Using EELs, a Practical Approach to Outerjoin and Antijoin Reordering

Jun Rao Bruce Lindsay Guy Lohman Hamid Pirahesh David Simmen
IBM Almaden Research Center
junrao,bruce,lohman,pirahesh,simmen@almaden.ibm.com

Abstract

Outerjoins and antijoins are two important classes of joins in database systems. Reordering outerjoins and antijoins with inner joins is challenging because not all the join orders preserve the semantics of the original query. Previous work didn't consider antijoins and was restricted to a limited class of queries.

We consider using a conventional bottom-up optimizer to reorder different types of joins. We propose extending each join predicate's eligibility list, which contains all the tables referenced in the predicate. An extended eligibility list (EEL) includes all the tables needed by a predicate to preserve the semantics of the original query. We describe an algorithm that can set up the EELs properly in a bottom-up traversal of the original operator tree. A conventional join optimizer is then modified to check the EELs when generating subplans. Our approach handles antijoin and can resolve many practical issues. It is now being implemented in an upcoming release of IBM's Universal Database Server for Unix, Windows and OS/2.

1. Introduction

Relational join combines information from two base tables by creating pairs of matching tuples. Tuples without any matches are simply discarded. We will refer to this kind of joins as *inner joins* in the rest of the paper. In addition to inner joins, there are two other types of joins commonly seen in relational database systems, namely *outerjoins* and *antijoins*.

Outerjoin [Cod79] is a modification of inner join that preserves all information from one or both of its arguments. Outerjoins can be further categorized into *left*, *right* (single-sided) or *full* (two-sided) outerjoin, depending on which side needs to be preserved. For example, the following SQL query will return all the department names and employees within each department. For those departments without employees, the department names are listed with the employee name set to NULL.

```
SELECT   department.dname, employee.ename
FROM     department LEFT JOIN employee
         ON department.no=employee.dno
```

Outerjoins are important because they are frequently used in the following traditional applications [GLR97]: (a) certain OLAP queries where we need to preserve tuples from the fact table with unknown (or missing) dimensional values; (b) constructing *hierarchical views* that preserve objects with no children and (c) queries generated by external tools and query rewriting unit. XML is becoming the standard of data exchange. Outerjoins are quite useful for expressing XML paths in SQL and exporting relational data into XML documents [RLL+00].

Antijoin is useful for handling negated nested queries. Straightforward evaluation of those queries would require using the nested iteration method, which may be very inefficient. [Kim82] proposed to transform negated nested queries into antijoins. Since join methods other than the nested loops join could potentially be used, this transformation gives the optimizer more freedom. Example 1.1 shows such a transformation. An antijoin preserves a tuple from the outer relation if there is no match from the inner relation. Otherwise, the tuple is discarded. Antijoin transformations happen a lot in commercial systems. For example, negated nested queries are often used to maintain referential integrity.

Example 1.1: list all departments with no employees.
Original query:

```
SELECT   department.dname
FROM     department
WHERE    NOT EXISTS
         (SELECT * FROM employee
         WHERE department.no=employee.dno)
```

After transformation:

```
SELECT   department.dname
FROM     department ANTIJOIN employee
         ON department.no=employee.dno
```

When there are only inner joins in a query, a query optimizer considers all possible join orders and selects the cheapest execution plan. Changing the order of

1063-6382/01 $10.00 © 2001 IEEE

join evaluation is a powerful optimization technique and can improve execution time by orders of magnitude. However, when outerjoins and/or antijoins are present in addition to inner joins, changing the order of evaluation is complicated. This is because these three types of joins are not always associative with each other (more specific examples in Section 4.1). So, not all orders will give the same answer as the original query, unless it is given special consideration.

The problem of outerjoin reordering has been studied in [RGL90, GLR92, BGI95, GLR97], with [GLR97] being the most comprehensive. In [GLR97], the authors identify a special class of query called *simple* join/outerjoin queries. A simple query has the property that its query graph (without specifying the join orders) unambiguously determines the semantics of the query. A conflicting set for each join predicate p is then computed through some graph analysis, which includes all join predicates that conflict with p. The information stored in the conflicting set can be used to form proper join orders in a conventional bottom-up join optimizer.

There are two limitations in the approach used in [GLR97]. First of all, it provides solutions to simple queries only. While simple queries are an important class of query, there are many real world queries that are not simple. For example, predicates with more than one conjunct, predicates referencing more than two tables, and Cartesian products are not allowed in simple queries. This limits the application of the technique in commercial systems. As a matter of fact, many commercial database systems either evaluate outerjoin queries in the order specified by the user or only allow limited reordering. Second, reordering with the presence of antijoins is not considered in [GLR97].

In this paper, we propose a new reordering approach that can handle a more general class of queries and more types of joins. We consider how to do reordering in a conventional bottom-up optimizer using dynamic programming [SAC+79]. Commercial systems such as DB2 [Cor99] associate with each join predicate an *eligibility list*. Normally, the eligibility list of a join predicate includes only those tables that are referenced in this join predicate. During the bottom-up join enumeration, the optimizer checks if there is a join predicate p whose eligibility list is a subset of all the tables in the two subplans to be merged. If so, the two subplans are combined using p. Otherwise, the two subplans can't be joined (unless we want to introduce a Cartesian product).

To incorporate reordering with outerjoins and antijoins, we extend the normal eligibility list. For each join predicate, we calculate an *extended eligibility list* (referred to as *EEL*), which includes additional tables referenced in those conflicting join predicates. Intuitively, an EEL gives all the tables needed by a predicate to preserve the semantics of the original query. EELs are precomputed during one traversal of the operator tree of the original query. Such an extension is transparent to the optimizer. Instead of the normal eligibility list, the optimizer now checks the EEL for each join predicate. We distinguish two kinds of approaches to the reordering problem. The first approach only allows join orders that are valid under associative rules. The optimizer simply refuses to combine subplans using a join predicate whose EEL is not covered. We refer to this approach as *without compensation*. The second approach is more aggressive. When a join predicate p's EEL is not covered, it allows two subplans to be combined using p as long as p's normal eligibility list is covered. Some compensation work must be done later to correct the join result. We refer to this approach as *with compensation*.

