Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins

Arnon Rosenthal
ETH Zurich and Xerox AIT
arnie@xait xerox com

César Galindo-Legaria *
Aiken Computation Laboratory
Harvard University
cesar@harvard harvard edu

Abstract

We determine when a join/outerjoin query can be expressed unambiguously as a query graph, without an explicit specification of the order of evaluation. To do so, we first characterize the set of expression trees that implement a given join/outerjoin query graph, and investigate the existence of transformations among the various trees Our main theorem is that a join/outerjoin query is freely reorderable if the query graph derived from it falls within a particular class, every tree that "implements" such a graph evaluates to the same result

The result has applications to language design and query optimization Languages that generate queries within such a class do not require the user to indicate priority among join operations, and hence may present a simplified syntax And it is unnecessary to add extensive analyses to a conventional query optimizer in order to generate legal reorderings for a freely-reorderable language

1 Introduction

1.1 Motivation

Relational query languages and optimizers are designed to exploit the properties of Restrict/Project/Join (R/P/J) queries In particular, they depend on the fact that Join is associative[1], so it is possible to abstract a query as a *query graph* The graph does not impose a partial ordering on the operations, but instead shows relations as nodes, and join predicates as edges Because execution order is inessential to the query's semantics, languages and optimization of R/P/J queries are particularly simple

- From the language point of view, users need only provide enough information to produce the query graph, e g , the SQL Select-From-Where clauses There is no need to supply parentheses to specify an execution order for joins.
- From the optimizer's point of view, every tree that "implements" the query graph is known to evaluate to the same result, and the input to later stages of optimization is known to be a query graph

But there are applications of matching elements of two relations that cannot be expressed properly by means of join A problem arises if we want to preserve all elements of one (or both) of the relations in the result, even if there is no matching element in the other relation As an example, when we want a listing of departments and their employees, we often want to see *all* departments, even those without employees. Currently, to obtain this rather natural result, it is necessary to use SQL embedded in a programming language —a complicated process For this reason, [DATE83] proposed outerjoin constructs to be added to SQL There are also cases where outerjoins arise in existing languages

- Daplex allows the specification of loops that are equivalent to left outerjoins [SHIP81]
- It can be used to obtain a relational (flattened) representation of a hierarchy where some parent instances have no children [SCHO87,OZSO89]
- It can be used for processing queries with Count operations [MURA89]

Unfortunately, the natural extension of an R/P/J query graph does not provide an unambiguous expression of a Join/Outerjoin query Sometimes the same query graph can be derived from two queries, i e operator trees, that compute different results (example 2) Yet the benefits of having a simple query graph as an order-independent abstraction are too significant to be casually discarded In particular, outerjoin reordering may reduce significantly the cost of a query (example 1) The task of this paper is to characterize situations in which these benefits remain available In addition, section 6.2 presents a generalized-outerjoin operation that appears useful when the query is not freely reorderable

*This author is supported by a CONACYT scholarship from the Mexican government, register number #52224

[1] Join does not exactly obey the associative law, which allows parentheses to be moved without altering the operators With joins sometimes a new parenthesization forces predicate conjuncts to be moved to a different operator Our formulation in terms of "implementing trees of a query graph" is more precise

1 2 Basic Definitions

Definition. A *scheme* is a finite set of attribute names A *tuple t on scheme S* is an assignment of values to attibute names in S The scheme of tuple t is denoted sch(t) For $X \subseteq$ sch(t), we say t is a tuple *over* X A *null tuple* on scheme S has a null value assigned for all attributes, and is denoted null$_S$

Definition. Tuples t_1, t_2 on disjoint schemes S_1, S_2, respectively, can be *concatenated* into a tuple on $S_1 \cup S_2$, whose assignment coincides with that of t_1 for attributes in S_1, and with that of t_2 for attributes in S_2 Concatenation of tuples t_1, t_2 will be denoted (t_1, t_2) If t is a tuple on scheme S, we may obtain a tuple t' on scheme $S' \supseteq S$ by *padding*, i e concatenating t with null$_{S'-S}$

Definition. A *relation* on scheme S is a finite set of tuples on scheme S The scheme of relation R is denoted sch(R) A *database* is a set of relations whose schemes are mutually disjoint, they will be called *ground relations*

Definition. A *simple predicate* p is defined on some set of attributes S, it maps {tuples over S} to $\{True, False\}$ For any t over S, $p(t)$ is defined and depends only on the values of attributes in S A *join predicate* p is defined similarly, on a pair of sets of attributes S_1, S_2 (from disjoint sets of ground relations), such that $p(t_1, t_2)$ depends only on the values of attributes in S_1 and S_2

Definition. The *regular join* (also called simply "join"), denoted $JN[p](R_1, R_2)$, yields the concatenations of tuples from R_1, R_2 that satisfy the join predicate p, it discards tuples t_1 of R_1 for which there is no t_2 in R_2 satisfying $p(t_1, t_2)$ The *outerjoin*, denoted $OJ[p](R_1, R_2)$ consists of $JN[p](R_1, R_2)$ plus the non-matched tuples of R_1 padded with nulls on the attributes of R_2, R_1 is called the *preserved* relation and R_2 the *null-supplied* relation In infix form, we also denote outerjoin by an arrow toward the null-supplied relation, e g , $R_1 \rightarrow R_2$, and join by $R_1 - R_2$ [2] Two-sided outerjoin will not be discussed

