Outerjoin Simplification and Reordering for Query Optimization

CÉSAR GALINDO-LEGARIA
Harvard University, Cambridge, MA
and
ARNON ROSENTHAL
The MITRE Corporation, Bedford, MA

Conventional database optimizers take full advantage of associativity and commutativity properties of join to implement efficient and powerful optimizations on select/project/join queries. However, only limited optimization is performed on other binary operators. In this article, we present the theory and algorithms needed to generate alternative evaluation orders for the optimization of queries containing outerjoins. Our results include both a complete set of transformation rules, suitable for new-generation, transformation-based optimizers, and a bottom-up join enumeration algorithm compatible with those used by traditional optimizers.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Outerjoins, query optimization, query reordering

1. INTRODUCTION

1.1 Motivation for Outerjoin Optimization

Relational join combines information from two tables by creating pairs of matching tuples. If a tuple in one relation has no corresponding tuple in the other, data in this tuple does not appear in the join result. *Outerjoin* is a modification of join that preserves all information from one or both of its arguments [Codd 1979; Lacroix and Pirotte 1976]. For instance, the left outerjoin of tables R_1, R_2 contains the join of those tables, plus the unmatched tuples of R_1. Since they have no corresponding R_2 tuples, unmatched tuples from R_1 are padded with null values on R_2 columns.

Authors' addresses: C. Galindo-Legaria, Microsoft Corp., One Microsoft Way, Redmond, WA 98052; email: cesarg@microsoft.com. A. Rosenthal, The MITRE Corporation, Bedford, MA 01730, email: arnie@mitre.org.

Outerjoins are available today in a number of implementations of SQL, and they are included in the ISO-ANSI standard draft for SQL2 [ANSI 1992]. Among the systems that currently support outerjoin are Sybase, SQL Server, NonStop SQL of Tandem, SSQL of ShareBase, and ORACLE/SQL.

As with joins, changing the order of evaluation of outerjoins can improve the performance of query execution dramatically. However, associativity properties of outerjoin are limited, which makes the generation of alternative evaluation plans a much more difficult task. Below, we discuss three topics: applications of outerjoins, difficulties in outerjoin reordering, and the potential cost reduction attained by such reordering.

Example 1. For a database with tables containing information about CUSTOMERS and ORDERS, we want to find customers who live in New York, and their current orders. Outerjoin is used to retain tuples for New York customers who have no orders

```
Select   All
From     CUSTOMERS Left Outerjoin ORDERS
         on CUSTOMERS.cust# = ORDERS.cust#
Where    CUSTOMERS.city = "New York"
```

The keyword **Left** specifies which relation must be preserved.

Outerjoins can be directly specified by a user, as in the query of the example above, or they can be introduced by a processor as part of a query evaluation strategy. Our examples are relational, but the ideas also apply to object-oriented databases—a specific case is mentioned in Section 5.5. Outerjoins are used in the following applications:

Database merging. In large companies, it is common to find that groups have developed independent databases. The goal of *federated databases* is to provide a global view that combines all information from these independent databases. The subject is an active, complicated research topic [Sheth 1991; Sheth and Larson 1990]. For query processing, tables from different databases must be combined via two-sided outerjoins, rather than joins [Dayal and Hwang 1984; Wang and Madnick 1990].

Hierarchical views. Sometimes it is convenient to think of *parent* objects as having an associated set of *children*. For instance, each department in a company has an associated set of employees. If a join combines tables, DEPT and EMP, departments with no employees are discarded. In general, to construct hierarchical views that preserve objects with no children, we need outerjoin rather than join [David 1991; Lee and Wiederhold 1994; Rosenthal and Galindo-Legaria 1990; Roth et al. 1989; Scholl et al. 1987].

Nested queries. Both SQL and Object Query Languages allow *query nesting*—that is, to test a predicate on a single *outer* object, one evaluates a subquery. But straightforward evaluation would correspond to using the nested loops method for a series of joins, and may be very inefficient. To give the query optimizer some freedom, the query must be represented algebraically [Kim 1982]. In the general case, outerjoin is necessary for this algebraic representation [Dayal 1987; Muralikrishna 1989].

Universal quantifiers. Queries with universal quantifiers can be translated into algebraic expressions with relational division, using Codd's proof of equivalence between calculus and algebra [Ullman 1982]. But this strategy introduces large intermediate results of Cartesian products. An alternative strategy using outerjoins reduces significantly the size of intermediate results [Dayal 1983].

For select/project/join queries, changing the order of evaluation of joins is a powerful and commonly used optimization technique, which often improves execution time by orders of magnitude. Joins can be evaluated in any order due to associativity and commutativity properties of the operator, so generating alternative query evaluation orders is relatively easy. However, when both joins and outerjoins are present in a query, changing the order of evaluation is complicated in two ways: First, instead of always using join, we need to determine which operator to apply at each step of the new evaluation order. Second, join and outerjoin are not always associative with respect to each other, so not all orders of evaluation may be possible. The following example illustrates the associativity problem.

Example 2. Consider the database of CUSTOMERS and ORDERS used in Example 1, and assume there is another table, ITEMS, with the items in stock. If CUSTOMERS_NY denotes the view consisting of New York customers, the following query returns these customers and their orders for items in stock (join predicates are straightforward equalities, and are omitted for readability).

Select **All**
From CUSTOMERS_NY **Left Outerjoin** (ORDERS **Join** ITEMS)

If the query is evaluated directly as stated, it produces a huge intermediate result, (ORDERS Join ITEMS). And if New Yorkers account for a relatively small fraction of ORDERS, most of the tuples in this intermediate are irrelevant. A better evaluation order would first eliminate these irrelevant tuples as early as possible, by combining CUSTOMERS_NY with ORDERS. Since we need to preserve all New York customers in the result, the operator to combine them must be the following outerjoin:

Select **All Into** CUST_NY_ORDER
From CUSTOMERS_NY **Left Outerjoin** ORDERS

But we can use neither join nor outerjoin to combine CUST_NY_ORDER with ITEMS, to produce the result of the original query. Computing (CUST_NY_ORDER **Join** ITEMS) wrongly eliminates customers without orders in CUST_NY_ORDER, because they do not join with ITEMS. On the other hand, computing (CUST_NY_ORDER **Left Outerjoin** ITEMS) wrongly preserves orders in CUST_NY_ORDER for out-of-stock items, which do not join with ITEMS.

Note, however, that if we had available an efficient operator to compute the original query from CUST_NY_ORDER and ITEMS, the second order of evaluation is arbitrarily superior to the original—the smaller the fraction of ORDERS for New York customers, the larger the improvement of the second evaluation order. In this article we show how to exploit this potential.

1.2 Relational Definitions and Notation

Our theorems require that base relations be duplicate-free. In Section 3.1, we discuss how to handle duplicates. Also, we assume base relations have no tuple with nulls in *every* column; they may contain tuples with nulls in *some* columns.

Operators. We use relational operators *selection* on a predicate ($\sigma_p R$), duplicate-removing *projection* on a set of attributes (π_a), *Cartesian product* of relations with disjoint schemas ($R_1 \times R_2$), and set *union* and *difference* of relations with identical schemas ($R_1 \cup R_2$, $R_1 - R_2$) [Ullman 1982]. *Relational join* ($\bowtie$) applies a *match predicate* p on the product of two relations.

We use the *outerunion* ($\uplus$) operator [Codd 1979] to introduce *null-padding*, necessary in outerjoins. Assume the schema S_1 of R_1 is different from $S_2 = \text{sch}(R_2)$, but they may have some attributes in common. The outerunion $R_1 \uplus R_2$ delivers tuples in R_1 plus tuples in R_2, *null-padded* to the schema $S_1 \cup S_2$:

$$R_1 \uplus R_2 = (R_1 \times \{\text{null}_{S_2 - S_1}\}) \ \cup \ (R_2 \times \{\text{null}_{S_1 - S_2}\}),$$

where null_A is a tuple with null values in all attributes of A.

The outerjoin of two relations includes the result of join, plus all unmatched tuples from one or both of its arguments [Codd 1979; Lacroix and Pirotte 1976]. *Left outerjoin* ($\rightarrow$) and *full outerjoin* ($\leftrightarrow$) are defined, respectively, as follows:

$$R_1 \overset{p}{\text{——}} R_2 = (R_1 \overset{p}{\bowtie} R_2) \uplus (R_1 - \pi_{\text{sch}(R_1)}(R_1 \overset{p}{\bowtie} R_2)).$$

$$R_1 \overset{p}{\text{——}} R_2 = (R_1 \overset{p}{\bowtie} R_2) \uplus (R_1 - \pi_{\text{sch}(R_1)}(R_1 \overset{p}{\bowtie} R_2)) \uplus (R_2$$
$$- \pi_{\text{sch}(R_2)}(R_1 \overset{p}{\bowtie} R_2)).$$

The *right outerjoin* is $R_1 \xleftarrow{p} R_2 = R_2 \xrightarrow{p} R_1$. Each outerjoin variety *preserves* the relation on the side opposite an arrow, and *introduces nulls* in the attributes of the relation on the arrow side. If an operator introduces nulls in attributes A it also introduces nulls in any subset of A.

Left and right outerjoin each preserves only one of its arguments, so they are called one-sided outerjoins. Except for associative identities, where it is

R

A	B	C
a	c	b
d	f	a
c	d	b

S

C	D	E
b	g	a
d	a	f

$R \overset{R.C=S.C}{\bowtie} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a

$R \overset{R.C=S.C}{\rightarrow} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d	f	a	–	–	–

$R \overset{R.C=S.C}{\leftrightarrow} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d	f	a	–	–	–
–	–	–	d	a	f

Fig. 1. Join and outerjoin operators.

convenient to use right outerjoins, we assume without loss of generality that expressions do not use right outerjoins, only left. Full outerjoin is also called two-sided outerjoin, because it preserves information from both arguments. We use *combine* operator as a generic term for join or outerjoin. Figure 1 shows examples of combine operators. A dash ("—") is used for null values.

Null-rejection. We say a predicate p *rejects nulls* in attribute set A if it evaluates to FALSE or UNDEFINED on every tuple in which all attributes in A are null. An operator *rejects nulls* if tuples in which all attributes in A are null do not affect the operator's result. For example, in

$$\sigma_{\text{City}=\text{“New York”}}\text{CUSTOMERS}$$

both the predicate and the selection operator reject nulls on City, and on any superset of {City}, (e.g., sch(CUSTOMERS)).

For selection and join, the operator rejects nulls if the predicate rejects nulls. For one-sided outerjoin, $(R_1 \rightarrow R_2)$ rejects nulls on A if p rejects null on A *and* $A \subseteq R_2$. Full outerjoin never rejects nulls, because it lets through all tuples from its inputs.