Our "without compensation" approach is logically equivalent to that used in [GLR97]. However, our framework can be extended to resolve many practical issues. This is because EELs exploit the join order information in the original operator tree. The query graph, on the other hand, has lost all the orders specified in the original query. Additionally, our framework allows reordering with the presence of antijoins and can be used to support other types of joins as well.

Our "with compensation" approach is through *nullification* and *best match*. Although similar to generalized outerjoins [GLR97], our approach allows multiple compensations to be merged and to be done at any time, whereas generalized outerjoins must always perform the compensation immediately. Thus, our compensation approach considers a superset of plans of generalized outerjoins. By separating the two approaches, we provide a framework that can be seamlessly incorporated into an existing system in two phases.

The rest of the paper is organized as follows: Section 2 introduces the notations and assumptions used in this paper. We describe related work in Section 3. Section 4 and Section 5 describes our reordering approach without compensation and with compensation respectively. We discuss the extension needed in an optimizer in Section 6. We talk about our future work in Section 7 and conclude in Section 8.

2. Notation and Assumptions

Definition 2.1: The *inner join*, denoted by $R \overset{p_{rs}}{\bowtie} S$, is defined as $\{(r,s)|r\in R, s\in S$ and $p_{rs}(r,s)$ is true$\}$. The subscript in the predicate represents all the tables referenced in the predicate.

Definition 2.2: The *single-sided outerjoin*, denoted by $R \xrightarrow{p_{rs}} S$, is defined as $\{(r,s)|r\in R, s\in S$ and $p_{rs}(r,s)$

is true}∪{(r,null)|r∈R and no tuple of S satisfies $p_{rs}(r,s)$}. We refer to R as the *preserving* side and S as the *null-producing* side.

Definition 2.3: The *antijoin*, denoted by $R \overset{p_{rs}}{\triangleright} S$, is defined as {r∈R|no tuple of S satisfies $p_{rs}(r,s)$}. Again, we refer to R as the *preserving* side and S as the *null-producing* side (although the nulls from S are not in the output).

Definition 2.4: The *two-sided outerjoin*, denoted by $R \overset{p_{rs}}{\longleftrightarrow} S$, is defined as $(R \overset{p_{rs}}{\longrightarrow} S) \cup ((R \overset{p_{rs}}{\triangleleft} S) \times \{null\})$.

Definition 2.5: A predicate p is *null-intolerant* if it evaluates to false whenever there is a null value in any of the attributes it references. Otherwise, the predicate is called *null-tolerant*. For example, $(R.a = S.a)$ is null-intolerant while $(R.a = 5$ or $R.b = 6)$ is null-tolerant (since a null value in $R.a$ doesn't necessarily disqualify that tuple) . A Cartesian product introduces a predicate that's always true and thus is also null-tolerant.

Definition 2.6: A *normal eligibility list* (NEL) of a predicate p includes all the tables (referred to as TABs) that p references. The NEL is used by an optimizer to determine whether a predicate can be applied. For example, the NEL for predicate $R.a = S.a$ is $\{R, S\}$.

Definition 2.7: We call predicates referencing two TABs *binary predicates*. Predicates that reference more than two TABs are called *hyper-predicates*. We assume that predicates are broken into conjuncts (which are ANDed together).

We also use the relational operator $\sigma_p R$ for applying predicate p on relation R and $\pi_a R$ for duplicate-removing projection of attribute a from relation R.

Expressing an outerjoin query requires the user to specify the join order. The following query specifies that table B be joined with table C first, then with table A.

```
SELECT   *
FROM     A LEFT JOIN
         (B LEFT JOIN C ON B.b=C.b)
         ON A.a=B.a
```

The order of an antijoin is determined at query rewriting time. Normally, the antijoin is placed as the last join between the inner and the outer query block.

We assume there is only one query block after query rewriting. Queries with multiple query blocks won't change our results since normally the optimizer will be invoked on each block individually. We assume that after parsing and query rewriting, an *operator tree* is generated for the original query. An operator tree consists of a number of binary joins (including inner join, outerjoin and antijoin) with a specific order. Although this order gives the correct result, it may not be optimal in terms of execution time. It's the optimizer's responsibility to find the optimal (valid) plan, which is also an operator tree.

Semantically, predicates in the WHERE clause should be applied after all joins. When there are outerjoins and antijoins, local predicates can't always be pushed down as is the case for inner joins. We assume these predicates are initially placed in a *filter* operator at the top of the original operator tree. We will discuss how to handle those predicates in Section 4.3.

As has been considered in [GLR97], null-intolerant predicates can simplify a query. For example, if p_s is null-intolerant, the following rule holds:

$$(0)\ \sigma_{p_s}(R \overset{p_{rs}}{\longrightarrow} S) = \sigma_{p_s}(R \overset{p_{rs}}{\bowtie} S)$$

Simplification is always a good thing to do since it makes operations cheaper. We assume all possible simplifications have been done on the original operator tree using an algorithm described in [GLR97].

Single-sided outerjoins are much more common in real world queries and can be implemented by more join methods. As a matter of fact, some commercial systems [Cor99] translate two-sided outerjoins into the union of a single-sided outerjoin and an antijoin. For the sake of simplicity, our presentation focuses on single-sided outerjoins. However, our result still holds for two-sided outerjoins. We will elaborate on this in Section 7.

3. Related Work

[Day83] gives some initial rules on valid evaluation orders for joins and one-sided outerjoins. [Day87] points out that one-sided outerjoin—and other operators useful for nested subqueries—can be implemented by minor modifications to join algorithms.

Galindo-Legaria and Rosenthal [GL92, GLR92, GLR97] have done pioneering work in the area of outerjoin simplification and reorderability. However, their work didn't handle antijoin and was restricted to *simple queries*. A simple query assumes the following:

- all the simplifications have been done.
- all the predicates are null-intolerant.
- all outerjoin predicates have only one conjunct and are binary.
- there are no Cartesian products.

Real world queries are complex and the last three assumptions may not hold. In later sections, we will discuss how to loose these restrictions in our approach.

