Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers

Umeshwar Dayal

Computer Corporation of America
4 Cambridge Center
Cambridge, Massachusetts 02142-1489

Abstract

Existing query optimizers focus on Restrict-Project-Join queries. In practice, however, query languages such as SQL and DAPLEX have many powerful features (eg., control over duplicates, nested subqueries, grouping, aggregates, and quantifiers) that are not expressible as sequences of Restrict, Project, and Join operations. Existing optimizers are severely limited in their strategies for processing such queries; typically they use only tuple substitution, and process nested subquery blocks top down. Tuple substitution, however, is generally inefficient and especially so when the database is distributed. Hence, it is imperative to develop alternative strategies. This paper introduces new operations for these difficult features, and describes implementation methods for them. From the algebraic properties of these operations, new query processing tactics are derived. It is shown how these new tactics can be deployed to greatly increase the space of interesting strategies for optimization, without seriously altering the architecture of existing optimizers. The contribution of the paper is in demonstrating the feasibility and desirability of developing an integrated framework for optimizing all of SQL or other query languages that have similiar features.

1. Introduction

Most research on query optimization has focused on conjunctive queries, i.e. queries that can easily be translated into restrict-project-join expressions of the relational algebra [CODD70]. However, practical query languages, such as SQL [CHAM76, DATE85] and DAPLEX [SHIP81], [SMIT83] have many features (e.g. nested subquery blocks, control over duplicates, aggregation functions, grouping and quantifiers) that cannot be mapped to the restrict-project-join subset of the relational algebra. Such languages pose an important challenge for query optimization. The semantics of queries that use these features are often described procedurally, and existing query optimizers are severely limited in their tactics for processing such queries.

Consider, for example, the following relations:

EMP (Emp#, Name, Dept#, Sal)
DEPT (Dept#, Name, Loc, Mgr)

and the following SQL query, which contains a nested subquery block:

Query 1

```
SELECT   E.Name
FROM     EMP E
WHERE    E.Dept#  IN
    SELECT   D.Dept#
    FROM     Dept D
    WHERE    D.Loc = 'Denver' AND
             E.Emp# = D.Mgr
```

The semantics of SQL prescribe that the tuples of the EMP relation be substituted in turn into the inner subquery block: for each tuple E of EMP, the inner block is evaluated to yield a list of Dept# values; if E. Dept# is in this list, then E.Name is inserted into the result. The system R optimizer follows this prescription quite literally, optimizing only the execution of the inner block (after the substitution, the inner block contains two selections and the optimizer considers strategies for efficiently evaluating them) [SELI79].

In [KIM82], Kim showed that some nested SQL queries could be transformed into equivalent "canonical" queries that did not contain nesting; for example, query 1 could be transformed into query 2 (the queries are not quite equivalent, but more on this issue later):

Query 2

```
SELECT    E.Name                          ..(P)
FROM      EMP E, DEPT D
WHERE     E.Dept# = D.Dept# AND           ..(J1)
          D.Loc = 'Denver' AND            ..(R)
          E.Emp# = D.Mgr                  ..(J2)
```

Kim argued that query 2 was in a better form for optimization, because it allows the optimizer to consider more strategies. First, expressing the "nesting" predicate (... IN SELECT ...) between query blocks as an explicit join enables the optimizer to consider alternative methods (e.g. sort-merge) for implementing the join instead of always using tuple substitution. (Note that tuple substitution corresponds to the nested iteration method of join implementation [BLAS77].) This is especially important in distributed database systems, because -- as the experimental results of [MACK86a, MACK86b] show -- it is inefficient to do tuple substitution across a network. Second, it is easier to see from the form of query 2 that the query contains two joins between the EMP and DEPT relations; the optimizer may decide that it is less expensive to join the two relations on the E.Emp# = D.Mgr predicate first (this is probably the more restrictive predicate anyway), and to apply the other predicate as a restriction.

In general, expressing a query more non-procedurally or in algebraic form, gives the optimizer more leeway in selecting efficient strategies. The goal of this paper is to describe tactics for increasing the strategy space considered by an optimizer for queries that contain nested blocks, control over duplicates, aggregate functions, grouping, and quantifiers. We use DB2 SQL (as described in [DATE85]) to describe these tactics. However, the approach works just as well for other query languages (e.g. DAPLEX, QUEL) that enjoy some or all of these features.

Our approach is quite eclectic: it builds upon previous work on nested subqueries [KIM82, LOHM84, GANS87], aggregate queries [KLUG82], quantified queries [DAYA83], and outerjoins [ROSE84], but shows the feasibility and desirability of integrating these ideas in a common framework. [KIM82] and [GANS87] transform a nested SQL query into a collection of queries that are still expressed in SQL (or, in the case of [GANS87], a simple extension of SQL that includes outerjoins). However, existing SQL optimizers are not good at optimizing collections of queries. [CERI85] shows how to compile SQL queries into an extended relational algebra (which includes one very interesting operator corresponding to the GROUP BY and aggregation constructs of SQL). However, while this algebra may be useful for defining the approximate semantics of SQL, it is of limited applicability to the query optimization problem because it ignores the tricky semantics of duplicate control in SQL, and its transformation rules can produce algebraic expressions that are difficult to optimize.

In contrast, our approach is to compile SQL queries into a more powerful internal form based on an algebra of duplicate elimination, generalized join, generalized restriction, and generalized aggregation operations. We show that these generalized operations can be implemented by simple extensions to algorithms that are already implemented in existing query processors. Using a unified algebraic framework makes it possible to optimize entire queries, instead of optimizing subquery blocks piecemeal. Finally, we argue that the new operators can be accommodated without causing great violence to the tactics for strategy enumeration and cost estimation employed by existing optimizers.

The focus of this paper is on the operations and how their properties can be used to increase the strategy space for optimization; detailed heuristics for strategy enumeration and cost modelling are orthogonal issues. For concreteness, the reader may assume that strategies are enumerated by dynamic programming and that a cost model similar to that in [SELI79, LOHM84, MACK86a,b] is used. Also, a formal framework for proving the correctness of these tactics is beyond the scope of this paper.

In Section 2, we briefly review the conventional approach to processing simple conjunctive queries, which do not contain nested blocks, aggregates, or quantifiers. In Section 3, we describe our approach for queries that may contain nested subquery blocks but are free of aggregates and quantifiers. Aggregates are considered in Section 4, and quantifiers in Section 5. Section 6 describes extensions to the basic technique to deal with other syntactic features of SQL such as nesting predicates other than IN, and disjunctions and unions.

2. Simple Non-Nested Conjunctive Queries

For queries that consist of a single block (i.e., with no nested subquery blocks), processing is straightforward. For simplicity, assume conjunctive queries (i.e., queries in which AND is the only logical connective); we relax this assumption in Section 6. Such queries can be processed completely by sequences of Restriction, Projection, and Join operations. Our syntax for these operations is: Restrict(relation; predicate), Project(relation; attribute-list), and Join(relation1, relation2; join-predicate). For example, Query 2 of the last section can be evaluated by Restricting DEPT on R, Joining the result with EMP on J1 AND J2, and then Projecting on P.