In the three-valued logic of SQL, the result of comparing a NULL value is UNDEFINED [ANSI 1992; Melton and Simon 1993]. Logical connectives (e.g., AND, NOT) produce UNDEFINED whenever any of its arguments is UNDEFINED. The net result is: If a null-valued attribute is compared in the predicate of a **Where** clause that contains no OR connectives, the predicate evaluates to UNDEFINED, and the tuple is rejected. Therefore, conjunctive SQL predicates usually reject nulls in every attribute they reference. The SQL **IsNull** primitive is used to avoid such rejection. The following query finds customer orders that can most easily be deferred; it rejects nulls on ORDERS.priority, but not on due date.

```
Select  All
From    ORDERS
Where   ORDERS.priority = “low” or IsNull(ORDERS.duedate)
```

The set of attributes referenced by a predicate p is called the *schema* of p, and denoted $\mathrm{sch}(p)$.[1] As a notational convention, we annotate predicates to reflect their schema. If $\mathrm{sch}(p)$ includes attributes of both R_i, R_j and only those relations, we can write the predicate as $p^{R_iR_j}$, or simply p^{ij}, when the relations are clear from the context.

1.3 Summary of Results and Comparison with Previous Work

This article focuses on the problem of changing the order of evaluation of queries containing both joins and outerjoins. Query graphs represent nonnested Select/Project/Join queries unambiguously, but are generally insufficient to express queries involving outerjoins (or even nesting). We therefore assume that the initial query is available as an unambiguous operator tree.

We present four major results:

(1) A *simplification algorithm* that allows the replacement of outerjoins by joins, in some cases.
(2) *Associative identities* that justify changing the order of evaluation. New evaluation orders sometimes require a *generalized outerjoin* operator, in addition to join and outerjoin (e.g., to deal with cases such as that of Example 2).

The next two results depend on additional assumptions that predicates reject null tuples, and that no outerjoin predicate references more than two base relations.

(3) A proof that our list of identities is *complete*, in the sense that they can generate any query reordering consistent with the connectivity heuristic.
(4) Instructions for extending conventional optimizers to handle joins and outerjoins, including the logic to choose the correct operator in each step of the bottom-up construction of execution trees.

The results presented here consolidate and simplify partial results reported earlier by the authors in Galindo-Legaria [1992]; Galindo-Legaria and Rosenthal [1992]; Rosenthal and Galindo-Legaria [1990]. Prior to those papers, results on outerjoin reordering were fragmentary, and insufficient to guide optimizer builders. For example, Rosenthal and Reiner [1984] describe a theory that allows reassociation, but use excessively low-level operators that do not match strategy enumeration optimizers such as

---

[1] Semantically, the schema of a predicate is the set of attributes it depends on, i.e. for any two tuples t_1, t_2 that coincide in attributes $\mathrm{sch}(p)$ we have $p(t_1) = p(t_2)$. This “semantic definition” is difficult to test, in the general case, so we rely on the more conservative “syntactic definition.”

System R. And in rule-driven optimizers, generating reassociations would require a large number of rules to be applied in the proper sequence.

Other papers deal with outerjoins in the context of specific applications. Though focused on universally quantified queries, Dayal [1983] gives some initial rules on valid evaluation orders for joins and one-sided outerjoins. Dayal [1987] points out that one-sided outerjoin—and other operators useful for nested subqueries—can be implemented by minor modifications to join algorithms. The paper also gives some rules to change the order of processing for joins and one-sided outerjoin, but those rules have glitches and do not generate all processing orders. Muralikrishna [1989] gives a pipelining algorithm to evaluate one-sided outerjoins, in the context of nested SQL. The paper expands on how to use outerjoin to model not only linear-nesting but also tree-nesting in SQL, but its treatment on how to change the evaluation order is limited. Basically, it follows the rule of performing first all joins and then all outerjoins, as outlined in Dayal [1983; 1987]. Ozsoyoglu et al. [1989] gives some identities for processing natural outerjoins, as part of the optimization strategies used by a system that stores and computes statistical information.

Papers that use outerjoins to evaluate nested SQL usually proceed in two steps: They first map the SQL query into a query graph with directed and undirected edges, and then define permissible orders of evaluation of the operators represented as graph edges. But query graphs do not posses a natural evaluation rule, and do not denote a unique result, in general. For this reason, one cannot separate the two steps when understanding the algorithms. Two orthogonal issues (modeling nested SQL with outerjoins, and selecting an order of evaluation) often intertwine, and typically one gets an incomplete treatment of each issue. In contrast, we focus on the generation of new evaluation orders, starting with an initial operator tree that unambiguously specifies the desired query.

Canonical declarative forms for expressing outerjoin queries were described in Galindo-Legaria [1994]. These canonical forms led to simpler proofs of our many identities, but the focus of that work was not query optimization. Recently, Bhargava et al. [1995] studied how to relax our assumptions for points 3 and 4 above. We briefly summarize their results in Section 3.1.

Section 2 presents our outerjoin simplification algorithm, associative identities for joins and outerjoins, and the generalized outerjoin operator. Section 3 establishes a framework for reordering, based on query graphs and operator trees, and states our main results. Section 4 describes how to build reorderings of join/outerjoin queries bottom-up. Section 5 discusses how to integrate our algorithms in current systems, and Section 6 presents our conclusions.

2. TACTICS FOR OUTERJOIN OPTIMIZATION

In this section we describe modifications that can be done on queries containing outerjoins. Subsections deal with outerjoin simplification, associativity of joins with outerjoins, and finally reassociation using a generalized outerjoin operator that enables the optimization suggested in Example 2.

2.1 Outerjoin Simplification

It is helpful to rewrite outerjoins as joins, whenever possible, for several reasons. First, the rewritten query often has smaller intermediate results. Second, the set of candidate implementations is largest for join, smaller for outerjoin, and smallest for full outerjoin. For example, the optimizer is free to choose the inner and outer relations for joins. Simplification based on null rejection is our first step.[2]

The simplification idea is that if a later operator discards the null-padded tuples introduced by an outerjoin, such outerjoin can be rewritten as regular join without altering the result. For example,

$$\sigma_{\text{ORDERS.date}<1/1/95}(\text{CUSTOMERS} \rightarrow \text{ORDERS})$$

$$= \sigma_{\text{ORDERS.date}<1/1/95}(\text{CUSTOMERS} \bowtie \text{ORDERS}).$$

Our development of this approach begins with identities that apply to all selections:

$$R_1 \overset{p_1 \wedge p_2}{\text{———}} R_2 = R_1 \overset{p_1}{\text{——}} (\sigma_{p_2} R_2), \text{ if } \text{sch}(p_2) \subseteq \text{sch}(R_2). \tag{1}$$

$$\sigma_{p_1}(R_1 \overset{p_2}{\text{——}} R_2) = (\sigma_{p_1} R_1) \overset{p_2}{\text{——}} R_2, \text{ if } \text{sch}(p_1) \subseteq \text{sch}(R_1). \tag{2}$$

Note that we could not push down a selection on the arrow side of an outerjoin (either side of full outerjoin). However, if the predicate p rejects nulls, then the following straightforward identities simplify by *removing an arrow*.

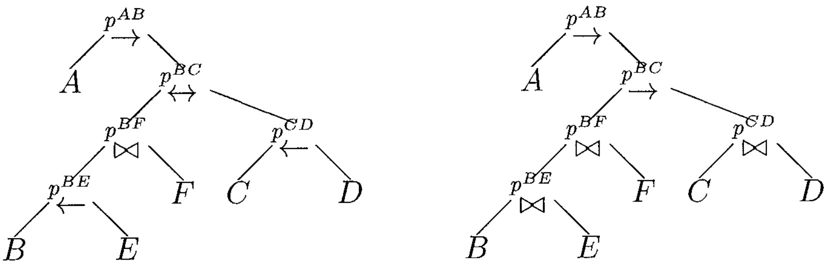
$$\sigma_{p_1}(R_1 \overset{p_2}{\text{——}} R_2) = \sigma_{p_1}\left(R_1 \overset{p_2}{\bowtie} R_2\right), \text{ if } p_1 \text{ rejects nulls on sch}(R_2). \tag{3}$$

$$\sigma_{p_1}(R_1 \overset{p_2}{\text{——}} R_2) = \sigma_{p_1}(R_1 \overset{p_2}{\text{——}} R_2), \text{ if } p_1 \text{ rejects nulls on sch}(R_2). \tag{4}$$

To simplify an entire operator tree, it is convenient to rewrite a predicate p that rejects nulls on attribute a as $p = p \wedge \neg\mathbf{IsNull}(a)$. Now $\neg\mathbf{IsNull}(a)$ can simplify an outerjoin using identities (3) and (4). It can then be moved

---

[2] Simplification based on null rejection was first introduced in Galindo-Legaria and Rosenthal [1992], and later used for a specific application in Lee and Wiederhold [1994]. Simplification based on integrity constraints and projection are described in Chen [1990]; Lee and Wiederhold [1994] and Rosenthal and Galindo-Legaria [1990].



(a) Original query. (b) Simplified query.

Fig. 2. Query simplification.

down the tree to the child that supplied a, using identities (2) and (1), plus the familiar rewrite for selection and join. When the process completes, the redundant predicates are removed.

Algorithm **A** below more directly produces the simplifications that would be obtained by creating $\neg$**IsNull**( ) predicates, pushing them down the tree, and simplifying outerjoins.

Algorithm A. Outerjoin simplification using null rejection.
Input. An operator tree Q with joins and outerjoins.
Output. A simplified query Q' equivalent to Q.
Procedure. Traverse the operator tree top-down; for each operator $\odot_1$ do
  For each operator $\odot_2$ descended from $\odot_1$ do
  — If, for some R, $\odot_1$ rejects nulls on sch(R) and $\odot_2$ introduces nulls in sch(R), then remove the arrow from outerjoin $\odot_2$, on the side of R, to convert it either to a one-sided outerjoin or to a join.[3]

Algorithm A performs all simplifications resulting from null-rejection. A second pass of the algorithm will not produce further simplifications.

Example 3. Figure 2(b) shows the result of the simplification algorithm on the query of Figure 2(a). The root operator $\xrightarrow{p^{AB}}$ rejects nulls on sch(B), so it simplifies $\xleftrightarrow{p^{BC}}$ to $\xrightarrow{p^{BC}}$; it also simplifies $\xleftarrow{p^{BE}}$ to $\overset{p^{BE}}{\bowtie}$. The newly obtained $\odot = \xrightarrow{p^{BC}}$ rejects nulls on sch(C), so it simplifies $\xleftarrow{p^{CD}}$ to $\overset{p^{CD}}{\bowtie}$.

