Extending the Algebraic Framework of Query Processing to Handle Outerjoins

Arnon Rosenthal
David Reiner

Computer Corporation of America
Four Cambridge Center
Cambridge, Massachusetts 02142 USA
(617-492-8860)
(Arpanet: {rosenthal, reiner}@cca-unix
Usenet: ucb!decvax!cca!{rosenthal, reiner})

Abstract

A crucial part of relational query optimization is the reordering of query processing for more efficient query evaluation. The reordering may be explicit or implicit. Our major goal in this paper is to describe manipulation rules for queries that include outerjoins, and views or nested subqueries. By expressing queries and processing strategies in terms of relational algebra, one can use the ordinary mechanisms of query optimization and view substitution with a minimum of disruption. We also briefly examine aggregate operators, universal quantifiers, and sorting.

1. Introduction

Relational query optimization is based on the fact that the order of evaluating predicates and joins is immaterial to the result of a query. A crucial part of optimization is the reordering (explicitly or implicitly) of query trees for more efficient query evaluation. Manipulation rules for these trees are straightforward.

Our intent in this paper is to describe tree-manipulation rules for queries including outerjoins. We have succeeded in defining rules that can be added fairly straightforwardly to existing mechanisms for query optimization and view substitution. It is not necessary to add a new paradigm for treating queries with outerjoins. We also briefly examine aggregates, universal quantifiers, and sorting.

We begin, in Section 2, with a short exposition of familiar query processing algorithms, from the point of view of relational algebra. Queries may reference previously-defined views, and may include certain types of nested subqueries. The relational algebra point of view enables us to see clearly what must be done to process new algebraic operations during optimization.

In Section 3, we define outerjoin and introduce several new operators needed to fully optimize queries that contain them. Extensions to process aggregate operators, more complex nested queries, universal quantifiers, and sorting are discussed in Section 4.

2. Query Processing

In this section we give an overview of relational query optimization from an algebraic point of view. [SMIT75] and [DAYA83b] took a similar approach to rearranging the operations in a query. See [KIM84] or [REIN82a] for a more detailed treatment of query optimization.

A relational query may be represented as a tree, having relations as leaves, and operators as internal nodes. For query optimization purposes, {multiway Cartesian product, selection, projection, union} is an easily manipulated set of operators. Most queries in SQL and QUEL [DATE81] map directly to a tree composed of these operators. For simplicity, we do not present simplification rules for moving union operators. Thus, our attention is restricted to conjunctive queries [DAYA83b].

A query without nesting or references to views has at most one product node. Selections and projections are at the top of the tree, above

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984

the product. A multiway product plus its associated selections is equivalent to a (multiway) join. Such a product does not imply any order for combining the relations.

Multilevel tree structures (or "multilevel queries") arise in several ways:

- A query Q references a view V. Query modification modifies Q's query tree; the leaf for V is replaced by the tree for the query which defines V. (For simplicity, we assume relations in V do not appear in Q.)
- The query includes a nested subquery. (We consider only subqueries that do not reference the enclosing query. See [KIM82] and [LOHM84] for more general treatments.)
- A parenthesization mechanism produces nested subqueries that can be referenced from the enclosing query [DATE83].

Since an access strategy uses only two-way joins, the query optimizer must choose an association. It first tries to reduce the number of product nodes, "flattening" the multilevel query tree [STON75], [KIM82], [LOHM84]. It permits choice among a greater variety of join orders, including orders that are inconsistent with the nesting in the original multilevel query.

The optimizer then (explicitly or implicitly) generates operator trees in which each multiway product has been replaced by two-way products. Operators that reduce the size of the intermediate results (selections and projections) are then pushed down the trees as far as possible (subject to the laws of relational algebra [SMIT75]). The choice among alternative query trees is made heuristically, or by using an explicit cost model. (See [REIN82b] for a comparison of how various optimizers generate alternatives and choose among them.)

An algebraic exposition of query optimization needs to address:

1. The tree flattening process. The flattening step moves operators (e.g., select and project) up through the higher product node or below the lower product node. When two product nodes become adjacent, we combine them. The multiway product offers greater freedom in generating reassociations.

2. Generating reassociations. We have no changes to propose in the way an optimizer generates and tests alternative associations. Our algebraic transformations can be applied within: (a) an optimizer that chooses a tree by some heuristic, or (b) one that enumerates and examines trees one at a time [KOOI82], or (c) one that uses dynamic programming to optimize substructures common to several trees (e.g., a join of R1 and R2) [SELI79], [ROSE82].

