Hypergraph based reorderings of outer join queries with complex predicates

Gautam Bhargava   Piyush Goel   Bala Iyer

IBM Santa Teresa Laboratory
555 Bailey Avenue
San Jose CA 95141

{gautam, pgoel, balaiyer}@vnet.ibm.com

Abstract

Complex queries containing outer joins are, for the most part, executed by commercial DBMS products in an "as written" manner. Only a very few reorderings of the operations are considered and the benefits of considering comprehensive reordering schemes are not exploited. This is largely due to the fact there are no readily usable results for reordering such operations for relations with *duplicates* and/or outer join predicates that are other than "*simple.*" Most previous approaches have ignored duplicates and complex predicates; the very few that have considered these aspects have suggested approaches that lead to a possibly exponential number of, and redundant intermediate joins. Since traditional query graph models are inadequate for modeling outer join queries with complex predicates, we present the needed hypergraph abstraction and algorithms for reordering such queries with joins and outer joins. As a result, the query optimizer can explore a significantly larger space of execution plans, and choose one with a low cost. Further, these algorithms are easily incorporated into well known and widely used enumeration methods such as dynamic programming.

1 Introduction

1.1 Motivation

The outer join operation provides a method to combine tuples from two tables, based on some matching condition, without having to lose tuples from one (or, both) table(s) that do not match with any tuples in the other table [DATE83]. This operation has been included in SQL2 and is being supported in leading commercial RDBMS products such as IBM's DB2, Tandem's NonStop SQL, Oracle/SQL, Sybase, etc. Additionally, outer joins are useful in myriads of other contexts, such as multi-database systems which try to integrate local schemas into global schemas [CHEN90], transformation of nested queries with the COUNT aggregation function into (outer) join queries [GANS87], construction of relational representations of hierarchies when some parent instances have no children [SCHO87], and instantiating objects from relational databases through views [LEE94].

For queries involving only the join operation (along with selections and projections, i.e., SPJ queries), reordering techniques are well known [SELI79, LAFO86] and lead to possibilities of many orders of magnitude improvement in performance. However, queries involving joins and outer joins are not totally reorderable and such improvements are not always possible. Galindo-Legaria and Rosenthal [GALI92a, GALI92b, ROSE90] have done pioneering work in the area of outer join reorderability, but their work makes the following restrictive assumptions:

1. Projection removes all duplicates.

   Most commercial RDBMSs support the data manipulation language SQL, featuring two kinds of projections: one that removes duplicates (SELECT DISTINCT) and another that does not (SELECT). Then, based upon the intended semantics, it is up to the database application to choose the kind of projection it needs.

2. There are no duplicates in base relations.

   All the identities involving the *GOJ* operator from [GALI92a, GALI92b, ROSE90] are applicable only if the relations do not have duplicates. In "real life" databases, duplicates are a fact of life and results that do not work for duplicate rows have limited applicability. [GALI94] suggests the use of a "system assigned unique identifier" for dealing with duplicates. However, it is worth noting here that whereas using such tuple identifiers is a straightforward idea for base relations, for more general relational expressions it requires careful formulation. Our paper provides these new results in Section 3. [KOO94] (in Korean) has identified six

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery.To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGMOD ' 95,San Jose , CA USA
© 1995 ACM 0-89791-731-6/95/0005..$3.50

cases involving base relations with duplicates that were previously not known to be reorderable. For one of these cases, a method is proposed to combine these base relations in a different order, using tuple identifiers. This approach does not seem to address reorderability of general relational expressions.

3. Outer join predicates are "simple", i.e., they refer to exactly two relations.

   SQL allows outer and full outer join predicates to reference more than two relations. Also, if an outer join predicate references attributes of a view or table expression, it is possible for these attributes to be from more than two different base relations. We call such predicates *complex*.

   [GALI94] has looked at complex predicates. But the approach there is aimed at providing a promising intuition for a new, canonical representation of outer join queries, and not for addressing the *efficient* reorderability of such queries. That work achieves reorderability for outer joins by considering any query result to be the glued together result of groups of joins, which themselves are freely reorderable. However, as acknowledged there, this has the potential of an exponential explosion in the number of joins, with even the same relation being used in multiple intermediate joins.

Our work solves these problems in a theoretically unified and efficient manner, while remaining amenable to easy inclusion in existing optimizers. The restriction on duplicates in relations is removed by the introduction of a novel binary operator (called $MGOJ$, or *Modified* GOJ) to replace the GOJ operator of [ROSE90]. The restriction that outer and full outer join predicates reference no more than two base relations is removed by introducing a reordering algorithm that uses the concept of *query hypergraphs* and *precedence sets*. These concepts are easily integrated into query optimizers that use methods such as dynamic programming, greedy algorithms, etc. to enumerate the various reorderings for the query.

Results in this paper require that binary operators (joins, outer joins, and full outer joins) be adjacent to each other in the query expression tree. A given user query may not have this adjacency property, since unary operators (selection, projection, and distincts-projection) can appear anywhere. We address this problem in [BHAR94a] where an algorithm is provided to "push-up/push-down" the unary operators in such a way that the binary operators appear adjacent to each other, without any intervening unary operators. Thus, without loss of generality, we can assume that binary operators are adjacent to each other.

The rest of this paper is organized as follows. In Section 1.2 we provide basic definitions of relevant relational concepts, needed to handle presence of duplicates and complex predicates in outer join query expressions. In Section 1.3 we introduce the query format expected by our algorithms, along with references to work that would transform any given query to this format. In Section 2 we introduce the concept of hypergraphs and association trees and motivate the use of hypergraphs for modeling such queries. In Section 3 we present the definition of the $MGOJ$ operator, and give the reordering identities for this operator. Then, by extending the strategy detailed in [GALI93], we provide the method for generating reordering schemes in Section 4 and motivate the proofs of correctness of these in Section 5. We summarize our work in Section 6.

1.2 Basic definitions

To model the presence of "duplicate" rows in relations, we use the concept of additional, *virtual* attributes. Besides the set of real attributes of a tuple, which is the same as the schema of the tuple in the traditional relational algebra, we use virtual attributes to provide unique *conceptual* tuple-ids to tuples [DAYA87]. The distinction between real and virtual attributes is made to emphasize the difference between the real attributes available for manipulation (to the DBMS user) and the virtual attributes used (by the DBMS) for bookkeeping only.

A *tuple* t is a mapping from a finite set of attributes, $R \cup V$, to a set of atomic (possibly null) values, where R is a non-empty set of *real attributes* and V is a non-empty set of *virtual attributes*, $R \cap V = \phi$, and the mapping t maps at least one virtual attribute $v \in V$ to a non-null value. For a set of attributes X, $t[X]$ represents the values associated with attributes X under the mapping t, where $X \subseteq R \cup V$ and $X \neq \phi$.

A *relation* r is a triple $\langle R, V, E \rangle$ where R is a non-empty set of real attributes, V is a non-empty set of virtual attributes, and E, the *extension* of relation r, is a set of tuples such that $(\forall t_1 \in E)(\forall t_2 \in E)(t_1 \neq t_2 \Rightarrow t_1[V] \neq t_2[V])$. $R \cup V$ is called the *schema* of relation r. Note that this definition does not preclude $t_1[R] = t_2[R]$ for any $t_1, t_2 \in E$, thus allowing "duplicates" in the sense of traditional relations.