SQL semantics require two types of Project operators, one that preserves duplicates (Project) and another that eliminates them (Delta-Project). (Note that Delta-Project is the projection operator of the "standard" relational algebra [CODD70].) Query 2 requires the use of Project; however, if the SELECT statement were changed to SELECT DISTINCT E.Name, then we would have to change the last step to Delta-Project on E.Name.

Note that Project is easily implemented in one scan of the relation. However, Delta-Project requires that the relation be grouped or sorted by the attributes over which we are projecting, so that duplicates can easily be detected; alternatively, an index or hash function on the projection attributes may be used.

The properties of Restrict, Join, and Delta-Project are well known. The properties of Project are similar to those of Delta-Project, except that Project and Delta-Project do not commute (see [DAYA82] for details). These properties are used to develop tactics for query optimization. Because joins commute and associate, any permutation of the joins in a query is feasible; restrictions and projections can be positioned anywhere relative to the joins (except, of course, that attributes needed for later processing cannot be projected out).

Thus, optimization of single-block queries consists of choosing the best permutation of the joins, positioning restrictions and projections relative to joins, choosing the best implementation method for each join, and choosing access paths (e.g., indices, hash keys) [SELI79].

To allow the optimizer maximum flexibility in selecting join permutations, we use a canonical internal representation of the query, called a *query graph* [BERN81]. There is one node in

the query graph for each relation variable (synonym) appearing in the query; each node is labelled with any restriction conditions involving that variable, and with any attributes of that variable that appear in the SELECT list of the query. There is one edge between nodes R and S in the query graph for each *simple join predicate* of the type (R.A op S.B) in the query. For *complex predicates*, e.g., (R.A = S.B + T.C), we introduce special "restriction nodes" labelled with the complex condition, and "precedence edges" directed from the variables involved in the condition to the restriction node. (The reason for treating these conditions separately is that there are unlikely to be any special access paths for efficiently evaluating them, so they will have to be evaluated as restrictions during or after a join.) Completely disconnected nodes should be connected to every other node by join edges labelled "True"; these correspond to Cartesian Product operations. Figure 2.1 shows the query graph for Query 2.

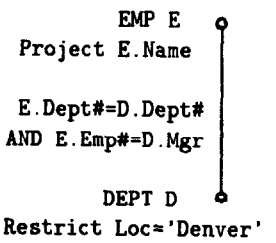

Figure 2.1 Query Graph for Query 2

The optimizer can select any permutation of the join edges that form a spanning tree of the query graph. The remaining joins and complex restrictions are all evaluated as restrictions. In practice, restrictions and projections are evaluated as early as possible. The following property is used to combine a restriction with the last join that brings together all the attributes required to evaluate the restriction: Restrict(Join(R,S;J); P) = Join(R,S;J AND P).

The output of the optimizer is an *execution plan*, which may be represented abstractly by a directed acyclic graph whose leaves are relations stored in the database, whose internal nodes are operators of the algebra, and whose edges prescribe an order of execution.

3. Nested Queries without Aggregates or Quantifiers

To extend the concepts of Section 2 to nested queries without aggregates or quantifiers we add a semijoin operator to our algebra. The *semijoin of relation R by relation S on condition J* is defined as the subset of R-tuples for which there are matching S-tuples that satisfy J, i.e., Semijoin (R, S; J) = { r ϵ R | $\exists$ s ϵ S (J (r,s))}. Note that the semijoin operation is not symmetric.

Semijoins were introduced as a tactic for distributed query processing in [BERN 81]. It was shown in [CERI 85] that semijoins are essential to correctly interpreting nested queries in SQL. In fact, the only difference between queries 2 and 1 of Section 1 is that where the former requires a join between DEPT and EMP, the latter requires a semijoin: if an EMP tuple E joins with more than one DEPT tuple, that E.Name will be repeated in query 2's result as many times as there are matching DEPT tuples, but will show up only once in query 1's result.

Semijoins can be implemented by only slightly modifying join implementation methods. First, remark that although the join operation is defined symmetrically, all join implementation methods are asymmetric: for each tuple of one relation (called the "outer") perhaps ordered in some sequence, an access path is used to find all matching tuples of the other relation (called the "inner") also perhaps ordered in some sequence; these matching tuples of the inner relation are concatenated to the tuple of the outer relation to produce tuples of the result. If no matching tuples of the inner relation are found, then nothing is output.

To produce the semijoin of the outer relation by the inner relation, we merely modify any join method as follows: output the tuple of the outer relation as soon as the *first* matching tuple of the inner relation is encountered, and then advance to the next tuple of the outer relation. When the two relations are stored at different sites in a distributed system, it may be cheaper to delta-project the inner relation on its join attributes, and to move this projection to the site of the outer relation, instead of moving the entire inner relation. (The trade-off is in the extra cost of the delta-project versus the reduction in the amount of data moved. Other implementations are also possible.) The cost of a semijoin is easily estimated by modifying the cost of the join.

The rules for constructing the query graph of a nested query are simple extensions of those for a simple conjunctive query. Relation nodes and join edges are introduced for each relation and join condition occurring inside the nested block; in addition, for each *nesting predicate* of the type R.A IN (SELECT S.B ...), a "semijoin edge" is introduced between the nodes for R and S, and is labelled semijoin(R, S; R.A = S.B); finally, for each *correlation predicate* of the type (R.C op S.D), where R and S are in different blocks, a semijoin edge is introduced. Figure 3.1 shows the query graph for the following query:

Query 3

```
SELECT   E.Name
FROM     EMP E
WHERE    E.Dept# IN                          ..(J1)
    SELECT   D.Dept#
    FROM     DEPT D
    WHERE    D.Name = 'R&D'
             AND D.Mgr IN                    ..(J2)
         SELECT   E2.Emp#
         FROM     EMP E2
         WHERE    E.Sal > E2.Sal              ..(J3)
```

The optimizer can consider semijoin edges together with join edges in choosing permutations. Sometimes, for cyclic queries, a semijoin has to be turned into a join (if attributes required for later processing must be retained). For Query 3, for example, if the permutation [J1, J2] is used, J1 must be replaced by a join, because its Mgr attribute is needed for the second join; however, J2 and J3 can be performed together as a semijoin. To produce

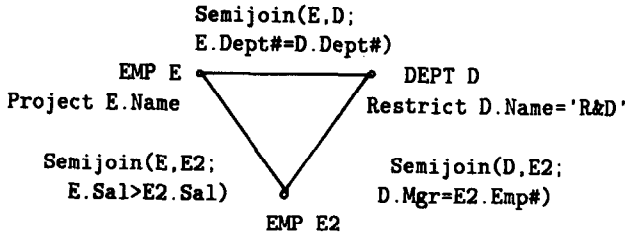

Figure 3.1 Query Graph for Query 3

the correct result, a Delta-Project on E.* is necessary before the final Project.