Figure 1 shows two equivalent product expressions that use different associations. R1 and R2 are joined. (R1 join R2), R3, and R4 are then joined in some unspecified order.

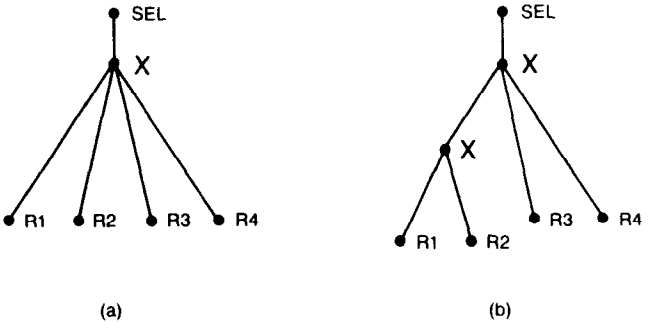

Figure 1. Splitting An M-Way Product Node

3. Moving unary operators. The purpose of reassociating is to obtain an efficient join ordering. The sizes of intermediate results are then reduced by performing selections and projections as early as possible. Thus, one wishes to push such operators down the tree, while operators that increase the size of the output should be moved up the tree and applied as late as possible.

Control can alternate between generating a new association and moving the unary operators. The combination of actions will be called unflattening the tree.

3. Extending Query Optimization to Outerjoins

3.1 Incorporating New Constructs into an Optimizer

To integrate a new construct (e.g., Outerjoin, Total, Unique, Generalization) into the query optimization process, one must:

1. Add operators to the relational algebra so that the construct may be expressed algebraically.
2. Determine useful operator-shuffling transformations in the extended algebra, so the operators may be moved aside during flattening, and moved around after the association has been chosen.

The algebraic approach keeps the query optimizer's logic from getting more complex as features are added. Rules for shuffling newly-introduced operators are added without changing the already-established framework.

We strive to keep all algebraic operators simple. Complex constructs are expressed as a composition of operators that have simple manipulation rules. Thus, product (rather than join) is our primitive for combining relations, and outerjoins are built out of more elementary single task operators.

3.2 Outerjoin Definitions

Information from two relations often must be combined in ways different from standard relational joins, to preserve information from one relation that is not matched by entries in the other. The "information-preserving join" or "outerjoin" permits this to be done [LACR76], [CODD79].

Proposals for extending SQL to handle outerjoins in SQL were presented in [CHAM80] and [DATE83]. The one-sided outerjoin of two relations is expressed below in a syntax adapted from [DATE83]:

```
Select <output list>
From R1, R2    Preserve R1
Where P
```

The result is the projection onto <output list> of

(R1 join(on P) R2) U {(t1, null) | no tuple of the join included t1}.

The result consists of the join, plus unmatched tuples of R1 padded with nulls. All tuples of R1 appear in the result. R2 could be similarly preserved. If both relations are listed in the Preserve clause, the query result is a two-sided outerjoin, consisting of

(R1 join(P) R2)
U {(t1, null) | no tuple of the join included t1}
U {(null, t2) | no tuple of the join included t2}

An interesting use of two-sided outerjoins appears in [DAYA83b], where queries involving generalizations are reduced to queries using the simpler operators of "aggregate" and outerjoin.

We provide syntax for two-operand outerjoins, since multiway outerjoins are not well-defined [DATE83]. More complex expressions involving outerjoins can be obtained by allowing R1 or R2 to be a view.

A straightforward implementation of the definition of outerjoin (i.e., using two or three queries to create the pieces of the union) is a poor processing strategy. Instead, one-sided outerjoins are implemented by nested loops much like an ordinary join [DAYA83a]. The join logic is modified so that if t1 has not successfully been joined, (t1,null) is included in the result. There is no need to perform a separate set union, and only the Preserved relation needs to be explicitly materialized.

Two sided outerjoins require a separate implementation. They sometimes can be computed efficiently by merging the two relations being outerjoined.

3.3 General Notations

R, S, T, R1, R2, Z: Relation names. If a relation name appears where a set of attributes is needed, we mean the attributes of the relation.

attrib(R): Attributes of relation R.

attrib(R&S): Attrib(R) intersect attrib(S).

Proj[R,S]: The projection of R onto the attributes in S. Proj[R,S] will mean the projection of R onto the attributes in R-S.

