Dealing with Duplicate Tuples in Multi-Join Query Processing

Roberto J. Bayardo Jr.
University of Texas at Austin
Department of Computer Sciences and Applied Research Laboratories
Taylor Hall Rm. 141, C0500
Austin, TX USA 78712

E-mail: bayardo@cs.utexas.edu
Web: http://www.cs.utexas.edu/users/bayardo/

ABSTRACT: This paper presents and evaluates several schemes for handling duplicate tuple elimination during optimization and execution of large select-project-join queries. The primary issues investigated are (1) precisely when to apply duplicate tuple removal during query evaluation, and (2) how an optimizer should predict the effects of removing duplicates. We also develop a realistic model of multiple join queries inspired by a proposed data-mining application. Through experiments on this model, we find two critical techniques for high performance execution of select-project-join queries: First, the optimizer should decide where duplicates are removed within the query plan independent of the projections creating them. Second, join algorithms should remove duplicates when sorting or hashing their input, and the optimizer should be capable of predicting its effects.

1. Introduction

We address the problem of optimizing and executing select-project-join (SPJ) queries requiring joins of five or more relations. A proposed health care data-mining application being developed at Microelectronics and Computer Corporation in Austin, Texas is expected to require joins of up to 100 relations. Similarly, applications from logic programming are known to result in expressions with hundreds of joins [10]. The use of nested view definitions, object-oriented databases, and rule-based and knowledge base systems with database back ends are also expected to increase the number of joins typically required in answering a query.

In anticipation of the up-and-coming need for multi-join query support, several [4,7,8,10,14] have investigated the problems involved with optimizing multi-join queries. However, the problem of executing multi-join queries has been largely ignored. In this paper, we take a preliminary stab at the problem by investigating schemes for dealing with duplicate tuples created by application of project. For queries that reference several tables, it is unlikely that every column of every referenced relation is required in the final result. We thereby expect multi-join queries to make liberal use of projections to eliminate the unwanted information.

Since column removal can create duplicate tuples, we wish to know the best way to deal with them. The naive approach of dealing with duplicates is to either (1) never eliminate duplicate tuples or (2) always eliminate duplicate tuples immediately with each projection. We find that because the number of duplicates often explodes exponentially with join arity, the first technique is not a practical option. The second, while keeping a tight bound on the size of intermediate relations, can impose unnecessary overhead due to interruption of pipelining and the hashing or sorting required to identify duplicates. We suggest an alternative method that instead delays the elimination of duplicates until the query optimizer deems it cost effective. The idea is to have the implementation of the project operator simply remove unneeded columns from a given relation, and to add another purely physical operator which removes duplicate tuples independent of the projections creating them. Doing so gives the optimizer utmost flexibility in determining when duplicates are to be removed, preventing unnecessary interruptions of pipelining, and allowing duplicate tuple removal to be performed by physical join operations in order to minimize (and sometimes eliminate entirely) its overhead. Through experiments on random queries, we find this more sophisticated scheme reduces disk IO across a wide range of selectivities.

We apply our ideas by implementing the two-phase optimization algorithm of Kang [9] along with extensions necessary for the duplicate removal technique. We show how techniques in the literature for predicting the size of relations after removing duplicates require modification in order to apply to relations derived by several joins, and discuss the fixes we employ. To develop experimental results, we formulate a method for random query generation approximating expected queries from the proposed health-care data mining application.

In related work, Bhargava, Goel and Iyer [1] investigate pushing projections up as well as down for SPJ queries involving both inner and outer joins. They provide algorithms for exhaustively and heuristically enumerating query plans with different schedulings of project operations. Our idea is similar, though we focus on the multi-join case. We believe the technique is more applicable in the multi-join case because of the potential for exponential blowup in duplicates and the effectiveness of pipelined query plans. They provide algebraic identities for pushing projections up and down, while we implement the technique through transformations so that it may be applied to (non-exhaustive) multi-join query optimizers. Their presentation is cost-model independent and provides empirical support through examples. We evaluate effects of the technique on a wide range of randomly generated queries with respect to a cost-model accounting for pipelining, duplicate removal by buffered join operations, and other high-performance query evaluation methods.