Before we give the formal rule to describe this transformation, we must introduce some notation. We assume that each relation R has a TID ("tuple identifier") attribute that serves as a primary key. (A user-supplied primary key attribute may be substituted for TID in the following discussion.) We also assume in practice that the operands of a join are duplicate-free. Relations stored in the database can be made duplicate-free by concatenating the TID attributes. Then the result of the join is also duplicate-free; the TID of the result may be obtained by concatenating the TIDs of the operands. Whenever a Project operation is used to reduce the width of an operand, the TID is included, so that the operand is still duplicate-free. (Only the final Project may introduce duplicates.) We write R.* to mean all attributes of R. The rule for converting a semijoin to a join then is:

Semijoin (R, S; J) =
Delta-Project (Join (R, S; J) ; R.*)

Because Delta-Project commutes with Join and Restrict, but not with Project, the optimizer can delay the Delta-Project operation, but not beyond the final Project on the attributes in the SELECT clause. In particular, it may be advantageous to delay the Delta-Project if at some subsequent stage, the intermediate result is going to be sorted or grouped on the projection attributes (this will reduce the cost of duplicate elimination).

Even though Delta-Project and Project cannot be commuted, the following rule still allows us to reduce the size (width) of intermediate results by projection:

Project(Delta-Project(R;X2); X1) =
Project(Delta-Project(Project(R;X3); X2); X1),

where X1 is contained in X2, which is contained in X3. Thus, we can project out attributes that are not required for future Delta-Projects or the final projection.

Another useful transformation rule allows us to absorb Delta-Project into a subsequent Semijoin:

Semijoin(R, Semijoin(S, T; J2) ; J1) =
Semijoin(R, Join(S, T ; J2) ; J1)

Using this rule in the permutation [J2, J1] for Query 3, allows us to solve J2 as a join, and then J1 and J3 together as a semijoin, without the need for a Delta-Project on D.* (the semijoin on J1 AND J3 eliminates duplicates anyway, and hence produces the correct result).

4. Queries with Aggregates

To describe tactics for processing aggregates, we introduce in Section 4.1 three new operators: generalized aggregation, generalized join, and generalized restriction. We define each of these operators in turn, describe implementation methods, and illustrate its use in processing SQL queries. Then, in Section 4.2, we describe tactics that use these operators for processing queries that contain aggregates.

4.1 Operators for Queries with Aggregates

Generalized-Aggregation

The syntax of this operation is: G-Agg(R; X; fnvector), where R is a relation, X is a list of attributes of R (called the *grouping attributes*), and fnvector is a vector of aggregation functions applied to attributes of R. An *aggregation function* is a function that takes a set of tuples and computes a single value. The built-in aggregation functions in SQL are COUNT, SUM, MAX, MIN, AVERAGE and their DISTINCT variants.

The meaning of the G-Agg operation is as follows: partition R is such a way that each block (group) of tuples has the same X-value; then evaluate each aggregation function in fnvector over the tuples in a group. The result of this operation is a relation whose attributes are the grouping attributes X together with as many new attributes (named for the respective aggregation functions) as there are aggregation functions in fnvector. (This is the FN operation introduced in [CERI85].) By convention, if the list of grouping attributes, X, is empty, then all the tuples of R are in a single group. Note that the grouping attributes constitute a primary key of the result.

G-Agg is implemented by modifying the implementation of Delta-Project as follows: Use any access paths (sorting, hashing, indexing, clustering) on the grouping attributes to scan the tuples in one group of the partition; accumulate the values of the aggregate functions; at the end of the group, output the grouping attribute values concatenated with the aggregated values; then move on to next group. Accumulating values is easy: for example, for COUNT, start with the initial value 0, and then increment the accumulator as each new tuple is scanned; for MAX, start with the initial value NULL, and then for each new tuple, replace the accumulator by the max of current value and the new tuple value (where MAX(NULL,a) = a, for a $\neq$ NULL); for AVERAGE, accumulate both COUNT and SUM. Accumulating the DISTINCT variants is a little more complicated. The solution is to use an additional access path (eg. minor sort, second hash function) that further subgroups the tuples in each group by the attributes to be aggregated. This will simplify the detection of duplicates in the group; only the first value in each subgroup is accumulated, and the scan then skips to the next subgroup. (Note that SQL disallows the occurrence of more than one DISTINCT variant in a SELECT clause. If this condition were relaxed, the processing of Delta-Project and G-Agg would become more complicated,

because additional passes would be needed.)

To see how G-Agg is used in processing SQL queries, consider the following query.

Query 4

```
SELECT E.Dept#, AVERAGE(E.Sal),
       COUNT DISTINCT(E.Name)
FROM   EMP E
GROUP BYE.Dept#
```

This query is equivalent to G-Agg(EMP; Dept#; AVERAGE(Sal), COUNT DISTINCT(Name))

The next example shows how G-Agg can be pipelined with the evaluation of Join.

Query 5

```
SELECT    D.Dept#, AVERAGE(E.Sal)
FROM      EMP E, DEPT D
WHERE     E.Dept# = D.Dept#
GROUP BY  D.Dept#
```

The query consists of a Join followed by a G-Agg. An efficient execution plan can be obtained by making the following important observation: all the join evaluation methods implemented by existing systems have the property that the result is grouped by tuples of the outer relation (or, equivalently, by the TID or other key of the outer relation). Hence, if DEPT is chosen as the outer relation, and if Dept# is its key, then the result of the join will already be grouped by D.Dept#, and so can be pipelined into the G-Agg computation.

In general, pipelining will not be possible if the query requires grouping by non-key attributes. For example, if the grouping attribute was D.Loc, then the result of the join will not be grouped correctly for the G-Agg computation (unless DEPT was already grouped by Loc). That G-Agg could be implemented by accumulating aggregate values during a scan, and could be pipelined with Join, was described in [KLUG 82]. However, the necessary condition for pipelining, viz. that the grouping attributes form a key of the outer relation, is new.

A similar strategy works for nested queries in which aggregates appear in the WHERE clause. Consider, for example, the following query:

Query 6

```
SELECT   D.Name
FROM     DEPT D
WHERE    D.Budget  < =
    SELECT   1000 * AVERAGE(E.Sal)
    FROM     EMP E
    WHERE    E.Dept# = D.Dept#
```

This query can be solved by pipelining the following sequence of operations: the join of E and D on predicate E.Dept# = D.Dept#; followed by the computation of the AVERAGE (grouped by D.TID, D.Budget); followed by the restriction D.Budget <= 1000 * AVERAGE; followed by the projection on D.Name. (Note that we had to include D.Budget in the grouping attributes of the G-Agg operation, because we need D.Budget in order to do the restriction; this does not affect the computation of G-Agg, since D.Budget will be unique for each D.TID anyway.)