Sel(P)(R): Returns the tuples of R that satisfy predicate P. We reserve the term "selection" for situations in which P is evaluated by looking at a single tuple. Operators that return a subset of the input relation will be called subsetting operators. Examples include selections, Unique, and Omit (introduced later).

query tree: As shown in the figures, a query tree is a relational algebra expression that represents a query. Each node of the tree represents a relation, either an input to the query, or a calculated result. We assume that each node has a distinct name, generated either by a human (for base relations and views) or by the system (for nested subqueries or system-generated tuple-variables). To simplify notation, we assume that if the query references a relation twice, it uses two different names. Attributes in a view will be given the same name as in their underlying relations.

3.4 Algebraic Definitions for Outerjoins

labelled nulls: In the original query, the allowable patterns of nulls are expressed by Preserve and by the query nesting. Optimization can change the nesting. It causes partly-null tuples to be introduced, joined, and eliminated in a way that preserves the original query's result. When manipulating the rearranged query tree, we must know which operand in the original query corresponds to each null. We then can ensure that all other attributes in the operand are assigned that same null.

Each relation for which a null tuple may be used is given a distinct labelled null. The null value assigned to attributes of R is denoted #R. The system is not required to retain the labels after query processing is completed.

Aug<R|#S>: R U {the tuple (#S,...,#S)}. Attrib(S) must contain attrib(R). The original tree for a query contains only augmentations of the form Aug<R|#R>.

(P|#S): The relaxed predicate (P|#S) is defined as "P OR (all attributes of S that appear in P have value #S)". The associated selection, applied to R, is written Sel(P|#S)(R).

dominates: For scalar values x and y, x #S-dominates y if x=y or x is nonnull and y is #S. For distinct tuples t′ and t, t′ #S-dominates t if for every attribute A, t′.A #S-dominates t.A.

A join that uses a relaxed predicate yields tuples not present in the outerjoin. The Omit operator below throws away dominated tuples.

Omit<R|#S>: = {t in R | there is no t′ in R which #S-dominates t}.

We are now able to express an outerjoin algebraically. The steps are:

- Allow null tuples in one or both joined relations (Augment).
- Relax the join predicate when nulls crop-up in Cartesian products (relaxed selections).
- Eliminate "dominated" tuples from the result (Omit).

The query processor translates the one-sided outerjoin

"Select <output-list> From R1, R2
Preserve R1 Where P"
to the algebraic expression:
Proj[Omit<Sel(P|#R2)(R1 X Aug<R2|#R2>) |#R2>
|output-list]

For a two-sided outerjoins of R1 and R2 with join predicate P, the algebraic expression is:

Proj[Omit<Omit<Sel((P|#R1)|#R2})
(R1 X Aug<R2|#R2> | #R1> | #R2>, output-list]

3.5 Examples

Examples in this paper refer to the schema in Figure 2. Contracts are composed of Tasks, which are staffed by (Fractions of) Employees.

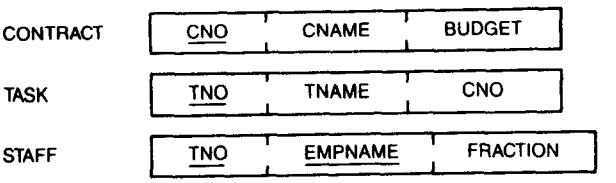

Figure 2. Relational Schema Used in Examples (Keys are underlined)

A view defined by an outerjoin query is shown in Figure 3. The query tree that results when the view is referenced in a query is shown in Figure 4. The figures use the tree structure to determine operands, rather than mentioning them explicitly. The bar over a set of attributes indicates that the indicated attribute is projected away.

```
VIEW TS IS
    SELECT TASK.TNO, TNAME, EMPNAME, FRACTION, CNO
    FROM TASK, STAFF PRESERVE TASK
    WHERE TASK.TNO = STAFF.TNO
```

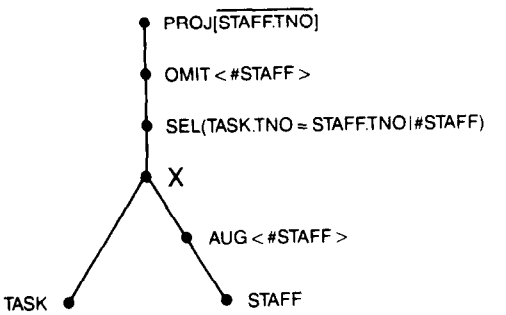

Figure 3. A View Containing An Outerjoin

```
Q1: SELECT TNO, TNAME, EMPNAME
    FROM TS
    WHERE TNAME = 'REPORT WRITER'
```