2. Problem Statement

Precisely when--and how to determine when--to eliminate duplicate tuples during evaluation of large SPJ queries is the question we address. The motivating application, being developed at MCC in Austin, Texas, involves applying the LDL++ system to allow formulation of complex queries. Patient data alone is captured in 10 base relations, and reference tables used to decode diagnosis codes, procedure codes, business rules concerning legal claims, etc. number approximately 75. One goal of the project is to correlate patient data with location data describing patient residence and hospital service area in order to discover regional differences in patient care. An important component of the expected queries consists of repeated application of selection, projection, and join on at least (though usually many more than) five base relations. Given the complexity of the joins, it is clear that the desired output will comprise a subset, in fact a small subset, of columns in referenced relations.

This task of removing undesired columns by way of the project operator can and often does produce duplicate tuples. There are two naive methods for dealing with duplicates which are nevertheless commonly employed. The first involves simply ignoring the presence of duplicates unless the user specifies otherwise (e.g. via the DISTINCT SQL keyword). The second involves unconditionally removing duplicate tuples through an explicit sort or hash with each projection applied through the common heuristic of “pushing projections down as far as possible”. An advantage of the first technique is simply that no overhead is incurred by duplicate removal. The overhead results from interrupting pipelining in order to materialize intermediate relations on which duplicate removal is performed, and the sorting or hashing required to identify duplicates. Another advantage of ignoring the presence of duplicates is that since pipelining can be better exploited, the first tuples in a query result can be provided quickly [5]. An advantage of unconditionally removing duplicates is that intermediate result sizes are kept as small as possible, potentially improving the performance of the other query operators [1].

When the number of relations to be joined (*join arity*) is small, there is little potential for significant numbers of duplicate tuples to arise. Many commercial databases therefore choose the first naive method and ignore the presence of duplicates unless told otherwise. For instance, the DB2 products avoid the use of sorting and materializing intermediate results as much as possible [5]. In evaluating larger-arity queries, intermediate results are often materialized due to the improved performance of bushy query plans [9] and the reduced IO of buffered merge or hash join [13]. The advantages of pipelining become less important in comparison to IO savings realized by these techniques. Further, the number of duplicates tuples can explode exponentially with join arity. The added cost of removing duplicates can easily pay off by reducing the cost of subsequent join operations.

3. Concept

We have made extensive use of the words “can” and “potentially” in describing the naive methods for dealing with duplicates since each is advantageous in different situations. Given the distinct advantages of each of these schemes, it seems there should be some ideal trade-off between ignoring and removing duplicates that is query dependent. We therefore propose to allow the optimizer to decide when duplicate tuple removal is to be applied. In order to allow the optimizer the utmost flexibility in dealing with duplicates, we suggest that the physical project operator and a purely physical duplicate removal operator be made separate entities. The physical project operator, which will be used according to the usual heuristic of “pushing projections down as far as possible”, is to be responsible only for removing unneeded columns. The duplicate removal operator can be applied independently from the projections creating the duplicates. The optimizer will determine where to apply duplicate tuple removal according to cost-estimates of query plans considered. The operators are not completely independent since there should be at most one application of the duplicate removal operator for each application of project.

It is easy to find several specific cases where flexibility in duplicate removal is warranted [1]. For instance, relations may be too large to be processed by a fast buffered merge or hash join algorithm unless duplicates are removed previous to its application. Similarly, in some cases the benefits of pipelining may far outweigh the benefits of size-reduction from duplicate removal. The optimizer could allow pipelining along a stretch of joins and projections without removing any duplicates until it is determined to reduce the number of tuples enough to be cost-effective.

Flexibility in removing duplicate tuples is further warranted because in delaying it to the appropriate point, it can often be performed with little or no additional cost. For instance, duplicates can be removed during the merge phase of merge sort [2]. Merge join algorithms therefore allow cheap duplicate removal on both join arguments [6]. The only additional work required by duplicate removal when performed as such is due to sorting on a slightly larger key and a few extra comparisons during the merge phase to skip over the duplicates. This cost will typically more than be made up for by the reduced size of the join result. A similar case holds for the various hash join methods. Most hash join methods take their inner argument and hash its tuples into buckets according to their join column values. Duplicate tuples will be hashed to the same bucket since the join attributes must be among the attributes remaining after the projection in order for the join to be performed. At the point a tuple is placed in a bucket, the hashing scheme can be extended to check and see if an equivalent tuple already exists. If so, the tuple is simply discarded. Duplicate removal on the inner argument of a hash join can therefore be performed with no additional IO cost and a small amount of additional CPU overhead (assuming hash buckets do not grow excessively).