[BGI95] adapted the previous framework to deal with predicates with more than one conjunct. Basically, such a predicate is treated as a single predicate so that the conjuncts within it can't be broken up.

4. Reordering without Compensation

[RGL90] introduced a complete set of valid and invalid reordering rules involving outerjoins, antijoins and inner joins. We refer to them as *associative rules* and *conflicting rules* respectively. These rules are summarized in [RLL+00]. Our approach is to extend the normal eligibility list of each predicate with some additional information and use it to determine the correct join order. Section 4.1 discusses how to extend the eligibility list when there are outerjoins and inner joins only. Section 4.2 extends the idea to include antijoins. In both sections, we assume that we are dealing with simple queries and we don't consider introducing Cartesian products. We will address various practical issues after loosening those restrictions in Section 4.3.

4.1. Reordering Outerjoins and Inner Joins

In contrast to inner joins, not all the join orders are valid when outerjoins are involved. Let's take a look at a more detailed example. Suppose the original query is $R \overset{R.a=S.a}{\longrightarrow} (S \overset{S.b=T.b}{\bowtie} T)$ and R, S and T have tuples shown in Figure 1 (tid gives the unique tuple ID). The result of the query is shown in Figure 2(a) (we only show the tids in the result). However, evaluating the query in another order will result in a different answer set as shown in Figure 2(b).

R

tid	a
r1	1
r2	3
r3	5

S

tid	a	b
s1	1	1
s2	1	2
s3	3	3
s4	3	4

T

tid	b
t1	1

Figure 1. Contents of table R, S and T

r1	s1	t1
r2	-	-
r3	-	-

(a) $R \longrightarrow (S \bowtie T)$

r1	s1	t1

(b) $(R \longrightarrow S) \bowtie T$

Figure 2. Query Results (- is a null value)

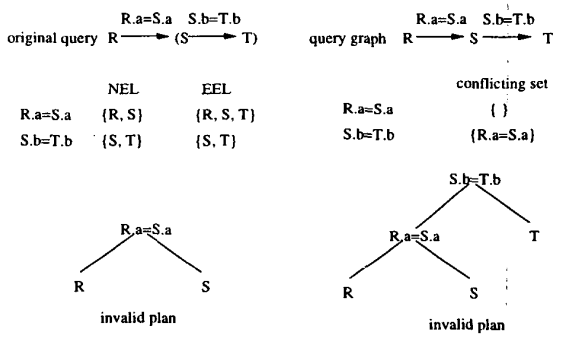
The reason is because an outerjoin conflicts with an inner join in its null-producing side as specified by the following conflicting rule:

$$(1)\ R \overset{p_{rs}}{\longrightarrow} (S \overset{p_{st}}{\bowtie} T) \neq (R \overset{p_{rs}}{\longrightarrow} S) \overset{p_{st}}{\bowtie} T$$

By extending the normal eligibility list, we can capture this kind of conflict and provide the information to be used by the optimizer. We now give the formal definition of EEL.

Definition 4.1: An *extended eligibility list* (EEL) of a predicate p includes all the tables needed as the input to p in order to get the correct answer.

In the above example, the EEL of predicate $R.a = S.a$ (call it p_{rs}) should be set to $\{R, S, T\}$ (its NEL is $\{R, S\}$) since T is referenced in the conflicting inner join predicate $S.b = T.b$ (call it p_{st}). The EEL of p_{rs} means that in order to apply the predicate p_{rs}, the subplans have to include all three tables. The EEL of the inner join predicate p_{st}, on the other hand, is the same as its NEL. If we require that two subplans be joined only if all the tables in the EEL of the join predicate exist in the two subplans, the optimizer won't be able to form a plan including R and S only. This is shown in Figure 3(a).



(a) Using EEL (b) Using Conflicting Set

Figure 3. Comparison between EEL and Conflicting Set

We now illustrate the approach used in [GLR97] using the same example. It first generates a query graph of the original query. The query graph doesn't contain any information about the original join order (remember for simple queries, the query graph uniquely identifies the query semantics). Then a *conflicting set* for each predicate is computed using the query graph. A conflicting set of an inner join predicate p includes only the first outerjoin predicate in a path connected to p with the arrow pointing inward. So the conflicting set of p_{st} equals to $\{p_{rs}\}$. The conflicting set of a single-sided outerjoin predicate is empty when there are no two-sided outerjoins (otherwise it will include some two-sided outerjoin predicates). The corresponding "without compensation" approach will prevent two subplans from being joined using predicate q if any join predicate in the two subplans is in q's conflicting set. Figure 3(b) shows this approach.

In comparison, EELs represent conflicts using table references, while conflicting sets represent conflicts using predicates. Although logically equivalent, there are many benefits of using the EEL's representation: (1) The idea of using the EEL can be easily extended to antijoins (discussed in Section 4.2); (2) Other practical issues such as predicates with multiple conjuncts, hyper-predicates and allowing Cartesian products can also be handled efficiently using EELs (discussed in Section 4.3); (3) The computation of conflicting sets requires some graph analysis and can be complicated.

On the other hand, the computation of EELs requires only one traversal of the original operator tree (as we will see shortly). Although such computation takes much less time than the evaluation of the query, the implementation of EEL will be much easier.

When there are only outerjoins and inner joins, Rule (1) is the only conflicting rule for any query. Rule (1) means that an outerjoin can't be pushed through any inner joins in the null-producing side. Intuitively, we should extend the eligibility list of a predicate p with TABs referenced in inner joins conflicting with p.

Now, we describe our algorithm to set up the EELs. Initially, the EEL of each predicate is set to its NEL, i.e., the TABs referenced in the predicate. During the bottom-up traversal of the original operator tree, we keep for each TAB a *companion set*. The companion set for TAB t has t itself initially. During the traversal, if we see an inner join predicate, we union the companion set of TABs references in the predicate. The companion set of each TAB in the unioned result (call it w) is set to be w. The companion set of a TAB t essentially includes all the TABs that are linked to t (directly or indirectly) through inner join predicates. On the other hand, if we see an outerjoin predicate p, we update its EEL by adding elements in the companion set of TABs referenced in p that are from the null-producing side of p. Once the EELs have been set, we can generate query plans using an "adapted" bottom-up join optimizer, which requires the EEL of each join predicate be covered in the two subplans. The details of the algorithms and the proof can be found in [RLL+00].