An alternative plan for solving this query is to pipeline the following sequence of operations: compute G-Agg(E; E.Dept#; AVERAGE(E.Sal)); then join with D on the combined condition D.Budget <= 1000 * AVERAGE AND D.Dept# = E.Dept#; followed by the final projection on D.Name. This strategy is an adaptation of Kim's rule [KIM82], but unlike the latter does not produce two separate queries; including all operations in a single plan makes it possible to optimize the whole query.

There is a tradeoff between these two plans. The first may compute the aggregate many times for the same D (if Dept# were not the key); the second requires that E be grouped or sorted by Dept#. Both alternatives must be considered by the optimizer.

Generalized-Join and Generalized-Restrict

The motivation for these operations proceeds in two steps. First, we show that outerjoins are necessary for certain queries that contain aggregates. Next, we show that regular joins (hitherto simply called joins) and outerjoins do not ordinarily commute, but treating them as special cases of a generalized join results in a workable sort of commutativity.

Consider Query 6 but with AVERAGE(E.Sal) replaced by COUNT(E.Emp#). We would like to be able to solve this query by plans similiar to the two described above for query 6. However, we would not quite get the correct answer: the join would delete departments that have no employees, so the names of these departments would not appear in the result; evaluating the query using strict SQL semantics, however, would return 0 for the COUNT of the employees in such departments, and of these, the ones with budgets <= 0 should appear in the result. This is precisely the "count bug" in the rules of [KIM82] that is described in [GANS87]. There is no clean way of making the second plan (based on Kim's rule) work; the first plan, however, can be modified to use an outerjoin instead of the regular join.

The *outerjoin of R by S on predicate J* is the union of the regular join and tuples formed by padding out the unjoined tuples of R with NULL values. Formally, Outerjoin(R,S;J) = Union(Join(R,S;J), Product(Antijoin(R,S;J), NULL-S)), where NULL-S is an all-null tuple of the same degree as S, and Antijoin(R,S;J) is the complement in R of Semijoin(R,S;J). Note that this operation is asymmetric [CODD79, LACR78, ROSE84].

To implement Outerjoin(R,S;J), modify any method for Join(R,S;J) with R as the outer relation as follows: for any tuple of R for which the set of matching S-tuples is empty, output the tuple of R padded with NULL values.

If we replace the join of DEPT and EMP in our example by the outerjoin, then we will retain all DEPT tuples, and the subsequent COUNT and restriction will come out correct. Any restrictions on D (e.g., D.Loc = 'Boston') that occur inside the

nested block are treated as "outerrestrictions" and are combined with the join predicate of the outerjoin. Note that this problem arises only when the aggregate function is COUNT. The other aggregate functions return NULL when evaluated over the empty set, and since NULLs cannot satisfy any conditions in the WHERE clause, these DEPT tuples cannot possibly contribute to the query's result. (In section 5 we shall see that Outerjoins also arise when the NOT EXISTS quantifier is used.)

While [GANS87] correctly underscored the importance of outerjoins, its transformation rules are of limited utility for several reasons. First, its rules use symmetric outerjoins, which are more difficult to implement than our asymmetric outerjoin. Second, its rules would transform the above query into two queries, one containing the outerjoin, group by, and aggregation, and the other containing the <= restriction and final projection. Unless the optimizer can simultaneously optimize a collection of queries, it will not discover that the join, aggregation, restriction and projection can all be pipelined. Third, a different rule is needed for queries where the join condition is different from equality. Finally, [GANS87] does not describe tactics that mix joins and outerjoins, as we do.

A complex query may be transformed into an expression that contains both regular joins and outerjoins. In [DAYA83], we described strategies for such queries. We showed that it is always correct to process all regular joins before any outerjoin. In general, it is not possible to directly commute a regular join and an outerjoin (see Fig. 4.1), i.e., Outerjoin(R, Join(S,T;J2); J1) $\neq$ Join(Outerjoin(R,S;J1), T; J2). However, this restricts the space of permutations that can be considered by the optimizer. Sometimes it may be better to process an outerjoin before a regular join (e.g., in Figure 4.1, the outerjoin J1 will knock out most of relation S, thus making the later regular join J2 cheaper.) To allow the optimizer to consider such permutations, we introduced in [DAYA83] a graft operator, which operated on trees and generalized both regular join and outerjoin. We did not, however, describe how to implement grafts. In this paper, we introduce, instead, a generalized-join operator that can be implemented easily by join-like methods on relations.

Let X be a subset of R's attributes and let Y be R.* - X. Then the *generalized-join of R by S on predicate J preserving attributes X* is written G-Join(R,S;X;J) and is equal to the union of the regular join of R and S, together with tuples formed by delta-projecting the unjoined tuples of R on X and then padding them out with NULL values. Formally, G-Join(R,S;X;J) = Union(Join(R,S;J), Product(Delta-Project(Antijoin(R,S;J); X), NULL-YS)), where NULL-YS is an all-null tuple over the union of attributes in Y and S. By convention, if R is duplicate-free, G-Join(R,S;$\emptyset$;J) is the regular join, and G-Join(R,S; R.*; J) is the outerjoin. Like the outerjoin, this operation is asymmetric. A symmetric version is easy to define, but unnecessary for our purposes.

Now, the reader can convince himself that, for the query in Figure 4.1, G-Join(R, G-Join(S,T;$\emptyset$;J2); R.*; J1), which corresponds to doing the regular join J2 before the outerjoin J1, is equivalent to G-Join(G-Join(R,S; R.*; J1), T; R.*; J2), which corresponds to doing the outerjoin J1 first and then a join on condition J2 that is neither an outerjoin nor a regular join. We will discuss later the rule for determining which attributes to preserve when we move a regular join past an outerjoin.

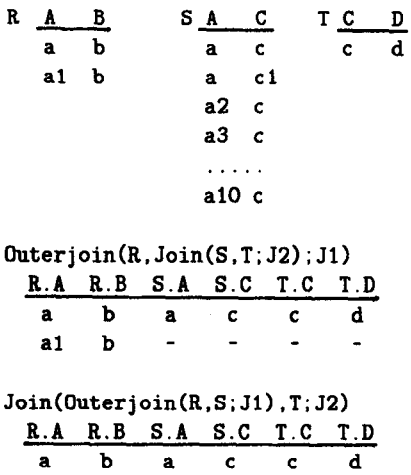

Figure 4.1 Joins and Outerjoins Do Not Commute

G-Join is implemented by a straightforward modification to any method for implementing Outerjoin. Note that the Delta-Project step requires grouping the outer relation R by the preserved attribute set X.

We also need a *Generalized-Restrict* operator, which is analogous to the generalized-join and generalizes restrict and outerrestrict. G-Restrict(R; X; P), where X is a subset of R.* and P is a predicate, is the union of Restrict(R; P) together with the remaining tuples of R delta-projected on X and then padded out with NULLS.

4.2 Tactics for Queries that Contain Aggregates

We describe in four steps the tactics for processing queries that contain nested subquery blocks and aggregates. First, we show how to construct query graphs for these queries. Second, we describe which enumerations of G-Joins are legal for the optimizer to consider. Third, we show how to position G-Agg operations in a sequence of G-Joins (and G-Restricts). Fourth, we describe a generally useful tactic, pipelining.