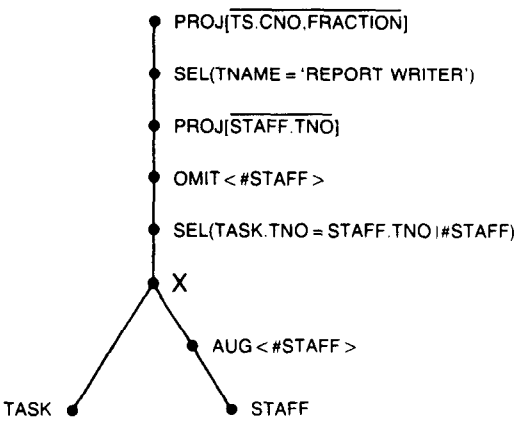

Figure 4. A Query to a View Containing an Outerjoin

Figure 5 shows a more complicated query, in which the view is itself an operand of an outerjoin. The query tree has three levels of nesting. The innermost level is the view TS from Figure 3. This is outerjoined with Contract to produce all available information about all contracts. Finally, we need another level to impose on the result a nonrelaxed predicate that is not part of any join.

Note that if (Cname="Database design workbench") had been included in the predicate defining the outerjoin, it would have been relaxed w.r.t. #TS, permitting information on all contracts. One might argue that the query processor should recognize single-relation predicates and separate them from the outerjoin. But the parser would not be able to detect single-relation predicates within user-supplied code for a predicate. Query semantics should not depend on how the predicate is presented to the system.

3.6 Flattening Rules

The basic step in flattening a query tree is to combine two Cartesian product nodes (referred to as "HIGH" and "LOW") that are separated only by unary operators. The process can be repeated to further flatten the query. The familiar query modification procedure [STON75] accomplishes this for query trees whose operators are product, projection, or selection. Because the commutativity conditions for those operators were so clear, Stonebraker did not use a formal algebraic model of query processing to justify the algorithm.

We flatten queries involving outerjoins, using an algorithm that extends [STON75]. First, operators between HIGH and LOW move up (Proj, Sel, Omit) or down (Aug). Some operators are modified as they move, to preserve the query result. Adjacent Cartesian product nodes then can be combined.

The query in Figure 4 did not require flattening, as it had only one product node. The result of flattening the query in Figure 5 is shown in Figure 7. When an operator moves down from a view to an underlying relation, references to attributes are changed to use the names from the underlying relation.

Flattening Algorithm:

If an augmentation appears between HIGH and LOW, then:

1. Move Aug below all projections, selections, and other operators above LOW, until it is immediately above LOW. The rules used are:

   1.1 Aug<Sel(P)(R)|#S> = Sel(P|#S)(Aug<R|#S>).

   1.2 Aug<Proj[R,Z]| #S> = Proj[Aug<R|#S>, Z].

   1.3 Omit<Aug<R|#S> |#T> = Aug<Omit<R|#T> | #S> if S and T are distinct.

(The equalities in 1 and 2 assume that on each path in the tree, at most one Augment will be permitted to introduce each labelled null.)

2. Move Aug below LOW, creating a new uniformity selection, denoted Unif(#S)(R). This enforces the predicate "If any attribute in attrib(R&S) has value #S, all of attrib(R&S) have value #S." The rule is:

   Aug<R1 X R2 X ...| #S> =
   Unif(#S)(Aug<R1|#S> X Aug<R2|#S> X ...)

Regardless of whether Aug appeared between HIGH and LOW:

3. Move the selections, projections, and omits above HIGH. (This includes any uniformity selections created in rule 2.) Rules are:

   3.1 Sel(P)(R1) X R2 = Sel(P)(R1 X R2).

   3.2 Proj[R1,T] X R2 = Proj[R1 X R2, T U attrib(R2)].

   3.3 If R2 does not include #S, then
   Omit<R1|#S> X R2 = Omit<R1 X R2|#S>
   Otherwise, Omit<R1|#S> X Omit<R2|#S> = Omit<R1 X R2|#S>.

(The above rules are applied in reverse in the unflattening process. For the reverse transformation to be defined, R1 must include all necessary attributes for the select, project, or omit.)

3.7 Unflattening Rules

This section discusses how unary operators are moved to better positions after a multiway product node has been split. (The optimizer makes association decisions by splitting an m-way product node into two nodes, with k and (m-k+1) inputs, for some k>1.) Operators that reduce the size of the result should be executed as early as possible. Thus, we push selections and projections down the tree. Operators that increase the size of the operand are moved upward (e.g., Aug). Where possible, simplifications are made and operators removed.

