Improving Join Reorderability with Compensation Operators

TaiNing Wang  
National University of Singapore  
taining_wang@u.nus.edu

Chee-Yong Chan  
National University of Singapore  
chanyc@comp.nus.edu.sg

ABSTRACT
A critical task in query optimization is the join reordering problem which is to find an efficient evaluation order for the join operators in a query plan. While the join reordering problem is well studied for queries with only inner-joins, the problem becomes considerably harder when outerjoins/antijoins are involved as such operators are generally not associative. The existing solutions for this problem do not enumerate the complete space of join orderings due to various restrictions on the query rewriting rules considered. In this paper, we present a novel approach for this problem for the class of queries involving inner-joins, single-sided outerjoins, and/or antijoins. Our work is able to support complete join reorderability for this class of queries which supersedes the state-of-the-art approaches.

CCS CONCEPTS
• Information systems → Query optimization;

KEYWORDS
Query optimization, join reordering

ACM Reference format:
https://doi.org/10.1145/3183713.3183731

1 INTRODUCTION
An important task in the optimization of join queries is to decide on a join order, which determines a partial ordering in which the binary join operations are computed. As picking the right join order has a significant impact on the efficiency of query evaluation, the problem of finding optimal join orderings (known as the join reordering problem) is a well-studied problem for inner-join queries [15]. However, queries involving outerjoins and/or antijoins are considerably more difficult to optimize because these join operators do not possess the nice properties of being both commutative and associative.

Not surprisingly, the join reordering problem for complex join queries has been intensively researched by both the industry and academia (e.g., [1, 2, 4–14]). The early research focused on outer-join simplification and reordering [4–6, 13, 14]. These works do not consider antijoins and are restricted to simple queries1. [1] extended these approaches to support complex join predicates that could refer to more than two relations. Extensions to also handle antijoins are presented in [9–11], but reorderings supported are rather limited.

Another recent work [7] is an approach to convert outerjoin queries to inner-join queries. The idea is to first compute a derived relation from each join operand of an outerjoin, and then compute an inner join of the derived relations. A derived relation for a join operand R is a copy of R with additional columns and additional tuples. While this approach is theoretically interesting, it is limited to queries with binary join predicates and the cost for computing derived relations could be as high as that for computing the original outerjoin query. Moreover, it also does not handle antijoins and semijoins.

In this paper, we use \( \leftarrow \), \( \leftrightarrow \), \( \Rightarrow \), \( \Leftrightarrow \), \( \Leftarrow \), \( \Rightarrow \), \( \Leftrightarrow \), and \( \Leftarrow \) to denote, respectively, the outer-union, left outerjoin, right outerjoin, full outerjoin, left semijoin, right semijoin, left antijoin, and right antijoin operators with \( P_{ij} \) denoting the join predicate between relations \( R_i \) and \( R_j \). We use \( C_J \) to denote the class of join queries with join operators from the set \( J = \{ \Leftarrow \Rightarrow \Leftrightarrow \Leftarrow \Rightarrow \Leftrightarrow \} \) and with null-intolerant join predicates; \( C_J^{nr} \) to denote the subset of join queries in \( C_J \) that do not involve \( \Rightarrow \); and \( C_J^{nr,\Leftrightarrow,\Leftarrow} \) to denote the subset of join queries in \( C_J \) that do not involve \( \Leftrightarrow \) and \( \Leftarrow \). Finally, we use \( \circ \) to denote some join operator from \( J \).

There are currently two state-of-the-art approaches for reordering complex join queries. The first approach is a transformation-based approach (denoted by TBA) that enumerates all valid join reorderings using the associativity and commutativity properties of the join operators [8]. For example, left outerjoins are known to be non-associative and non-commutative, and for the query \( Q = R_3 \Rightarrow (R_2 \Leftrightarrow R_1) \), \( Q' = (R_1 \Rightarrow R_3) \Rightarrow R_2 \) is an invalid reordering of \( Q \).

The second approach is a compensation-based approach (denoted by CBA) that permits certain invalid join reorderings so long as they could be compensated to become valid [12]. Continuing with the earlier example, the invalid reordering \( Q' \) can be made valid by rewriting it to \( \beta(\lambda R_2 R_3 Q') \) using the compensation operators \( \beta \) and \( \lambda \). However, CBA has certain limitations in that certain

1A simple query is a query that satisfies the following properties: all outerjoin predicates have only one conjunct and must be binary predicate referring to only two relations, there are no cartesian products in the query, and all predicates are null-intolerant. A predicate is classified as null-intolerant if it cannot evaluate to true when referencing a null value; otherwise, it is a null-tolerant predicate.
valid reorderings that are supported by TBA are not possible with CBA.