4. Estimating Sizes of Projection Results

In order to effectively determine when and if duplicate removal is cost effective, an optimizer must have a method of estimating the size of a relation after duplicates are removed. There are several techniques in the literature for estimating the size of projection results which can be used for the task, but each is problematic in the multi-join context. For instance, the technique of Chen and Yu [3] fails because it does not consider the fact that value combinations within columns in a relation derived by several joins can be highly correlated. Their technique is as follows: we are given a relation R with column attributes A which we wish to project onto column attributes $A' \subseteq A$ (denoted $R[A']$). We wish to calculate the size of $R[A']$ in tuples ($\{R[A']\}$). The answer is given by a difficult to compute probabilistic function $G(m, n, k)$ where

$$m = \prod_{a \in A_i} \{R[a]\}$$

and $k = \{R\}^{*}$. An inexpensive approximation for G used by Swami [15] is $G(m, n, k) = \min(m, k)$. This approximation provides an upper-bound on the actual value that is typically sufficiently accurate. In computing m, note that $\{R[a]\}$ is simply the distinct value count of column a in relation R -- a statistic maintained by most database systems.

As an example of the problems from assuming column independence in the multi-join context, consider using the above method to estimate the size of $(R \bowtie S)[A_R]$ where A_R specifies the columns of R only. For most reasonable distinct value counts and approximations of G, the result approaches $\{R \bowtie S\}$. However, this combination of operations (equivalent to the semijoin of R and S) produces a relation that must be no larger than $\{R\}$. The problem persists if we compute G with total accuracy instead of approximating it. An optimizer using this approach will usually conclude there is no reduction in size by performing duplicate removal on any join result.

To fix this problem, we must account for the fact that value combinations within columns obtained from the same base relation are highly correlated. The number of distinct tuples within such a combination of columns cannot exceed the cardinality of the base relation containing them. Instead of treating each column independently, then, we treat clusters of columns obtained from the same base relation as a single column. We estimate the number of "distinct values" in such a clustered column as the minimum of the number of tuples in the base relation, or the result of multiplying the individual distinct value counts of the column within the cluster. It is easy to see that this technique produces an estimated result for the size of $(R \bowtie S)[A_R]$ that is $\{R\}$ or less: we treat the columns in A_R as a single "clustered" column since they all come from the same base relation. We multiply the distinct value counts of each column in A_R and use the smaller of this value or $\{R[A_R]\} = \{R\}$ as its distinct value count. We now have that m is at most $\{R\}$ from which the result follows. The idea can be generalized to relations derived from joining more than two base relations in a straightforward manner.

Other (potentially more accurate) techniques in the literature (e.g. [11]) assume that statistics are maintained reflecting the number of values in one column

* We do not trouble the reader with the details of computing n since it is not needed by the simple approximation of $G(m, n, k)$ that we use.

paired with values in another column. Maintaining such statistics in a database of multiple relations, each relation containing multiple columns, can be impractical. Furthermore, it is not at all clear how to propagate this information in order for it to apply to relations derived by several joins. We therefore leave their application to multi-join query processing as an open problem beyond the scope of this paper.

5. Application

Because exhaustive optimization of complex queries is infeasible due to exponential complexities, we implemented the two-phase optimization algorithm of Kang [9] for determining query plans according to each of the two naive duplicate-tuple removal schemes and our proposed technique. This randomized optimization algorithm is designed to efficiently and effectively optimize multi-join queries by applying transformation rules to a given query plan. We chose the version which considers bushy query plans as well as linear plans due to its superiority over linear-only schemes [7]. We had to add one additional transformational rule which swaps in and out an explicit duplicate removal operator. Again, we treat the removal of duplicates independent of column-removal phase of projection, which we assume is performed according to the usual heuristic of pushing projections down as far as possible. The duplicate removal operator can be swapped in at any point following a projection as long as another duplicate removal operator is not already present for the given projection.

