either heuristically based on syntactical or semantic patterns of the input query trees, or cost based. The latter is possible as all cost information needed is already calculated in the previous phase. Elapse time of each subtree is first derived from its total cost and is recomputed after each transformation. Every transformation is validated to make sure that total resource usage does not increase and no interesting order is lost.

4.3 Run-time Adjustment

To cope with possible inefficient or incorrect execution plans due to inaccurate statistical data, the Pegasus QE and PA modules have been extended with the following functions:

- Join method adjustment (from NLJ to SMJ or HJ) when the number of tuples returned from the outer table is large;
- Automatic generation and execution of adjustment query for SMJ and HJ when statistical data used in query modification are not consistent with real data; and
- Collection of statistical data and observation of unusual behavior of LDBSs.

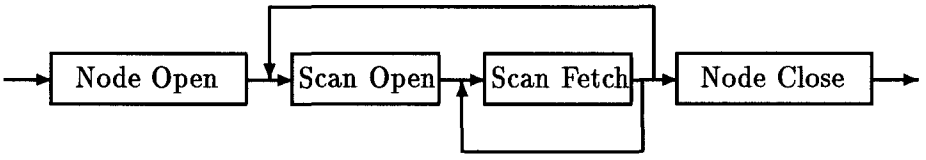

Figure 5: Query evaluation in Pegasus

At logical level, QE and PA consist of a set of node evaluators, one for each kind of query tree node. A node evaluator evaluates a node in four phases: Node Open, Scan Open, Scan Fetch, and Node Close (Figure 5). At Node Open phase, all one time initialization tasks are performed. For example, for external table scan node, the associated query is PREPAREd and DESCRIBEd and data buffers are allocated. Binding arguments are validated at the second phase to make sure that they are in right type and format. A cursor is also OPENed at the phase for table scan node. At Scan Fetch phase, a number of tuples are returned (to the parent node). The Scan Open phase is invoked once for each argument (or a block of arguments in the case of block NLJ), while the Scan Fetch phase may be invoked several times for the same argument(s) to return all qualified tuples. Each invocation of Scan Fetch of the child node specifies the *min* and *max* number of tuples it should return. The child node evaluator is free to return either n tuples, where $min \leq n \leq max$ when there are at least *min* tuples, or all tuples otherwise.

Run-time adjustment of join methods can be implemented by modifying NLJ node evaluator to check the number of tuples returned from the first fetch of the outer table. If it is greater

than the threshold set by the QO, the NLJ node evaluator will invoke the SMJ/HJ node evaluator with the same argument, along with the tuples it already retrieved from the outer table.

There are however several things worth noting. First, the run-time join method adjustment is possible because NLJ node (SPQ and block implementations) have the same leftmost child as SMJ and HJ nodes. They however have different right child node. To reduce run-time overhead, the QO will decide at optimization time the alternative join method (SMJ or HJ) and will also generate the corresponding right child query. Second, in order for the QE to make right decision, the first fetch of the outer table scan needs to fetch enough tuples (i.e., *min* must be greater than the threshold set by the QO). This is clearly justified for the SPQ or block implementations of NLJ when the left child node is a simple table scan. There are however cases in which the left child node is a complex and time consuming subtree (e.g., a subquery) and it is inadequate to request a large number of tuples in one fetch. Therefore, Pegasus will only do run-time adjustment for those NLJs whose child nodes are simple operations (e.g., external table scans).

Another important extension to QE and PA is to collect min and/or max values of columns of external tables. The data will be used to validate the corresponding data in the Pegasus system catalog. In the case where an inconsistent is detected, the statistical data will be updated and marked dirty. Only those statistical data that are infrequently marked dirty will be used in query modification.

To avoid run-time overhead, statistical data will be collected only for those columns that are requested entirely by Pegasus (e.g., join columns of some SMJs and HJs). Note that this is different from the explicit collection of LDBS statistical data periodically performed by the Pegasus statistics update utility.

5 Conclusions

In this paper, we have studied several implementations of global joins in multidatabase systems and have proposed a query optimization and execution strategy for multidatabase queries. We also showed a simple technique which reduces number of tuples returned from external databases using actual query results and statistical information. Experiments in Pegasus have shown that the strategy and the technique proposed in the paper, though simple, are effective in reducing overall execution (elapsed) time of multidatabase query.

Although efficient query optimization and execution is essential to multidatabase systems, little effort has been devoted to the topic. The focus of this paper is on the implementation of traditional join methods (nested loop, sort merge and hash) and their impacts on query optimization. There are other related issues that need to be investigated. For example, outerjoins are widely used to resolve inconsistent data values between equivalent tuples from different tables [CHEN90, DAYA87]. We also need new operations to resolve inconsistencies between tuples from the same external table. Tuples from the same table may be equivalent because they may all be equivalent to the same tuple from another external table. Efficient optimization and execution of outer-join and other new operations are very important and much research is still needed.

6 References

[BE77] M. Blasgen and K. Eswaran. Storage and access in relational data bases. In *IBM System Journal*, No. 4, 1977.

[BRAT84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of International Conference on Very Large Data Bases*, Singapore, 1984.

[CHEN90] A. Chen. Outerjoin optimization in multidatabase systems. In *Proceedings of International Symposium on Distributed and Parallel Database Systems*, Ireland, 1990.

[DAYA87] U. Dayal. Query processing in a multidatabase system. In *Query Processing in Database Systems* (Kim, Batory and Reiner (eds.)), 1985.

[DG85] D. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of International Conference on Very Large Data Bases*, Stockholm, Sweden, 1985.

[DKS91] W. Du, R. Krishinamurthy and M. Shan. Query optimization in heterogeneous DBMS. In *Proceedings of International Conference on Very Large Data Bases*, Vancouver, Canada, 1991.

[GHK92] S. Ganguly, W. Hasan and R. Krishinamurthy. Query optimization for parallel execution. In *Proceedings of International Conference on Management of Data*, San Diego, CA. 1992.

[GRAE94] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proceedings of International Conference on Data Engineering*, Houston, Texas, 1994.

[LMH+84] G. Lohman, C. Mohan, L. Haas, B. Lindsay, P. Selinger, P. Wilms and D. Daniels. Query processing in R*. In *Query Processing in Database Systems* (Kim, Batory and Reiner (eds.)), 1985.

[MS91] M. Murphy and M. Shan. Execution plan balancing. In *Proceedings of International Conference on Data Engineering*, Kobe, Japan, 1991.

[SAD+94] M. Shan, R. Ahmed, J. Davis, W. Du and W. Kent. The Pegasus project - A heterogeneous information management system. In *Modern Information Computer* (Kim (ed.)), Addison-Wesley Publishing Company, 1994