2.2 Join/Outerjoin Associativity

In addition to simplification, a primary tool for query processing is the ability to change the order of operations, using some form of associativity. Associative identities for join, outerjoin, and full outerjoin are the following (for detailed proofs, see Galindo-Legaria [1992]):

$$(R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{13}\wedge p^{23}}{\bowtie} R_3 = R_1 \overset{p^{12}\wedge p^{13}}{\bowtie} (R_2 \overset{p^{23}}{\bowtie} R_3). \tag{5}$$

---

[3] For a full outerjoin, the test is applied to each side separately, and in some cases may remove arrows on both sides. The simplification is valid for any set of attributes satisfying the conditions, not just for relation schemes.

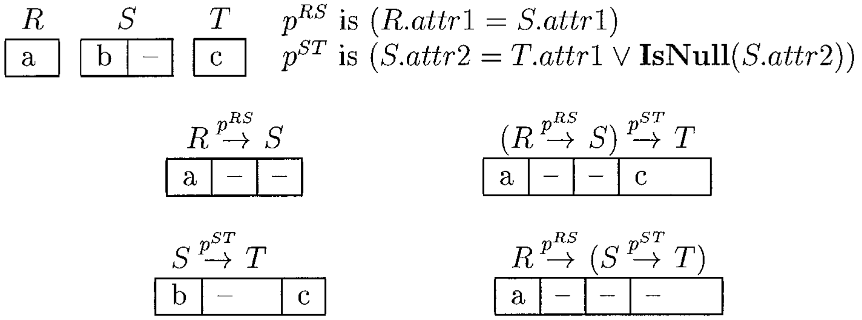


Fig. 3. Effect of a predicate that does not reject nulls.

$$(R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\text{——}} R_3 = R_1 \overset{p_{12}}{\bowtie} (R_2 \overset{p^{23}}{\text{——}} R_3). \tag{6}$$

$$(R_1 \overset{p^{12}}{\text{——}} R_2) \overset{p^{23}}{\text{——}} R_3 = R_1 \overset{p^{12}}{\text{——}} (R_2 \overset{p^{23}}{\text{——}} R_3),$$

$$\text{if } p^{23} \text{ rejects nulls on } \mathrm{sch}(R_2). \tag{7}$$

$$(R_1 \overset{p^{12}}{\text{——}} R_2) \overset{p^{23}}{\text{——}} R_3 = R_1 \overset{p^{12}}{\text{——}} (R_2 \overset{p^{23}}{\text{——}} R_3). \tag{8}$$

$$(R_1 \overset{p^{12}}{\text{——}} R_2) \overset{p^{23}}{\text{——}} R_3 = R_1 \overset{p^{12}}{\text{——}} (R_2 \overset{p^{23}}{\text{——}} R_3),$$

$$\text{if } p^{12} \text{ and } p^{23} \text{ reject nulls on } \mathrm{sch}(R_2). \tag{9}$$

$$(R_1 \overset{p^{12}}{\text{——}} R_2) \xrightarrow{p^{23}} R_3 = R_1 \overset{p^{12}}{\longleftrightarrow} (R_2 \xrightarrow{p^{23}} R_3),$$

$$\text{if } p^{23} \text{ rejects nulls on } \mathrm{sch}(R_2). \tag{10}$$

Example 4. Predicates that do not reject nulls as required by specific associative identities probably occur in a small fraction of queries, but they do violate those identities. For example, consider the case shown in Figure 3. Predicate p^{ST} does not reject nulls on sch(S). Then, associative identity 7 does not hold, and in fact $(R \xrightarrow{p^{RS}} S) \xrightarrow{p^{ST}} T \neq R \xrightarrow{p^{RS}} (S \xrightarrow{p^{ST}} T)$.

2.3 Generalized Outerjoin

The identities listed in Section 2.2 do not allow all join/outerjoin expressions to be reassociated. Example 2 in Section 1 showed that $R_1 \xrightarrow{p^{12}} (R_2 \overset{p^{23}}{\bowtie} R_3)$ is equal to neither $(R_1 \xrightarrow{p^{12}} R_2) \overset{p^{23}}{\bowtie} R_3$ nor to $(R_1 \xrightarrow{p^{12}} R_2) \xrightarrow{p^{23}} R_3$. Yet, if $(R_2 \overset{p^{23}}{\bowtie} R_3)$ is large, first combining R_1 with R_2 can greatly reduce the execution cost. *Generalized outerjoin* is used to reorder in this case.

R

A	B	C
a	c	b
d	f	a
c	d	b
c	f	a
c	e	f

S

C	D	E
b	g	a
d	a	f

$R \overset{R.C=S.C}{\rightarrow} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d	f	a	–	–	
c	f	a	–	–	
c	e	f	–	–	

$R\ GOJ[R.C = S.C, \{A\}]\ S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d	–	–	–	–	

Fig. 4. Generalized outerjoin.

Generalized outerjoin *preserves projections* of its operands, much as outerjoin preserves complete operands. The operator was first defined in Rosenthal and Galindo-Legaria [1990], repairing a small error in the generalized join from Dayal [1987]. Then it was extended to deal with full outerjoins in Galindo-Legaria and Rosenthal [1992].

The *generalized outerjoin* on p of relations R_1, R_2, *preserving* attributes $A \subseteq R_1$ is defined as:

$$R_1\ GOJ[p, A]R_2 = (R_1 \overset{p}{\bowtie} R_2) \uplus (\pi_A R_1 - \pi_A(R_1 \overset{p}{\bowtie} R_2)).$$

Figure 4 shows an example of outerjoin and generalized outerjoin. The first of these identities (see Galindo-Legaria [1992] for their proofs) allows early evaluation of one-sided outerjoins:

$$R_1 \overset{p^{12}}{\relbar\relbar\relbar} (R_2 \overset{p_{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\relbar\relbar\relbar} R_2)\ GOJ[p^{23}, \mathrm{sch}(R_1)]R_3,$$

$$\text{if } p^{23} \text{ rejects nulls on } \mathrm{sch}(R_2). \qquad (11)$$

$$R_1 \overset{p^{12}}{\relbar\relbar\relbar} (R_2 \overset{p_{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\relbar\relbar\relbar} R_2)\ GOJ[p^{23}, \mathrm{sch}(R_1)]R_3,$$

$$\text{if } p^{23} \text{ rejects nulls on } \mathrm{sch}(R_2). \qquad (12)$$

Example 5. Returning to Example 2 of Section 1.1, assume CUSTOMERS_NY has 3 tuples, each with two orders, and ORDERS has 1 million tuples, most of them for items in stock. Using identity 11, the original query can be rewritten as

CUSTOMERS_NY → (ORDERS $\bowtie$ ITEMS) =
(CUSTOMERS_NY → ORDERS) GOJ[sch(CUSTOMERS_NY)] ITEMS.

The GOJ expression produces an intermediate result of cardinality 6, while the intermediate result of the original expression has cardinality close to 1 million. The query is evaluated more efficiently using the GOJ reordering.

What GOJ does is deliver the join of (CUSTOMERS_NY → ORDERS) with ITEMS, plus those customers in (CUSTOMERS_NY → ORDERS) that do not appear in the join with ITEMS, without duplication. A customer that appears padded with nulls in the result does not appear also with a matching order and item.

Implementation of GOJ. To show that GOJ is practical, we describe a simple execution algorithm. It is an open problem to find more efficient algorithms.

GOJ can be implemented using two simpler primitives: A modified one-sided outerjoin algorithm, plus a modified duplicate elimination algorithm.

Say we want to compute $R_1\ GOJ[p, A]\ R_2$, with $A \subset \mathrm{sch}(R_1)$. A left outerjoin algorithm will take every tuple t_1 in R_1, and if it has no matching R_2 tuple then t_1 is output padded with nulls on R_2 attributes. Modify the algorithm so that for such tuples, only the A-attributes of t_1 are output, instead of the whole t_1.

On the result of the modified outerjoin, use a modified duplicate-elimination algorithm to eliminate subsumed tuples. For example, sort the table using A-columns as primary key, and non-A-columns as secondary key. Then for each group with common A values, eliminate padded tuples if there is a non-padded tuple in the group. Otherwise, output a single padded tuple.

GOJ preserving multiple sets. Early evaluation of full outerjoins can require preserving multiple sets of attributes. The operator definition is rather daunting; in Section 5.4 we consider optimizers that do not support this construct.

The *generalized outerjoin* on p of relations R_1, R_2, *preserving* disjoint sets of attributes $A_{11}, \ldots, A_{1n}, A_{21}, \ldots, A_{2m}$, such that $A_{1i} \subseteq \mathrm{sch}(R_1)$, $A_{2j} \subseteq \mathrm{sch}(R_2)$ is

$$R_1\ GOJ[p, A_{11}, \ldots, A_{1n}, A_{21}, \ldots, A_{2m}]R_2 =$$

$$(R_1 \overset{p}{\bowtie} R_2) \uplus$$

$$(\pi_{A_{11}}R_1 - \pi_{A_{11}}(R_1 \overset{p}{\bowtie} R_2)) \uplus \cdots \uplus (\pi_{A_{1n}}R_1 - \pi_{A_{1n}}(R_1 \overset{p}{\bowtie} R_2)) \uplus$$

$$(\pi_{A_{21}}R_2 - \pi_{A_{21}}(R_1 \overset{p}{\bowtie} R_2)) \uplus \cdots \uplus (\pi_{A_{2m}}R_2 - \pi_{A_{2m}}(R_1 \overset{p}{\bowtie} R_2)).$$

We can now reorder the following join/outerjoin/GOJ cases (again, see

Galindo-Legaria [1992] for proofs):

$$R_1 \overset{p^{12}}{\text{——}} (R_2 \overset{p^{23}}{\text{——}} R_3) = (R_1 \overset{p^{12}}{\text{——}} R_2)\ GOJ[p^{23}, \text{sch}(R_1), \text{sch}(R_3)]\ R_3,$$

$$\text{if } p^{23} \text{ and } p^{12} \text{ reject nulls on sch}(R_2). \tag{13}$$

Finally, if p^{23} rejects nulls on sch(R_2), and p^{12} rejects nulls on a set of attributes $A_s \subseteq \text{sch}(R_2)$, which is disjoint from $A_{21}, \ldots, A_{2n}$, we have

$$R_1 \overset{p^{12}}{\text{——}} (R_2\ GOJ[p^{23}, A_{21}, \ldots, A_{2n}, A_{31}, \ldots, A_{3m}]R_3)$$

$$= (R_1 \overset{p^{12}}{\text{——}} R_2)\ GOJ[p^{23}, \text{sch}(R_1), A_{21}, \ldots, A_{2n}, A_{31}, \ldots, A_{3m}]R_3. \tag{14}$$

3. TREES, GRAPHS, AND QUERY REORDERING

In Section 2 we gave identities to reorder queries, and showed that they are useful to reduce query execution cost. However, it is not obvious from this list of associative identities what are the reorderings allowed on a given query—or which reorderings *cannot* be obtained. In this section we present reordering properties of *entire* queries, based on our set of associative identities.

3.1 Notation and Restrictions

If we can obtain some query Q' from Q using associative identities (5) through (14), we say that Q' is *I-equivalent* to Q. Q^I denotes the set of all queries I-equivalent to Q.