A predicate p over a set of real attributes $sch(p)$, called the *schema* of p, is *null-intolerant* in attributes $R \subseteq sch(p)$ if p evaluates to FALSE for tuples that have null values on one or more attributes in R. In this paper we assume that all predicates are null-intolerant. This is a reasonable assumption in light of the fact that most predicates specified in data manipulation languages such as SQL are null-intolerant.

1.2.1 Algebraic operators

In addition to traditional relational operators, two new operators (π^c, and $MGOJ$, defined in Section 3) are included in our relational algebra.

Let $r = \langle R, V, E\rangle$, $r_1 = \langle R_1, V_1, E_1\rangle$ and $r_2 = \langle R_2, V_2, E_2\rangle$ denote relations such that $R_1 \cap R_2 = \phi$ and $V_1 \cap V_2 = \phi$.

The *projection*, $\pi_X(r)$, of relation r onto attributes X is the relation $\langle X, V, E'\rangle$ where $X \subseteq R$ and $E' = \{t \mid (\exists t' \in E)(t[X] = t'[X] \;\wedge t[V] = t'[V])\}$. π does not remove "duplicates" in the real attributes part of the source expression. *All* the virtual attributes of the source expression are included in the virtual schema of the result expression.

The *projection*^{c}, $\pi^c_{X_R X_V}(r)$, of relation r on attributes $X_R X_V$ is the relation $\langle X_R, X_V, E'\rangle$, where $X_R \subseteq R$, $X_V \subseteq V$ and $E' = \{t \mid (\exists t' \in E)(t[X_R] = t'[X_R] \;\wedge\; t[X_V] = t'[X_V])\}$. In contrast to π, π^c allows a *choice* in the selection of the virtual attributes from the source expression. This operation is needed for defining "modified generalized outer join" and other similar operations for relations with duplicates.

Example 1.1 Consider the relation $r = \langle A_1 A_2, v_1 v_2, E\rangle$, where $A_1 A_2$ are the real attributes and $v_1 v_2$ are the virtual attributes. Figure 1 shows the extension of the relation, and also the evaluation of the expressions $\pi_{A_1}(r)$ and $\pi^c_{A_1 v_1}(r)$. □

The *delta-projection*, $\delta_X(r)$, of relation r on attributes X is the relation $\langle X, V_{new}, E'\rangle$, where $X \subseteq R \cup V$, and tuples in E' are obtained by projecting out unique values on attributes X from tuples in E, and then assigning a new, unique value to the V_{new} attribute for each of these tuples. In other words, δ produces a result relation which has distinct values on the attributes X and a new virtual attribute.

The *selection*, $\sigma_p(r)$, of relation r on predicate p is the relation $\langle R, V, E'\rangle$, where $sch(p) \subseteq R$, and $E' = \{t \mid t \in E \;\wedge\; p(t)\}$.

The *union*, $r_1 \cup r_2$, of two union compatible relations r_1 and r_2 is the relation $\langle R, V, E_1 \cup E_2\rangle$. The *outer union*, $r_1 \uplus r_2$, of relations r_1 and r_2 is the relation $\langle R_1 \cup R_2, V_1 \cup V_2, E'\rangle$, where

$$\begin{aligned} E' = \{t | &(\exists t' \in E_1)(t[R_1 V_1] = t' \wedge \\ &(\forall A \in (R_2 - R_1) \cup (V_2 - V_1))(t[A] = \mathsf{NULL})) \\ \vee\; &(\exists t'' \in E_2)(t[R_2 V_2] = t'' \wedge \\ &(\forall A \in (R_1 - R_2) \cup (V_1 - V_2))(t[A] = \mathsf{NULL}))\}. \end{aligned}$$

Note, rows in $r_1 \uplus r_2$ are padded with nulls for attributes that are not present either in relation r_1 or in relation r_2.

We assume in the following definitions that if predicate p is associated with join/outer join/full outer join of relations r_1 and r_2 then $sch(p) \cap R_1 \neq \phi$, $sch(p) \cap R_2 \neq \phi$, and $sch(p) \subseteq R_1 \cup R_2$.

The *join*, $r_1 \overset{p}{\bowtie} r_2$, of relations r_1 and r_2 is the relation $\langle R_1 R_2, V_1 V_2, E'\rangle)$, where $E' = \{t \mid t \in (E_1 \times E_2) \wedge p(t)\}$. For the purpose of differentiation from outer joins, we also refer to joins as "inner" joins.

Notational convention: If $r = \langle R, V, E\rangle$ is a relation, we use $ext(r)$ to represent E, the extension part of r. In order to keep our definitions more readable, in the following few definitions we adopt the shorthand convention of simply writing $E_1 \bowtie E_2$ instead of $ext(r_1 \bowtie r_2)$, or $\pi_X(E_1 \bowtie E_2)$ instead of $ext(\pi_X(r_1 \bowtie r_2))$, etc.

The *left outer join*, $r_1 \xrightarrow{p} r_2$, of relations r_1 and r_2 is the relation $\langle R_1 R_2, V_1 V_2, E'\rangle$, where

$$E' = (E_1 \overset{p}{\bowtie} E_2) \;\uplus\; (E_1 \;-\; \pi^c_{R_1 V_1}(E_1 \overset{p}{\bowtie} E_2)).$$

Relation r_1 in the above definition is called the *preserved relation* and relation r_2 is called the *null supplying relation*. The *right outer join*, $r_1 \xleftarrow{p} r_2$, can similarly be defined in which r_1 is the null supplying relation and r_2 is the preserved relation. The *full outer join*, $r_1 \overset{p}{\leftrightarrow} r_2$, of relations r_1 and r_2 is the relation $\langle R_1 R_2, V_1 V_2, E'\rangle$, where

$$\begin{aligned} E' = (E_1 \overset{p}{\bowtie} E_2) \;&\uplus\; (E_1 \;-\; \pi^c_{R_1 V_1}(E_1 \overset{p}{\bowtie} E_2)) \\ &\uplus\; (E_2 \;-\; \pi^c_{R_2 V_2}(E_1 \overset{p}{\bowtie} E_2)). \end{aligned}$$

In the following, we use "outer join" to denote either of left outer join or right outer join. When referring to full outer joins, we explicitly use the word "full."

1.3 Simplification of queries

Our reordering algorithms assume that the query has only binary operators adjacent to each other in the query tree. Whereas this is not true for all queries, any query we consider can be transformed to an equivalent query for which this is true [BHAR94b, GALI92b]. This transformation is part of the process called "query simplification."

The first step in simplifying a given query is to remove selections by pushing them down to the base relations. This process may replace full outer joins by outer joins, and outer joins by joins, respectively. These simplifications are based on the following identities for expressions $X_i = \langle R_i, V_i, E_i\rangle$, for $1 \leq i \leq 2$:

$$\sigma_p(X_1 \rightarrow X_2) = \sigma_p(X_1 \bowtie X_2),$$

A_1	A_2	v_1	v_2
a	b	5	6
a	c	5	7
d	e	6	8

(a) r

A_1	v_1	v_2
a	5	6
a	5	7
d	6	8

(b)$\pi_{A_1}(r)$

A_1	v_1
a	5
d	6

(c) $\pi^c_{A_1 v_1}(r)$

Figure 1: Difference between π and π^c

$sch(p) \cap R_2 \neq \phi$, and p is null-intolerant in R_2
$\sigma_p(X_1 \leftrightarrow X_2) = \sigma_p(X_1 \rightarrow X_2)$,
$sch(p) \cap R_1 \neq \phi$, and p is null-intolerant in R_1

Next, projections (both π and δ) can be correctly moved up the query tree, shown for queries containing joins and outer join in [BHAR94a] and for queries without any outer joins in [DAYA87, PIRA92].