For modeling query cost, we used Kang's CM3 cost model with some modification. Kang's CM3 cost model uses the cost expressions from Shapiro [13] for buffered merge and hash join. For modeling nested loops, Kang did not account for buffering nor use of block nested loops, making it cost-ineffective in all but the rarest of situations. We reformulated the cost of nested loops to account for buffering: given a sequence of nested loop join operators joining n relations, buffer space was divided into $n-1$ equally sized portions. Each intermediate result (except for the last) is buffered by one portion, and the last portion is used to buffer the first input relation. The final result of the n-way nested loop join is assumed to be written to disk. For a 2-way nested loop join, the entire buffer can be used to read in as much of the first relation as possible. Nested loops is therefore assumed to be pipelined and performed block-at-at-time, with the size of the block being the amount of buffer space allocated to each portion.

Following Kang, our optimizers assumed 100 pages of available buffer space, pipelining of block nested loop joins only, a page size of 8k bytes, and an attribute value size of 4 bytes. Cost is determined to be the number of disk pages written and read, which we call disk IOs.

We denote the naive scheme that never removes duplicates as NEVER, and our proposed technique COST. We evaluate two implementations of the scheme which always attempts to remove duplicates. The first, ALWAYS_1, performs a hash for removing duplicates with each projection. The second, ALWAYS_2, performs a hash only when the duplicates cannot be removed by the subsequent join operation. COST also exploits the technique allowing duplicates to be removed for free when it can be combined with a subsequent join operation.

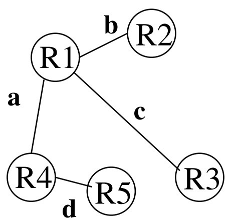
6. Test Suites

Formulating a “realistic” random model of queries and their associated data is a difficult problem since there are many variables involved with processing a query including relation cardinality, number of relations, selectivities, query graph structure, number of columns appearing in the query result, etc. For a particular test suite, each parameter can be fixed to a particular value, or made to vary over a given range and given probability distribution. Kang [9] provides four random test suites which we initially experimented with. Unfortunately, we found that all but the relcat1 catalog frequently produced queries with empty query results when join arity was increased beyond 10. The problem is that the join columns of two relations are filled in independently of one another according to some randomly determined distinct value count. Two relations of different size can thereby have vastly different join values. The net effect of several such situations was that tuples from one join column could easily have very few shared values with those from another. We therefore have chosen to design a set of experiments motivated by an application being developed at Microelectronics and Computer Corporation in Austin, Texas for data-mining on a health care database. Because the project is still in its infancy, we must still rely on randomly generated queries. However, knowledge of the database schema and structure of some expected queries allows us to model the application with some accuracy.

Proposed queries for the application seem to be formulated navigationally. They start from one relation and “link” the interesting tuples in that relation to others with particular join predicates, and continue recursively with the linked relations until all required data has been gathered. The tree-growing routine from Swami [15, pg. 18] for specifying query-graph structure fits this model nicely, since it starts by generating a node and adds nodes by linking them randomly to previously linked nodes. Given a query graph, each node is to represent a relation, and each edge a join predicate. We label every edge with a unique column name, and then generate a relation for each node containing those columns represented by the edges incident upon it, plus an additional uniquely named column to store unique tuple identifiers. The join predicates between two relations are the equality predicates implied between columns with equivalent labels. This ensures that each join predicate is independent of the others, as is assumed by most optimizers including ours. An example randomly-generated query with a join arity of five is displayed in Figure 1.