Query Graphs

The query graphs of aggregate queries must represent outerjoin, outerrestrict, and G-Agg operations. An outerjoin is represented by an edge directed from the outer relation to the inner. Outerrestrict and G-Agg operations are represented by special nodes that are connected to other nodes by precedence edges (as we did with restriction nodes in section 2). In addition, for outerrestricts, we must indicate the outer relations (i.e., the preserved set).

If the SELECT or HAVING clause of a block contains an aggregate function, we call it an *aggregate block*; if it contains a COUNT function, we also call it a *COUNT block*. The *scope* of a block is the block itself and all the blocks nested within it (to

any level). The *immediate scope* of an aggregate block is the set of blocks in its scope but not in the scope of any other aggregate block in its scope.

We define a predicate that is in the immediate scope of a COUNT block B to be an *outer predicate* if it involves any variables that are defined outside the scope of B. Such variables are said to be *outer variables* with respect to B. All other predicates are said to be *regular* and all other variables defined in B's scope are said to be *inner* with respect to B.

The query graph of an aggregate query contains one node for each variable in the query. For regular predicates, the rules described in sections 2 and 3 apply: i.e., we introduce regular join edges for simple join predicates involving variables defined in the same block; restriction nodes and precedence edges for complex predicates; restriction labels for one-variable restriction predicates; and semijoin edges for simple nesting predicates and simple correlation predicates. Nesting predicates of the form R.A op (SELECT AGG(S.B)...) and HAVING AGG1 (R.A) op (SELECT AGG2(S.B)...) are treated as complex predicates; also unless R and S are already connected by a semijoin edge, introduce an edge labelled Semijoin (R,S; TRUE).

For each simple outer predicate that involves an outer variable and an inner variable, introduce an outerjoin edge directed from the outer to the inner variable. For all other outer predicates, introduce outerrestriction nodes and precedence edges from all the variables involved; label "outer" the precedence edges from the outer variables.

Finally, for each aggregate block B, introduce an aggregation node labelled with a G-Agg operation as follows: the grouping attributes include the outermost GROUP BY attributes, if any, the TIDs of all variables outside B's scope that occur in correlation predicates in B's immediate scope, and any other attributes that may be needed for future restrictions or projections; the fnvector consists of all the aggregate functions in the SELECT and HAVING clauses. Introduce precedence edges from all nodes corresponding to variables whose attributes appear in the G-Agg label and from all nodes corresponding to variables and predicates in the immediate scope. Introduce precedence edges from the aggregation node to any restriction node in which the result of the G-Agg operation is used.

Figure 4.2 shows the query graph for the following query. Note the outerjoin edge and outerrestriction node corresponding to the predicates OJ and OR, respectively.

Query 7

```
SELECT      D.Name, SUM(E.Sal)
FROM        DEPT D, EMP E
WHERE       D.Dept# = E.Dept#
GROUP BY    D.Dept#, D.Loc
HAVING      AVERAGE(E.Sal) >=
    SELECT    1000 * COUNT(E2.Emp#)
    FROM      EMP E2
    WHERE     E2.Dept# IN
        SELECT    D2.Dept#
        FROM      DEPT D2
        WHERE     D2.Loc = D.Loc AND        ..(OJ)
                  D.Name ≠ Sales            ..(OR)
```

Note that G-Joins and G-Restricts do not appear on query graphs. They appear only in execution plans derived as a result of permuting joins and outerjoins as discussed below.

Legal Join Permutations

As usual, we have to pick permutations of the join edges of the query graph that form a spanning tree; the other joins are performed as restrictions. First, consider the (skeletal) query graph of Figure 4.4. Because outerjoins (and G-Joins) are asym-

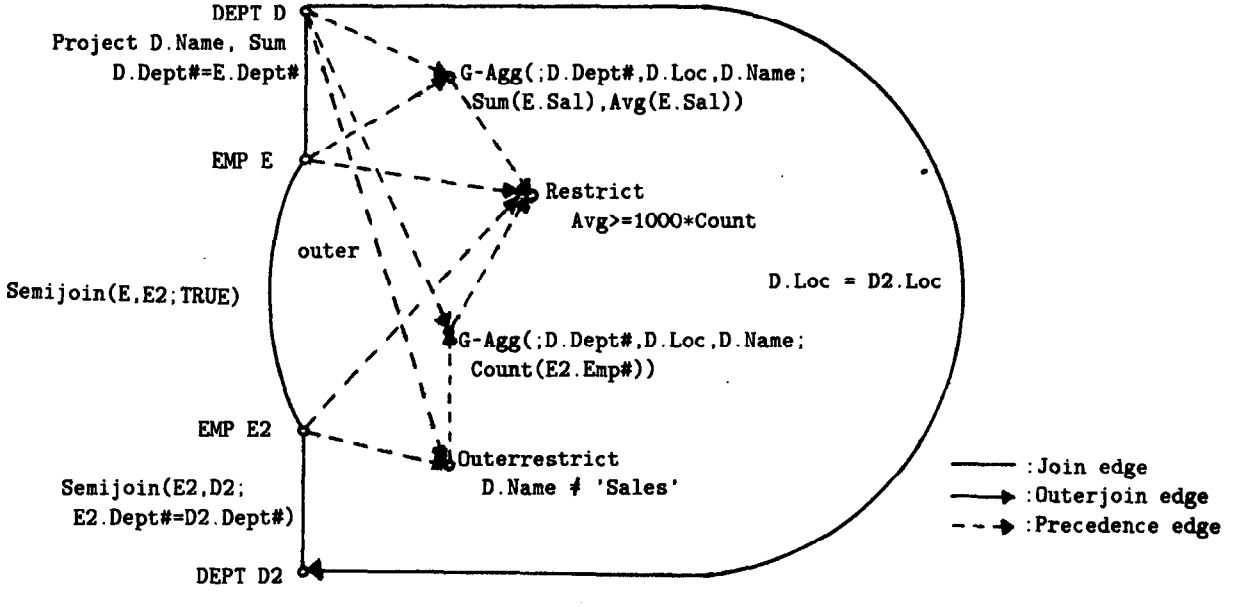

Figure 4.2 Query Graph of Query 7

metric operations, if we pick the spanning tree {J1,J3,J4}, then after performing any two of these joins, we will not be able to perform the third as a G-Join (nor J2 as a G-Restrict). However, selecting {J1,J2,J3} as the spanning tree will lead to correct permutations.

The algorithm for deciding whether a spanning tree will yield correct join permutations is as follows. Assign a level number to each node in the query graph by setting the level number of the variables and restriction nodes in the outermost block to be 0; let the length of each outerjoin edge and each precedence edge labelled "outer" be 1; and the length of all other edges be 0; the level number of a node is the length of the longest path into it. Then, the spanning tree should not contain any edge (X,Y) where (level number of Y - level number of X) > 1. (We call such edges *straddling edges*).