Outerjoin simplification and associativity identities of Section 2 apply to all queries, but we examine the extent of reordering achievable (i.e., the set Q^I) only for *simple* join/outerjoin queries. A join/outerjoin query Q is *simple* if it satisfies the following:

(1) *Q cannot be further simplified using Algorithm A.* In practice, Q will usually be obtained by the idempotent Algorithm A.
(2) *Predicates reject nulls on the scheme of every relation they reference.*
(3) *There are no Cartesian products.*
(4) *All input relations are free of null tuples and duplicates.*
(5) *All outerjoins are binary*—i.e., outerjoin predicates reference attributes of exactly two base relations.

Restriction 2 is present because some associative identities required that certain predicates reject nulls. While we expect that predicates for most practical queries will indeed reject nulls,[4] queries that coalesce values from multiple sources into a single column (e.g., natural outerjoin, [Date 1986]) and then compute predicates on the coalesced column will violate the assumption. Restriction 3 arises because we would need to represent Cartesian product as join with a TRUE predicate, which does not reject nulls. Even with violations of these two conditions, reorderings may be possible, but we are unsure whether the benefits will repay the extra complexity, and do not investigate further in this article.

Restriction 4 is normally satisfied by base relations, but might not be satisfied if the join-outerjoin query receives input from some other query. The problem can be solved by extending tuples by adding (and later discarding) a system-generated unique identifier. Then our results can be applied.

Restriction 5 also comes from the known reordering identities. Take, for example, identity (6), and modify the outerjoin predicate on the left-hand side to be nonbinary, e.g., $(R_1 \overset{p^{12}}{\bowtie} R_2) \xrightarrow{p^{13} \wedge p^{23}} R_3$. Reassociating naively we would obtain $R_1 \overset{p^{12} \wedge p^{13}}{\bowtie} (R_2 \xrightarrow{p^{23}} R_3)$, which simplifies—with the usual null-rejection assumption—to $R_1 \overset{p^{12} \wedge p^{13}}{\bowtie} (R_2 \overset{p^{23}}{\bowtie} R_3)$, and is not equivalent to the original expression.

It would be desirable to remove the restrictions, both to remove loose ends and to improve performance. It is difficult for us to judge the kinds of queries that application development environments will produce, but we strongly suspect that most queries will satisfy our restrictions. Some progress has already been made. Recently, Bhargava et al. [1995] adapted our framework to deal with nary outerjoin predicates. Instead of considering all evaluation orders consistent with the connectivity heuristic, they identify a subset of topologies such that reorderings require only join, outerjoin, and generalized outerjoin. They also discuss the issue of duplicates in base relations. Although their approach is basically that of extending tuples with an extra column, the details are somewhat more complicated.

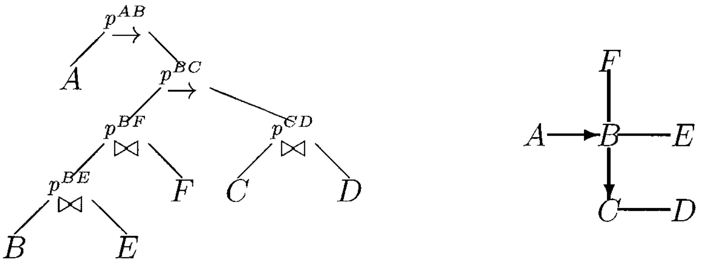
3.2 Extending Query Graphs for Simple Join/Outerjoin Queries

In Section 2 we assumed queries were represented as fully parenthesized *operator trees* (as SQL2 does for outerjoins), which define a unique result. As in most optimizers, it is convenient to show connections by deriving a *query graph* consisting of base relations as nodes, plus edges for predicates. The difficulty with outerjoins is that, unlike joins, a query graph without information on evaluation order is ambiguous, i.e., no longer denotes a unique result. We first extend the common query graph representation for outerjoins, and then show how to overcome the ambiguity problem for simple queries.

In the query graph of a simple join/outerjoin query, join conjuncts are represented, as usual, by undirected edges; one-sided outerjoin predicates

---

[4] See Galindo-Legaria [1994] for "intuitive specifications" of join/outerjoin queries, which are valid only when predicates reject nulls.



(a) Operator tree. (b) Query graph.

Fig. 5. A simple query and its graph.

are represented by directed edges, pointing towards the relation on whose scheme nulls are introduced; and full outerjoin predicates are represented by bidirected edges.

For simplicity, we consider only join conjuncts that are binary. Conjuncts referencing more than two relations can be handled with hyperedges [Ullman 1982] and do not affect our results.

Formally, for an operator tree Q, a *query graph* $G = (V, E) = \text{graph}(Q)$ is constructed as follows:

—The set of nodes $V = \text{leaves}(Q)$.

—Labeled edges correspond to predicate conjuncts. Take a subtree $(Q_j \overset{p}{\odot} Q_k)$, where $p = p_1^{R_{j1}R_{k1}} \wedge \cdots \wedge p_n^{R_{jn}R_{kn}}$, and R_{ji} (respectively R_{ki}) is a leaf of Q_j (respectively Q_k). For each $p_i^{R_{ji}R_{ki}}$ there is an edge $(R_{ji}, R_{ki}) \in E$ labeled by p_i. Also,

if the operator is join, i.e., $\odot = \bowtie$, then the edge is undirected $(R_{j_i} \text{—} R_{k_i})$;

if one-sided outerjoin, i.e., $\odot = \rightarrow$, then the edge is directed $(R_{j_i} \rightarrow R_{k_i})$;

if full outerjoin, i.e., $\odot = \leftrightarrow$, then the edge is bidirected $(R_{j_i} \leftrightarrow R_{k_i})$.

Example 6. Figure 5 shows the query graph of the simplified query of Example 3. Note that the query graph does not indicate, for example, whether outerjoin $\xrightarrow{p^{AB}}$ should be evaluated before or after join $\overset{p^{BE}}{\bowtie}$. And different evaluation orders produce different results.

Query graphs abstract away all information about order of evaluation. The concept of *free-reorderability* was first introduced in Rosenthal and Galindo-Legaria [1990] to replace the loosely-used term "associativity" in the context of combine operators. Although join is conventionally said to be associative, the assertion is not strictly true. Observe that when we change the order of evaluation of a query $((R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{13} \wedge p^{23}}{\bowtie} R_3)$ to obtain $(R_1 \overset{p^{12} \wedge p^{13}}{\bowtie} (R_2 \overset{p^{23}}{\bowtie} R_3))$, predicate conjuncts move between operators, thus, in a sense, modifying the operators themselves. We call a query Q *freely-reorderable* if any other query with the same query graph evaluates to the same result, i.e., the information on order of evaluation in Q is irrelevant for its semantics. We say that joins are freely-reorderable (rather than associative) because queries containing only joins are freely-reorderable, in the sense defined. Outerjoins are not, in general, freely reorderable.

3.3 Completeness of Identities for Simple Queries

Monotonic Queries. We order the three generic operators based on their *restrictiveness*, defined as the number of operands for which unmatched tuples are discarded (i.e., the nonarrow end of the symbol). Join is most restrictive, then one-sided outerjoin, then full outerjoin. We call a query *monotonic* if no parent is more restrictive than its child. That is, all joins are executed first, then one-sided outerjoins, and finally full outerjoins. For example, the query in Figure 5a is monotonic.

Examination of our associative identities shows that in a simple query, any operator can be pushed down past other operators that are equally or less restrictive, without introducing generalized outerjoin. Such reordering is the basis of the following theorem, proved in the appendix.

THEOREM 1. *Let Q be a simple query. Any monotonic query Q' such that graph(Q) = graph(Q') is in Q^I.*

In other words, if we are told that a query graph was obtained from a simple query, then we can compute the result of the original query by evaluating all joins in the query graph first, in any order, then one-sided outerjoins, in any order, and finally full outerjoins. This evaluation rule can help users formulate and understand queries. Of course the optimizer is free to (and often will) select a different evaluation order for actual execution.

Evaluations Consistent with the Connectivity Heuristic. The well-known connectivity heuristic restricts the search space of optimizers, avoiding Cartesian products in favor of joins [Ono and Lohman 1990; Selinger et al. 1979; Tay 1990]. Given a query graph $G = (V, E)$, an operator tree T satisfies the connectivity heuristic if for every subtree T', leaves(T') induces a connected subgraph of G.[5] Topologies that satisfy the connectivity heuristic (i.e., binary trees providing an order of evaluation but not the operator to use at each step) are called here *association trees*. assoc(G) denotes *association trees of G*. The following theorem, proved in the appendix, states the *completeness* of our set of identities.

THEOREM 2. *Let Q be a simple query. For any association tree T in assoc(graph(Q)), there is a query Q' in Q^I whose topology matches T.*

In other words, starting with a simple query, our associative identities can generate every order of evaluation consistent with the connectivity heuristic.

---

[5] Given a graph $G = (V, E)$ and a set of nodes $V' \subseteq V$, we denote the *induced subgraph* as $G|_{V'} = (V', E')$, where $E' = \{(u, v) | (u, v) \in E, u \in V', v \in V'\}$.

4. BOTTOM-UP ENUMERATION OF OPERATOR TREES

In this section we describe how to generate evaluation orders bottom-up, for simple join/outerjoin queries. First we present a general form of the algorithm to construct operator trees bottom-up. Then, based on an analysis of query graphs and reordering identities, we present a decision table to choose the operator to use at each step of the bottom-up tree construction.

4.1 Algorithm to Construct Operator Trees

Conventional optimizers in the style of System R [Selinger et al. 1979] enumerate operator trees directly, as alternative strategies for evaluating a query. Trees are built incrementally, bottom-up, from the query graph. Each step combines two subtrees, using a join node as the new root. Dynamic programming is used to prune the space of alternatives. The algorithm for enumerating *all association trees* of a query graph takes the following general form:[6]

```
Algorithm B. Enumeration of operator trees.
  Input. A query graph $G$ with $n$ relations.
  Output. A set of operator trees for $G$.
  Procedure.
    B-1. For each node in $G$ create a 1-leaf tree.
    B-2. For $k = 2, \ldots, n$: Choose $T_l$, $T_r$ such that
      leaves($T_l$) $\cap$ leaves($T_r$) $= \emptyset$.
      $|leaves(T_l)| + |leaves(T_r)| = k$.
      $G|_{\text{leaves}(T_l) \cup \text{leaves}(T_r)}$ is connected.
      B-2.1. Create a tree $T_s = (T_l \odot T_r)$.
    B-3. Output trees with $n$ leaves.
```

Some optimizers impose restrictions on the trees to be generated; these can be added straightforwardly to tree generation in our algorithm. For example, for "left-leaning trees," T_r in step B-2.1 is always a one-leaf tree. Following the connectivity heuristic, most optimizers insist that every join involve an edge in G. In our proofs of join/outerjoin reordering, connectivity is unfortunately a required assumption, rather than just a search-pruning mechanism.