The algorithm for moving operators down starts with the unary operator just above the product node (denoted X). It attempts to take each operator (denoted op) currently above X and apply it instead to some input to the product. If there is an operator between op and X with which op does not commute, then leave op in place.

Sometimes an operator that cannot move can be split into two operators, one of which can move down. For example, one may be able to move part of a projection below a join predicate or a Unique or Omit, if the attributes projected away are inessential.

The detailed rules for moving unary operators to the most advantageous positions are given below. Their results are illustrated in Figure 7, which shows the result of unflattening the query of Figure 4, and in Figure 8, which shows the result of unflattening the query of Figure 6.

Before the rules can be given, we need to restrict ourselves to well-behaved operators, namely subsetting operators which prefer non-null values to null values. A subsetting operator op(R) is #S-monotonic if [(t is in op(R) and (t' #S-dominates t)] implies that t' is op(R-{t} U {t'}). Unique and Omit are #S-monotonic.

The most common kind of monotonic subsetting operator is selection using an #S-monotonic predicate: A predicate pred is #S-monotonic if [t satisfies pred and t' #S-dominates t] implies that t' satisfies pred. Predicates that do not mention #S are #S-monotonic (e.g., "Fraction=0.5"). If P is #S-monotonic, (P|#T) is #S-monotonic if #T is different from #S. Other #S-monotonic predicates are "(T.A is not #S)" and also "(T.A is not null)".

Rules 4-6 below guide the operator movement. They are proved in [ROSE83].

4. If op is #S-monotonic, then:
   Omit<op(R)|#S> = op(Omit<R|#S>).

5. Proj[Omit<R|#S>, T] = Omit<Proj[R,T] | #S> if either:
   All deleted attributes are functionally dependent on attributes that are in T but not in S; OR
   (R represents a composition of projections and #S-monotonic operators applied to (R1 X R2), the projection preserves all attributes of R1, and attrib(R&S) $\subseteq$ attrib(R1)).

6. Rules from ordinary relational algebra are also used in unflattening [SMIT75].

6.1 Selections commute.

6.2 Projections commute.

6.3 Project commutes with selection, if the project does not discard an attribute needed by select.

3.8 Simplifications

The system can use other rules to make local or minor simplifications. Since all processing is done in the same tree model, one has considerable flexibility in choosing the time to perform them. Some simplifications involving outerjoins are:

Remove unnecessary augmentations: If Sel(P) appears above Aug<R|#S> and P is false for all tuples that contain #S for the attributes of R, then the null tuple cannot appear in the query output. The augmentation (and the corresponding Omit) can be removed. For example, if a query on TS includes a predicate (FRACTION=0.5), then only tuples with nonnull STAFF entries can satisfy the query. The operators Augment<#STAFF> and Omit<#STAFF> may in that case be removed.

Remove unnecessary joins: Suppose some relation R has no attributes in the output list, and none in any predicate except an outerjoin with a Preserved relation. Then R may be removed from the query. (However, the removal may change the number of duplicate tuples returned.)

Remove unnecessary uniformity predicates: If the tree includes Unif(#S) (Aug<R|#S>), the uniformity predicate can be omitted.

4. Handling Additional Constructs

Aggregates: For aggregate operators (Max, Sum, Median, ...), it is easy to deduce the necessary transformation rules (or the fact that no transformation is permitted). For example, Unique commutes with Omit, with selections, and with Augment; it commutes with projection as long as a key survives the projection; it cannot move up through a product.

Universal quantifiers: [KIM82] and [DAYA83a,b] point out that outerjoins may be used to process queries with universal quantifiers. ([DAYA83a] can produce better strategies for quantified queries than our general purpose rule, but he introduces considerable special-purpose machinery.)

Define a simple selection predicate Has_null(#S) to accept only tuples of R where all attributes of S have the value #S. Now

"{x in R1| for all y in R2, P(x,y)}"
is equivalent to:
Define View V as Select * From R1, R2 Preserve R1
    Where not P(R1,R2)]
Select R1.* From V Where Has_null(#R2)(V)

5. Directions for Further Research

Outerjoins: The conditions for moving operators can be weakened somewhat, and perhaps simplified.