We illustrate the algorithm using some examples. In Example 4.1, the two outerjoins are associative. So the EELs of the two join predicates are the same as the NELs. Thus join order $((R, S), T)$ is valid. If R is much smaller than the other two tables, this join order could be much cheaper than the original order. In Example 4.2, since none of the inner joins can be evaluated before the outerjoin, the outerjoin predicate will include all four tables in its EEL.

Example 4.1: $R \xrightarrow{p_{rs}} (S \xrightarrow{p_{st}} T)$

companion set				p_{rs}	p_{st}
R	S	T	NEL	$\{R,S\}$	$\{S,T\}$
$\{R\}$	$\{S\}$	$\{T\}$	EEL	$\{R,S\}$	$\{S,T\}$

Additional valid join order: $((R, S), T)$

Example 4.2: $R \xrightarrow{p_{rs}} ((S \overset{p_{st}}{\bowtie} T) \overset{p_{tu}}{\bowtie} U)$

companion set			
R	S	T	U
$\{R\}$	$\{S,T,U\}$	$\{S,T,U\}$	$\{S,T,U\}$

	p_{rs}	p_{st}	p_{tu}
NEL	$\{R,S\}$	$\{S,T\}$	$\{T,U\}$
EEL	$\{R,S,T,U\}$	$\{S,T\}$	$\{T,U\}$

Additional valid join orders: $(R, (S, (T, U)))$

4.2. Reordering Outer, Anti, and Inner Joins

In this section, we show how to set the EELs properly when there are outerjoins, antijoins, and inner joins in the operator tree. With antijoins, we have the following additional conflicting rules:

$$(2)\ R \xrightarrow{p_{rs}} (S \overset{p_{st}}{\triangleright} T) \neq (R \xrightarrow{p_{rs}} S) \overset{p_{st}}{\triangleright} T$$
$$(3)\ R \overset{p_{rs}}{\triangleright} (S \overset{p_{st}}{\bowtie} T) \neq (R \overset{p_{rs}}{\triangleright} S) \overset{p_{st}}{\bowtie} T$$
$$(4)\ R \overset{p_{rs}}{\triangleright} (S \xrightarrow{p_{st}} T) \neq (R \overset{p_{rs}}{\triangleright} S) \xrightarrow{p_{st}} T$$
$$(5)\ R \overset{p_{rs}}{\triangleright} (S \overset{p_{st}}{\triangleright} T) \neq (R \overset{p_{rs}}{\triangleright} S) \overset{p_{st}}{\triangleright} T$$
$$(6)\ (R \xrightarrow{p_{rs}} S) \overset{p_{st}}{\triangleright} T \neq R \xrightarrow{p_{rs}} (S \overset{p_{st}}{\triangleright} T)$$

We summarize all the conflicts in Figure 4. Basically, an outerjoin conflicts with inner joins and antijoins in its null-producing side. An antijoin conflicts with all kinds of joins in the null-producing side and outerjoins "pointing" inward in the preserving side. Inner joins, on the other hand, have no conflicts.

	preserving side	null-producing side
outer	{ }	{inner, anti}
anti	{outer (pointing inward)}	{inner, outer, anti}
inner	{ }	{ }

Figure 4. Conflicting Matrix

The conflicting matrix tells us: For an outerjoin predicate to be eligible, all the TABs in the conflicting inner join and antijoin predicates have to be available. For an antijoin predicate to be eligible, all the TABs from the null-producing side and all the TABs in the conflicting outerjoin predicates from the preserving side have to be available. For an inner join predicate, we just need the TABs referenced in itself to be available.

During the bottom-up traversal of the original operator tree, we keep for each TAB t the following: (a) an *outerjoin set* (will be added to the EEL of outerjoin predicates), which contains all the TABs that are linked together through inner join or antijoin predicates, and (b) an *antijoin set* (will be added to the EEL of antijoin predicates), which contains all the TABs that are linked to t through outerjoins pointing to t. These two sets can be computed in the same way as in Section 4.1. The EELs of join predicates are set up as follows: For each outerjoin predicate p, we add to p's EEL elements in the outerjoin set of the null-producing TABs referenced in p. For each antijoin predicate q, we add to q's EEL elements in the antijoin set of the preserving TABs referenced in q and all the TABs in the null-producing side. Due to space limit, the details of the algorithm are described in [RLL+00].

We illustrate the algorithm using two examples (we refer to outerjoin set and antijoin set as outer_T and anti_T respectively). In Example 4.3, the antijoin set of T includes both S and T since p_{st} points to T. Thus, the

EEL for p_{tu} is set to $\{S, T, U\}$. This means that U can't be joined with T until S has been joined with T. On the other hand, R can be joined with S either before or after the antijoin. In Example 4.4, the EEL for predicate p_{rs} is $\{R, S, T\}$, which requires table S be joined with table T before table R. The EEL for predicate p_{uv} includes $\{R, T, U, V\}$ and thus p_{uv} can only be applied at the end. The order of the two outerjoins can be switched.