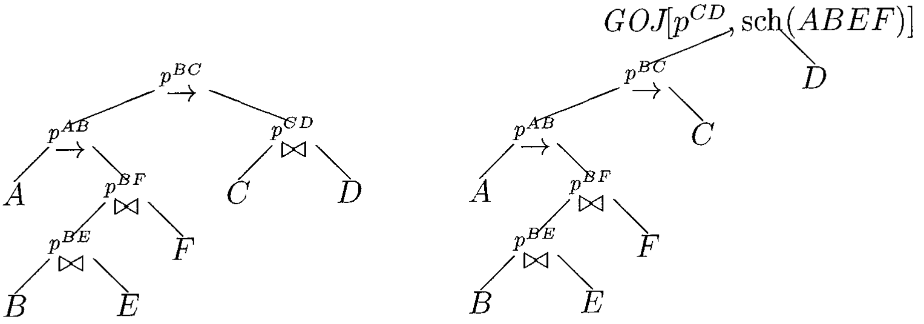
Step B-2.1 has to determine the operator to use to combine subtrees T_l, T_r, in every step. When the original query uses only join to combine relations, then the operator to use at each step is join. But when the original query contains both joins and outerjoins, it is not obvious which operator should be used at each step—join, outerjoin, or generalized outerjoin?

4.2 How Does GOJ Appear During Transformations?

To obtain an arbitrary reordering of a simple query, we can start with a convenient monotonic evaluation, and then apply transformations to move

---

[6] Actually, the implementation of the algorithm would produce a graph that embeds all the trees, rather than a set of trees.



(a) Starting monotonic tree. (b) New root, GOJ.

Fig. 6. Moving an operator to the root.

up an operator until it reaches the root of the tree. As a result, the new root may become a GOJ, but it turns out its subtrees always remain simple queries. This process of placing a new root can be continued recursively on the subtrees to obtain a desired topology.

Generalized outerjoin is introduced when an operator is moved above past a less restrictive parent, in the cases shown in identities (11), (12) and (13). We now illustrate the process of moving an operator up to the root, and show how GOJ is introduced.

Example 7. Take the query shown with its query graph in Figure 5, and consider an evaluation order in which join predicate p^{CD} is applied last. Such evaluation is achieved by moving operator $\overset{p^{CD}}{\bowtie}$ to the root.

In the query of Figure 5a, there are two operator operators above $\overset{p^{CD}}{\bowtie}$. A more convenient starting monotonic query to move up $\overset{p^{CD}}{\bowtie}$ is that shown in Figure 6a, where the join of interest has only one ancestor. Application of identity (11) obtains the desired root, as shown in Figure 6b.

Now, suppose we want to move operator $\overset{p^{BF}}{\bowtie}$ to the root. It has two ancestors in the query of Figure 5a as well as that of Figure 6a. But the order of the ancestors is different. The query in Figure 5a is more convenient, because the $\overset{p^{BF}}{\bowtie}$ can be moved past its first ancestor without introducing GOJ, using identity (6); then, a second transformation places the desired new root, this time introducing GOJ.

The critical information about the convenient monotonic query and the attributes preserved by the resulting GOJ at the root can be detected directly from the query graph.

Example 8. Note that maximal join subtrees in monotonic queries correspond to maximal connected subgraphs with undirected edges only. In the query graph in Figure 5b, there is a maximal connected, undirected subgraph that contains edge C—D. There is one directed edge, $B \rightarrow C$, adjacent to a node in such subgraph. This implies that outerjoin $\xrightarrow{p^{BC}}$ is an ancestor of join $\overset{p_{CD}}{\bowtie}$ in every monotonic operator tree for this query. In addition the directed edge points towards the undirected subgraph. Then, moving the join $\overset{p_{CD}}{\bowtie}$ up past the outerjoin $\xrightarrow{p_{BC}}$ requires the introduction of GOJ. Finally, the attributes preserved by the new GOJ must be those preserved originally by the outerjoin, i.e., in the query graph, those of relations on the other side of $B \to C$, away from C—D. Such GOJ is the root of the operator tree in Figure 6b.

For the edge B—F, there are two directed edges adjacent to the maximal connected, undirected component, and therefore the two corresponding outerjoins are ancestors of join $\overset{p_{BF}}{\bowtie}$ in every monotonic evaluation. Now, only one of the two edges, $A \to B$, points towards the maximal undirected subgraph. When the join is moved up past the outerjoins that corresponds to that edge, a GOJ is introduced. The preserved attributes are again those on the other side of $A \to B$, away from B—F.

4.3 Query Graph Analysis

The intuition provided in the above examples is now formalized and generalized. We use the following terms and notation. Let $G = (V, E)$ be a query graph. For an edge $e \in E$, the removal of e from G is abbreviated as $G - e = (V, E - \{e\})$. A path in G is a sequence of edges, per the usual definition, which can be traversed in either direction, and does not include the same edge more than once. A directed edge in a path appears either as "→", if traversed in the direction of the arrow, or as "←", if traversed opposite the arrow. Graph connectivity ignores the direction in which edges are traversed.

LEMMA 3. *Let G be the query graph of a simple query. Assume e is a directed or bidirected edge in G. Then $G - e$ is disconnected (and it has two connected components).*

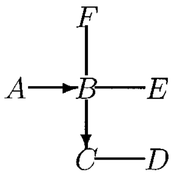
Observation 4. Let G be the query graph of a simple query, and e_1, e_2 be any two edges of G. Then any two paths from e_1 to e_2 include the same set of directed and bidirected edges, traversed in the same direction (including e_1, e_2). That is, only the undirected edges may differ.

Observation 4 follows from Lemma 3. Next, define a restrictiveness ordering for edges: Undirected edges are *more restrictive* than directed edges, which are in turn *more restrictive* than bidirected edges. Now, consider a path $P = e_M \cdots e_L$, where e_M is more restrictive than e_L. We say e_M *conflicts with* e_L, and call P a *conflict path*, if the following holds (with arrow directions exactly as shown):

—e_L is either $\leftarrow$ or $\leftrightarrow$; and
—all other edges in P are either — or $\to$.

The set of conflicting edges for a given edge are

$$\mathrm{conf}(e_M) = \{e_L | e_M \cdots e_L \text{ is a conflict path}\}.$$



$$\begin{aligned} \mathrm{conf}(B - E) &= \{(A \rightarrow B)\} \\ \mathrm{conf}(B - F) &= \{(A \rightarrow B)\} \\ \mathrm{conf}(C - D) &= \{(B \rightarrow C)\} \\ \mathrm{away}_{B-E}(A \rightarrow B) &= \{a\} \\ \mathrm{away}_{B-F}(A \rightarrow B) &= \{a\} \\ \mathrm{away}_{C-D}(B \rightarrow C) &= \{a, b, e, f\} \end{aligned}$$

(a) Query graph. (b) Result of graph analysis.

Fig. 7. A graph and its analysis.

Also define

> $\mathrm{away}_{e_M}(e_L)$ = {attributes of relations in the connected component of $G - e_L$ that does not contain e_M}.

In a monotonic evaluation, with the operator for e_L done last, all attributes in $\mathrm{away}_{e_M}(e_L)$ are preserved. If e'_L, e''_L, . . . , are also in conf(e_M), their away sets also need to be preserved when the operator of e_M moves upward. That is, every edge in conf(e_M) whose operator is moved below that of $\odot_M$ adds a set of attributes to the GOJ's preservation list.

Example 9. Figure 7 shows the result of analyzing the query graph of the monotonic query of Figure 5, to obtain both conf( ) and away( ) information. We assume that each relation A through F has a single attribute, whose name is the lower case letter of the relation name.

To compute conflict sets, it is useful to note that, by Observation 5, if $e_M \cdots e_L$ is a conflict path, then every path from e_M to e_L is a conflict path. One can compute conf(e_M) and each $\mathrm{away}_{e_M}(e_L)$ straightforwardly by depth first traversal from e_M. Devising an efficient way to mass produce all conflict sets and away sets is left as an open problem.

4.4 Choosing an Operator in the Bottom-Up Construction

To generate arbitrary reorderings of simple queries, consistent with the connectivity heuristic, Algorithm B is modified to take advantage of the information provided by the query graph analysis. The needed modification to step B-2.1 is shown in Figure 8. The idea is that when Algorithm B constructs association trees for the graph G of a simple query, each time a graph edge is used in combining two existing subtrees, we check if conflicting, less restrictive edges have already been used in the construction of the subtrees. In that case, the subtrees must be combined with GOJ, and the attributes to preserve are determined by the conflicting edges used in the subtrees.

In Figure 8, line 5 determines the conflicting, less restrictive edges $\{e_1, \ldots, e_n\}$ that have already been used in the construction of the subtrees T_l, T_r. It is sufficient to use conf(e_0), for any $e_0 \in E'$, chosen in

```
1. Let $T_l, T_r$ be subtrees selected in an iteration of step B-2.
2. Let $E'$ be the edges in $G$ between leaves($T_l$) and leaves($T_r$).
3. Let $e_0 \in E'$. Let $p$ be the conjunction of predicate labels in $E'$.
4. Let $S = \text{leaves}(T_l) \cup \text{leaves}(T_r)$. Let $A_s = \{\text{attributes of relations in } S\}$.
5. Let $\{e_1, \ldots, e_n\} = \{e_L \mid e_L \in \text{conf}(e_0) \text{ and } e_L \text{ is between two nodes in } S\}$.

6.1. If $e_0$ is bidirected, create $T_s = T_l \overset{p}{\leftrightarrow} T_r$.
6.2. If $e_0$ is directed, assume wlog $e_0 = R_l \rightarrow R_r$, $R_l \in \text{leaves}(T_l)$.
     If $n = 0$, create $T_s = T_l \overset{p}{\rightarrow} T_r$;
     otherwise, let $A_l = \{\text{attributes of relations in leaves}(T_l)\}$, and create
         $T_s = T_l \; GOJ[p, A_l, \text{away}_{e_0}(e_1) \cap A_s, \ldots, \text{away}_{e_0}(e_n) \cap A_s] \; T_r$.
6.3. If $e_0$ is undirected,
     if $n = 0$ create $T_s = T_l \overset{p}{\bowtie} T_r$;
     otherwise, create
         $T_s = T_l \; GOJ[p, \text{away}_{e_0}(e_1) \cap A_s, \ldots, \text{away}_{e_0}(e_n) \cap A_s] \; T_r$.
```

Fig. 8. New step B-2.1, to reorder simple join/outerjoin queries.

line 3. Observe that if E' is not a singleton then the edges in E' are all part of some cycle in G, because leaves(T_l) and leaves(T_r) each induces a connected subgraph of G. Then, by Lemma 4 all edges in E' are undirected. From the definition of conflicting edges, any two edges e_1, e_2 in a cycle have conf(e_1) = conf(e_2), so, all edges in E' have the same set of conflicting edges.