The query processor must recognize monotonic predicates. For predicates defined over the algebra understood by the query processor, this is easy. If one wishes to extend a system possessing outerjoins by adding predicates for new datatypes (e.g., time intervals, geometric regions), then the processor must be informed about the monotonicity of these predicates.

Complex nested queries: The operator tree model of queries would need considerable extension to handle nested queries which reference relations in the outer query. [LOHM84] explores these issues.

Sort and group by: An important line of research is to find a useful algebraic model for operators that are rather physical (e.g., Sort, access via index). One could then treat the rules for placing the physical operators as transformations on the query tree, reducing the amount of ad hoc code needed for physical optimization. One would specify how each physical operator moves through select, project, product, aggregates, etc. Note that when sort order or grouping are considered, the order of a product's operands affects the result.

Group-by has already had considerable attention. [OSZO] considers algebraic properties of grouped relations, using algebraic rules to improve query processing. [CERI83] presents a relational algebra for partitioned relations.

6. Conclusions

An explicit algebraic framework makes it easy to extend a query optimizer to handle new semantic constructs. One first expresses the new constructs with (possibly new) relational algebra operators. One can then specify the operators' algebraic properties (which can be quite complex) independently of the algorithms for flattening and unflattening query trees.

In the future, researchers will wish to add new datatypes and to use databases as a foundation for knowledge bases. The ability to add new constructs to the query language and to obtain powerful optimization will be important. Query processors will need to be extendible without major rewriting or changes to the conceptual framework. Our method of incorporating outerjoin into query optimization is an interesting, nontrivial example of that capability.

7. References

[CERI83] Ceri, S., and G. Pelagatti, "Correctness of Query Execution Strategies in Distributed Databases", TODS, Vol. 8 No. 4, December 1983.

[CHAM80] Chamberlin, D.D., "A Summary of User Experience with the SQL Data Sublanguage", Proc. Int'l. Conf. on Databases, Aberdeen, Scotland, July 1980.

[CODD79] Codd, E.F., "Extending the Database Relational Model to Capture More Meaning", TODS, Vol. 4, No. 4, December 1979.

[DATE81] Date, C.J., An Introduction to Database Systems, Third Edition, Addison-Wesley, 1981.

[DATE83] Date, C.J., "The Outer Join", Proc. Second Intl. Conf. on Databases, Cambridge, England, September 1983.

[DAYA83a] Dayal, U., "Processing Queries with Quantifiers: A Horticultural Approach", PODS, Atlanta, Georgia, March 1983.

[DAYA83b] Dayal, U., "Query Optimization in Multidatabase Systems", VLDB '83, Florence, Italy.

[KIM82] Kim, W., "On Optimizing a SQL-Like Nested Query", TODS, Vol. 7, No. 3, September 1982.

[KIM84] Kim, W., D. Reiner, and D. Batory, eds., Query Processing in Database Systems, to be published by Springer-Verlag in late 1984.

[KOOI82] Kooi, R., and D. Frankforth, "Query Optimization in INGRES", in [REIN82a].

[LACR76] Lacroix, M., and A. Pirotte, "Generalized Joins", SIGMOD Record, Vol. 8, No. 3, September 1976.

[LOHM84] Lohman, G., D. Daniels, L. Haas, and R. Kistler, Selinger, P., "Optimization of Nested Queries in a Distributed Relational Database", in VLDB '84.

[OSZO] Ozsoyogu, Z.M. and G. Ozsoyoglu, "An Extension of the Relational Algebra for Summary Tables", 2ond Int'l. Database Workshop,

[REIN82a] Reiner, D., ed., Database Engineering, Vol. 5, No. 3, September 1982.

[REIN82b] Reiner, D., and A. Rosenthal, "Strategy Spaces and Abstract Target Machines for Query Optimization", in [REIN82a].

[ROSE82] Rosenthal, A., and D. Reiner, "An Architecture for Query Optimization", SIGMOD, Orlando, Florida, June 1982.

[ROSE83] Rosenthal, A., and D. Reiner, "Proofs for Query Processing with Outerjoins and Views", working paper, June 1983.

[SELI79] Selinger, P.G., et al., "Access Path Selection in a Relational Database System", SIGMOD, Boston, Massachusetts, May 1979.

[SMIT75] Smith, J.M., and P.Y.-T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface", CACM, Vol. 18, No. 10, October 1975.