A *query* is an expression over operators in a relational algebra It is expressed as a tree whose leaves correspond to relation variables, and whose internal nodes contain joins, outerjoins, and other algebraic operators. The result of a query Q is denoted *eval(Q)*, and is defined by the usual bottom-up evaluation of expressions If $op_1, op_2,$ denote generic operators (e g , restrict, outerjoin), then an $op_1/op_2/$ query is a an expression in which only the mentioned operators may appear (e g , Restrict/Project/Join query)

Example 1. *Reordering can greatly reduce cost* Consider a query $R_1 - (R_2 \rightarrow R_3)$ in which the first predicate equijoin keys of R_1 and R_2, and the second predicate equijoins keys of R_2 and R_3 It is equivalent to the expression $(R_1 - R_2) \rightarrow R_3$) (as proved in section 2) Assume that these keys have indexes, and also that R_1 has one tuple, while R_2 and R_3 have 10^7 tuples each Then the first expression retrieves $2 * 10^7 + 1$ tuples, and the second retrieves only 3

It should be noted that the strategy of evaluating joins before outerjoins, as we did in the example above, is not necessarily the least expensive alternative for all cases For the same (freely-reorderable) expression $R_1 - R_2 \rightarrow R_3$, if the join predicate is $(R_1\ A > R_2\ B)$ and the outerjoin predicate is $(R_2\ C = R_3\ D)$, evaluating the join first would produce a large output, which then participates in the outerjoin The optimal strategy in this case is to do the outerjoin first

We now define query graphs that are used as abstractions of query expressions (removing the specification of execution order) For an R/P/J query, we use the conventional graph This graph is defined to have a node for each relation mentioned in the query And for each join operator in the query expression, for each predicate conjunct, there is an edge labeled with that conjunct, we assume conjuncts reference attributes of two ground relations —in section 4 we discuss some initial observation on how to handle more general cases We also assume that no relation is used more than once in a query, but several copies of the same relation with renamed attributes can be used

For Outerjoin/Join queries, each outerjoin operation on predicate p between ground relations adds a single edge, directed toward the null-supplied relation, and labeled with the *entire* outerjoin predicate p Each outerjoin predicate must reference attributes from exactly two ground relations, or else the graph is undefined

The graph obtained by the above construction may have parallel edges —i e two different edges between the same relations, like those obtained by conjuncts (R_1 *F-Name*=R_2 *F-Name*) and (R_1 *L-Name*=R_2 *L-Name*) But note that, in terms of evaluation, we are usually interested in applying both conjuncts at the same time For this reason, we will treat them as if they were a single conjunct, and the parallel edges will be collapsed into one The graph corresponding to query Q is denoted *graph(Q)*

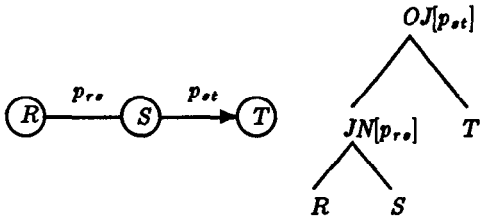
1.3 Outerjoin / Join Queries

In contrast to R/P/J queries, Outerjoin/Join query graphs do not necessarily capture the semantics of the query represented Example 2 shows two queries that have the same graph but yield different results, assume that the outerjoin predicate is over (sch(R_1), sch(R_2)) and the join predicate is over ($sch(R_2)$, sch(R_3))

Example 2. *Joins and outerjoins do not always associate* Despite having the same graph, $R_1 \rightarrow (R_2 - R_3)$ is not equivalent to $(R_1 \rightarrow R2) - R3$ To see this, suppose r_1, r_2, r_3 are the only tuples of R_1, R_2, R_3, respectively, and (r_2, r_3) does not satisfy the join predicate The first expression yields $\{(r_1, -, -)\}$ (i e , one tuple, consisting of r_1 padded by nulls), while the second yields the empty set

But being optimists, we think the class of Outerjoin / Join queries that *do* reassociate easily is large and useful The class of queries whose semantics are successfully represented by the graph is the class of those queries where all reasonable

[2] This notation for join is intended to reflect a graph representation that uses undirected edges, and should not be confused with set difference The notation for outerjoin should also reflect the corresponding directed edge used to represent it in query graphs



(a) Implementing Tree (b) Query Graph

Figure 1 Alternate representations of a query

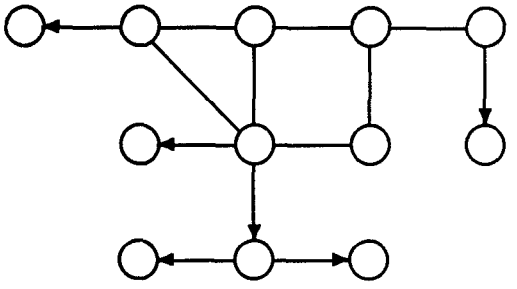


Figure 2 A "nice" topology for a query graph

execution orders yield the same result Such queries (and their graphs) will be called *freely reorderable* The remainder of this section summarizes our major results

Definition. An algebraic expression (i.e , query) is called an *implementing tree* (*IT* for short) of graph G if $G = \text{graph}(Q)$

The two representations —expressions and graphs— provide different levels of information and emphasize different aspects of the query An expression unambiguously specifies the inputs to each operation, and can be evaluated A query graph represents a collection of relations and the join predicates that connect them, but it does not directly possess an evaluation rule When it is known that all associations (i e , ITs) will yield the same result, then the graph is a natural query representation It can be obtained directly from a textual language that does not specify a particular association, or else by ignoring associations specified in a query presented as an algebraic expression Note also that ITs correspond only to *connectivity-preserving* parenthesizations, i.e , joins without graph edges (i e , Cartesian products) are excluded "Tree" terminology (root, ancestor, etc) will be used freely to describe expressions and their operands Fig 1 shows these two representations of a query, p_{xy} denotes a predicate between attributes in relations X and Y