Next, we consider the question of which permutations of a spanning tree's edges are legal. Any permutation in which all regular joins precede outerjoins is legal. However, to increase the space of strategies considered by the optimizer, we want to shuffle regular and outerjoins. Essentially, this means that we treat all joins as G-Joins; then any permutation of a correct spanning tree will be legal.

Given a permutation, we must determine what the preserved attributes for each G-Join operation should be. For an edge J = (R,S) in the permutation, let X be the preserved attribute set for the G-Join corresponding to J. X is derived as follows. Suppose J has no predecessor of the form J' = (T,R) in the permutation. Then, X = $\emptyset$ if J is a regular join or a semijoin edge, and X = R.* if J is an outerjoin edge. Suppose, on the other hand, that J does have such a predecessor and that J' = (T,R) is the nearest such predecessor. Let X' be the preserved set of the G-Join corresponding to J'. Then X = X' if J is a regular join or semijoin edge, and X = X' U R.* if J is an outerjoin edge. For G-Restricts, the preserved sets can be computed by an analogous rule. In practice, G-Restricts are combined with G-Joins as early as possible.

For lack of space, we cannot justify these algorithms in detail. However, the interested reader is encouraged to try the algorithms out on the query graph of Figure 4.3 and to understand for himself the reasoning behind them.

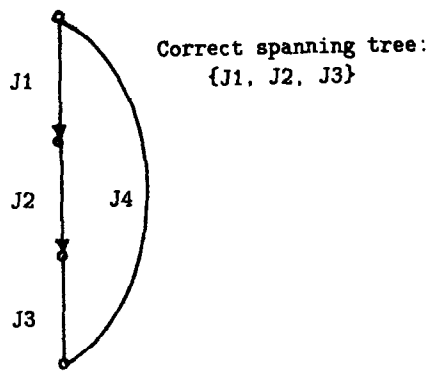

Figure 4.3 Legal Join Permutations

An optimizer such as that described in [SELI79, LOHM86a,b], which enumerates permutations and evaluates them by dynamic programming, must check which join or restriction predicates can be applied next. It should be easy to extend the optimizer to check also for the condition that no straddling edge is added, and to compute the preserved set for the G-Join corresponding to each new edge that is added.

Positioning G-Agg operations

The optimizer also has some flexibility in positioning G-Agg operations relative to G-Joins, G-Restricts, Projects and Delta-Projects. Three rules delimit this flexibility:

- In general, a G-Agg operation cannot be executed unless all predicates in its scope have been evaluated. This is guaranteed by the precedence edges introduced in the construction of the query graph.
- To delay a G-Agg past a G-Join, add to the grouping attributes the TID (or other key) of the relation being joined to. This rule follows from the property:
  G-Join(G-Agg(R;X;F), S; Y; J) = G-Agg(G-Join(R,S;Y;J), (X,S.*); F).
- In some special cases, Kim's rule applies and a G-Agg can actually be moved ahead of a G-Join. If the fnvector of the G-Agg operation involves attributes of only one relation and does not contain a COUNT, and if the join predicate of the G-Join contains only equalities, then the following rule applies:
  G-Agg(G-Join(R, S; X; R.A1 = S.B1 AND ... AND R.An = S.Bn); R.*; Agg(S.Y)) =
  Project(G-Join(R, G-Agg(S; S.B1,...,S.Bn; Agg(S.Y)); X; R.A1 = B1 AND ... AND R.An = Bn); (R.*, Agg))

Usually, it is a good idea to execute a G-Agg operation as soon as possible. For instance, applying this dictum to the following "uncorrelated" nested query means that the nested block is evaluated once, and the nesting condition is replaced by a restriction (this follows as a special case of the third rule above):

Query 8

```
SELECT   E.Emp#
FROM     EMP E
WHERE    E.Sal <
         SELECT AVERAGE(E2.Sal)
         FROM   EMP E2
```

However, sometimes it may be beneficial to delay a G-Agg operation, because subsequent operations may produce the correct grouping needed for the G-Agg, or because later joins inside a HAVING clause may reduce the number of groups to be aggregated. To see this, consider the following query, for which we have added a third relation FLEET(Location, #Cars) to our data base.

Query 9

```
SELECT     D.Loc, SUM(D.Budget)
FROM       DEPT D
GROUP BY   D.Loc
```

```
HAVING    COUNT(D.Mgr) =
    SELECT    F.#Cars
    FROM      FLEET F
    WHERE     F.Loc = D.Loc                ..(J)
```

Since the join condition J is likely to be quite restrictive (not many locations have fleets), it may be better to do this join first, before aggregating.

Moving G-Agg past a join requires same care, however, because the join may increase the number of tuples in each group. Our second rule above avoids this problem. If it is known a priori that each R-tuple will join with at most one S-tuple, then augmenting the grouping attributes is unnecessary. This is true of our previous example.

Pipelining

To reduce the cost of creating and storing temporary intermediate results, the query processor pipelines the results of one operation into the next whenever possible. We have seen several examples of this tactic already. The result of a G-Join can be pipelined into a G-Agg whose grouping attributes consist of a key of G-Join's outer relation. The result of a G-Join or a G-Agg can be pipelined into a Delta-Project (again if the grouping requirements are met). The result of any operation can be pipelined into a Project or a G-Restrict.

To effectively exploit pipelining, the optimizer must keep track of how intermediate results are grouped. The SQL optimizers of [SELI79, LOHM84] already keep track of interesting sort orders; extending them to record interesting groupings is straightforward.

5. Nested Queries with Quantifiers

In this section, we consider queries that contain the EXISTS and NOT EXISTS quantifiers. Queries that contain only the existential quantifier (EXISTS) require no new techniques: they can easily be transformed into expressions involving Restrict, Project, Join, Semijoin, and Delta-Project. For example, the following query is equivalent to Query 1 (of section 1):

```
Query 10
    SELECT    E.Name
    FROM      EMP E
    WHERE     EXISTS
              (SELECT *
              FROM        DEPT D
              WHERE       D.Dept# = E.Dept# AND
                          D.Loc = 'Denver' AND
                          E.Emp# = D.Mgr)
```

A little care is necessary when transforming queries where the nested SELECT clause lists a proper subset of attributes, instead of *. For example, if the nested SELECT clause in the above query had read SELECT D.Budget, then we would first have to transform the query by conjoining the additional restriction condition (D.Budget IS NOT NULL) to the WHERE clause of the inner block. We will assume that queries containing EXISTS and NOT EXISTS have already been transformed like this.

For queries with mixed EXISTS and NOT EXISTS quantifiers a different class of strategies is required. Again, we want to consider more options than the straightforward tuple substitution approach of [SELI79, LOHM84]. Most previous work transforms quantified queries into expressions involving set differences, Cartesian products, and division operators [CODD72, PALE72, CERI85], which are quite inefficient to evaluate. [JARK82] describes transformations based on logical identities (e.g., distributing quantifiers over disjunctions, switching the order of quantifiers), but does not relate these to detailed strategies for query optimization. [KIM82] describes one rule for efficiently processing a quantified query using division, but the rule does not generalize to arbitrary queries containing mixed quantifiers.