After removing selections/projections from a query, all "redundant" outer and full outer joins are recursively converted to joins and outer joins respectively, based on the null-intolerant properties of the predicates [GALI92b]. Henceforth, without loss of generality, we assume that a given query expression does not contain redundant (full) outer joins.

Example 1.2 Consider the expression $Q = r_1 \overset{p_{13}}{\bowtie} (r_2 \overset{p_{23}}{\leftrightarrow} r_3)$ where r_i, $1 \leq i \leq 3$, is a relation and p_{ij} is a predicate between relations r_i and r_j. The algorithm for eliminating "redundant" outer joins would convert this expression to $r_1 \overset{p_{13}}{\bowtie} (r_2 \overset{p_{23}}{\leftarrow} r_3)$ since the predicate p_{13} is null-intolerant and would not allow any rows having nulls for columns of r_3 in the subexpression $(r_2 \overset{p_{23}}{\leftrightarrow} r_3)$ to appear in the final result. □

2 Hypergraphs and association trees

Previous works on query optimization [PIRA92, ROSE90] have postulated and motivated the use of the query graphs. In an unrelated context, that of query evaluation by decomposition, [ULLM82] has postulated the use of hypergraphs. In this paper, we use the concept of hypergraphs, and association trees for these hypergraphs, for handling queries where (full) outer join predicates may reference more than two relations.

Before giving the formal definitions, let us describe the use of hypergraphs. We use a hypergraph to represent a query, where the set of nodes of the hypergraph correspond to relations referenced in the query, and a *hyperedge* represents an (inner, outer, or full outer) join operation involving a predicate between the sets of relations in its two *hypernodes*. A hyperedge that corresponds to an (inner, outer, or full outer) join operation, involving a predicate that references exactly two relations, is the same as an "edge" in traditional query graphs; on the other hand, more general hyperedges correspond to outer join operations involving predicates that reference more than two relations.

The use of hypergraphs to represent queries where outer join predicates may reference more than two relations is motivated as follows. Recall, the following well-known reordering identity for joins: $(X \overset{P_{XY}}{\bowtie} Y) \overset{P_{XZ} \wedge P_{YZ}}{\bowtie} Z = X \overset{P_{XY} \wedge P_{XZ}}{\bowtie} (Y \overset{P_{YZ}}{\bowtie} Z)$. However, its analog for outer joins is *not* true: $(X \overset{P_{XY}}{\rightarrow} Y) \overset{P_{XZ} \wedge P_{YZ}}{\rightarrow} Z \neq X \overset{P_{XY} \wedge P_{XZ}}{\rightarrow} (Y \overset{P_{YZ}}{\rightarrow} Z)$. In the traditional query graph approach, join predicates such as $P_{XY} \wedge P_{XZ}$ are represented by two edges, and identities such as the one above are used to "break-up" the predicate and still have an equivalent, reordered query. In graph terms, component pieces of different join predicates (each component being represented by a separate edge) are re-combined to have an equivalent, reordered query. Since the same does not hold true for outer join predicates, we propose the use of hyperedges and hypergraphs. In the hypergraph model, an *outer join* predicate such as $P_{XY} \wedge P_{XZ}$ is represented as *one* hyperedge, and is never subject to "breaking up" (on the other hand, a *join* predicate of the form $P_{XY} \wedge P_{XZ}$ would be represented by two hyperedges, since we do know the mathematical identity to break up and re-combine join predicates).

In our work, a query is represented by a ***hypergraph***, defined as follows.

Definition 2.1 A *hypergraph* H is defined to be the pair (V, E), where V is a non-empty set of *nodes* and E is the set of *hyperedges*, such that E is a mapping on non-empty subsets of V (i.e., $E : 2^V \rightarrow 2^V$). If hyperedge $e = \langle V_1, V_2 \rangle$, where $V_1, V_2 \in 2^V$, we call V_1 and V_2 *hypernodes*.

If $e = \langle V_1, V_2 \rangle$ and the cardinalities of both V_1 and V_2 are 1, we refer to hyperedge e as simply an *edge* and the hypernodes V_1 and V_2 as simply *nodes*.

In the context of query hypergraphs, we say a hyperedge is *directed* if it represents an outer join operation in the query. Further, we say a hyperedge is *bi-directed* if it

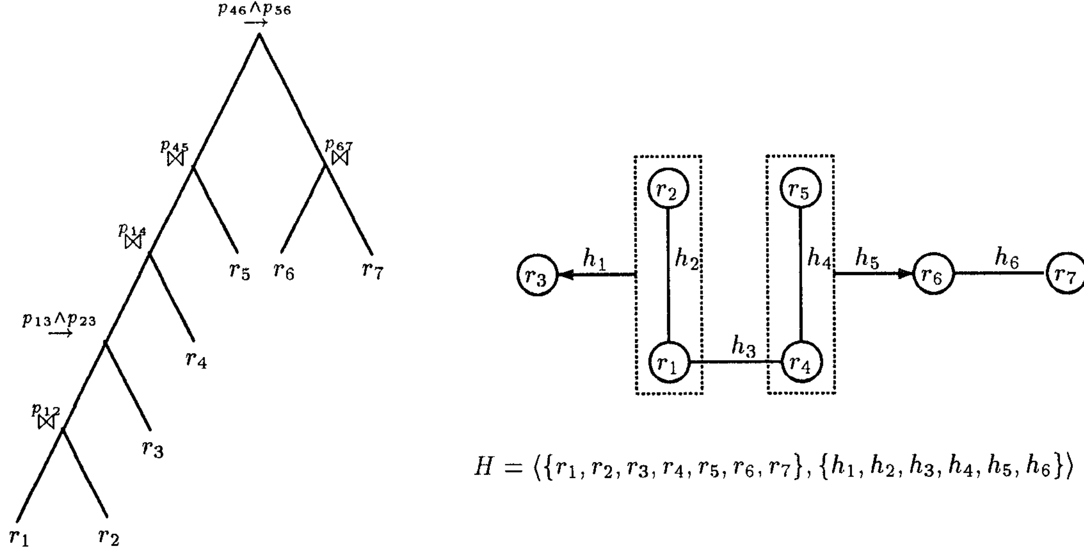


Figure 2: Expression tree and hypergraph for query from Example 2.1

represents a full outer join operation in the query. □

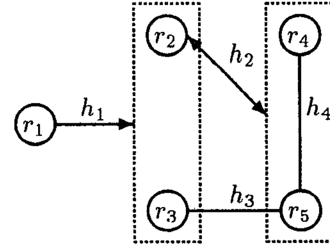
Example 2.1 Figure 2 shows the expression tree and hypergraph corresponding to the query $((((r_1 \overset{p_{12}}{\bowtie} r_2) \overset{p_{13}\wedge p_{23}}{\longrightarrow} r_3) \overset{p_{14}}{\bowtie} r_4) \overset{p_{45}}{\bowtie} r_5) \overset{p_{46}\wedge p_{56}}{\longrightarrow} (r_6 \overset{p_{67}}{\bowtie} r_7)$, where $r_i = \langle R_i, V_i, E_i \rangle$, for $1 \leq i \leq 7$, is a relation and p_{ij} is a predicate between r_i and r_j, for $1 \leq i, j \leq 7$. Dotted-line squares denote hypernodes. For this hypergraph the sets of nodes and hyperedges are $V = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$, and $E = \{h_1, h_2, h_3, h_4, h_5, h_6\}$, respectively, where $h_1 = \langle\{r_1, r_2\}, \{r_3\}\rangle$, $h_2 = \langle\{r_1\}, \{r_2\}\rangle$, $h_3 = \langle\{r_1\}, \{r_4\}\rangle$, $h_4 = \langle\{r_4\}, \{r_5\}\rangle$, $h_5 = \langle\{r_4, r_5\}, \{r_6\}\rangle$, and $h_6 = \langle\{r_6\}, \{r_7\}\rangle$. For directed hyperedge h_1, the hypernodes are $\{r_1, r_2\}$ and $\{r_3\}$; for h_5, the hypernodes are $\{r_4, r_5\}$ and $\{r_6\}$. Also, hyperedges h_2, h_3, h_4, and h_6 are edges in the classical, graph-theoretic sense. □