Implementing trees and their relationship to query graphs can be used to define the notions of "reasonable execution orders," "reassociation" and "associativity" Not every reassociation is permitted in an implementing tree —every join must be supported by a predicate in the original query In Fig 1, a reassociation joining R and T is disallowed A detailed treatment of query graphs, implementing trees, and tree rewrite rules (basic transformations) appears in Section 3

Definition. A query Q and its graph(Q) are called *freely reorderable* if graph(Q) is defined and for all Q' such that $\text{graph}(Q') = \text{graph}(Q)$, $\text{eval}(Q) = \text{eval}(Q')$

Theorem (Join/outerjoin free reorderability). *A graph G (and therefore its implementing trees) will be freely reorderable when the following two conditions hold*

- *G consists of a connected set of join edges, from which OJ-labeled trees go outward, and*
- *outerjoin predicates return False when all attributes of the preserved relation are null*

Fig. 2 shows a graph satisfying the conditions of the above theorem.

2 Identities for Join-like Operators

In this section we prove identities that imply that certain rewrites will not alter the value of a query Section 3 will prove that for "nice" graphs, a series of such rewrites can map any implementing tree to any other implementing tree for the same graph We choose a proof approach based on equivalence of algebraic expressions. In this approach, the intermediate results are still algebraic identities, and may be useful in other settings Also, an algebraic treatment should be preferred over proofs of equivalence by mutual set-containment in an environment where duplicates are permitted

2.1 Definitions

Definition. The *antijoin*, denoted $AJ[p](R_1, R_2)$, or $R_1 \triangleright R_2$, is defined as $\{r_1 \in R_1 \mid \text{no tuple of } R_2 \text{ satisfies } p(r_1, r_2)\}$

Definition. A predicate p is *strong* with respect to a set S of attributes if, whenever a tuple t has a null value for all attributes in S, $p(t)$ =False As a natural extension, we call a predicate strong with respect to a relation R if it is strong with respect to sch(R)

The algebraic identities in this section depend on the pattern of attribute references in different join predicates The following conventions ease the specification and handling of these patterns

Convention. We will use $\odot$ as an infix "generic join operator," in statements where join ($-$), outerjoin ($\rightarrow$), or antijoin ($\triangleright$) can appear Given subexpressions X, Y, $X \overset{p_{xy}}{\odot} Y$ denotes an operator whose predicate can reference only attributes in X and Y, moreover, any conjunct in the operator has to reference attributes in both X and Y Thus, in the expression $X \overset{p_{xy}}{-} (Y \overset{p_{yz}}{\rightarrow} Z)$ the first predicate reference relations in Y but not in Z This predicate information is omited

when clear from the context We say that P_{xy} is strong with respect to X if the predicate is strong with respect to the set of attributes it references from X.

Convention. Whenever we have an expression $X \overset{P_{xy}}{\odot} Y$, we assume that $sch(eval(X)) \cap sch(eval(Y)) = \emptyset$ Also, to have more freedom writing expressions, each operator has a "symmetric form" that reverses the treatment of left and right operands $X \overset{P_{xy}}{\rightarrow} Y = Y \overset{P_{yx}}{\leftarrow} X$, $X \overset{P_{xy}}{\triangleright} Y = Y \overset{P_{yx}}{\triangleleft} X$, and of course $X \overset{P_{xy}}{-} Y = Y \overset{P_{yx}}{-} X$

Convention. For comparing or computing the union of relations X, Y, we first pad the tuples of each relation to scheme $sch(X) \cup sch(Y)$ Using this convention it is legal to write $(R - S) \cup (R \triangleright S)$, we would pad the result of the antijoin and then compute the union with the join result The purpose is to allow an easier writing of expression (see identities 9 and 10 below)

For reasons of space, most proofs appear in [GALI89] rather than in this paper

2.2 Preliminary Identities

We will assume that join operators have precedence over union The following associativity and distributivity identities on joins follow fairly easily from the definitions Recall that — indicates join Identities 1–3 are roughly associative laws, 4–7 are distributive Identity 1 requires the existence of predicates P_{xy} and P_{yx}, while P_{xx} is optional, and if it exists then it has to be moved between operators after reassociation —the corresponding query graph has a cycle For all other identities, operators always remain unchanged after reassociation

$$(X \overset{P_{xy}}{-} Y) \overset{P_{xz} \cup P_{yz}}{-} Z = X \overset{P_{xy} \cup P_{xz}}{-} (Y \overset{P_{yz}}{-} Z) \quad (1)$$

$$(X \overset{P_{xy}}{-} Y) \overset{P_{yz}}{\triangleright} Z = X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{\triangleright} Z) \quad (2)$$

$$(X \overset{P_{xy}}{\triangleleft} Y) \overset{P_{yz}}{\triangleright} Z = X \overset{P_{xy}}{\triangleleft} (Y \overset{P_{yz}}{\triangleright} Z) \quad (3)$$

$$X - (Y \cup Z) = (X - Y) \cup (X - Z) \quad (4)$$

$$(Y \cup Z) - X = (Y - X) \cup (Z - X) \quad (5)$$

$$(Y \cup Z) \triangleright X = (Y \triangleright X) \cup (Z \triangleright X) \quad (6)$$

$$X \overset{P_{xy}}{\triangleright} Y = X \overset{P_{xy}}{\triangleright} (Y \overset{P_{yz}}{-} Z \cup Y \overset{P_{yz}}{\triangleright} Z) \quad (7)$$