Thus, both the state-of-the-art approaches are incomparable in terms of the join reorderings that could be supported. In this paper, we present a novel approach to solve the join reordering problem and make four contributions. First, we introduce a precise formalization of complete join reorderability. Second, we develop a new approach for the join reordering problem that strictly subsumes the join reorderability of both the state-of-the-art solutions TBA and CBA. For the class of queries $C_J$, our approach provides complete join reorderability (in contrast to the partial join reorderability supported by both TBA and CBA). For the class of queries $C_F$, although all three approaches support only partial join reorderability, the set of join reorderings enabled by our approach strictly subsumes that supported by TBA and CBA. Third, we develop a novel top-down plan enumeration algorithm to compute optimal query plans with compensation operators. Fourth, we perform an experimental evaluation of our approach and our results show that our approach can improve query execution time by up to factors of 2.84 and 6.14 on PostgreSQL and a commercial database system, respectively.

The rest of this paper is organized as follows. Section 2 presents the key ideas behind the two state-of-the-art approaches. In Section 3, we formalize the notion of complete join reorderability and compare the different approaches in terms of their completeness for join reorderability. We present our new rewriting approach for join reordering in Section 4. Section 5 presents a novel top-down query plan enumeration algorithm for query plans with compensation operators. We discuss implementation issues in Section 6. Section 7 presents an experimental evaluation of our approach. Finally, Section 8 concludes the paper.

We end this section by introducing some of the basic notations used in this paper.

**Notations.** Given a relation $R_i$, we use $\text{attr}(R_i)$ to denote the set of all attributes in the schema of $R_i$; and given a set of relations $\mathcal{R}$, we use $\text{attr}(\mathcal{R})$ to denote $\bigcup_{R_i \in \mathcal{R}} \text{attr}(R_i)$. For a predicate $p$, we use $\text{attr}(p)$ to denote the set of attributes referenced by $p$. For convenience, we write $\pi_{\text{attr}(R_i)}$ as $\pi_{R_i}$ to denote the projection of all the attributes in the schema of $R_i$. Furthermore, we use $\pi_{\mathcal{R}}$ to denote $\pi_{\mathcal{S}}$ where $S = \bigcup_{R_i \in \mathcal{R}} \text{attr}(R_i)$.

2 RELATED WORK

In this section, we describe the key ideas behind the two state-of-the-art approaches TBA[8] and CBA[12]. These two works are the most closely related to our work as our approach uses some of their ideas which are further discussed in Section 4.

2.1 Transformation-Based Approach (TBA)

TBA’s approach for reordering joins is based on three basic transformations derived from the commutativity/associativity properties of join operators as illustrated in Figure 1. We use $e_i$ to denote some relational expression; $\sigma_{\mathcal{S}}$ to denote some join operator; and $p_{ij}$ to denote a join operation $\sigma_{e_i}$ between $e_i$ and $e_j$ with join predicate $p_{ij}$. The three basic properties are defined as follows:

1. $\sigma_{\mathcal{S}}$ and $\sigma_{\mathcal{S}'}$ satisfy the associativity property (denoted as $\text{assoc}(\sigma_{\mathcal{S}}, \sigma_{\mathcal{S}'})$ if $(e_1 \sigma_{\mathcal{S}} e_2 \sigma_{\mathcal{S}'} e_3) = (e_1 \sigma_{\mathcal{S}'} e_2 \sigma_{\mathcal{S}} e_3)$.
2. $\sigma_{\mathcal{S}}$ and $\sigma_{\mathcal{S}'}$ satisfy the left asscom property (denoted as $\text{l-asscom}(\sigma_{\mathcal{S}}, \sigma_{\mathcal{S}'})$ if $(e_1 \sigma_{\mathcal{S}} e_2 \sigma_{\mathcal{S}'} e_3) = (e_1 \sigma_{\mathcal{S}'} e_2) \sigma_{\mathcal{S}} e_3$.
3. $\sigma_{\mathcal{S}}$ and $\sigma_{\mathcal{S}'}$ satisfy the right asscom property (denoted as $\text{r-asscom}(\sigma_{\mathcal{S}}, \sigma_{\mathcal{S}'})$ if $(e_1 \sigma_{\mathcal{S}} e_2 \sigma_{\mathcal{S}'} e_3) = e_1 \sigma_{\mathcal{S}} (e_2 \sigma_{\mathcal{S}'} e_3)$.