Definition 2.2 A *path* in a hypergraph $H = (V, E)$ from node v_1 to node v_n is an alternating sequence of nodes and hyperedges, $v_1 e_1 \cdots v_i e_i \cdots e_{n-1} v_n$ such that

1. $v_j \in V$, for $1 \leq j \leq n$ and $e_k \in E$ for $1 \leq k \leq (n-1)$.
2. $v_i \neq v_j$, for $1 \leq i, j \leq n$ and $i \neq j$.
3. $e_j \neq e_k$, for $1 \leq j, k \leq (n-1)$ and $j \neq k$.
4. If $e_j = \langle V_{j_1}, V_{j_2} \rangle$, where $V_{j_1}, V_{j_2} \subseteq V$, then $v_j \in V_{j_1}$ and $v_{j+1} \in V_{j_2}$, (or vice-versa), for $1 \leq j \leq (n-1)$.

We call v_1 the starting node of the path and v_n the ending node of the path. □

In the definition of path, above, if we relax Condition 2 to allow the starting and ending nodes to be the same, we get a *cycle*.



$H = \langle\{r_1, r_2, r_3, r_4, r_5\}, \{h_1, h_2, h_3, h_4\}\rangle$

Figure 3: Hypergraph for Example 2.2

Example 2.2 Consider the query $Q = r_1 \overset{p_{12}\wedge p_{13}}{\longrightarrow} (r_2 \overset{p_{24}\wedge p_{25}}{\longleftrightarrow} ((r_4 \overset{p_{45}}{\bowtie} r_5) \overset{p_{35}}{\bowtie} r_3))$, where $r_i = \langle R_i, V_i, E_i \rangle$, for $1 \leq i \leq 5$, is a relation and p_{ij} is a predicate between r_i and r_j, for $1 \leq i, j \leq 5$. Figure 3 shows the hypergraph for this query. This hypergraph has no cycles (even though there are 3 paths — $r_1 h_1 r_3 h_3 r_5$, $r_1 h_1 r_2 h_2 r_5$, and $r_1 h_1 r_2 h_2 r_4 h_4 r_5$ — between nodes r_1 and r_5). Note, for bi-directed hyperedge h_2, the hypernodes are $\{r_2\}$ and $\{r_4, r_5\}$. □

A query hypergraph is a useful abstraction for representing predicate connectivity, as well as the semantic information regarding outer join predicates that reference more than two relations and, therefore, *cannot*[1] be broken up into smaller components. For a given query, we can construct its hypergraph and then determine those reorderings of the operations that satisfy the hyperedge connectivity criterion. With this in mind, we define the notion of association trees for hypergraphs.

[1] Using currently known operators and identities

These enforce legal operand-pairing orders, based on the above mentioned properties of hypergraphs.

Definition 2.3 For a query hypergraph $H = (V, E)$, we define an *association tree* T for H as a binary tree in which:

1. $leaves(T) = V$, and no relation appears in more than one leaf[2].
2. for any subtree T_s of T, $H\mid_{leaves(T_s)}$[3] is connected.
3. for any subtree $T_s = (T_l.T_r)$ of T, if E_{T_s} denotes the set of all hyperedges in E that connect leaves of T_l with leaves of T_r, then $\forall e = \langle V_1, V_2\rangle \in E_{T_s}$ either $V_1 \subseteq leaves(T_l)$ and $V_2 \subseteq leaves(T_r)$, or $V_2 \subseteq leaves(T_l)$ and $V_1 \subseteq leaves(T_r)$. □

Of special interest in the above definition is Item 3, which enforces the requirement that the individual nodes within the hypernodes of a (bi-) directed hyperedge must be combined, before the two hypernodes are connected via the hyperedge. Thus, such association trees enforce a sort of "precedence" requirement.

It is easy to envision an enumeration algorithm for association trees, constructed bottom up from the given query hypergraph. The first step is to create one leaf trees. Then, at each subsequent step, two subtrees are combined to obtain a larger tree, only if all the conditions specified in the above definition of association trees are satisfied. This check is easily included in, say, the dynamic programming approach of existing RDBMS optimizers. [BHAR94a] presents details of such extensions to the dynamic programming enumeration technique.

Example 2.3 Consider query $Q = r_1 \overset{p_{13}}{\rightarrow} ((r_2 \overset{p_{23}}{\bowtie} r_3) \overset{p_{24}\wedge p_{34}}{\leftrightarrow} r_4)$, where r_i, $1 \leq i \leq 4$, is a base relation and p_{ij} is a predicate between relations r_i and r_j. Figure 4 shows all the valid association trees for the hypergraph of Q. Note, in all the association trees, r_2 and r_3 had to be combined with each other before either could be combined with r_4 — a consequence of the complex predicate $\overset{p_{24}\wedge p_{34}}{\leftrightarrow}$. □

Different hypergraph association trees correspond to different possible reorderings of the binary operators. Then, for a given association tree, we want to assign correct operators at the internal nodes of the tree so as to compute the same answer as that of the original query. As suggested in [DAYA87, ROSE90] some of the reorderings require the use of a "compensation operator". In the next section we give the definition of this operator.

[2] As in SQL and other optimization literature, we assume that relations occurring more than once in a query expression have been renamed to have unique names

[3] We use $H\mid_{leaves(T_s)}$ to denote the induced sub-hypergraph $(leaves(T_s), E')$, where $E' = \{\langle V_1, V_2\rangle \in E \mid V_1 \subseteq leaves(T_s) \wedge V_2 \subseteq leaves(T_s)\}$

3 Modified generalized outer join

Certain hypergraph association trees correspond to evaluation orders that require the use of the *Modified Generalized Outer Join*, or simply $MGOJ$, operator in order to compute the same result as the original query. First, we define this operator, contrasting it with earlier similar definitions [DAYA87, ROSE90] and explaining how these earlier definitions did not quite work for relations with duplicates. Then, in the next subsection we provide the association identities that employ the $MGOJ$ operator.