In [DAYA83], we introduced an algebra of graft and prune operations on trees that yields efficient strategies for quantified queries. Our approach now is to replace the graft and prune operators on trees by the G-Join and G-Agg operations on relations. Then, we can use the implementation methods and tactics for optimization that we have described in sections 2 - 4. Also, by using the same algebraic operators as for other nested queries, we can cast the optimization of all SQL queries into a common framework.

First, we introduce two Boolean-valued aggregation functions EXISTS and NOTEXISTS. EXISTS(R) iff $R \neq \emptyset$. NOTEXISTS(R) = NOT(EXISTS(R)) These aggregation functions can be used in G-Agg exactly like any other aggregation function.

EXISTS is accumulated by initialising to FALSE, and then replacing the accumulator with TRUE when the first tuple is encountered; of course, for each group, the accumulation can cease when the accumulator is first set to TRUE. (Not surprisingly, this is very much like the computation of semijoin.) NOT EXISTS is computed by accumulating EXISTS and then negating the accumulator at the end of each group.

To illustrate the use of these aggregation functions, consider Query 10 with EXISTS changed to NOT EXISTS). This query can be solved by replacing the Semijoin we used for Query 10 with an Antijoin. Alternatively, we can first perform the Outerjoin of EMP by DEPT on the conditions in the WHERE clause, then G-Agg(; E.TID, E.Name; NOT EXISTS), then Restrict where NOT EXISTS = True, and finally Project on E.Name. Of course, since the join and G-Agg require the same grouping, we can pipeline these operations. Note that while the EXISTS aggregate requires a regular join, the NOTEXISTS aggregate requires an outerjoin. Hence, for queries with mixed quantifiers, we will use G-Joins as in Section 4. The G-Join step corresponds to the graft operation, and the G-Agg, G-Restrict sequence to the prune operation of [DAYA83].

While it is clearly preferable to use Semijoin or Antijoin operations rather than the G-Join, G-Agg etc. sequence, it is not always possible to do so when they are nested and the query graph is cyclic.

The construction of the query graph uses the same rules as in Section 4.2, with NOT EXISTS behaving like COUNT.

The optimizer enumerates join edges and positions G-Agg operations using the various tactics described in Section 4.

Additionally, we have the tactic of using semijoins and antijoins wherever possible. In an execution plan in which the last G-Join in the scope of a quantifier is immediately followed by its corresponding G-Agg, this sequence of operations can be replaced by a Semijoin or an Antijoin, respectively, depending upon whether the quantifier is an EXISTS or NOT EXISTS. Formally stated, the rule is:
Project(G-Agg(G-Join(R,S;∅;J); R.TID; EXISTS); R.*)
= Semijoin(R, S; J)

An analogous rule is obtained for NOT EXISTS by substituting R.* for ∅ as the preserved set in the G-Join on the left hand side, and substituting Antijoin for Semijoin on the right hand side.

Finally, the tactics described in this section will work for other syntactic constructs of SQL such as ANY and IS NOT IN; we assume that these constructs are first transformed into EXISTS and NOT EXISTS as described in [DATE85].

6. Extensions

6.1 Nesting Predicates other than IN

Consider query 1 with the nesting predicate IN replaced by =, thus: ... E.Dept# = (SELECT D.Dept# ...). SQL semantics dictate that if the nested subquery returns more than one tuple, then an exception should be raised and the query should be aborted. To capture these semantics in our algebra, we need an additional operator: Test-FD. Let X, Y subsets of R.*, and let F:X–>Y be a functional dependency statement. Then, Test-FD(R;F) = R if F holds in R; otherwise, raise an exception and abort.

Test-FD can be implemented by modifying the method for implementing G-Agg(R; X; COUNT DISTINCT(Y)) to abort whenever more than one distinct value of Y is encountered for a group.

The Test-FD operation can be delayed past G-Join and Project operations, but not past Restrict, Delta-Project, or G-Agg. Hence, the above query can be evaluated by pipelining the result of a regular join between EMP and DEPT into the Test-FD, and then into Delta-Project to get rid of duplicates for the semijoin.

On query graphs, the nesting predicate (R.A op (SELECT S.B ...)) is represented by a semijoin edge, labelled (R.A op S.B), between R and S, together with a Test-FD node labelled with the FD R.TID –> S.B, and precedence edges to this node from R and from the variables and restrictions in the scope of S's block.

6.2 Disjunctions and Unions

Queries that contain disjunctions and unions are usually expensive to process, because their results are quite large, and database systems do not have access pathes to support their evaluation.

Instead of describing in detail the construction of query graphs and execution plans for such queries, we highlight the salient features of our approach. Disjunctions that appear in one-variable restriction conditions (e.g., R.A=a or R.B=b) are processed during one scan of relation R. Disjunctions that appear in join conditions (e.g., R.A=S.B or R.C=S.D) can be processed in several ways: (a) as a restriction when R and S are joined on some other condition; (b) as two joins followed by a union (this is useful only if both joins are supported by fast access paths); or (c) more intelligently, by a join on one condition followed by a join of the unjoined portions on the other condition. (For (c), we would need a symmetric operation that simultaneously produces the regular join, and the two antijoins of R by S and vice-versa.)

The last tactic is especially useful for queries that require the union of two semijoins of the same relation by two different relations. To illustrate this point, consider the following query:

Query 11

```
SELECT   R.A
FROM     R
WHERE  EXISTS    (SELECT *
                  FROM S
                  WHERE R.B=S.C)
OR       EXISTS  (SELECT *
                  FROM T
                  WHERE R.D=T.E)
```

The query graph of this query is shown in Figure 6.1 (we introduce OR nodes to represent disjunctions). The query can be processed in one of three ways:

(a) Use the tactics of section 5 to evaluate the two nested subqueries, which will result in the computation of an Agg attribute for each; the final answer is obtained by restricting R on the disjunction of these attributes.

(b) Use a variant of the rule for replacing EXISTS with a semijoin to process the two subqueries by semijoins, and then construct the union.

(c) Simultaneously do one semijoin and its corresponding antijoin (this operation was called a semiouterjoin in [DAYA85]; then use the antijoin in the subsequent semijoin.