FIGURE 1. Example Randomly Generated Query

Query Graph



Relation Schema

R1: $\{u_1, a, b, c\}$
R2: $\{u_2, b\}$
R3: $\{u_3, c\}$
R4: $\{u_4, a, d\}$
R5: $\{u_5, d\}$

SQL Query

SELECT u_5,a,c
FROM R1, R2, R3, R4, R5
WHERE R1.a = R4.a
AND R1.b = R2.b
AND R1.c = R3.c
AND R4.d = R5.d

The types of predicates used to link relations are one of three main types: (1) interesting tuples match with exactly one tuple in another relation, (2) interesting tuples match [1-n] tuples in another relation for some value n, and (3) interesting tuples match [0-m] tuples in another relation for some value m. (1) is clearly a common case due to the use of foreign keys to probe the primary key of another relation. Case (2) arises in situations such as "towns located within service area {x}". An example of case (3) is "children of patient {x}". The frequency of each type of join predicate is not known precisely, nor are the expected values of n or m for most relations. We therefore experiment on several classes intended to cover different degrees of mean selectivity. They are described in the table below. The high selectivity class of queries has most (75%) of its join predicates of the type (1) variety. Note that this does not imply that there is a one-to-one relationship between join columns! If the linked relation is smaller than the originating relation, then tuples in the originating relation may join with the same tuple in the linked relation. Similarly, there are cases where tuples in the (larger) linked relation need not join with any tuples in the originating relation. The other type of predicate, accounting for the remaining 25% of predicates, links a tuple in one relation with 0-2 tuples in the other. This provides some potential for blowup, but the blowup is limited by the fact that some tuples will not join at all. The medium and low selectivity classes gradually increase the potential for blowup in intermediate results. Added for these classes is the join predicate of type (3) which produces a join result bigger than the smaller of the two input relations.

TABLE 1. Classes of Test Queries

Selectivity	Join Predicate Distributions
High	75% 1 -> 1, 25% 1 -> [0-2]
Medium	50% 1 -> 1, 25% 1 -> [0-2], 25% 1 -> [1-2]
Low	50% 1 -> 1, 25% 1-> [0-3], 25% 1 -> [1-2]

The number of tuples in each relation is expected to vary wildly. Some relations such as those containing patient records are very large (many megabytes), whereas those for decoding diagnostic procedures are relatively small. In order to allow experiments over several queries, we chose to allow the number of tuples in each relation to vary (uniform randomly), but over the somewhat limited range [1000-10000]. This range is intended to capture the expected number of tuples in most relations after selections have been performed. In practice we might expect far larger base relations, but this range is large enough to illustrate the general effects of our schemes. We can expect that larger relations would magnify the performance benefits of intelligent duplicate removal schemes since effective use of buffer space becomes even more critical.

The attributes required in the final result are expected to vary depending on the query. However, it is highly unlikely that every attribute within a referenced relation will be necessary. For instance, sometimes we will want to find patients receiving a particular type of treatment without needing the treatment description appearing in the final result. We decided to randomly choose 25% of attributes to appear in the final query result. The value seems higher than what would be needed in for most large queries, but small enough to adequately demonstrate the effects of each scheme.

7. Experimental Results

The results of various experiments are plotted in Figures 1-3. The number of Disk IOs was calculated by optimizing the query and then simulating execution of the resulting plan. Simulating the execution involved performing the necessary projections and joins to determine the actual (as opposed to estimated) sizes of intermediate results, and then plugging in these relation values into the cost formula used by the optimizer. Assuming the cost-formula provide accurate representations of algorithm cost, this calculated cost should accurately represent the cost of executing the query on a DBMS fully implementing each operation modelled. Each data point in the figure represents the average cost of 200 queries optimized and executed in this fashion.

The first experiment (Figure 2) demonstrates the effects of queries with high selectivity. For these queries, the number of duplicates, even given liberal applications of project, remains small across all join arities. Notice that the cost of ALWAYS is always higher than both NEVER and COST, and the disparity increases with larger join arities. The ALWAYS schemes perform poorly because they are constantly interrupting pipelines to remove duplicates, even though doing so has no performance advantage. COST and NEVER exploit pipelining to its fullest extent. ALWAYS_1 performs particularly bad because it is unable to have join operations remove duplicates, and must resort to explicit hashes. For clarity, we do not present data this scheme in the following figures.

For queries where selectivity is such that growth in intermediate results is sometimes possible (Figure 3), the NEVER policy becomes unacceptable at larger join arities. Since several joins still remain relatively selective, pipelining is sometimes cost-effective, so COST is able to outperform ALWAYS. The performance difference again increases with increasing join arity.