Figure 1: Basic transformation rules in TBA: (a) assoc rule (b) l-asscom rule (c) r-asscom rule.

Corresponding to the above three basic properties, we have three basic types of transformations: $\text{assoc}(\sigma_{\mathcal{S}}, \sigma_{\mathcal{S}'}), \text{l-asscom}(\sigma_{\mathcal{S}}, \sigma_{\mathcal{S}'}),$ and $\text{r-asscom}(\sigma_{\mathcal{S}}, \sigma_{\mathcal{S}'})$.

A specific type of transformation for a pair of join operators is said to be valid if the corresponding property holds; otherwise, it is said to be invalid. Examples of valid transformations include $\text{assoc}(\rightarrow, \rightarrow)$ and $\text{l-asscom}(\rightarrow, \rightarrow)$, while examples of invalid transformations include $\text{assoc}(\rightarrow, \leftarrow)$ and $\text{r-asscom}(\rightarrow, \leftarrow)$. Table 1 from [8] summarizes all the valid (+) and invalid (−) assoc/l-asscom/r-asscom transformations introduced by TBA’s approach. Note that because l/r-asscom are symmetric properties, Table 1(b) is symmetric across the diagonal.

<table>
<thead>
<tr>
<th>$\sigma_{\mathcal{S}}$</th>
<th>$\sigma_{\mathcal{S}'}$</th>
<th>$\text{assoc}$</th>
<th>$\text{l-asscom}$</th>
<th>$\text{r-asscom}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{assoc}$</td>
<td>$\text{assoc}$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
</tr>
<tr>
<td>$\text{assoc}$</td>
<td>$\text{l-asscom}$</td>
<td>$+$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{assoc}$</td>
<td>$\text{r-asscom}$</td>
<td>$+$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{l-asscom}$</td>
<td>$\text{assoc}$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
</tr>
<tr>
<td>$\text{l-asscom}$</td>
<td>$\text{l-asscom}$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
</tr>
<tr>
<td>$\text{l-asscom}$</td>
<td>$\text{r-asscom}$</td>
<td>$+$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{r-asscom}$</td>
<td>$\text{assoc}$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
</tr>
<tr>
<td>$\text{r-asscom}$</td>
<td>$\text{l-asscom}$</td>
<td>$+$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{r-asscom}$</td>
<td>$\text{r-asscom}$</td>
<td>$+$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

Table 1: (a) Valid and invalid assoc transformations. (b) Valid and invalid l/r-asscom transformations.

TBA is able to enumerate all reordered query plans by applying valid transformations for all conventional join types. However, TBA forbids join reorderings involving any invalid transformation.
2.2 Compensation-Based Approach (CBA)

Given a query \( Q \) on a set of relations \( \mathcal{R} = (R_1, \cdots, R_n) \), CBA aims to rewrite \( Q \) into an equivalent canonical form:

\[
\beta(\lambda_{p_1,A_1}(\cdots(\lambda_{p_n,A_n}(R_1 \times \cdots \times R_n))))
\]

Here, \( \times \) denotes the outer variant of the cartesian product operator that preserves all tuples from non-empty relations. The operators \( \lambda_{p_i,A_i} \) and \( \beta \) are the two new unary relational operators introduced by CBA. They are used as compensation operators to compensate for invalid join reordering. The first new operator, \( \lambda_{p_i,A_i}(R_1 \times \cdots \times R_n) \), which is referred to as the nullification operator, is defined with respect to a predicate \( p_i \) and a set of attributes \( A_i \subseteq \text{attr}(\mathcal{R}) \) such that for each tuple \( t \in R_1 \times \cdots \times R_n \), if \( p_i \) does not evaluate to true for \( t \), then the values for all the attributes in \( A_i \) will be set to \( \text{null} \) for tuple \( t \).