Example 4.3: $((R \overset{p_{rs}}{\bowtie} S) \overset{p_{st}}{\longrightarrow} T) \overset{p_{tu}}{\triangleright} U$

	R	S	T	U
anti_T	$\{R\}$	$\{S\}$	$\{S,T\}$	$\{U\}$
outer_T	$\{R,S\}$	$\{R,S\}$	$\{T,U\}$	$\{T,U\}$

	p_{rs}	p_{st}	p_{tu}
NEL	$\{R,S\}$	$\{S,T\}$	$\{T,U\}$
EEL	$\{R,S\}$	$\{S,T\}$	$\{S,T,U\}$

Additional valid join orders: $((R,(S,T)),U)$ and $(R,((S,T),U))$

Example 4.4: $((R \overset{p_{rs}}{\longrightarrow} (S \overset{p_{st}}{\bowtie} T)) \overset{p_{tu}}{\longrightarrow} U) \overset{p_{uv}}{\triangleright} V$

	R	S	T	U	V
anti_T	$\{R\}$	$\{R,S\}$	$\{R,T\}$	$\{R,T,U\}$	$\{V\}$
outer_T	$\{R\}$	$\{S,T\}$	$\{S,T\}$	$\{U,V\}$	$\{U,V\}$

	p_{rs}	p_{st}	p_{tu}	p_{uv}
NEL	$\{R,S\}$	$\{S,T\}$	$\{T,U\}$	$\{U,V\}$
EEL	$\{R,S,T\}$	$\{S,T\}$	$\{T,U\}$	$\{R,T,U,V\}$

Additional valid join order: $((R,((S,T),U)),V)$

4.3. Practical Issues

More complex predicates such as hyper-predicates, null-intolerant predicates and Cartesian products can be found in real world queries. EELs can exploit the join order information in the original operator tree and thus be used to handle these practical issues. None of the previous work has considered all these practical issues thoroughly.

Hyper-predicates: Predicates referencing more than two tables can occur in reality. Our algorithm can also handle hyper-predicates. Example 4.5 shows how the EEL of predicate p_{rst} can be set up properly. To support hyper-predicates, the graph analysis approach used in [GLR97] would require handling hyper-edges, which would be more complicated and less intuitive.

Example 4.5: $R \overset{p_{rst}}{\longrightarrow} ((S \overset{p_{st}}{\bowtie} T) \overset{p_{tu}}{\bowtie} U)$

	companion set		
R	S	T	U
$\{R\}$	$\{S,T,U\}$	$\{S,T,U\}$	$\{S,T,U\}$

	p_{rst}	p_{st}	p_{tu}
NEL	$\{R,S,T\}$	$\{S,T\}$	$\{T,U\}$
EEL	$\{R,S,T,U\}$	$\{S,T\}$	$\{T,U\}$

Additional valid join order: none

Predicates in the Top Filter Operator: Initially, all the conjuncts in the where clause are applied by the top filter operator. These predicates are either local selection predicates or inner join predicates (if they reference more than one table). When there are outerjoins or antijoins, predicates in the where clause can't always be pushed down. Consider the query in Example 4.6. Suppose after joining S and T, there is only one tuple in the join result and $S.c$ in the result tuple is less than 10. Also suppose there is only one tuple in R and it matches the previous join result. The correct result set will be empty since the tuple in the join result will be filtered out by the predicate (call p_s) in the WHERE clause. However, if p_s is applied before the left outerjoin, we will get one tuple with nulls in the S and T part. The reason is because p_s is null-tolerant.

Example 4.6:

```
SELECT  *
FROM    R LEFT JOIN
        (S INNER JOIN T ON S.b=T.b)
        ON R.a=S.a
WHERE   (S.c>10 OR S.c IS NULL )
```

So, we have to distinguish between null-tolerant and null-intolerant predicates in the top filter operator. If a predicate p is null-intolerant, it can be applied as early as possible. This is because any TAB t referenced in p can't be in the null-producing side of any outerjoin predicates (simplifiable using rule (0)) or antijoin predicates (impossible since the null-producing side is not in the output). The following rules allow predicates to be pushed into the preserving side of outerjoins or antijoins and either side of inner joins.

$$(7)\ \sigma_{p_r}(R \longrightarrow S) = \sigma_{p_r}(R) \longrightarrow S$$
$$(8)\ \sigma_{p_r}(R \triangleright S) = \sigma_{p_r}(R) \triangleright S$$
$$(9)\ \sigma_{p_r}(R \bowtie S) = \sigma_{p_r}(R) \bowtie S$$

Thus, for null-intolerant predicates in the filter operator, their EELs are the same as their NELs.

A null-tolerant predicate q, on the other hand, can't be pushed down arbitrarily since the TABs it references could be in the null-producing side of an outerjoin. It's only safe to evaluate q at the very end after all the joins have been performed. As a result, the EEL of a null-tolerant predicate should include all the TABs in the operator tree. Alternatively, we can separate those null-tolerant predicates into a new query block on top of the original query block (which consists of the operator tree). This will also guarantee that those null-tolerant predicates are evaluated after the join.

Null-tolerant Join Predicates: There are two kinds of problems when join predicates are null-tolerant. First, some of the associative rules will break. For example, associative rule $(R \overset{p_{rs}}{\longrightarrow} S) \overset{p_{st}}{\longrightarrow} T = R \overset{p_{rs}}{\longrightarrow} (S \overset{p_{st}}{\longrightarrow} T)$ will not hold if p_{st} is null-tolerant. Second, simplification is limited. For example, query $(R \overset{p_{rs}}{\longrightarrow} S) \overset{p_{st}}{\bowtie} T$ can't be simplified if p_{st} is null-tolerant. As a result,

it's not always safe to change the order of the evaluation of null-tolerant join predicates. So the EEL of a null-tolerant predicate will include all the TABs from below it in the original operator tree. Similarly, we can put those predicates in separate query blocks. By doing so, we fix the evaluation order of this particular join. However, joins below or above it may still be reorderable. A slight improvement can be made for null-tolerant inner join predicates. If there are no outerjoins or antijoins below the inner join predicate in the original operator tree, we can still use its NEL as its EEL.

Conjuncts in the Join Predicates: Join predicates specified in the ON clauses can have more than one conjunct. Although we can treat all the conjuncts as a single predicate, breaking the conjuncts may allow us to evaluate some of the conjuncts earlier. Here, we assume that all the conjuncts are null-intolerant (null-tolerant join predicates are handled in the above case). We first consider outerjoin predicates. If a conjunct q_s is local to the null-producing side, it can be pushed down as specified by the following rule:

$$(10)\ R \overset{p_{rs} \wedge q_s}{\longrightarrow} S = R \overset{p_{rs}}{\longrightarrow} (\sigma_{q_s} S)$$

In this case, the EEL of q_s is its NEL. For each of the rest of the conjuncts (including those local to the preserving side and those referencing both sides), the EEL is set to be the union of the EELs of all the conjuncts. For example, the EEL of conjunct p_r and p_{rs} will both be set to $\{R, S, T\}$ for query $R \overset{p_r \wedge p_{rs}}{\longrightarrow} (S \overset{q_{st}}{\bowtie} T)$.