3.1 Definition of $MGOJ$

Consider a query expression of the form $r_1 \overset{p_{12}}{\rightarrow} (r_2 \overset{p_{23}}{\bowtie} r_3)$. Both Dayal [DAYA87] and Rosenthal & Galindo-Legaria [ROSE90] propose to reorder this query as $(r_1 \overset{p_{12}}{\rightarrow} r_2) GOJ[p_{23}, sch(r_1)] r_3$, where each of them defined their GOJ (generalized outer join) differently:

1. Dayal's generalized outer join operator of relation r_1 with relation r_2 preserving attributes X, written $r_1\ GOJ[p_{12}, X]\ r_2$, where $X \subseteq sch(r_1)$, is
$$(r_1 \overset{p_{12}}{\bowtie} r_2) \uplus \delta_X(r_1 - \pi_{sch(r_1)}(r_1 \overset{p_{12}}{\bowtie} r_2)).$$
2. Rosenthal & Galindo-Legaria redefined Dayal's generalized outer join operation as follows:
$$(r_1 \overset{p_{12}}{\bowtie} r_2) \uplus (\delta_X(r_1) - \delta_X(r_1 \overset{p_{12}}{\bowtie} r_2)).$$

Since both these definitions remove duplicates in the anti-join part, they cannot be used to reorder query expressions on relations with duplicates. This is illustrated in the following example.

Example 3.1 Consider the following relations, shown here with only their real attributes:

r_1:

A_1	A_2
a	b
d	e
d	e

r_2:

B_2	B_3
b	c

r_3:

C_3	C_4
c	f

For these relations with duplicates, the following identity, in terms of the binary operator GOJ defined in [ROSE90], does not hold:

$$r_1 \overset{p_{12}}{\rightarrow} (r_2 \overset{p_{23}}{\bowtie} r_3) = (r_1 \overset{p_{12}}{\rightarrow} r_2)\ GOJ[p_{23}, \text{sch}(r_1)]\ r_3,$$

where p_{12} is $A_2 = B_2$ and p_{23} is $B_3 = C_3$.

This is illustrated in the following evaluations of these two expressions:

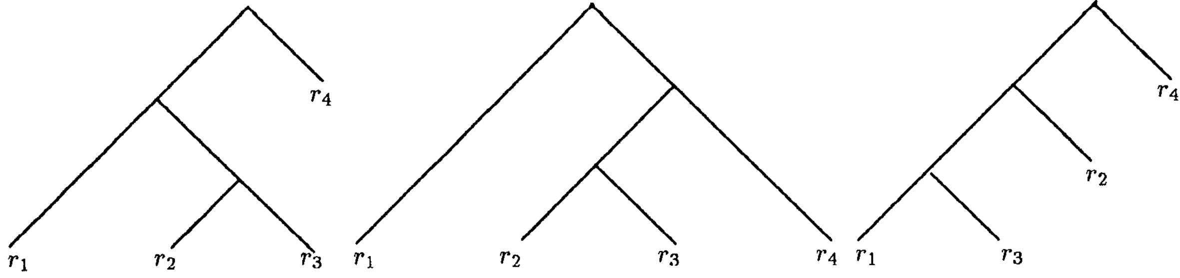


Figure 4: Association trees for Example 2.3

$r_1 \overset{p_{12}}{\rightarrow} (r_2 \overset{p_{23}}{\bowtie} r_3) =$

A_1	A_2	B_2	B_3	C_3	C_4
a	b	b	c	c	f
d	e	-	-	-	-
d	e	-	-	-	-

while,

$(r_1 \overset{p_{12}}{\rightarrow} r_2)\ GOJ[p_{23}, \text{sch}(r_1)]\ r_3 =$

A_1	A_2	B_2	B_3	C_3	C_4
a	b	b	c	c	f
d	e	-	-	-	-

This illustrates why Rosenthal & Galindo-Legaria's *GOJ* does not work for relations with duplicates. For this example, Dayal's *GOJ* operator also produces the same results. Our new operator, *MGOJ*, will handle duplicates in relations, as per SQL semantics. □

In the following definition, let $r_1 = \langle R_1, V_1, E_1 \rangle$ and $r_2 = \langle R_2, V_2, E_2 \rangle$ be two relations such that $R_1 \cap R_2 = \phi$ and $V_1 \cap V_2 = \phi$. Further, let $X_i = \langle R_{X_i}, V_{X_i}, E_{X_i} \rangle$, $1 \leq i \leq n$, and $Y_j = \langle R_{Y_j}, V_{Y_j}, E_{Y_j} \rangle$, $1 \leq j \leq m$, be relations such that $R_{X_i} \cap R_{X_k} = \phi = V_{X_i} \cap V_{X_k}$, $R_{Y_j} \cap R_{Y_s} = \phi = V_{Y_j} \cap V_{Y_s}$, $R_{X_i} \subseteq R_1$ and $R_{Y_j} \subseteq R_2$, where $i \neq k$, $j \neq s$, $1 \leq i, k \leq n$ and $1 \leq j, s \leq m$.

Definition 3.1 The *modified generalized outer join*, $r_1\ MGOJ[p, R_{X_1}, \ldots, R_{X_n}, R_{Y_1}, \ldots, R_{Y_m}]\ r_2$, of relations r_1 and r_2 while preserving attributes R_{X_i} and R_{Y_j} is the relation $\langle R_1 R_2, V_1 V_2, E' \rangle$, where $1 \leq i \leq n$, $1 \leq j \leq m$ and E' is given by:

$$E' = (E_1 \overset{p}{\bowtie} E_2)$$
$$\uplus_{i=1}^{n} \{t \mid t \in (\pi^c_{R_{X_i}, V_{X_i}}(E_1) - \pi^c_{R_{X_i}, V_{X_i}}(E_1 \overset{p}{\bowtie} E_2))\}$$
$$\uplus_{j=1}^{m} \{t \mid t \in (\pi^c_{R_{Y_j} V_{Y_j}}(E_2) - \pi^c_{R_{Y_j} V_{Y_j}}(E_1 \overset{p}{\bowtie} E_2))\} \square$$

For the query expression in Example 3.1, the use of this *MGOJ* operator in the expression $(r_1 \overset{p_{12}}{\rightarrow} r_2)\ MGOJ[p_{23}, \text{sch}(r_1)]\ r_3$ results in the same answer as the original expression $r_1 \overset{p_{12}}{\rightarrow} (r_2 \overset{p_{23}}{\bowtie} r_3)$.

3.2 *MGOJ* Identities for join/outer join association

Let $r_i = \langle R_i, V_i, E_i \rangle$, where $1 \leq i \leq 3$, be an expression, p_{ij} denote the predicate between expressions r_i and r_j. Then, we have the following association identities based on the definition of the *MGOJ* operator to replace the identities from [GALI92a] that use the *GOJ* operator:

- $r_1 \overset{p_{12}}{\rightarrow} (r_2 \overset{p_{23}}{\bowtie} r_3) = (r_1 \overset{p_{12}}{\rightarrow} r_2)\ MGOJ\ [p_{23}, R_1]\ r_3$
- $r_1 \overset{p_{12}}{\leftrightarrow} (r_2 \overset{p_{23}}{\bowtie} r_3) = (r_1 \overset{p_{12}}{\leftrightarrow} r_2)\ MGOJ\ [p_{23}, R_1]\ r_3$
- $r_1 \overset{p_{12}}{\leftrightarrow} (r_2 \overset{p_{23}}{\leftarrow} r_3) = (r_1 \overset{p_{12}}{\leftrightarrow} r_2)\ MGOJ\ [p_{23}, R_1, R_3]\ r_3$
- $r_1 \overset{p_{12}}{\leftrightarrow} (r_2\ MGOJ\ [p_{23}, X_2, X_3] r_3) =$
  $(r_1 \overset{p_{12}}{\leftrightarrow} r_2)\ MGOJ\ [p_{23}, R_1, X_2, X_3] r_3$,
  where $X_2 \subseteq R_2$ and $X_3 \subseteq R_3$.