The second new operator, \( \beta(R) \), which is referred to as the best-match operator, removes all the dominated or duplicated tuples in its relational operand \( R \). Given two tuples \( t, t' \in R \), \( R \) is dominated by \( t' \) if for every non-null attribute value in \( t \), \( t' \) has the same value in the corresponding attribute, and \( t \) has more attributes with null values than \( t' \). We refer to the tuples being eliminated in \( R \) by \( \beta \) as spurious tuples.

Example 2.1. This example illustrates the \( \beta \) operator on the relation \( \mathcal{R}(A, B, C) \) shown below. Observe that the last three tuples in \( \mathcal{R} \) are all spurious tuples dominated or duplicated by the first tuple, while the second tuple is a non-spurious tuple.

<table>
<thead>
<tr>
<th>( R )</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td></td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( \text{null} )</td>
<td>( c_2 )</td>
<td></td>
</tr>
<tr>
<td>( \text{null} )</td>
<td>( b_1 )</td>
<td>( \text{null} )</td>
<td></td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( \text{null} )</td>
<td>( c_1 )</td>
<td></td>
</tr>
</tbody>
</table>

For notational convenience, given relations \( R_1, R_2 \) and a predicate \( p_i \), we use \( \lambda_{p_i,R_1} \) to denote \( \lambda_{p_i,\text{attr}(R_1)} \), and use \( \lambda_{p_i,R_1 \cup R_2} \) to denote \( \lambda_{p_i,\text{attr}(R_1 \cup R_2)} \). Given a set of relations \( \mathcal{R} \), we use \( \lambda_{p_,\mathcal{R}} \) to denote \( \lambda_{p_,\text{attr}(\mathcal{R})} \).

Join queries are rewritten into the canonical form by applying the following rewriting rules, where \( p_1 \) is a predicate that references only \( R_1 \) and \( A_1 \subseteq \text{attr}(R_1) \).

\[
\begin{align*}
R_1 \overset{p_1}{\rightarrow} R_2 & = \beta(\lambda_{p_2,R_1 \cup R_2}(R_1 \times R_2)) \quad (1) \\
R_1 \overset{p_2}{\rightarrow} R_2 & = \beta(\lambda_{p_1,R_2}(R_1 \times R_2)) \quad (2) \\
\beta(\beta(R_1)) & = \beta(R_1) \quad (3) \\
\beta(R_1 \times R_2) & = \beta(R_1 \times R_2) \quad (4) \\
\lambda_{p_1,A_1}(R_1 \times R_2) & = \lambda_{p_1,A_1}(R_1 \times R_2) \quad (5) \\
\beta(\lambda_{p_2,A_2}(\beta(R_1))) & = \beta(\lambda_{p_1,A_1}(R_1)) \quad (6)
\end{align*}
\]

The key idea is that with the rewritten query in the canonical form, different join reorderings could be derived with the following two-step approach. First, since the cartesian product operator is both commutative and associative, the cartesian product ordering is first reordered to a desired ordering and the reordered form is then converted back to a form with join operators by applying the rewriting in the reverse direction. As an example, CBA is able to reorder the joins in \( R_1 \overset{p_1}{\rightarrow} (R_2 \overset{p_2}{\rightarrow} R_3) \) to the equivalent query \( \beta(\lambda_{p_2,R_1}(R_1 \overset{p_1}{\rightarrow} R_2) \overset{p_2}{\rightarrow} R_3)) \). Note that as \( \text{assoc}(\rightarrow, \overset{p}{\rightarrow}) \) is an invalid transformation, \( \text{CBA} \) is unable to reorder the joins. However, CBA is able to achieve the reordering by adding two compensation operators \( \lambda_{p_2,R_1} \) and \( \beta \).

CBA’s main focus is to maximize join reorderability for queries with join operators in \( \{\text{-, }\rightarrow, \overset{p}{\rightarrow}\} \). CBA also briefly discussed a possible approach to handle antijoins and full-outjoins by basically transforming the antijoins and full outjoins in the query into single-sided outjoins. However, the reorderability achieved by this approach using only the \( (\beta, \lambda) \) compensation operators is rather limited in contrast to our approach.

3 JOIN REORDERABILITY

In this section, we formalize the notion of complete join reorderability and compare this completeness for the two state-of-the-art approaches and our approach.