At lower selectivities (Figure 4), pipelining is not as often an option since the large intermediate results favor buffered merge and hash joins. Still, COST is able to significantly outperform ALWAYS due do the presence of several highly selective predicates. In examining its query plans, we found a reason outside pipelining that COST outperforms ALWAYS_2: COST makes more use of the faster buffered hash join algorithm than ALWAYS_2 because it can delay the duplicate removal for its left argument until a later operation. ALWAYS_2 prefers the merge join since it removes duplicates from both arguments, thereby preventing the need for an explicit duplicate removal phase outside the join algorithm.

In examining query plans produced by the COST optimizer for our queries, we found that despite the performance advantage of hash join over merge join with respect to two-way joins [13], merge join is often preferable since it can remove duplicates from both arguments. Hash join is still well exploited, particularly on the medium and high selectivity queries. Because our initial relations are relatively small, pipelining could be well exploited when few relations are to be joined. This can be seen from the data at join-arity five points: NEVER and COST perform nearly identically across all test suites. Most query plans at these points are left-deep trees applying pipelined block nested loops. We can expect that larger base relations would result in a disparity at smaller join arities, with COST outperforming NEVER. Small join queries which benefit from flexible duplicate removal schemes appear in [1].

FIGURE 2. Performance Results for High Selectivity Queries

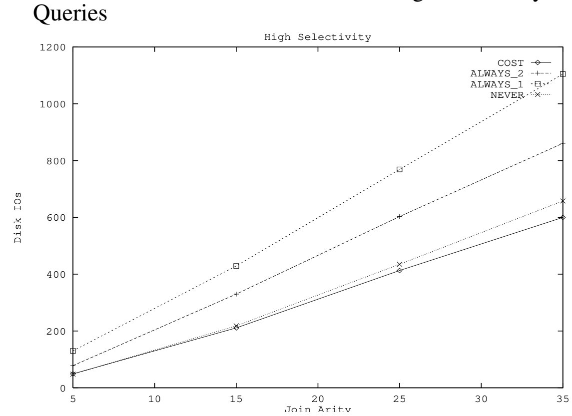


FIGURE 3. Performance Results for Medium Selectivity Queries

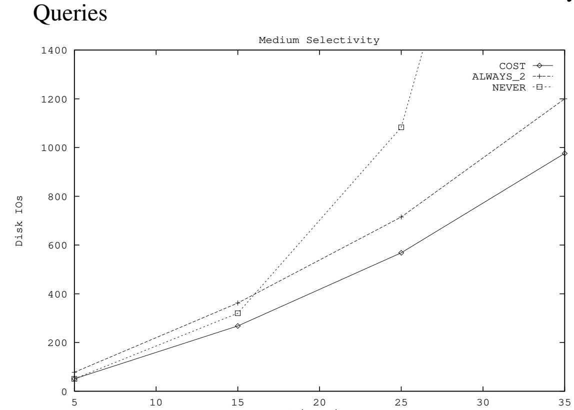
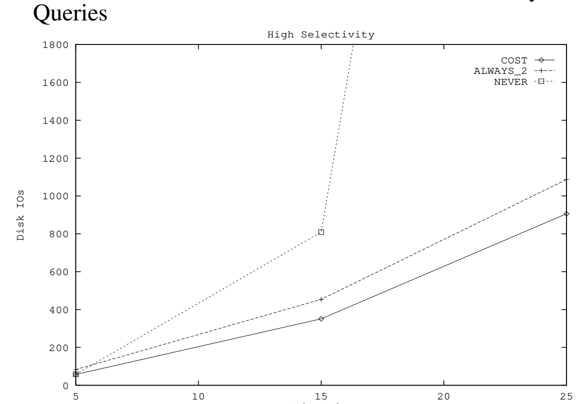


FIGURE 4. Performance Results for Low Selectivity Queries



8. Conclusion and Future Work

We have shown that allowing flexibility in dealing with duplicate tuple elimination can increase the effectiveness of multi-join query processing across a wide range of queries. The keys into realizing its effectiveness are accurately estimating the sizes of projected relations by knowing that columns from the same base relation are highly correlated, and applying duplicate removal when it is free given the subsequent join operation. The lower the join selectivity and the higher the join arity, the more improvement provided by the technique. Another interesting result of our investigation is that merge join is often preferable to hash join algorithms when processing multi-join queries even though hash join is always preferable when join arity is 2 [13].