If the predicate P_{yz} is strong with respect to Y, then we have

$$(X \overset{P_{xy}}{\triangleright} Y) \overset{P_{yz}}{-} Z = \emptyset \quad (8)$$

$$(X \overset{P_{xy}}{\triangleright} Y) \overset{P_{yz}}{\triangleright} Z = X \overset{P_{xy}}{\triangleright} Y \quad (9)$$

Finally, note that outerjoin can be expressed in terms of join and antijoin as

$$X \overset{P_{xy}}{\rightarrow} Y = X \overset{P_{xy}}{-} Y \cup X \overset{P_{xy}}{\triangleright} Y \quad (10)$$

$(X \overset{P_{xy}}{\rightarrow} Y) \overset{P_{yz}}{\rightarrow} Z$

(Expand outerjoin, eqn 10)

$= (X \overset{P_{xy}}{\rightarrow} Y) \overset{P_{yz}}{-} Z \cup (X \overset{P_{xy}}{\rightarrow} Y) \overset{P_{yz}}{\triangleright} Z$

(Expand outerjoin, eqn 10)

$= (X \overset{P_{xy}}{-} Y \cup X \overset{P_{xy}}{\triangleright} Y) \overset{P_{yz}}{-} Z \cup$
$(X \overset{P_{xy}}{-} Y \cup X \overset{P_{xy}}{\triangleright} Y) \overset{P_{yz}}{\triangleright} Z$

(Distribute join and antijoin over union, eqn 5, 6)

$= (X \overset{P_{xy}}{-} Y) \overset{P_{yz}}{-} Z \cup (X \overset{P_{xy}}{\triangleright} Y) \overset{P_{yz}}{-} Z \cup$
$(X \overset{P_{xy}}{-} Y) \overset{P_{yz}}{\triangleright} Z \cup (X \overset{P_{xy}}{\triangleright} Y) \overset{P_{yz}}{\triangleright} Z$

(Simplify for strong predicate P_{yz}, eqn 8, 9)

$= (X \overset{P_{xy}}{-} Y) \overset{P_{yz}}{-} Z \cup (X \overset{P_{xy}}{-} Y) \overset{P_{yz}}{\triangleright} Z \cup X \overset{P_{xy}}{\triangleright} Y$

(Reassociate join and antijoin, eqn 1, 2)

$= X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{-} Z) \cup X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{\triangleright} Z) \cup X \overset{P_{xy}}{\triangleright} Y$

(Complete by pseudo-distributivity of antijoin, eqn 7)

$= X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{-} Z) \cup X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{\triangleright} Z) \cup$
$X \overset{P_{xy}}{\triangleright} (Y \overset{P_{yz}}{-} Z \cup Y \overset{P_{yz}}{\triangleright} Z)$

(Factor out join from union, eqn 4)

$= X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{-} Z \cup Y \overset{P_{yz}}{\triangleright} Z) \cup$
$X \overset{P_{xy}}{\triangleright} (Y \overset{P_{yz}}{-} Z \cup Y \overset{P_{yz}}{\triangleright} Z)$

(Rewrite as outerjoin, eqn 10)

$= X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{\rightarrow} Z) \cup X \overset{P_{xy}}{\triangleright} (Y \overset{P_{yz}}{\rightarrow} Z)$

(Rewrite as outerjoin, eqn 10)

$= X \overset{P_{xy}}{\rightarrow} (Y \overset{P_{yz}}{\rightarrow} Z)$

Figure 3 Algebraic proof of identity 12

2.3 Reassociation Rules for Outerjoins

The main results of Section 2 are the following identities on the associativity of "three operand" outerjoin and join queries The analysis of whether join predicates must be strong appears to be new

$$(X \overset{P_{xy}}{-} Y) \overset{P_{yz}}{\rightarrow} Z = X \overset{P_{xy}}{-} (Y \overset{P_{yz}}{\rightarrow} Z) \quad (11)$$

$$(X \overset{P_{xy}}{\rightarrow} Y) \overset{P_{yz}}{\rightarrow} Z = X \overset{P_{xy}}{\rightarrow} (Y \overset{P_{yz}}{\rightarrow} Z), \quad (12)$$
if P_{yz} is strong w r t Y

$$(X \leftarrow_{P_{xy}} Y) \overset{P_{yz}}{\rightarrow} Z = X \leftarrow_{P_{xy}} (Y \overset{P_{yz}}{\rightarrow} Z) \quad (13)$$

As an example, the proof of identity 12 is shown in Fig 2 The algebraic proofs of identities 11 and 13 are similar

Identity 12 does not extend to arbitrary predicates In the following example, predicate P_{bc} is not strong with respect to B.