[STON75] M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," ACM-SIGMOD Conf., 1975, 65-7.

```
Q2: SELECT ALL
    FROM (SELECT CONTRACT.CNO, CNAME, BUDGET, TNO, TNAME, EMPNAME
          FROM CONTRACT, TS PRESERVE CONTRACT
          WHERE CONTRACT.CNO = TS.CNO)
    WHERE CONTRACT.CNAME = 'DATABASE DESIGN WORKBENCH'
```

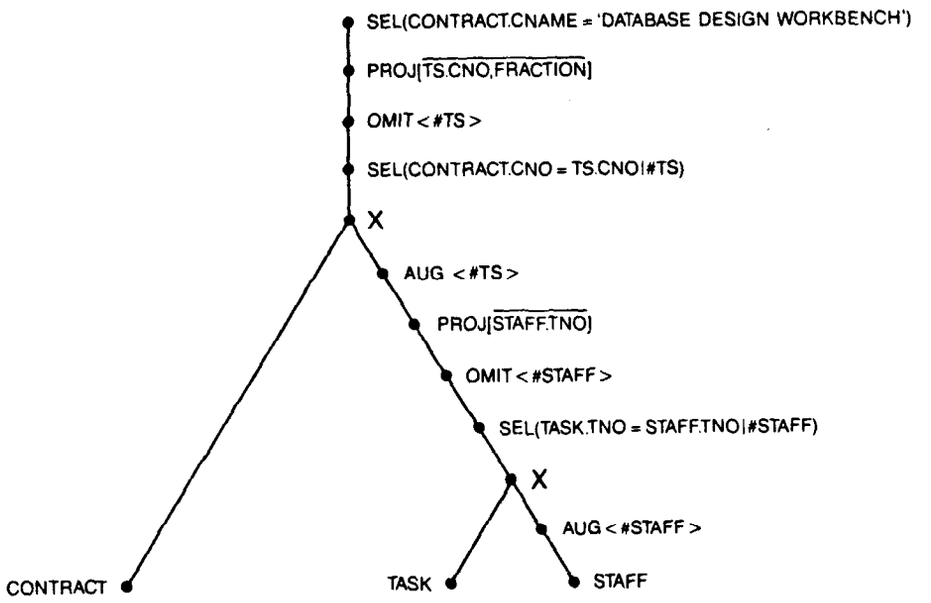

Figure 5. Initial Tree for a Query with Two Outerjoins