3.3 Notes on *MGOJ*

1. In the definition of *MGOJ*, above, we have presented the preserved attributes, R_{X_i}, R_{Y_j}, in a manner which forces them to be the *entire* real schema of some (base) relation. Although the *MGOJ* operator can be defined without this restriction, we have chosen this approach to highlight the fact that this is the expected use in reordering expressions.

2. [GALI92b] also presents a definition of *GOJ* that preserves multiple attribute sets (as opposed to the single attribute set preservation illustrated in Example 3.1). However, that multiple attribute set definition is also geared toward relations without duplicates.

3. The *MGOJ* operator and its use in the reordering identities is correct even when the relations have tuples with all attributes as NULL (a definite possibility with SQL relations).

4. In an implementation, the *MGOJ* operator may be realized by extensions of existing outer join methods [DAYA87, PIRA93], which in themselves are modifications of join methods.

4 Generation of the optimal schedule

Recall, association trees are generated subject to the connectivity and predicate constraints of the hypergraph. Given association trees and the correct reordering identities, it is possible to assign operators to the internal nodes of the association trees to produce corresponding expression trees. Then, the optimal expression tree with the minimum cost can be selected from these expression trees.

Algorithm *conflictFreeAssignment* generates expression trees which do not employ the $MGOJ$ operator. This algorithm is especially useful for DBMS products which do not support materialization of $MGOJ$s. Algorithm *generalAssignment* explores a much larger alternative space of reorderings by including $MGOJ$s as one of the possible operators. These algorithms extend Galindo-Legaria's direct edge and generalized outer join selection tables in [GALI93]. These extensions take into account duplicates in relations and the property of hypergraphs. For reasons of space, we will not present the full details of these algorithms here. Rather, we will rely on examples and an outline of algorithm *conflictFreeAssignment.* Detailed write-ups for these extended algorithms are provided in [BHAR94b].

First, we motivate and provide the definition for the supporting concept of ***hypergraph conflict sets*** (for graphs, this definition is due to Galindo-Legaria [GALI93]). These sets are computed once from the query hypergraph and are then used for all association trees. For the following discussion, let Q be the given query expression and let H be the hypergraph corresponding to Q.

For a join edge e_0, the set of *closest conflicting outer joins*, denoted $ccoj(e_0)$, is the set $ccoj(e_0) = \{e \mid R_k \stackrel{e}{\rightarrow} R_l \stackrel{e_1}{\bowtie} \cdots \bowtie R_i \stackrel{e_0}{\bowtie} R_j \text{ is a path in } H\}$. As shown later, there can be at most one closest conflicting outer join hyperedge in $ccoj(e_0)$. Also, if removing a set of join edges $e_0, e_1, \cdots, e_n$ from H disconnects H into exactly two connected hypergraphs, then all these join edges have the same closest conflicting outer join hyperedge.

Definition 4.1 The *hypergraph conflict set* for a hyperedge, e_0, denoted by $conf(e_0)$, is defined as follows:

$$conf(e_0) = \begin{cases} \phi & \cdots \text{ if } e_0 \text{ is bi-directed} \\ \{e \mid R_i \stackrel{e_0}{\rightarrow} R_j \stackrel{e_1}{\sim} \cdots \stackrel{e_n}{\sim} R_k \stackrel{e}{\leftrightarrow} R_l \text{ is a path in } H\} & \cdots \text{ if } e_0 \text{ is directed} \\ \{e \mid R_i \stackrel{e_0}{\bowtie} R_j \stackrel{e_1}{\sim} \cdots \stackrel{e_n}{\sim} R_k \stackrel{e}{\leftrightarrow} R_l \text{ is a path in } H\} & \cdots \text{ if } e_0 \text{ is undirected and } ccoj(e_0) = \phi \\ \{e\} \cup conf(e) & \cdots \text{ if } e_0 \text{ is undirected and } ccoj(e_0) = \{e\} \end{cases}$$

where $\stackrel{e_i}{\sim}$ is a join or a left/right outer join. □

Intuitively, if hyperedge e_1 belongs to $conf(e_0)$, then the operator corresponding to e_1 cannot be a descendant of the operator for e_0 in any expression tree that does not use the $MGOJ$ operator.

In the above definitions, it is important to note the dependence on the definition of "path" for hypergraphs. This subtle, yet significant, difference from the graph model of [GALI92b] helps us extend the class of queries that can now be reordered.

Algorithm *conflictFreeAssignment:* (outline)
This algorithm works by recursively assigning an operator to the root of an given association (sub-) tree T, for the hypergraph H. It does so by considering the set of hyperedges in H that connect the nodes corresponding to the leaves in the left subtree, with the nodes corresponding to the leaves in the right subtree of T. (The Lemmas in Section 5 establish that for (bi-)directed hyperedges, this is a singleton set.) If the conflict sets for the hyperedges in this set are empty, then

- if the set contains undirected hyperedges, then assign the join operator to the root, along with the conjunctive predicate corresponding to all the join hyperedges in this set.
- otherwise, assign the operator and predicate to the root that corresponds to the single (full) outer join hyperedge in this set. □

Algorithm *conflictFreeAssignment* only works if the conflict sets for the edges being considered is empty. Otherwise, an $MGOJ$ operator is needed and algorithm *generalAssignment*[BHAR94b] is used.

[GALI93] has shown for the graph model, that certain association trees, called *conflict free*, do not require the use of GOJ operators during operator assignment. We extend this classification to the hypergraph model as follows (this is precisely the class of association trees for which algorithm *conflictFreeAssignment* can successfully assign operators).

Definition 4.2 An association tree T of a query hypergraph H is *conflict free* if for each subtree $T_s = T_l.T_r$ of T, whenever a hyperedge e_0 of H connects a node in $leaves(T_l)$ with one in $leaves(T_r)$, then $H\,|_{leaves(T_s)}$ does not contain any hyperedge in $conf(e_0)$. □

Example 4.1 Consider again the query $Q = r_1 \stackrel{p_{13}}{\leftrightarrow} ((r_2 \stackrel{p_{23}}{\bowtie} r_3) \stackrel{p_{24} \wedge p_{34}}{\longrightarrow} r_4)$, from Example 2.3. Figure 4 shows the association trees for this query. Figures 5(a) and (b) show the expression trees corresponding to conflict-free association trees for this query; the expression tree in Figure 5(c) is not conflict free and requires the use of the $MGOJ$ operator. □

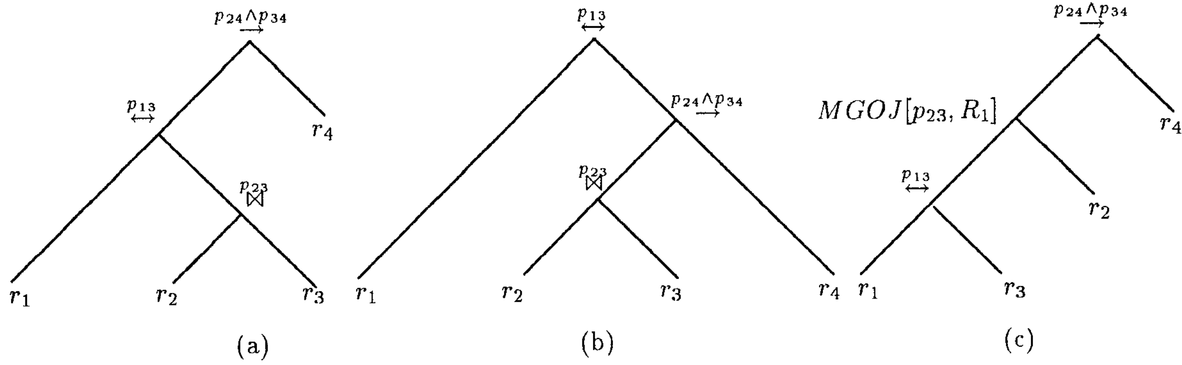


Figure 5: Expression trees for query in Example 4.1

Example 4.2 Consider again the query from Example 2.1. Figure 6(a) shows an expression trees without $MGOJ$, while Figure 6(b) shows an expression tree with the $MGOJ$ operator. □

5 Correctness of algorithms

All the expression trees in Figure 5 are equivalent. In order to prove this assertion, we present properties of query hypergraphs (as Lemmas; proofs, based on the approach of [GALI93], can be found in [BHAR94b]).