Conjuncts in an antijoin predicate can be treated in the same way as outerjoin predicates. Conjuncts in inner join predicates can always be treated separately and their EELs are the same as their NELs.

Introducing Cartesian Products: Although introducing Cartesian product is not a good idea in general, sometimes it can generate better plans. Consider a query $R \overset{p_{rs}}{\bowtie} S \overset{p_{st}}{\bowtie} T$ and suppose S is a fact table and R and T are dimension tables. If both R and T are small and there exists a multi-column index on S, it might be cheaper to combine R and T first using a Cartesian product, followed by an indexed nested loops join with S. We can't introduce Cartesian products freely when there are joins other than inner joins. For example, if we have a query whose query graph is shown in Figure 5 (the original order is not important since the query is freely-reorderable), we can't introduce a Cartesian product between R and V since then it's not clear how to combine T with (R, V) (neither outerjoin nor inner join is appropriate). However, for two TABs that are connected through inner joins, we can introduce a Cartesian product between the two. In Figure 5, we can introduce a Cartesian product between T and S. So the rule is: we can introduce a Cartesian product on two TABs as long as they have the same companion set.

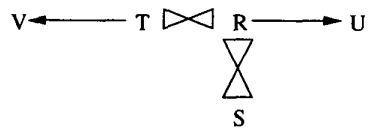


Figure 5. A Query Graph

5. Reordering with Compensation

Instead of allowing only valid join orders, we can take a more aggressive approach to allow invalid orders and later compensate the incorrect result. Consider again the query $R \overset{R.a=S.a}{\longrightarrow} (S \overset{S.b=T.b}{\bowtie} T)$ we used in Section 4.1. Suppose both S and T are large, but R is very small. If we don't allow compensation, we are forced to join S and T first, which will probably generate a large intermediate result. On the other hand, the result size of R joining S could be much smaller. If we can do some compensation later to obtain the correct result, we may get an overall better plan.

The *Generalized Outerjoin* approach (GOJ) proposed in [GLR97] is also a compensation approach. The following shows the definition of generalized outerjoin and the transformation using generalized outerjoin:

$$R\ GOJ[p, A]S$$
$$= (R \overset{p}{\bowtie} S) \cup ((\pi_A R - \pi_A(R \overset{p}{\bowtie} S)) \times \{null\})$$
$$R \overset{p_{rs}}{\longrightarrow} (S \overset{p_{st}}{\bowtie} T) = (R \overset{p_{rs}}{\longrightarrow} S)\ GOJ[p_{st}, R]T$$

Generalized outerjoin is an expensive operation though, which may require two passes of the input and some additional sorting. Unfortunately, for each compensation performed during the optimization, a generalized outerjoin will be introduced. It would be nice if we can reduce the number of these expensive operations. The compensation approach we take is to evaluate a query that's less restrictive and then remove the spurious tuples later. Our approach has the benefit that multiple compensation operations (which are also expensive) can be merged and applied only once.

Definition 5.1: We say that tuple $t1$ is subsumed by $t2$ if for all the non-null fields in $t1$, $t2$ has the same values, and $t2$ has more non-null fields than $t1$. For example, tuple $(1, -, -)$ is subsumed by $(1, 2, -)$.

We now illustrate our compensation approach using query $R \overset{R.a=S.a}{\longrightarrow} (S \overset{S.b=T.b}{\bowtie} T)$ again. The contents in table R, S and T are the same as shown in Figure 1. We know that the alternative join order $(R \overset{R.a=S.a}{\longrightarrow} S) \overset{S.b=T.b}{\bowtie} T$ will give an incorrect result (since all the tuples with nulls in S are eliminated). To avoid losing tuples, we first "promote" the inner join to an outerjoin and evaluate $(R \overset{R.a=S.a}{\longrightarrow} S) \overset{S.b=T.b}{\longrightarrow} T$ instead (result shown in Figure 6(a)). We then perform compensation in two steps. The first one is *nullification*, which sets to

null the S part of those tuples where $S.b = T.b$ evaluates to false (result shown in Figure 6(b)). Clearly, we now have more tuples than the correct result set. These are duplicated tuples (e.g., the 4th row in Figure 6(b)) and subsumed tuples (e.g., the 2nd row in Figure 6(b)). We introduce a new operator called *best match* to remove tuples that are duplicated or subsumed. The final result after best match is shown in Figure 6(c). This gives us exactly the same result as the original query. The formal definition of nullification and best match are give in Definition 5.2 and 5.3.

r1	s1	t1
r1	s2	-
r2	s3	-
r2	s4	-
r3	-	-

(a)$(R \rightarrow S) \rightarrow T$

r1	s1	t1
r1	-	-
r2	-	-
r2	-	-
r3	-	-

(b) Nullification

r1	s1	t1
r2	-	-
r3	-	-

(c) Best Match

Figure 6. Reordering with Compensation

Definition 5.2: A nullification operator on a set of tuples S, for a given predicate p and a set of attributes A is defined as $Null[p, A](S)$ ={for each tuple s in S, set all the attributes in set A to null if predicate p is true}. When a relation name is used for A, we mean to nullify all the attributes coming from that relation.

Definition 5.3: A best match operator is defined as BM(S)={the set of tuples in S less all duplicated tuples and subsumed tuples}. To guarantee the correctness, when using best matches, we assume that we keep the TID of all the participating relations during query evaluation. Best match can be implemented by hashing the inputs on one of the attributes (not nullified) and then removing duplicated and subsumed tuples by sorting tuples in each bucket.