Example 3. *Nonstrong predicates preclude outerjoin reassociation* Consider the relations $A = \{(a)\}$, $B = \{(b, -)\}$, $C = \{(c)\}$ Let P_{ab} denote (A attr1=B attr1), and P_{bc} denote (B attr2=C attr1 or B attr2=null) Then

$$A \overset{P_{ab}}{\rightarrow} (B \overset{P_{bc}}{\rightarrow} C) \text{ yields } \{(a, -, -, -)\}, \text{ but}$$
$$(A \overset{P_{ab}}{\rightarrow} B) \overset{P_{bc}}{\rightarrow} C \text{ yields } \{(a, -, -, c)\}$$

3 Query Graphs and Implementing Trees

Implementing trees can be modified by reordering adjacent operators to make one of them "jump" one level up the tree, the resulting tree implements the same graph This modification is called a *basic transform* (*BT* for short). Although two trees that implement the same graph need not yield the same result, BTs corresponding to the identities of section 2 are result-preserving The closure under those BTs is a set of trees that evaluate to the same result Proving free reorderability of a query Q reduces to show that all queries implementing graph(Q) can be obtained from Q by result-preserving BTs

Note that our proofs depend on equivalence of operator trees (which possess a well-defined bottom-up evaluation rule), rather than on equivalence of query graphs, which do not possess a general evaluation rule

3.1 Definitions and Basic Properties

Recall that in a query $(Q_1 \odot Q_2)$, we can call $\odot$ the root, Q_1 the left subtree, and Q_2 the right subtree

Observation. If IT Q implements graph G, then subtrees of Q implement connected subgraphs of G Suppose $Q = (Q_1 \odot Q_2)$ Edges of conjuncts of operator $\odot$ determine a cut in G, removing them, we are left with two connected subgraphs, say G_1 and G_2 Either Q_1 implements G_1 and Q_2 implements G_2, or Q_2 implements G_1 and Q_1 implements G_2

Observation. If G = graph(Q_1) = graph(Q_2) and their corresponding roots are equal (including operator and predicate), the subtrees of both roots implement the same subgraphs of G

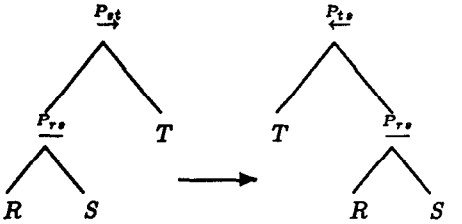
Observation. Let G be a query graph and G' an induced subgraph If G' is connected, then ITs of G' are subtrees of ITs of G

Definition. A query graph G is called *"nice"* if the following holds

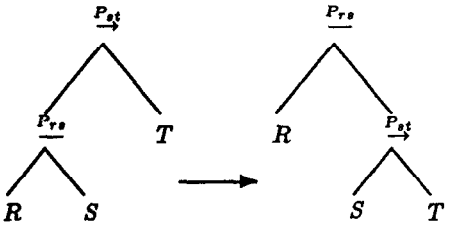
- G can be expressed as the union of graphs G_1, G_2, where G_1 is connected and has only join edges, and G_2 is a forest of outerjoin edges, and
- the intersection of G_1 and G_2 is the set of nodes that are roots of the forest G_2

Lemma 1. *G is "nice" if and only if it satisfies*

- *There are no cycles composed of outerjoin edges,*
- *there is no path of the form $X \rightarrow Y - Z$, and*
- *there is no path of the form $X \rightarrow Y \leftarrow Z$*



(a) Reversal BT



(b) Reassociation BT

Figure 4 Basic Transforms on Implementing Trees

Observation. If G' is a connected subgraph of a "nice" graph G, then G' is also "nice"

3.2 Basic Transforms

Definition. There are two *basic transforms* of trees, reversal and reassociation They are illustrated in Fig 4 for the same IT of Fig 1

Reversal Exchange the left and right subtree of a node as shown in Fig 4 Since the operator of that node may not be commutative, it is replaced by its "symmetric form" (e g $\overset{P_{xy}}{\rightarrow}$ is replaced by $\overset{P_{yx}}{\leftarrow}$)

Reassociation. Exchange a pair of parent/child nodes as shown in Fig 4 $((Q_1 \odot_1 Q_2) \odot_2 Q_3)$ is transformed to $(Q_1 \odot_1 (Q_2 \odot_2 Q_3))$ If there is a conjunct in $\odot_2$ referencing Q_1, then it has to be moved to the operator $\odot_1$ after the reassociation (see identity 1 of section 2) This BT is *valid* or *applicable* only if the predicate in $\odot_2$ references some relation in Q_2, and if a conjunct was moved between operators then both operators must be regular joins We refer to this BT as $[X \odot_1 Y \odot_2 Z]$

Observation. If Q' is obtained from Q by a reversal or reassociation BT, then graph(Q') = graph(Q)

We now turn our attention to those BTs which do not modify the result of the query

Definition. Suppose we obtain Q' from Q by applying one BT This BT is called *result preserving* if eval(Q') = eval(Q) for all values of the ground relations

A reassociation BT $[X \odot_1 Y \odot_2 Z]$ is result preserving whenever the "three relations" identity $((X \odot_1 Y) \odot_2 Z) = (X \odot_1 (Y \odot_2 Z))$ holds It follows from the definition that reversal BTs are always result preserving

Lemma 2. *If $G = graph(Q)$ is "nice" and outerjoin predicates are strong with respect to the null-supplied relation then all BTs applicable to Q are result preserving*

Proof sketch. The only non-preserving BTs that may occur with the operators we have are

- $[X \rightarrow Y - Z]$
- $[X \rightarrow Y \leftarrow Z]$

We will show that if $[X \rightarrow Y - Z]$ is applicable on Q then G is not a "nice" graph The other case is handled similarly Without loss of generality, assume that $[X \rightarrow Y - Z]$ is applicable on the root of Q, which can then be written as $((Q_1 \rightarrow Q_2) - Q_3)$