5.1 Properties of query hypergraphs

A query Q is called *simple* if it has been simplified via the transformations mentioned in Section 1.3 and every predicate is null-intolerant.

Lemma 1 *Let H be the query hypergraph of a simple query Q. There are no cycles in H that include directed or bi-directed hyperedges.*

Corollary 1 *A directed or bi-directed hyperedge e is in* **every** *path from a node within one hypernode of e to a node in the other hypernode of e.*

Corollary 2 *Let H be the hypergraph of a simple query Q, $E_J = \{e_1, \ldots, e_m\}$ be a set of join edges in H such that removing edges $\{e_1, \ldots, e_m\}$ from H partitions it into two connected subgraphs, and $V_J = \{v_i \mid v_i \in V_1^i$ or $v_i \in V_2^i, e_i = \langle V_1^i, V_2^i \rangle \in E_J\}$, $1 \leq i \leq m$. Then any path, P, connecting two vertices in V_J contains join edges only.*

Corollary 3 *Let H be the hypergraph of a simple query, h_0, h_n be directed hyperedges, $h_1, \cdots, h_{n-1}$ be undirected edges in H. Then, H has no path of the form $v_0 \xrightarrow{h_0} v_i \overset{h_1}{\bowtie} \cdots \overset{h_{n-1}}{\bowtie} v_j \xleftarrow{h_n} v_n$.*

It follows from Corollaries 2 and 3 that if removing $\{h_1, h_2, \ldots, h_m\}$ join edges disconnects H into two hypergraphs, then there is at most one path of the form $v_0 \xrightarrow{h_0} v_1 \overset{h_1}{\bowtie} \cdots \overset{h_m}{\bowtie} v_{m+1}$ in H and, consequently, $ccoj(h_1) = ccoj(h_2) = \cdots = ccoj(h_m)$. The above results establish the fact that there can be exactly one (bi-)directed edge that connects the nodes in the left subtree of an association tree with the nodes in the right subtree. This fact is key to the correctness of the operator assignment algorithms *conflictFreeAssignment* and *generalAssignment* [BHAR94b].

5.2 Proof of equivalence of expression trees

We establish the proof of equivalence of expression trees by first describing a canonical representation into which all simple queries can be transformed. Then we show that queries represented in this canonical form can be transformed to an equivalent query whose topology matches that of a given association tree and whose operators have been assigned using algorithm *conflictFree-assignment* or *generalAssignment*.

From Corollary 1, if (bi-)directed hyperedge e is on a path connecting two vertices of hypergraph H of a simple query, then it is on *every* path that connects the two vertices. This observation leads to the following definition.

Definition 5.1 For a (bi-)directed hyperedege e, we define $precedes(e)$ to be the set $\{e_1 \mid e_1$ is (bi-)directed and e_1 is in the paths connecting the nodes within either hypernode of $e\}$. □

Intuitively, if a (bi-)directed hyperedge e_0 belongs to $precedes(e)$ then the operator corresponding to e_0 must be a descendent of the operator corresponding to e in every expression tree for the query represented by hypergraph H.

Definition 5.2 For a (bi-)directed hyperedge e in a query hypergraph H we define the extended-precedes set, denoted $ext_precedes(e)$, as follows:

1. A (bi-)directed hyperedge $e_1 \in ext_precedes(e)$ if $e_1 \in precedes(e)$.
2. If bi-directed hyperedge $e_1 \in ext_precedes(e)$ and $e_1 \in conf(e_2)$, where e_2 is a directed hyperedge, then all hyperedges in $ext_precedes(e_2)$ are in $ext_precedes(e)$. □

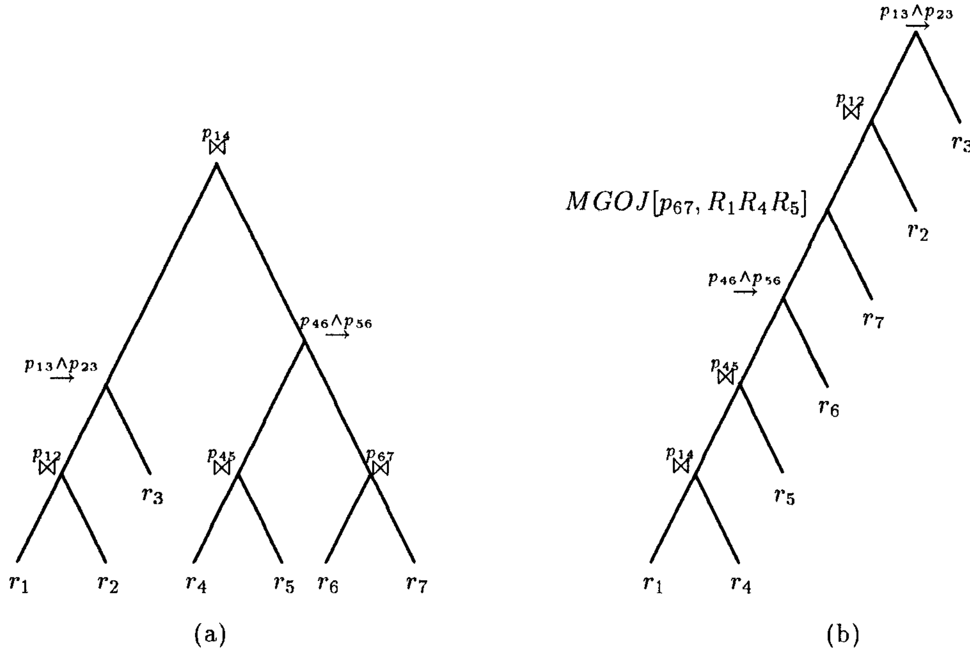


Figure 6: Reordering possibilities for query from Example 2.1

Example 5.1 For the hypergraph shown in Figure 3, $ext_precedes(h_1) = precedes(h_1) = \{h_2\}$, since the directed hyperedge h_2 is part of the path $r_2h_2r_5h_3r_3$, connecting nodes r_2 and r_3 within hypernode $\{r_2, r_3\}$ of h_1. □

Extended-precedes sets capture the required precedence of operations, as well as the notion of conflicts for operations, in the sense defined in Section 4. For hyperedge e, $precedes(e) \subseteq ext_precedes(e)$. Intuitively, if a hyperedge e_0 belongs to $ext_precedes(e) - precedes(e)$ then e cannot be a descendent of e_0 in any expression tree that does not use $MGOJ$s. These extended-precedes sets can be used to provide a canonical representation for a simple query, as follows.