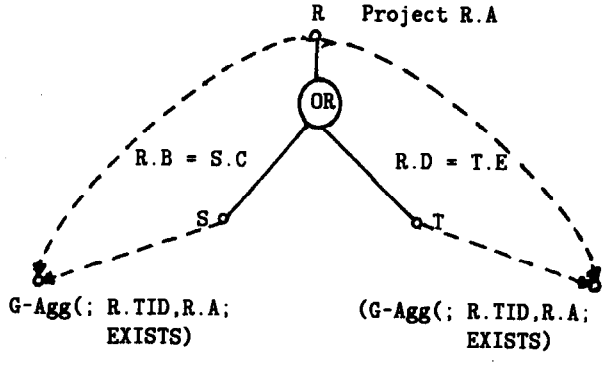

Figure 6.1 Query Graph of a Disjunctive Query

For more complicated cases (e.g., when disjunction of quantified nested subqueries occur inside a quantifier), we may not be able to use semijoins or antijoins. However, we can adapt the G-Join, G-Agg technique of section 5 as in (a) above: essentially, evaluate each parallel nested subquery to return an Agg

value. The G-Agg for the quantified subquery at the next level up then tests the disjunction of these Agg values instead of just testing for the empty set. This last tactic is extensible to other Boolean connectives (e.g., =>) as well.

7. Conclusions

Existing query optimizers consider many strategies for Restrict-Project-Join queries. They use heuristics to enumerate permutations of the joins, and to position restrictions and projections relative to the joins, and use sophisticated cost models to compare strategies.

In practice, however, query languages such as SQL contain many powerful features (e.g., control over duplicates, nested subqueries, aggregates, quantifiers, and disjunctions) that are not expressible as Restrict-Project-Join sequences. Existing query processors are severely limited in their strategies for processing such queries: typically, they just use tuple substitution. Unfortunately, however, tuple substitution is often inefficient, especially in a distributed system. Hence, there is a pressing need to expand the space of query processing strategies for such queries.

In this paper, we introduced a collection of new operators, and described how they can be implemented by straightforward extensions to existing implementation methods. From the algebraic properties of these operators, we derived new query processing tactics that greatly increase the space of strategies considered by the optimizer, without changing the basic architecture of the optimizer. Instead of joins, the optimizer must now enumerate G-Joins, and must position G-Aggs, G-Restricts, Projects, and Delta-Projects relative to the G-Joins. The optimizer can consider the relative cost of tuple substitution (nested iteration) for implementing the G-Joins and other (e.g., sort-merge) implementation methods. Finally, the optimizer can often pipeline operations if the intermediate results are correctly grouped or ordered, thereby avoiding the cost of storing temporaries (which is basically the only advantage of tuple substitution). Hence, we now have a unified framework for optimizing all of SQL (and other languages with similar features).

Future work consists of working out detailed heuristics for enumeration, heuristics as in [WONG 76] for decomposing complex queries into simpler queries (to control the exponential growth in the size of the strategy space), formal proofs of the transformation rules, and a detailed cost model.

Our style in this work is algebraic. Hence, it should be easy to add the new operations and rules into rule-driven and extensible optimizers such as those described in [FREY 87, DEWI 87].

Acknowledgements

The author wishes to thank his colleagues, Arvola Chan, Anil Nori, Joh Hee, Sara Haradhvala, Arnie Rosenthal, and Ken Abbott for fruitful discussions of these ideas, and Andrea Fisher and Julie Goyette for text processing support.

8. References

[BERN81] Bernstein, P.A. and D.M. Chiu, "Using Semijoins to Solve Relational Queries," *J ACM 28*, 1 (Jan. 1981).

[BLAS77] Blasgen, M.W., and K.P. Eswaran, "On the Evaluation of Queries in Relational Database Systems," IBM Research Report RJ1745, April 1976.

[CERI85] Ceri, S., and G. Gottlob, "Translating SQL in Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries," *IEEE Trans.* SE-11, 4 (April 1985).

[CHAM76] Chamberlin, D.D., et al., "SEQUEL 2: A Unified Appraoch to Data Definition, Manipulation, and Control," *IBM J. Res. and Dev.* 20, 6 (Nov. 1976).

[CODD70] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *CACM 13*, 6 (June 1970).

[CODD72] Codd, E.F., "Relational Completeness of Database Sublanguages," in *Database Systems*, Current Comp. Sci. Symp. 6 (R.Rustin, ed.), Prentice-Hall, Englewood Cliffs, N.J. 1972.

[CODD79] Codd, E.F., "Extending the Database Relational Model to Capture More meaning," *ACM TODS* 4, 4 (Dec. 1979).

[DATE85] Date, C.J., *A Guide to DB2*, Addison-Wesley Pub. Co., Reading Mass., 1985.

[DAYA82] Dayal, U., N. Goodman, and R.H. Katz, "An Extended Relational Algebra with Control Over Duplicate Elimination," *Proc. ACM PODS* 1982.

[DAYA83] Dayal, U., "Processing Queries With Quantifiers: A Horticultural Approach," *Proc. ACM PODS* 1983.

[DAYA85] Dayal, U., "Query Optimization in a Multidatabase System," in *Query Processing in Database Systems*. (W. Kim, D. Reiner, D. Batory, eds.), Springer-Verlag, 1984.

[DEWI87] DeWitt, D.J., and G. Graefe, "The EXODUS Optimizer Generator," *Proc. ACM SIGMOD* 1987.

[FREY87] Freytag, J.C., "A Rule-Based View of Query Optimization," *Proc. ACM SIGMOD* 1987.

[GANS87] Ganski, R., and H.K.T. Wong, "Optimization of Nested SQL Queries Revisited," *Proc. ACM SIGMOD* 1987.

[JARK82] Jarke, M., and J. Schmidt, "Query Processing Strategies in the PASCAL/R Relational Database Management System," *Proc. ACM SIGMOD* 1982.

[KIM82] Kim, W., "On Optimizing an SQL-Like Nested Query," *ACM TODS* 7, 3 (September 1982).

[KLUG82] Klug, A., "Access Paths in the ABE Statistical Query Facility," *Proc. ACM SIGMOD* 1982.

[LACR76] Lacroix, M., and A. Pirotte, "Generalized Joins," *SIGMOD Record* 8, 3 (Sept. 1976).

[LOHM84] Lohman, G.M., et al., "Optimization of Nested Queries in a Distributed Relational Database," *Proc. VLDB* 1984.

[MACK86a] Mackert, L.F., and G.M. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proc. ACM SIGMOD* 1986.

[MACK86b] Mackert, L.F., and G.M. Lohman, "R* Optimizer Validation and Evaluation for Distributed Queries," *Proc. VLDB* 1986.

[PALE72] Palernio, F.P., "A Database Search Problem," IBM Research Report RJ1072, July, 1972.

[ROSE84] Rosenthal, A., and D. Reiner, "Extending the Algebraic Framework of Query Processing to Handle Outerjoins," *Proc. VLDB* 1984.

[SELI79] Selinger, P.G., et al., "Access path Selection in a Relational Database Management System," *Proc. ACM SIGMOD* 1979.

[SHIP81] Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX," *ACM TODS* 6, 1 (March 1981).

[SMIT83] Smith, J.M., S. Fox, and T. Landers, "ADAPLEX Rationale and Reference Manual," Tech Report CCA-83-08, Computer Corp. of America, Jan. 1983.

[WONG76] Wong, E., and K. Youssefi, "Decomposition – A Strategy for Query Processing," *ACM TODS* 1, 3 (Sept. 1976).