- The outerjoin predicate references some relation A in Q_1 and B_1 in Q_2, the join predicate references some relation B_2 in Q_2 and C in Q_3
- Leaves of Q_1, Q_2, Q_3 induce connected subgraphs G_1, G_2, G_3, respectively There is in G a directed edge from A to B_1, and an undirected edge between B_2 and C
- Since B_1, B_2 are both in a connected subgraph, G_2, there is a path between them in G This path can be extended to obtain a path between A and C, where the conditions of lemma 1 are violated □

Lemma 3. *If $graph(Q)$ is "nice" and $graph(Q) = graph(Q')$ then a sequence of BTs can obtain Q' from Q*

Proof sketch. Suppose $Q' = (Q'_1 \odot Q'_2)$ If $\odot$ is an inner node k levels below the root in Q and its predicate is a single conjunct, the application of k reassociations will map Q to an expression of the form $(Q_1 \odot Q_2)$, in which $\odot$ is the root We know that (up to a single reversal BT) $graph(Q_1) = graph(Q'_1)$, and $graph(Q_2) = graph(Q'_2)$, and each of these graphs is smaller than the original By induction, Q_1 can be mapped to Q'_1 by a sequence of BTs, and Q_2 can be mapped to Q'_2 by a sequence of BTs When $\odot$ has more than one conjunct, i e it is a general cutset in the graph, a generalization of the above argument applies □

Theorem 1. *If $graph(Q)$ is "nice" and outerjoin predicates are strong with respect to the null-supplied relation then Q is freely reorderable*

Proof. Take any Q' such that $graph(Q) = graph(Q')$ By lemma 3, Q' can be obtained from Q by a sequence of BTs, say

$$Q = Q_0 \overset{BT}{\rightsquigarrow} Q_1 \overset{BT}{\rightsquigarrow} \quad \overset{BT}{\rightsquigarrow} Q_n = Q'$$

By lemma 2, $\forall i = 0 \quad n-1 \quad eval(Q_i) = eval(Q_{i+1})$ Finally, by transitivity $eval(Q) = eval(Q')$, and therefore Q is freely reorderable □

4 Join/Outerjoin/Restrict Queries: Initial Observations

In this section, we sketch how Restriction operations can be added to the theory presented In general, these operations cannot be freely reordered in the presence of outerjoins, i e , it does make a difference when they are executed We analyze only the case where all Restrictions and Projections in the original query occur after all outerjoins have been performed Unlike joins, we do not usually want to explore alternative positions, but instead just want to do restrictions as early as possible.

It is well known that a restriction on the preserved operand of an outerjoin can be moved into the outerjoin predicate Difficulties arise only with moving restrictions passed a null-supplied operand But a Restriction with a predicate that is strong on some attribute A will discard tuples in which nulls are used to pad that attribute Hence, if the results of an outerjoin are subject to such a restriction, there is no point in introducing the padded tuples —regular join would suffice A similar argument can be used to convert 2-sided outerjoin to one-sided uterjoin

The simplification rule is Suppose the query includes a predicate (restriction or regular join) that is strong in some attributes of relation R Consider the path *in the implementing tree* going from that predicate to R' If an outerjoin is in that path and R is in its null-supplied subtree, then replace the operator by regular join This simplification is carried out before creation of the query graph

We conjecture that if the restriction predicate occurs after all outerjoins, then the simplification cannot introduce new violations of free reorderability That is, on the path where it operates, it can never introduce a join as an operand of an outerjoin, nor add a join with a null-supplied attribute

While this transformation is guaranteed to simplify query processing, some other simplifications do pull us out of the class of free reorderability For example, suppose we know that some outerjoin operation yields the same result as a regular join, i e , that there will be no unmatched tuples to be padded with nulls —a referential integrity constraint could supply this information It is legal to replace the outerjoin operator by a join operation (and the corresponding edge of the query graph by a join edge) However, the resulting query may not be freely reorderable, an implementing tree of the graph of the revised query may not give the correct result

Consider the freely reorderable expression $R_1 \rightarrow R_2 \rightarrow R_3$ If a referential integrity constraint between R_2 and R_3 allowed us to replace $R_2 \rightarrow R_3$ by $R_2 - R_3$, the resulting query $R_1 \rightarrow (R_2 - R_3)$ would no longer be freely reorderable Reassociation for general graphs is still possible using generalized outerjoin, as shown in Section 6, but this requires a more sophisticated optimizer

5 Languages That Generate Freely-Reorderable Queries

Freely-reorderable outerjoins go substantially beyond Restrict/Project/Join queries, but impose relatively little bur-

den on language syntax or optimization To show that the class may be more than a theoretical curiosity, we present some operators that extend SQL to handle relations whose attributes may be set- or entity-valued They were devised by J Bauer (unpublished), as part of an effort to design a new commercial DBMS with somewhat more than relational power[3] In this section we describe the operators, show how they can be expressed in terms of Outerjoins, and show that the resulting expressions are freely reorderable The syntax is approximate, to enhance readability

5.1 The Operators UnNest and Link

Two new operators, denoted * and ->, will be used in the From-List of an SQL query We often use "entity" as a synonym for "tuple", to emphasize that in this section's model, tuples (i e , entities) have identity, repeating fields, and entity-valued attributes