Definition 5.3 An association tree T of query hypergraph H is called *partially monotonic* if for each subtree $T_s = (T_l.T_r)$ the following conditions hold:

1. If undirected edges connect $leaves(T_l)$ with $leaves(T_r)$ in H, then $H\ |_{leaves(T_s)}$ contains only undirected edges.
2. If a directed hyperedge e connects a node in $leaves(T_l)$ with a node in $leaves(T_r)$ in H, then $H\ |_{leaves(T_s)}$ contains undirected edges, directed hyperedges, or bi-directed hyperedges that are in $ext_precedes(e)$. □

Partial monotonic trees represent evaluation orders in which joins are done first, outer joins are done next, and full outer joins are done last, with the exception that a full outer join will be done *before* an outer join *if* there is a precedence requirement, as captured by the notion of precedes sets. For partial monotonicity, we only need to worry about $ext_precedes$ sets for directed hyperedges. Furthermore, we are only interested in the bi-directed hyperedges contained in the $ext_precedes$ sets for these directed hyperedges.

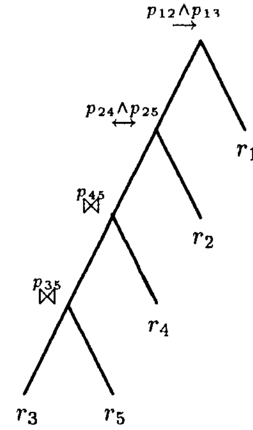


Figure 7: Partial monotonic association/expression tree for query in Example 2.2

Example 5.2 Figure 7 shows an expression tree corresponding to a partial monotonic association tree for the query in Example 2.2. Notice, in this evaluation order, joins are done first, outer joins are done next, and full outer joins are done last, with the exception that a full outer join is done before an outer join if there is a precedence requirement. □

Lemma 2 *Any simple query Q can be transformed into a simple partially monotonic query.* □

The proof is based on the argument that using the reassociation identities we can always change the order of operations as long as the *ext_precedes* set sequencing is not violated. Then, the transformed equivalent expression is simple partially monotonic. With this in place, it is easy to establish the correctness of the hypergraph model and the precedence sets by observing that:

1. Any simple partially monotonic query can be transformed into an equivalent query whose topology matches a conflict free association tree and whose operators have been assigned using algorithm *conflictFreeAssignment*.

2. Any simple partially monotonic query can be transformed into an equivalent query whose topology matches an association tree and whose operators have been assigned using algorithm *generalAssignment*.

This leads us to the following observation:

Theorem 1 *Let Q be a simple query and T be an association tree of query hypergraph(Q). We can apply association identities to Q to obtain Q′ such that Q′ computes the same answer as Q and the expression tree for Q′ has the same topology as T.* □

This theorem establishes the fact that a query having complex outer join predicates can be reordered to produce an equivalent query in which the order of evaluation of binary operators is different, provided the topology for the reordered expression tree corresponds to a legal association tree for the hypergraph of the query.

6 Summary

Queries with outer join predicates that refer to more than two relations were executed in an "as written" manner in the past, largely due to the fact that there were no practically usable results available to reorder operations in such queries. In this paper we have presented results to enumerate other plans for such complex queries. Since traditional query graph models are not sufficiently powerful, we model such queries by hypergraphs, thereby making it possible to explore an expanded state space from which the optimal plan can be selected. Hypergraphs provide a useful abstraction for employing known operator association identities along with the ones presented in this paper. These new identities employ a new operator, *MGOJ*. The *MGOJ* operator can handle duplicates in relations, thereby making the results useful for commercial SQL RDBMS products. Combined, these two aspects make our results apply to a whole new, large class of queries that would previously have been executed in an "as written" manner only. Our results are *easily* implementable in *existing* RDBMS optimizers that use, for example, dynamic programming or greedy algorithms. Finally, our initial experiments indicate that the methods proposed in this paper can improve the performance of complex queries by orders of magnitude.

7 Acknowledgements

We would like to thank César Galindo-Legaria, Guy Lohman, V. Srinivasan, Yun Wang, and the anonymous referees for their comments on earlier drafts. Thanks also to Harry Campbell, Atul Chadha, Josephine Cheng, Doug DeGraffenreid, Gopal Krishnan, and Jay Tang for their help and support.

References

[BHAR94a] Bhargava, G., Goel, P. and Iyer, B., "An algorithm for complete view merging and reordering of SQL queries," *Working draft, to be published.*

[BHAR94b] Bhargava, G., Goel, P. and Iyer, B., "Reordering of complex queries involving joins and outer joins," *IBM Technical Report TR03.567*, July 1994.

[CHEN90] Chen, A.L.P., "Outerjoin optimization in multidatabase systems," *2nd International Symposium on Databases in Parallel and Distributed Systems*, 1990

[DATE83] Date, C. J., "The outer join," *Proceedings of the second International Conference on Databases*, Cambridge, England, September 1983.

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

[GANS87] Ganski, R., and Wong, H.K.T., "Optimization of nested SQL queries revisited," *SIGMOD*, 1987.

[GALI92a] Galindo-Legaria, C., and Rosenthal, A., "How to extend a conventional optimizer to handle one- and two-sided outerjoin," *Proceedings of Data Engineering*, pp. 402-409, 1992.

[GALI92b] Galindo-Legaria, C. A., "Algebraic optimization of outerjoin queries," Ph.D. dissertation, Dept. of Applied Science, Harvard University, Cambridge, 1992.

[GALI93] Galindo-Legaria, C., and Rosenthal, A., "Outerjoin simplification and reordering for query optimization," *Manuscript dated July 26, 1993 - Submitted for publication.*

[GALI94] Galindo-Legaria, C. A., "Outerjoins as disjunctions," *SIGMOD*, pp. 348-358, 1994.

[KOO94] Koo, H.S. and Bae, H.Y., "A partition method for query optimization on outer join," *Journal of the Korea Information Science Society*, Vol.21, No.5, pp.931-43, May 1994.

[LAFO86] Lafortune, S. and Wong, E., "A state transition model for distributed query processing," *ACM Transactions on Database Systems*, Vol 11, No. 3, pp. 294-322, Sept. 1986.

[LEE94] Lee, B.S. and Wiederhold, G., "Outer joins and filters for instantiating objects from relational databases through views," *IEEE Transactions on Knowledge and Data Engineering*, pp. 108-119, Vol 6, No. 1, Feb 1994.

[PIRA92] Pirahesh, H., Hellerstein, J. M. and Hasan, W., "Extensible/rule based query rewrite optimization in Starburst," *SIGMOD*, pp. 39-48, San Diego, Ca., June 1992.

[PIRA93] Pirahesh, H., Mohan, C., Cheng, J., "Sequential and parallel algorithms for unified execution of outer join and subqueries," Technical Report, IBM Almaden Research Center, San Jose, June 1993.

[ROSE90] Rosenthal, A. and Galindo-Legaria, C., "Query graphs, implementing trees, and freely-reorderable outerjoins," *SIGMOD*, pp. 291-299, 1990.

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

[SELI79] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G., "Access path selection in a relational database management system," *SIGMOD*, pp. 23-34, 1979.

[ULLM82] Ullman, J.D. "Principles of database systems," 2nd Edition, Computer Science Press, 1982.