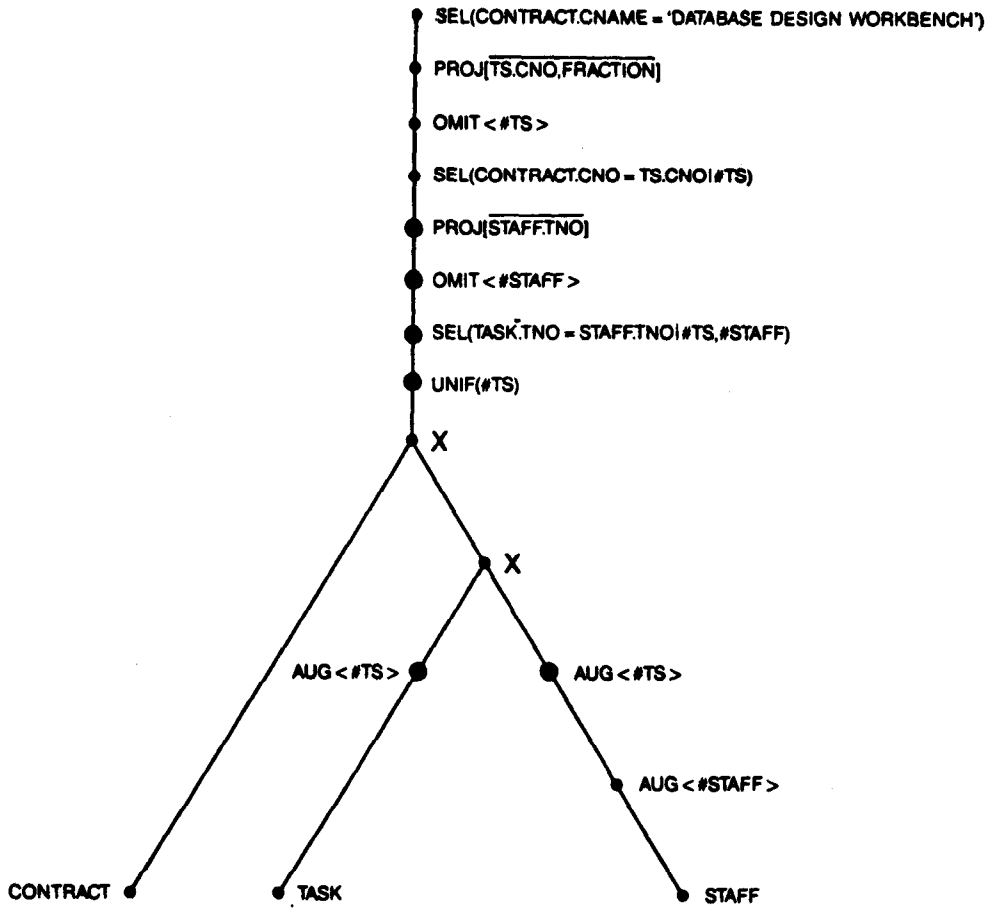

Figure 6. Q2 During Flattening; Just Before Product Nodes are Combined

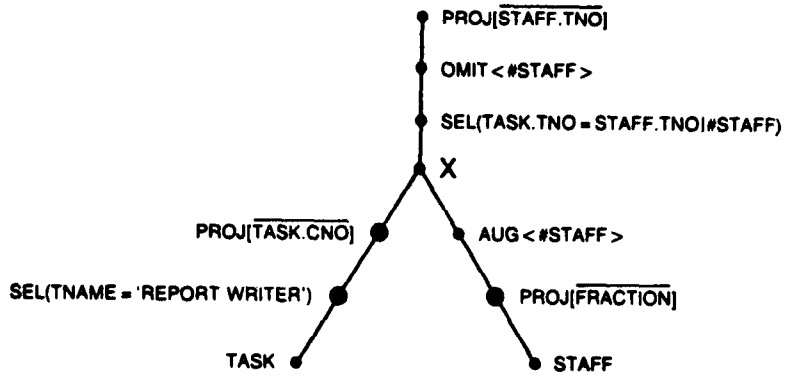

Figure 7. Query Q1, After Unflattening
(Large nodes indicate operators which have moved since the previous figure)
("TS.CNO" is renamed "TASK.CNO" when the projection is moved down)

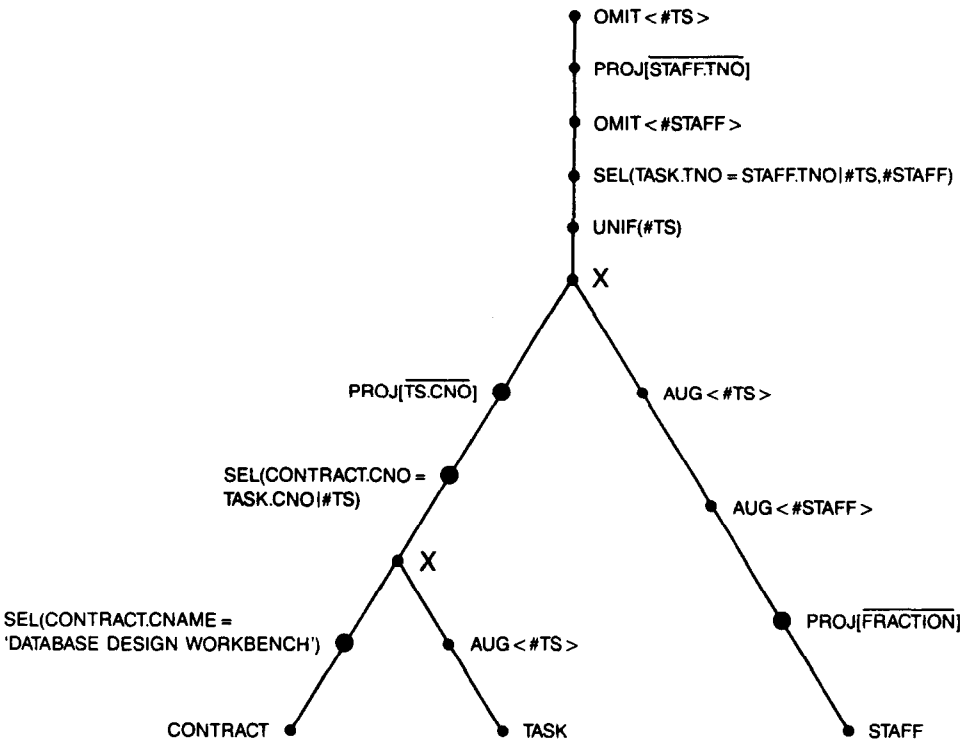

Figure 8. Query Q2, After Unflattening