For correctness, the clauses for $n = 0$ (i.e., no conflicting, less restrictive edges used earlier) are not necessary, as they are a particular case of the general GOJ expressions. We include these clauses explicitly because an optimizer should recognize these cases and issue simpler operators than the general GOJ.

The bottom-up construction we have outlined is based on the application of associative identities of Section 2, as described in the next theorem.

THEOREM 5. *Let G be the graph of a simple query Q. Let Q' be an operator tree generated by Algorithm B using the rule of Figure 8. Then Q' is in Q^I.*

Example 10. The bottom-up construction of an operator tree for the query graph of Figure 7a is illustrated in Figure 9. The example constructs a single operator tree (greedily); an optimizer doing join enumeration would actually generate many alternatives. To illustrate the gains from reordering, we qualitatively estimate sizes of intermediate results, under the following assumptions: Relations A and D are qualitatively **small** compared to the other relations, and join predicates are equalities on keys.

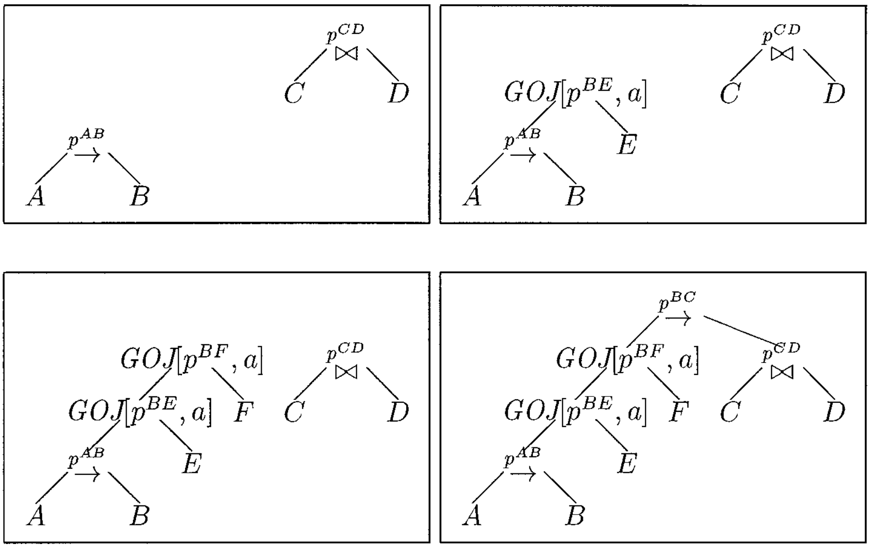


Fig. 9. Bottom-up construction of an operator tree.

First, we build subtrees with two leaves that combine the **small** relations A and D. Results of these two trees are **small**. Then, combine the A-B subtree with the one-leaf E subtree. The edge between subgraphs $\{A, B\}$ and $\{E\}$ is undirected, $e_M = (B—E)$, so Step 6.3 will determine the operator for the corresponding node in the operator tree. The only conflicting edge for $(B—E)$ is $(A \rightarrow B)$, which connects relations within a subtree, so its operator has been applied earlier. Therefore the operator to use is $GOJ[p^{BE}, \text{away}_{(B—E)}(A \rightarrow B) \cap \{a, b, e\}]$. Since the attributes preserved by GOJ belong to the **small** relation, the GOJ result is **small**.

Now, combine the A-B-E subtree with the F subtree. In this case, $e_M = (B—F)$, and again its conflicting edge $\text{conf}(e_M) = \{(A \rightarrow B)\}$ connects relations within a subtree. The operator to use is $GOJ[p^{BE}, \text{away}_{(B—F)}(A \rightarrow B) \cap \{a, b, e, f\}]$. The result of the new subtree is **small**.

Finally, combine subtrees A-B-E-F and C-D. Now $e_M = (B \rightarrow C)$ and $\text{conf}(e_M) = \emptyset$. The operator to combine the two trees is outerjoin. All intermediate results are **small** in the operator tree we have built.

5. SPECIFYING AND OPTIMIZING OUTERJOINS IN DATABASE SYSTEMS

The previous sections addressed the difficult, detailed technical problems. This section exploits their results. First, we describe how to apply our results of Sections 2 and 4 within the architecture of current database optimizers. Section 5.1 applies our results to rule-based optimizers, and Section 5.3 considers more conventional enumerative approaches. Then, in Section 5.5 we describe particular applications and restricted query languages that are particularly easy to reorder. Under such restrictions, both query language and optimization are simpler than the general case.

5.1 Rule-Based Optimizers

Rule-based query optimizers have been proposed as configurable modules that explore a space of execution plans, based on rules for transformation of expressions and cost estimation [Graefe and DeWitt 1987; Galindo-Legaria 1987; Graefe 1990; Haas et al. 1990; Rosenthal and Helman 1986]. Our processing tactics of Section 2—outerjoin simplification and associative identities—provide the necessary transformation rules to handle outerjoins.

Simplification identities are always beneficial, and we wish to apply them before exploring reassociations. Associative identities may be applied in either direction, and cost estimation is necessary to compare the alternatives. Most rule-based optimizers should accept such advice.

The completeness result of Theorem 2 guarantees that if predicates of the original query satisfy the conditions of Section 3.1, then our transformation rules can generate the complete space of evaluation orders consistent with the connectivity heuristic. That is, all reorderings of the query are within the reach of the optimizer.

5.2 Selections and Projections

Previous sections assumed that queries contained only joins and outerjoins. We now briefly examine projections and arbitrary selections, which are treated roughly as in join queries. The adaptation to join-enumeration optimizers also appears relatively similar to the conventional treatment.

Projections are straightforward. As in join queries, they can be either pushed down the operator tree to the earliest possible time, or done at the end.

Selections are used to remove arrows, as in Algorithm A, then pushed down past joins, and past outerjoins using identities (3) and (4). Selection predicates that reject nulls are eventually pushed into either base relations or match predicates. Then our join/outerjoin reordering technique can be applied.

Unfortunately, if selection predicates do not reject nulls, there is no guarantee they will be absorbed into base relations or match predicates. For example, take an outerjoin that introduces nulls on sch(R_1). A later selection that references columns in sch(R_1) but does not reject nulls can be pushed neither into the outerjoin match predicate, nor below the outerjoin. Pending further research, nonnull-rejecting predicates, in selects as well as in combine operators, force us to split the query into separately-optimized pieces.

5.3 Join-Enumeration Optimizers

This section describes a general structure for a join-enumeration optimizer that handles many outerjoin queries. It also indicates ways to reduce implementation effort by generating fewer reorderings.

The generation of evaluation orders in enumerative optimizers is abstracted by Algorithm B of Section 3.3. Section 4.4 refined the algorithm to handle simple join/outerjoin queries. This algorithm is an ingredient in a larger process that handles queries (or subqueries) that may not be simple but do satisfy the restrictions of binary and null-rejecting predicates of Section 3.1. In this case, the optimizer must perform the following steps:

—First, simplify the original query Q_1 using Algorithm A of Section 2.1—and any other simplification algorithms available—to obtain Q_2.

—Next, construct the query graph G of Q_2, and compute edge conflicts and away sets for each edge—functions conf( ) and away( ) defined in Section 4.3.

—Finally, apply Algorithm B on G to enumerate trees, using the logic of Figure 8 to choose the root operator of each subtree created in step B-2.

5.4 Generating Restricted Reorderings

To reduce the implementation effort in the execution engine, one might choose to implement a restricted set of GOJ operators. This section discusses the apparently modest importance of preserving multiple sets, and then considers how enumeration can be modified not to use the general case of GOJ.

Preserving multiple sets is necessary for completeness, to reorder full outerjoins. As with Cartesian product [Ono and Lohman 1990], there are special cases where an early placement is somewhat beneficial. However, the benefit from early evaluation of full outerjoins seems much smaller and less frequent than that of one-sided outerjoins. For example, consider $R_1 \overset{p^{12}}{\longleftrightarrow} (R_2 \overset{p^{23}}{\longleftarrow} R_3)$, where R_1 and R_2 are tiny, but R_3 is huge. Early evaluation of full outerjoin, using identity (13), can yield an improved strategy because a huge intermediate result is avoided. But since full outerjoin preserves all tuples in both inputs, the result of the query is huge, and the impact of evaluating full outerjoin early seems marginal. Contrast this case with that of Example 5, where early evaluation of one-sided outerjoin can produce a dramatic performance improvement.

Delaying evaluation of full outerjoins guarantees that any GOJ in the reordered query preserves only one set of attributes, thus excluding the more complicated, general case of GOJ. The following lemma is proved in the appendix.

LEMMA 6. *Let Q be an operator tree generated by Algorithm B using Figure 8 to choose operators. Assume ancestors of full outerjoins in Q are all full outerjoins. Then operators in Q may be join, one-sided outerjoin, full outerjoin, and GOJ preserving only one set of attributes.*

Lines 6.2 and 6.3 of Figure 8 can easily be modified so the tree enumeration algorithm refuses to combine subtrees that require a certain operator. It is possible to improve on this, modifying the algorithm to look ahead and refrain from creating subtrees that force the inclusion of undesired operators later on. To avoid general GOJ, a lookahead can be accomplished by modifying the rule in Figure 8 so that no bidirected edge is used to combine subtrees, unless those subtrees have already used all directed and undirected edges in the graph.

The algorithm can be modified similarly to avoid generating trees that use any form of GOJ, based on conflicting edge information conf( ). The scheme to generate restricted reorderings allows the optimizer to be easily upgraded by modifying (or discarding) lookahead code, to generate more reorderings when new operators are implemented in the execution engine.

5.5 Important Special Cases

We now describe restricted classes of join/outerjoin queries whose properties allow easier query languages and optimization algorithms. The motivation is that queries that do not specify an evaluation order are much easier to write than fully-parenthesized queries. As an example, compare the relative simplicity of an SQL "select-from-where" block versus a corresponding operator tree, written in whatever syntax.

"Simple" Queries. Users could provide a query graph rather than an operator tree, with the proviso that the graph be interpreted as a "simple" query—recall from Section 3.3 that a query graph with no information about processing order is an adequate specification of any simple query.

Users may think of a monotonic order of evaluation as performing first all joins, then one-sided outerjoins, and finally full outerjoins, and simply specify the operators of each type. Clearly, the optimizer is free to choose any other order of evaluation following the approach we have shown.

Hierarchic Queries. Hierarchic queries are used to collect parent-child information, preserving parents with no children. This pattern seems to be a frequent use of one-sided outerjoin in current SQL implementations [David 1991]. Examples 1 and 2 in the introduction show such queries, with New York customers as "parents" of a set of "children" orders.