Using nullification and best match, the following transformation is possible ($\sim$ represents negation):

$$R \xrightarrow{p_{rs}} (S \overset{p_{st}}{\bowtie} T) = BM(Null[\sim p_{st}, S]((R \xrightarrow{p_{rs}} S) \xrightarrow{p_{st}} T))$$

Although logically equivalent to what is done in the GOJ approach, best match has the benefit that multiple best matches can be merged. It's not difficult to verify the following rules (assuming that we keep the TID of all the relations during evaluation):

$BM(Null[p, A](BM(R))) = BM(Null[p, A](R))$,
where A is a subset of attributes in R,
p is a null-intolerant predicate

$BM(R) \bowtie S = BM(R \bowtie S)$

$BM(R) \longrightarrow S = BM(R \longrightarrow S)$

$BM(R) \longleftarrow S = BM(R \longleftarrow S)$

Using the above rules, we can now derive:

$$R \xrightarrow{p_{rs}} (S \overset{p_{st}}{\bowtie} T \overset{p_{tu}}{\bowtie} U)$$
$$= BM(Null[\sim p_{tu}, S \cup T](BM$$
$$(Null[\sim p_{st}, S]((R \xrightarrow{p_{rs}} S) \xrightarrow{p_{st}} T)) \xrightarrow{p_{tu}} U)) \quad \text{(a)}$$
$$= BM(Null[\sim p_{tu}, S \cup T]$$
$$(Null[\sim p_{st}, S]((R \xrightarrow{p_{rs}} S) \xrightarrow{p_{st}} T) \xrightarrow{p_{tu}} U)) \quad \text{(b)}$$

There are tradeoffs between the two plans above. Plan (a) has to perform two best matches. It is essentially the GOJ plan. Similar to GOJ, best match is an expensive operation. Plan (b) reduces the number of best match operations to one. However, since the compensation is delayed, spurious tuples may be carried along during the evaluation, which increases the size of intermediate results. It's up to the optimizer to decide which plan to pick by comparing their cost. Notice that our best match approach covers the plans considered by the generalized outerjoin. The generalized outerjoin approach doesn't have a corresponding plan (b).

5.1. Compensation with Outerjoins and Inner Joins Only

In this section, we discuss how to perform compensation when there are only outerjoins and inner joins in the query. We will talk about other kinds of compensation in Section 7. For the sake of simplicity, we consider only simple queries. We assume that the following have been calculated using the algorithm in Section 4.1: (a) the companion set for each TAB; (b) the NEL and the EEL for each join predicate. When no compensation is considered, the optimizer will prevent using a join predicate p if its EEL is not covered in the subplans. Now, we can loosen this restriction. If p's EEL is not covered, we still allow the two subplans to be joined as long as p's NEL is covered. However, we need to do three things for compensation. First, we have to identify those inner join predicates that need to be promoted to outerjoins (so that we won't lose tuples). Second, we need to determine which TABs need to be nullified and how. Last, we should consider when to introduce best matches.

We now sketch the algorithm of generating plans with compensation (details can be found in [RLL+00]). We keep in each subplan: (a) a *nullification set* for each TAB t. A nullification set includes a list of predicates that will be used to nullify t; (b) a *compensation set*, which includes all the predicates that need to be promoted from inner join to outerjoin. Predicates in the compensation set will be treated as outerjoin predicates, and we refer to them as *compensation predicates*. If the EEL of a predicate p is not covered (p is either an outerjoin or a compensation predicate), we find all the inner join predicates that are linked to the null-producing TAB in p. Since these inner join predicates will be evaluated after p, they have to be added to the compensation set.

A compensation predicate will inherit its EEL from p. When a compensation predicate q is used to combine two subplans, we have to use its negation to nullify some TABs in the companion set of the TAB referenced in q. These TABs are those that exist in the preserving side when q is evaluated. We also store in each subplan a compensation flag. The compensation flag is turned on as long as a compensation predicate has been applied. We should consider adding a best match to any subplan with the compensation flag turned on at some point. However, intermediate subplans have the option to delay the compensation.

We illustrate the algorithm using Example 5.1. We consider how to generate plans corresponding to a join order that needs compensation. In the first step, the optimizer tries to combine R and S first. It determines that the EEL of p_{rs} is not covered. The only inner join predicate linking to the null-producing TAB S is p_{st}. So we add it to the compensation set. The EEL of p_{st} becomes $\{R,S,T,U\}$ (inherited from p_{rs}). In the second step, we want to combine (R,S) with T using predicate p_{st}. Since p_{st} is a compensation predicate, it's used as an outerjoin predicate. As the EEL of p_{st} is not covered, we promote p_{tu} to a compensation predicate. We also add $\sim p_{st}$ to the nullification set of S, since S is in both the companion set of S and the preserving side of this join. The compensation flag is turned on at this step since a compensation predicate has been applied. We can consider adding a best match now or at a later time. In the third step, we need to add $\sim p_{tu}$ to the nullification set of S and T. We have to introduce a best match after this step. Figure 7(a) and 7(b) show two alternative plans for this particular join order. In Figure 7(a), a best match is performed immediately after each compensation. In Figure 7(b), two best matches are merged and applied only once at the end. We show other possible join orders using compensation at the end of Example 5.1.

Example 5.1: $R \overset{p_{rs}}{\longrightarrow} ((S \overset{p_{st}}{\bowtie} T) \overset{p_{tu}}{\bowtie} U)$

	companion set		
R	S	T	U
$\{R\}$	$\{S,T,U\}$	$\{S,T,U\}$	$\{S,T,U\}$
	p_{rs}	p_{st}	p_{tu}
NEL	$\{R,S\}$	$\{S,T\}$	$\{T,U\}$
EEL	$\{R,S,T,U\}$	$\{S,T\}$	$\{T,U\}$

Steps of generating join order $(((R,S),T),U)$.