We have not investigated the effect of using these techniques with alternate cost models or cyclic queries, and leave these issues open to future work. We do not expect alternate cost models to change things significantly since duplicate tuple removal is typically applied in combination with the various join algorithms, and thus imposes very little additional CPU cost and no additional disk IO. Cyclic queries will not allow projections to be pushed as far down as acyclic queries since join columns will generally have to be propagated further. The more dense the query graph, the more we can expect performance to be adversely affected since duplicates will not arise until later in the evaluation chain. Nevertheless, queries with only a few cycles should perform similarly to the acyclic cases evaluated here.

We also leave open to future study the effect of these techniques when distinct value counts are either incomplete or inaccurate, and/or distributions of join values are non-uniform. We suspect some form of duplicate tuple removal will become more desirable in these situations since the techniques provide better upper-bounds on intermediate result sizes regardless of the relation contents. For COST to perform well in such a situation, we imagine that several optimize-execute phases should be used. Graefe [6] suggests optimizing and executing several local portions of a large query independently to control error propagation. Given fast global optimization methods such as Kang's, it may be beneficial to instead optimize globally, begin query execution, and perform global re-optimization with updated information on intermediate result sizes and selectivities whenever the size of an intermediate result differs from its predicted value by a particular threshold.

While optimization of multi-join queries has been well studied, the problem of executing them has been largely ignored. This paper has taken a preliminary step at extending the standard arsenal of physical query operators for use in multi-join query processing. We anticipate several further advances along these lines.

ACKNOWLEDGMENTS

This work was supported by an AT&T Bell Ph.D. Fellowship. I thank Professors Daniel P. Miranker and Don S. Batory for their comments and assistance.

REFERENCES

1. G. Bhargava, P. Goel, and B. Iyer, No regression algorithm for the enumeration of projections in SQL queries with joins and outer joins, *Proceedings of CASCON-95*, Toronto, Ontario, Canada, 87-99, 1995.
2. D. Bitton and D. J. Dewitt, Duplicate record elimination in large data files, *ACM Transactions on Database Systems*, 8(2), 255-265, June 1983.
3. M. Chen and P. S. Yu, Combining join and semi-join operations for distributed query processing. *IEEE Transactions on Knowledge and Data Engineering,* **5**(3), 534-542, 1993.
4. C. Galindo-Legaria, A. Pellenkoft and M. Kersten, Fast, randomized join-order selection--why use transformations?. In *Proc. of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, 1994.
5. P. Gassner, G. M. Lohman, K. Bernhard Schiefer, Y. Wang, Query optimization in the IBM DB2 Family. *IEEE Bulletin of the Technical Committee on Data Engineering*, 16(4), 4-18, Dec. 1993.
6. G. Graefe, Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 73-170, June 1993.
7. Y. E. Ioannidis and Y. C. Kang, Left-deep vs. bushy trees: An analysis of strategy space and its implications for query optimization. In *Proc. of ACM-SIGMOD*, 168-177, 1991.
8. Y. E. Ioannidis and Y. C. Kang, Randomized algorithms for optimizing large join queries. In *Proc. of the 1990 ACM-SIGMOD Conference,* 312-321, 1990.
9. Y. C. Kang, *Randomized Algorithms for Query Optimization*. Ph.D. Dissertation, University of Wisconsin-Madison, TR 1053, October 1991.
10. R. Krishnamurthy, H. Boral, and C. Zaniolo, Optimization of nonrecursive queries. In *Proc. of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, 128-137, 1986.
11. R. Mukkamala and S. Jajodia, A note on estimating the cardinality of the projection of a database relation, *ACM Transactions on Database Systems*, 16(3), 564-566, Sept. 1991.
12. P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD*, 1979.
13. L. D. Shapiro, Join processing in database systems with large main memories, *ACM Transactions on Database Systems*, 11(3), 239-264, September 1986.
14. A. Swami, Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proc. of the 1989 ACM-SIGMOD Conference,* 367-376, 1989.
15. A. Swami, *Optimization of Large Join Queries*, Stanford University, Department of Computer Science Ph.D. Dissertation, May 1989.