From the query optimization perspective, hierarchic queries have two relevant properties: They have no full outerjoins, and they are simplified—i.e., their topology does not allow arrow removals by Algorithm **A**. Exploiting these properties, our optimization strategy becomes simpler in three ways:

—There is no need to apply Algorithm **A** for simplification.

—Query graph analysis is easier because there are no bidirected edges (and undirected edges conflict with at most one directed edge).

—No reordering requires GOJs that preserve more than one set of attributes.

Also, since hierarchic queries are simplified, they can be specified using only a query graph. David [1991] uses a graph as a conceptual visualization for parenthesized SQL2 outerjoins. Such graph does not specify an evaluation order, although it does suggest a monotonic evaluation.

Conflict-Free Queries. Queries whose graphs have no conflicting edges are easier to handle than hierarchic queries. The conflict sets conf( )

are always empty, so there is no need to compute and check these sets, and reordered queries will never require generalized outerjoin. Hence graphs are unambiguous query specifications, and there is no need to analyze the graph. A trivial class of conflict-free queries are those that combine relations using only joins.

A conflict-free class of join/outerjoin queries is presented in Rosenthal and Galindo-Legaria [1990]. This class is characterized by a query graph whose topology consists of a subgraph of undirected edges, from which directed trees grow outward. This class can handle an extended form of Select-From-Where blocks, for a data model with entity- and set-valued attributes. Attribute dereference and unresting is specified in the Select clause, and represented algebraically by means of outerjoins.

Another conflict-free class, which may arise when databases are merged, is characterized by query graphs consisting of a subgraph of bidirected edges from which directed trees grow outward.

From the definition of conflicting edges, simple queries that contain both joins and full outerjoins always have conflicts, regardless of the topology of their query graph.

6. CONCLUSIONS

Outerjoins are an important SQL2 operation, but are handled poorly by current optimizers. In this article, we provided a theory that allows join/outerjoin queries to be reordered so that inexpensive combinations can be performed before expensive ones. Equally important, we described how the theory can be adapted to fit within existing optimizers, both conventional—in the enumerative style of System R—and rule-based. The results were presented modularly, so that system designers are free to choose the techniques that are suitable for their environment, and to build incrementally.

We first presented a set of identities for simplifying queries composed of one- and two-sided outerjoins, joins, and selections. These rules can readily be implemented in a query preprocessor, so federated database systems that merge information using full outerjoin can benefit easily from this technique.

We next presented a set of algebraic identities that justify reassociation of pairs of joins and outerjoins. This set of identities was shown to be complete for *simple* queries (defined in Section 3.1). For those queries, our identities can generate any evaluation order consistent with the connectivity heuristic. We also described classes of queries where reassociations use solely the operators in the original query, so there was no need to introduce generalized outerjoin. For those queries, the result is independent of the query's association—i.e., parenthesization—so the query language syntax and query optimization algorithms can be simplified.

A major practical contribution of this article is the query enumeration algorithm to generate candidate evaluation orders for joins and outerjoins. We expect that the join enumeration component of traditional optimizers can be extended, without major redesign, to use the logic for operator selection we have described. The result will be an optimizer that generates good strategies for most outerjoin queries.

Further research is needed on how to relax the restrictions we have assumed (see Section 3.1) while preserving complete reorderability freedom, and on how to exploit the limited freedom allowed in more general classes. As mentioned in Section 3.1, some steps in this direction have been taken recently by Bhargava et al. [1995]. A problem we did not consider is how to choose operators when the bottom-up enumeration does not follow the connectivity heuristic, so that Cartesian products are required. Finally, more work is necessary on the efficient implementation of query simplification, query graph analysis, and generalized outerjoins.

We continue our work on declarative outerjoin specifications, to simplify reasoning about outerjoins, and to explore its use in logic languages (see Galindo-Legaria [1994]). Another unexplored and, we believe, promising research direction is the application of our framework of operator tree / query graph / association tree to reorder other operators that combine relations, such as semijoin and antijoin.

A. PROOFS OF LEMMAS AND THEOREMS

To prove the results presented in the body of the article we rely on a sequence of claims, numbered independently from the main lemmas and theorems. The terms “associative identities” and “associative transformations” refer to the identities in Section 2. More complete but somewhat less intuitive proofs can be found in Galindo-Legaria [1992].

In both Theorem 1 and Theorem 5, we seek to transform a given operator tree to a more desirable topology. In both cases, our strategy is to place the new root first, and then (recursively) to transform the subtrees. Transformation steps are supported by associative identities. Assume the goal topology is of the form $(T_l \odot T_r)$. Then placing the correct root in an operator tree implies sorting the leaves in two groups, left and right subtree, so they coincide with the leaves of T_l, T_r, respectively. In every case, the subtrees will remain simple queries, so we can apply the procedure recursively to transform them to match T_l, T_r.

Claim 1. Simple queries are closed under associative transformations.

PROOF SKETCH. For property 1, if a query has been simplified by Algorithm **A**, then reassociating its operators does not lead to further simplifications. Properties 2 through 5 of simple queries (see Section 3.1) are trivially invariant under associative transformations. □

PROOF SKETCH OF THEOREM 1. We show how to transform simple query Q into monotonic query $Q' = Q'_l \odot Q'_r$. If Q has only one operator, then, after commuting the operands, if necessary, Q is identical to Q'. Otherwise, move up operator $\odot$ in Q to the root, then subtrees can be transformed into Q'_l and Q'_r. To move this operator upward, observe that in simple queries, the associative identities always permit less restrictive operators to be moved up past equal or more restrictive operators. For example, if the ancestor of a full outerjoin is a one-sided outerjoin but identity (10) is not applicable, then we either have a predicate that does not reject nulls, or else a simplification by Algorithm **A** is possible—and Q would not be simple. □

The proof of Theorem 2 is presented after Theorem 5.

Claim 2. Let G be the query graph of a monotonic, simple query Q. All cycles in G consist of undirected edges.

PROOF. By induction on the structure of monotonic operator trees.

Base. For 1-node trees the lemma is trivially true.

Induction. Assume $Q = (Q_l \overset{p}{\odot} Q_r)$. Then G is obtained by adding edge(s) corresponding to p to the union of $G_l = \text{graph}(Q_l)$, $G_r = \text{graph}(Q_r)$. If $\odot$ is join then because Q is monotonic, all operators in Q_l, Q_r are joins. Thus all edges in G are undirected, including all cycles. Otherwise, if $\odot$ is outerjoin, then p is a binary predicate, so only one edge connecting G_l with Q_r is added. All cycles in G are already present in G_l or G_r and, by induction hypothesis, they consist of undirected edges. □

PROOF OF LEMMA 3. There is a monotonic query Q with graph G. Then, by Claim 2, e is not part of a cycle in G, and therefore $G - e$ is disconnected. Since G is connected, $G - e$ has two connected components. □

Claim 3. Let G be the query graph of a simple query, and e_M be an edge in G. In any monotonic query Q for G, operators associated to conf(e_M) are ancestors of the operator associated to e_M.

PROOF SKETCH. Suppose e_M is a directed edge. Because Q is monotonic and obeys the connectivity heuristic, a maximal subtree of Q having only joins and one-sided outerjoins corresponds to a maximal connected subgraph of G having only directed and undirected edges. Call Q_M the maximal join/one-sided outerjoin subtree of Q that includes the operator for e_M. Edges in conf(e_M) are all bidirected, and each points to a node in the maximal such subgraph that contains e_M. So, full outerjoins corresponding to edges in conf(e_M) must be ancestors of a relation in subtree Q_M. Since there are no full outerjoins in Q_M, they must be ancestors of the root of Q_M, and therefore of every operator in Q_M including the operator of e_M.

When e_M is an undirected edge, the situation is analogous. □

Claim 4. Let G be the query graph of a simple, monotonic query Q, and e_M be an undirected edge in G. Then conf(e_M) contains at most one directed edge.

PROOF SKETCH. Suppose there are two different directed edges in conf(e_M). By Claim 3, there are two different one-sided outerjoins ancestors of some join operator. But one of those one-sided outerjoins introduces null on some attributes, say A, while the other one-sided outerjoin, higher up, rejects nulls on those attributes. Simplification by Algorithm **A** is possible, contradicting the assumption that Q is simple.

Claim 5. Let G be the query graph of a simple, monotonic query Q. Assume $e_0 = R_i \overset{p}{\longleftrightarrow} R_j$ is a bidirected edge whose removal from G leaves connected components G_l, G_r. Using associative identities, Q can be transformed into $Q_l \overset{p}{\longleftrightarrow} Q_r$ such that Q_l, Q_r are simple, graph(Q_l) = G_l and graph(Q_r) = G_r.

PROOF. Pick any monotonic tree Q' that evaluates outerjoin $\overset{p^{R_1 R_j}}{\longleftrightarrow}$ last. By Theorem 1, Q can be transformed into Q'. $\square$

Claim 6. Let G be the query graph of a simple, monotonic query Q. Assume $e_0 = R_i \overset{p}{\longrightarrow} R_j$ is a directed edge whose removal from G leaves connected components G_l, G_r, with R_i in G_l. Let A_l be the union of attribute sets of all relations in G_l. Using associative identities, Q can be transformed into Q_l $GOJ[p, A_l, \text{pres}_{e_0}(e_1), \ldots, \text{pres}_{e_0}(e_n)]$ Q_r, where $\{e_1, \ldots, e_n\}$ = conf(e_0). Q_l, Q_r are simple, graph(Q_l) = G_l and graph(Q_r) = G_r.

PROOF SKETCH. First, note that G has a monotonic query Q_1 such that: (1) Outerjoin $\overset{p^{R_i R_j}}{\longrightarrow}$ is the root of a maximal subtree of one-sided outerjoins and joins. (2) Every ancestor of $\overset{p^{R_i R_j}}{\longrightarrow}$, say there are m, is a full outerjoin whose predicate references some relation below $\overset{p^{R_i R_j}}{\longrightarrow}$. That is, full outerjoins unrelated to e_0 have been moved aside, to be performed independent of this subtree). (3) Full outerjoin ancestors are ordered so that those corresponding to edges in conf(e_0) appear last in the tree. By Theorem 1, Q can be transformed into query Q_1. Now the first $m - n$ ancestors of $\overset{p^{R_i R_j}}{\longrightarrow}$ reference relations in the preserved subtree of the one-sided outerjoin, so $\overset{p^{R_i R_j}}{\longrightarrow}$ can be moved up using identity (10) without introducing GOJ. Then, move the operator up to the root using identity (13) once and (14) $n - 1$ times. The resulting query is that stated in the claim.