UnNest or Flatten (*). If R has a set-valued attribute *Field*, (R * *Field*) indicates the unnesting of the attribute[4] The result is a relation R' with a scheme identical to scheme(R), except that *Field* is now single-valued Each tuple r of R will produce one or more tuples in R' If r *Field* had $n > 0$ elements, then R' has n tuples for r, each tuple's Field containing a single value from the set, if r *Field* is empty then there is only one tuple for r in R' and it has a null value in *Field* For example,

```
Select All
From   EMPLOYEE*ChildName, DEPARTMENT
Where  EMPLOYEE D# = DEPARTMENT D# and
       DEPARTMENT Location = 'Queretaro'
```

returns at least one tuple for each employee in a Queretaro department For Queretaro employees with children, one tuple is returned for each child, otherwise, a tuple with null ChildName is returned

Link via (->). If R has an entity-valued attribute *Field*, (R-> *Field*) indicates the completion of each tuple r in R by concatenating to it the tuple referenced by r *Field* The scheme of the result is sch(R) ∪ sch(the entity type referenced by *Field*) − *Field* In the extension, for each tuple in R we will have exactly one tuple in the result If r *Field* is null, then a null tuple is concatenated to r

For example, suppose DEPARTMENT includes EMPLOYEE-valued attributes Manager and Secretary, and a REPORT-valued attribute Audit The following query returns, for all Zurich departments, the department information, plus the employee attributes of the manager, and the report information from the audit.

```
Select All
From   DEPARTMENT-->Manager-->Audit
Where  DEPARTMENT Location = 'Zurich'
```

The next example uses both Flatten and Link The order of the clauses is not essential —the parser can associate the attributes with their relations The query helps a prosecutor identify money siphoned to employees or their children, by adding EMPLOYEE and ChildName information to the department information of the previous query

```
Select All
From   EMPLOYEE*ChildName,
       DEPARTMENT-->Manager-->Audit,
Where  EMPLOYEE D# = DEPARTMENT D# and
       DEPARTMENT Location = 'Zurich' and
       EMPLOYEE Rank>10
```

Attributes obtained from the right side of -> and * operators cannot appear in the Where-List predicates because the position of the restriction predicate would be ambiguous, either before or after unnesting But they may be restricted in an enclosing query block

5.2 Expressing Link and UnNest with Outerjoins

We can make * and -> special cases of outerjoins by defining *traversable predicates* on these associated entity fields The reformulation is similar to that used in [ROSE85] to apply relational optimization to SQL over Codasyl structures Assume below that every tuple (i.e , entity), and also every field value, has a unique object identifier (e g , a physical address on disk), denoted by the prefix @ The restatement in terms of outerjoins requires introducing special predicates NestedIn and LinkedTo that connect an entity to its repeating values or related entities These predicates are evaluated only as part of a join or outerjoin, and require the optimizer to generate code to traverse to the related value or entity For example, given an EMPLOYEE instance denoted e, the access routines can find all DEPARTMENTS d such that Manager(d) = e. In any case, the implementation technique for these predicates is not relevant to correctness of query reordering

The operators are expressed in terms of outerjoin as follows

- For UnNest, suppose we have an expression (R * *Field*) Define an abstract, one-column relation *ValueOfField* containing every value appearing in r *Field* for any r in R Then define a predicate NestedIn(@r, @*value*) between identifiers of tuples of R and *ValueOfField*, true whenever *value* is in r *Field* The UnNest expression can be written now as

$$OJ[\text{NestedIn}(@r, @value)](R, ValueOfField)$$

- For Link, suppose we have an expression (R-> *Field*) where the entity-valued *Field* points to a tuple in relation *DomainOfField* Define a join predicate

[3] A Nori and J Hee helped refine the operators' semantics The "free reorderability" criterion emerged from an effort to validate these designers' intuition that this language would be easily optimizable

[4] This operation can easily be extended to allow *Field* to be relation-valued

LinkedTo(@r, @value) between identifiers of tuples of R and *DomainOfField*, true whenever r *Field* points to *value* The Link-via expression can be written as

$$OJ[\text{LinkedTo}(@r, @value)](R, DomainOfField)$$

Two examples are shown below To improve readability we abbreviate EMPLOYEE, DEPARTMENT, and ValueOfChildName as EMP, DEPT, and ChName respectively
`Select All From EMPLOYEE*Childname` becomes

$$\text{EMP } OJ[\text{NestedIn}(@EMP, @ChName)] \text{ ChName}$$

`Select All From DEPARTMENT-->EMPLOYEE*Childname` is written as the following expression, where the position of parenthesis is arbitrary

$$\text{DEPT } OJ[\text{LinkedTo}(@DEPT, @EMP)] \\ (\text{EMP } OJ[\text{NestedIn}(@EMP, @ChName)] \text{ ChName})$$

Join/UnNest/Link Queries Are Freely Reorderable

Observation. The rules above imply that all query blocks are freely-reorderable, i e , that they satisfy the preconditions of Theorem 1 (As in SQL, the query block is hence an attractive unit for optimization, in both languages, combining query blocks requires more subtlety [MURA89])

Proof. Relations that were the null-supplied input to an outerjoin were those introduced by the attributes in the right side of operands * and -> Each time a relation is obtained from a field, it was considered independent, i e , a new tuple variable Therefore there was no way to create two edges directed into a node, or a cycle Furthermore, these relations could not appear in the Where clause of the same query block Thus none of the forbidden subgraphs has arisen Finally, note that the special outerjoin predicates are strong So from Theorem 1 each query block is freely reorderable

It is possible to express non-reorderable queries, by imposing restrictions in enclosing query blocks As in SQL, transformations involving multiple blocks may be difficult But at least the main unit of optimization, the block, is freely reorderable