Step 1 $R \overset{p_{rs}}{\longrightarrow} S$:
new compensation predicates: p_{st}
compensation flag: off

	R	S
nullification set	$\{\}$	$\{\}$

Step 2 $(R \overset{p_{rs}}{\longrightarrow} S) \overset{p_{st}}{\longrightarrow} T$:
new compensation predicates: p_{tu}
compensation flag: on

	R	S	T
nullification set	$\{\}$	$\{\sim p_{st}\}$	$\{\}$

Step 3 $((R \overset{p_{rs}}{\longrightarrow} S) \overset{p_{st}}{\longrightarrow} T) \overset{p_{tu}}{\longrightarrow} U$:
new compensation predicates: none
compensation flag: on

	R	S	T	U
nullification set	$\{\}$	$\{\sim p_{tu}\}$	$\{\sim p_{tu}\}$	$\{\}$

Additional valid join orders:
$BM(Null[\sim p_{st}, S]((R \overset{p_{rs}}{\longrightarrow} S) \overset{p_{st}}{\longrightarrow} (T \overset{p_{tu}}{\bowtie} U)))$
$BM(Null[\sim p_{tu}, S \cup T](((R \overset{p_{rs}}{\longrightarrow} (S \overset{p_{st}}{\bowtie} T)) \overset{p_{tu}}{\longrightarrow} U))$
∎

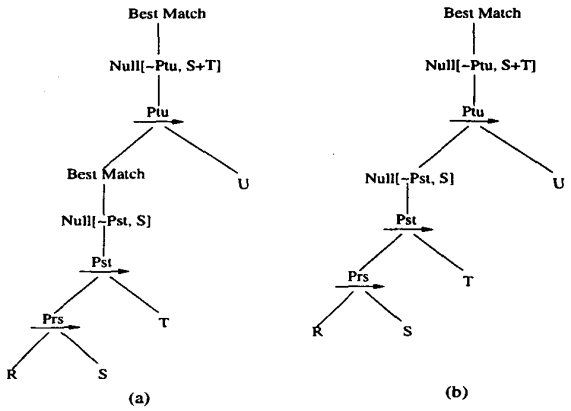


Figure 7. Alternative plans using best match for join order $(((R,S),T),U)$

6. Extending a Conventional Optimizer

In this section, we summarize all the extensions needed for a conventional query optimizer to handle outerjoin and antijoin reordering using EELs. Before doing join optimization, we make one bottom-up traversal through the operator tree of the original query to set up the EEL of each predicate properly.

When using the "without compensation" approach, the only additional information we need to keep is the EEL of each predicate (if we allow introducing Cartesian products, the companion set for each TAB also has to be kept). Similar to NEL, EEL is a property that's plan independent and won't change during join enumeration. This means no additional information needs to be kept in each subplan generated by the optimizer. When combining two subplans, the optimizer checks the EEL of a join predicate instead of its NEL.

The compensation approach is more complicated. First of all, we have to implement a best match operator.

Second, We have to keep in each subplan a nullification set for each TAB and a compensation set. Both properties are plan dependent. Since the number of plans enumerated by the optimizer can be exponential in the number of participating relations [OL90], this could introduce a non-trivial amount of storage overhead. [RLL+00] discusses this in more details.

In comparison, reordering without compensation is easier to implement. However, it considers a more restrictive search space. The compensation approach gives the optimizer more freedom in choosing join orders, but it introduces more space overhead and requires more implementation effort. By separating the two approaches, it's possible to first adapt an optimizer to support the "without compensation" approach and later extend it to the compensation approach. It's also easy to add a switch in the optimizer to enforce a specific approach.

7. Future Work

Our approach can be extended for two-sided outerjoins (or any other new types of join) as well. For reordering without compensation, we just need to identify conflicting rules involving two-sided outerjoins and build a four by four conflicting matrix. We can then set the EELs of all the join predicates properly using the matrix. For the compensation approach, we can consider other kinds of compensation. For example, the following rule allows us to perform an inner join before a full outerjoin.[1]

$R \overset{p_{rs}}{\leftrightarrow} (S \overset{p_{st}}{\bowtie} T) = BM(Null[\sim p_{st}, S]((R \overset{p_{rs}}{\leftrightarrow} S) \overset{p_{st}}{\rightarrow} T))$

We'd also like to investigate how to compensate when there are antijoins.

8. Conclusion

Outerjoins and antijoins are important types of joins in a database system. They will be very useful for new applications such as supporting XML queries. Reordering these joins together with inner joins is a challenging task. Previous work didn't consider antijoin reordering and was restricted in the kind of queries it can support. As a result, many commercial systems only have limited support for outerjoin and antijoin reordering.

In this paper, we propose a solution that extends the eligibility list of join predicates. Our solution handles all three types of join and supports a more general class of queries. Extending the eligibility list is a powerful technique and can be used for other new types of join. Our framework separates the "without compensation" and "with compensation" approach. This allows an existing system to smoothly incorporate our techniques in two phases. Our compensation approach provides an optimizer with more opportunities and thus could lead to overall better plans. We believe that our work will have a major impact on commercial database systems in adopting outerjoin and antijoin reordering techniques.

[1] The BM involving two-sided outerjoins will eliminate a tuple where all the fields are null.

References

[BGI95] Gautam Bhargava, et al. Hypergraph based reorderings of outer join queries with complex predicates. In *ACM SIGMOD Conference*, 1995.

[Cod79] E. F. Codd. Extending the relational database model to capture more meaning. *Transactions on Database Systems*, 4(4):397–434, 1979.

[Cor99] IBM Corporation. DB2 universal database version 6.1. 1999.

[Day83] Umeshwar Dayal. Processing queries with quantifiers. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Conference on Principles of Database Systems*, pages 125–136, 1983.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates and quantifiers. In *Proceedings of the 13th VLDB Conference*, 1987.

[GL92] Cesar A. Galindo-Legaria. *Algebraic optimization of outerjoin queries*. PhD thesis, Department of Applied Science, Harvard University, 1992.

[GLR92] Cesar A. Galindo-Legaria and Arnon Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 402–409, 1992.

[GLR97] Cesar A. Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *Transactions on Database Systems*, 22(1):43–73, 1997.

[Kim82] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th VLDB Conference*, pages 314–325, 1990.

[RGL90] Arnon Rosenthal and Cesar A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proceedings of the ACM SIGMOD Conference*, pages 291–299, 1990.

[RLL+00] Jun Rao, et al. Using eels, a practical approach to outerjoin and antijoin reordering. Technical report #RJ1020395077, IBM Almaden Research Center, Dec. 2000.

[SAC+79] Patricia G. Selinger, et al. Access path selection in a relational database management system. In *ACM SIGMOD Conference*, 1979.