Claim 7. Let G be the query graph of a simple, monotonic query Q. Assume E' is a minimal set of undirected edges whose removal from G leaves connected components G_l, G_r. Let $p_1, \ldots, p_k$ be the predicates labeling edges in E' and $p = p_1 \wedge \cdots \wedge p_k$. Let $e_0 \in E'$. Using associative identities, Q can be transformed into Q_l $GOJ[p, \text{pres}_{e_0}(e_1), \ldots, \text{pres}_{e_0}(e_n)]$ Q_r, where $\{e_1, \ldots, e_n\}$ = conf(e_0). Q_l, Q_r are simple, graph(Q_l) = G_l and graph(Q_r) = G_r.

PROOF SKETCH. G has a monotonic query Q_1 such that: (1) Join $\overset{p_1 \wedge \cdots \wedge p_k}{\bowtie}$ is the root of a maximal subtree of joins. (2) Every one-sided outerjoin ancestor of $\overset{p_1 \wedge \cdots \wedge p_k}{\bowtie}$ references a relation below the join. (3) Every full outerjoin ancestor of $\overset{p_1 \wedge \cdots \wedge p_k}{\bowtie}$ references some relation in the maximal join/one-sided outerjoin subtree that includes $\overset{p_1 \wedge \cdots \wedge p_k}{\bowtie}$. As in the previous claim, outerjoins unrelated to e_0 have been moved aside. (4) Ancestors are ordered so that those corresponding to edges in conf(e_0) appear as late as possible in the tree. By Theorem 1, Q can be transformed into such Q_1. Now use identity (6) to move $\overset{p_1 \wedge \cdots \wedge p_k}{\bowtie}$ up the operator tree as high as possible. If conf(e_0) is empty, the join is able to move to the root. Otherwise we have two cases.

—If there is a directed edge in conf(e_0), then the parent of the join is a one-sided outerjoin corresponding to this directed edge. Use identity (10) to move the one-sided outerjoin upward, until the identity no longer applies. Now the ancestors of $\overset{p_1 \wedge \cdots \wedge p_k}{\bowtie}$ all correspond to edges in conf(e_0), first the one-sided outerjoin, then the full outerjoins. Use (11) once, introducing a GOJ, then (14) until the operator reaches the root.

—If there are only bidirected edges in conf(e_0), then the ancestors of $\overset{p_1 \wedge \cdots \wedge p_k}{\bowtie}$ all correspond to edges in conf(e_0). Use (12) once, introducing a GOJ, then (14) until the operator reaches the root.

In both cases, the resulting query is that stated in the claim. □

Claim 8. Let G be the graph of a simple query. Let Q' be an operator tree generated by Algorithm **B** using the rule of Figure 8. For each subtree Q'_s of Q', any simple query Q_s with graph $G|_{\text{leaves}(Q'_s)}$ can be transformed into Q'_s by means of associative identities.

PROOF. By induction on the structure of subtrees of Q'.

Base. The claim is trivially true for one-leaf subtrees of Q'.

Induction. Let $Q'_s = Q'_{ls} \odot Q'_{rs}$. Let E' be the set of edges in G between leaves(Q'_{ls}) and leaves(Q'_{rs}). If E' contains bidirected, directed, or undirected edges, then by Claim 5, 6, or 7, respectively, simple query Q_s can be transformed by associative identities into $Q_{ls} \odot Q_{rs}$; Q_{ls}, Q_{rs} are simple, with graphs $G_{\text{leaves}(Q'_{ls})}$, $G_{\text{leaves}(Q'_{rs})}$, respectively. In each case, the root operator chosen by the rule in Figure 8 for Q'_s is the same as the root $\odot$ obtained transforming Q_s into $Q_{ls} \odot Q_{rs}$. Now, by induction hypothesis, Q_{ls}, Q_{rs} can be transformed into Q'_{ls}, Q'_{rs}, thus completing the transformation of Q_s into Q'_s. □

PROOF OF THEOREM 5. Follows from Claim 8. □

PROOF OF THEOREM 2. Let G be the query graph of a simple query Q, and let $T \in \text{assoc}(G)$. Using the operator selection rule of Figure 8, the modified Algorithm **B** generates an operator tree Q' having association tree T. By Theorem 5, associative identities can transform Q into Q'. □

PROOF OF LEMMA 6. Assume $T_s = T_l \odot T_r$ is a subtree of Q, and examine now how the operator $\odot$ was chosen using Figure 8. Let e_0 be the edge used to combine subtrees T_l, T_r. If e_0 is bidirected, then $\odot$ is full outerjoin; otherwise, no bidirected edge was used in the generation of T_l, T_r, for it would have created an early full outerjoin. Now, if e_0 is directed, then conf(e_0) consists of bidirected edges only, but we have said that no bidirected edge was used in T_l or T_r; then, no conflicting, less restrictive edges of e_0 were evaluated early and operator $\odot$ is one-sided outerjoin. Finally, if e_0 is undirected, it conflicts with at most one directed edge, by

Claim 4. Again, no bidirected edge in conf(e_0) was used early, but now there may be one directed edge in conf(e_0), which could have been used in T_l or T_r. In that case, operator $\odot$ is GOJ preserving one set of attributes; otherwise, $\odot$ is join. Therefore, in no case does GOJ need to preserve more than one set of attributes. □

REFERENCES

ANSI. 1992. Working draft of SQL2/SQL3. Tech. Rep., American National Standards Institute.

BHARGAVA, G., GOEL, P., AND IYER, B. 1995. Hypergraph based reorderings of outer join queries with complex predicates. In *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data,* (San Jose, CA, May), 304–315.

CHEN, A. L. P. 1990. Outerjoin optimization in multidatabase systems. In *2nd International Symposium on Databases in Parallel and Distributed Systems.*

CODD, E. F. 1979. Extending the relational database model to capture more meaning. *ACM Trans. Database Syst. 4,* 4 Dec. 397–434.

DATE, C. J. 1986. *An Introduction to Database Systems,* Vol. II. Addison-Wesley, Reading, MA.

DAVID, M. 1991. Advanced capabilities of the outer join. *ACM SIGMOD Rec. 21,* 1 Mar. 65–70.

DAYAL, U. 1983. Processing queries with quantifiers. In *Proceedings of ACM SIGACT-SIGMOD-SIGART 1983 Conference on Principles of Database Systems* (Atlanta, GA, March), 125–136.

DAYAL, U. 1987. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the Thirteenth International Conference on Very Large Databases* (Brighton, England, Sept.) 197–208.

DAYAL, U. AND HWANG, H. 1984. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. Softw. Eng. 10,* 6 (Nov.), 628–645.

GALINDO-LEGARIA, C. A. 1987. Rule-based optimization of database queries. Master's thesis, CINVESTAV-IPN, Mexico City (Sept.). In Spanish.

GALINDO-LEGARIA, C. A. 1992. Algebraic optimization of outerjoin queries. PhD thesis, Technical Rep. TR-12-92, Harvard Univ., Cambridge, MA, May.

GALINDO-LEGARIA, C. A. 1994. Outerjoins as disjunctions. In *Proceedings of ACM SIGMOD 1994 International Conference on Management of Data,* (Minneapolis, MN), 348–358.

GALINDO-LEGARIA, C. A. AND ROSENTHAL, A. 1992. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *Proceedings of the Eighth International Conference on Data Engineering.*

GRAEFE, G. 1990. Volcano, an extensible and parallel query evaluation system. Tech. Rep. CU-CS-481-90, Univ. of Colorado, Boulder.

GRAEFE, G. AND DEWITT, D. J. 1987. The exodus optimizer generator. In *Proceedings of ACM SIGMOD 1987 International Conference on Management of Data,* (San Francisco, CA, May), 160–172.

HAAS, L., CHANG, W., LOHMAN, G. M., MCPHERSON, J., WILMS, P. F., LAPIS, G., LINDSAY, B., PIRAHESH, H., CAREY, M., AND SHEKITA, E. 1990. Starburst mid-flight: As the dust clears. *IEEE Trans. Knowl. Data Eng. 2,* 1 143–160.

KIM, W. 1982. On optimizing an sql-like nested query. *ACM Trans. Database Syst. 7,* 3 (Sept.).

LACROIX, M. AND PIROTTE, A. 1976. Generalized joins. *ACM SIGMOD Rec. 8,* 3, (Sept.).

LEE, B. S. AND WIEDERHOLD, G. 1994. Outer joins and filters for instantiating objects from relational databases through views. *IEEE Trans. Knowl. Data Eng. 6,* 1 108–119.

MELTON, J. AND SIMON, A. R. 1993. *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann, San Francisco, CA.

MURALIKRISHNA, M. 1989. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of the Fifteenth International Conference on Very Large Databases* (Amsterdam, The Netherlands, Aug.), 77–85.

ONO, K. AND LOHMAN, G. M. 1990. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the Sixteenth International Conference on Very Large Databases* (Brisbane, Australia, Aug.), 314–325.

OZSOYOGLU, G., MATOS, V., AND OZSOYOGLU, Z. M. 1989. Query processing techniques in the summary-table-by-example database query language. *ACM Trans. Database Syst. 14,* 4 (Dec.), 526–573.

ROSENTHAL, A. AND GALINDO-LEGARIA, C. A. 1990. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data,* (Atlantic City, NJ, May), 291–299.

ROSENTHAL, A. AND HELMAN, P. 1986. Understanding and extending transformation-based optimizers. *Database Eng. Bull. 9,* 4 (Dec.), 44–51.

ROTH, M. A., KORTH, H. F., AND SILBERSCHATZ, A. 1989. Null values in nested relational databases. *Acta Inf. 26,* 7, 615–642.

ROSENTHAL, A. AND REINER, D. 1984. Extending the algebraic framework of query processing to handle outerjoins. In *Proceedings of the Tenth International Conference on Very Large Databases,* (Singapore, Aug.), 334–343.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, P. A., AND PRICE, T. G. 1979. Access path selection in a relational database system. In *Proceedings of ACM SIGMOD 1979 International Conference on Management of Data* (Boston, MA, May), 23–34.

SHETH, A. P. ED. 1991. *ACM SIGMOD Record Special Issue on Semantic Issues in Multidatabase Systems. 20*, 4 (Dec.).

SHETH, A. P. AND LARSON, J. A. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv. 22,* 3 (Sept.) 183–236.

SCHOLL, M. H., PAUL, H.-B., AND SCHEK, H.-J. 1987. Supporting flat relations by a nested relational kernel. In *Proceedings of the Thirteenth International Conference on Very Large Databases,* (Brighton, England, Sept.), 137–146.

TAY, Y. C. 1990. On the optimality of strategies for multiple joins. In *Proceedings of ACM SIGACT-SIGMOD-SIGART 1990 Conference on Principles of Database Systems* (Atlantic City, NJ, May), 124–131.

ULLMAN, J. D. 1982. *Principles of Database Systems.* 2nd ed., Computer Science Press, Rockville, MD.

WANG, Y. R. AND MADNICK, S. E. 1990. A polygen model for heterogeneous database systems: The source tagging perspective. In *Proceedings of the Sixteenth International Conference on Very Large Databases,* (Brisbane, Australia, Aug.), 519–538.