6 Conclusions and Future Work

6.1 Extending an Optimizer to Handle Freely Reorderable Queries

For designers of query optimizers, freely-reorderable queries are much simpler than the general case For language constructs that are known to generate only freely-reorderable outerjoins, the extension seems small, at least conceptually Optimizers already implement a query graph by generating expression trees with different associations of the graph edges, now it must fill in Join or else Outerjoin (preserving the operator direction) There is no need to insert additional operators, or perform a subtle analysis

For more general languages, it may be possible to extend this approach to reorder freely-reorderable subqueries of the given query. But further research is needed to determine how to combine reordering results for queries that contain arbitrary combinations of operators, e g , Restrict/Project/Join/Outerjoin/Semijoin queries

6.2 Generalized Outerjoins

The result-preserving basic transformations based on identities of section 2 are unable to reassociate an expression of the form $X \rightarrow (Y - Z)$, like the one we had in example 2 A full algorithm for reordering arbitrary join/outerjoin queries is beyond our scope, but we present a generalized outerjoin operator (denoted *GOJ*) that may be an important element in the solution

Definition. Let S denote a set of attributes contained in $\text{sch}(R_1)$, and π denote projection *with* removal of duplicates Then the *generalized outerjoin* consists of tuples that appear in the join, plus S-projections of tuples whose S-projection did not appear in the join, the latter are padded with nulls for attributes outside S
The formal definition (with the predicate p omitted for readability, and $-$ used for set difference is

$$GOJ[S](R_1, R_2) = \\ JN(R_1, R_2) \cup \\ (\pi[S](R_1) - \pi[S]\, JN(R_1, R_2)) \times \text{null}_{\text{sch}(R_1) \cup \text{sch}(R_2) - S} \tag{14}$$

Our definition refines the Generalized-Join operator [DAYA87] by omitting unmatched R_1 tuples whose S-projection appeared in the join We use the term "generalized *outer*join" because *GOJ* generalizes join and outerjoin, but not semijoin or antijoin As with Generalized-Join, *GOJ* can be computed by a slightly modified join algorithm

We have proved several reassociation identities for *GOJ* [GALI89], under the assumption that relations are duplicate free, predicates are strong and of the form P_{xy} and P_{yz}, those identities include

$$X \; OJ \; (Y \; JN \; Z) = \\ (X \; OJ \; Y) \; GOJ[\text{sch}(X)] \; Z \tag{15}$$

$$X \; JN \; (Y \; GOJ[S] \; Z) = \\ (X \; JN \; Y) \; GOJ[S \cup \text{sch}(X)] Z, \tag{16}$$

if $S \subseteq \text{sch}(Y)$ and it contains all X–Y join attributes

6.3 Theoretical Remarks

Studies of join-like operators (outerjoins, nested queries, and semijoins) have had an unusual number of erroneous claims For example, [MURA89] identifies three papers with errors To us, this indicates the need for additional definitions and lemmas to make sound reasoning easier We hope

that our analysis of association and implementing tree may be useful

Note that it may be helpful to be able to reason about both the graph and the query trees, as each representation makes different information accessible The graph shows the pattern of join predicates, but only the implementing trees show execution strategies, or have a general evaluation rule that can justify transformations Thus far, our conditions for reorderability applied to graphs, we conjecture that there are also simple conditions on the expression trees. For example, the null-supplied input of an operand should not be created by a regular join, nor involved later as an operand of a regular join

The "folk theorems" about reorderability do not tell what detailed preconditions are needed for each rule, e g , that predicates be strong, or that relations be duplicate-free. A formal analysis highlights these preconditions and gives a framework to examine algebraic extensions

We hope that similar free reorderability theorems can be proved of other classes of expressions, using the tools developed here For example, for join/semijoin queries, it appears that fewer basic transforms preserve the result, and therefore a smaller set of graphs will be freely reorderable —semijoin edges in series appear to be an additional forbidden subgraph

References

[DATE83] C J. Date, "The Outer Join," *Proceedings of the Second International Conference on Databases*, Cambridge, England, September 1983

[DAYA83] U Dayal, "Processing queries with quantifiers," *PODS 1983*, pp 125–136.

[DAYA87] U Dayal, "Of nests and trees A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," *VLDB 1987*, pp 197–208.

[GALI89] C Galindo-Legaria, A Rosenthal, E Kortright, "Expressions, Graphs, and Algebraic identities for joins, antijoins, outerjoins, and generalized outerjoins", working paper, available from the authors

[MURA89] M Muralikrishna, "Optimization and dataflow algorithms for nested tree queries," *VLDB 1989*, pp 77–85

[OZSO89] G Ozsoyoglu, V Matos, Z M. Ozsoyoglu, "Query processing techniques in the Summary-Table-by-Example database query language," *ACM Transactions on Database Systems*, Vol 14, No 4, December 1989, pp 526–573

[ROSE84] A Rosenthal, D Reiner, "Extending the algebraic framework of query processing to handle outerjoins," *VLDB 1984*, pp. 334–343

[ROSE85] A Rosenthal, D Reiner, "Querying relational views of networks" in Kim et al, *Query Processing*, Springer-Verlag, New York, 1985

[SCHO87] M. H. Scholl, H -B Paul, H -J Schek, "Supporting flat relations by a nested relational kernel," *VLDB 1987*, pp 137–146.

[SHIP81] D W. Shipman, "The functional data model and the data language DAPLEX," *ACM Transactions on Database Systems*, Vol 6, No 1, March 1981, pp 140–173