Consider a query \( Q \) on the set relations \( \{R_1, \cdots, R_{n+1}\} \) with \( n \) join operators \( \{\alpha_1, \cdots, \alpha_n\} \) where each join operator \( \alpha_i \) is associated with a join predicate \( p_i \). A join ordering for \( Q \) is an unordered binary tree \( T \) with internal nodes \( \{p_1, \cdots, p_{n+1}\} \) and leaf nodes \( \{R_1, \cdots, R_{n+1}\} \) such that for each internal node \( p_i \) in \( T \), the relations referenced by \( p_i \) are all contained in the subtree rooted at \( p_i \) and \( p_i \) must reference some relation in each of its child subtrees.

Given a join query \( Q \), we use \( \text{JoinOrder}(Q) \) to denote the set of all join orderings of \( Q \).

It is important to emphasize that our definition of join ordering focuses on the order in which the join operands are combined but not on the specific join operators used.

Given a join ordering \( \theta \in \text{JoinOrder}(Q) \) and a set of compensation operators \( O \), we say that \( Q \) is \( O \)-reorderable if \( Q \) can be rewritten into an equivalent query \( Q' \) (possibly using operators in \( O \)) where the join operands in \( Q' \) are joined following the order given by \( \theta \).

Example 3.1. Consider the query \( Q = R_1 \overset{p_1}{\rightarrow} (R_2 \overset{p_2}{\rightarrow} R_3) \). We have \( \text{JoinOrder}(Q) = \{R_1 \overset{p_1}{\rightarrow} (R_2 \overset{p_2}{\rightarrow} R_3), (R_1 \overset{p_1}{\rightarrow} R_2) \overset{p_2}{\rightarrow} R_3\} \). Consider \( \theta = \{R_1 \overset{p_1}{\rightarrow} R_2, R_2 \overset{p_2}{\rightarrow} R_3\} \) and \( O = \{\beta, \lambda\} \), which are the compensation operators used by CBA. It is well known that \( Q \) is not \( O \)-reorderable; however, \( Q \) is \( \theta \)-reorderable. Specifically, CBA could reorder \( Q \) to \( Q' = \beta(\lambda_{p_2,R_1}(R_1 \overset{p_1}{\rightarrow} R_2) \overset{p_2}{\rightarrow} R_3)) \).

As illustrated by Example 3.1, in the reordered query \( Q' \), the join ordering \( \theta \) only determines the order in which the join operands in \( Q' \) are combined, but the join operators themselves could change in \( Q' \). In Example 3.1, observe that \( R_2 \) and \( R_3 \) are combined using \( \overset{p_2}{\rightarrow} \) in \( Q \) but they are combined using \( \overset{p_2}{\rightarrow} \) in \( Q' \).

Given a query class \( C \) and a set of compensation operators \( O \), we say that \( C \) is completely reorderable with respect to \( O \) if for every query \( Q \) that belongs to \( C \) and for every join ordering \( \theta \in \text{JoinOrder}(Q) \), \( Q \) is \( \theta \)-reorderable.
We conclude this section with a discussion on the reorderability completeness of CBA, TBA and our approach. For TBA, $O = \emptyset$ since it does not rely on any compensation operators. The join reorderability of TBA is rather limited as the approach imposes a strong requirement that the join operators in reordered queries remain unchanged. Thus, TBA is unable to reorder the query given in Example 3.1.

CBA uses two compensation operators given by $O = \{\beta, \lambda\}$, and it could produce join reorderings where the reordered join operators are different from the initial join operators. CBA can be easily extended to achieve complete join reorderability for the query class $C_J^{\gamma^*}$, i.e., $C_J^{\gamma^*} \not\leftrightarrow$. However, CBA has very limited reorderability for queries with join operators from $\{\ast, \ast, \ast\}$.

Our approach introduces new compensation operators to support all valid and invalid transformations. The approach subsumes both TBA and CBA in terms of join reorderability for the query classes $C_J$, $C_J^{\gamma^*}$, and $C_J^{\gamma^*}$. Specifically, we have the following results.

**Theorem 3.2.** (a) For $C_J^{\gamma^*}$, our approach has complete join reorderability, but both TBA and CBA only have partial join reorderability.

(b) For $C_J$, all three approaches have only partial join reorderability but our approach strictly supersedes both TBA and CBA.

The soundness of the above theorem is based on the following properties. First, in terms of the structural transformations to query plans for reordering adjacent join nodes in query plans, the assoc, l-asscom and r-asscom properties discussed in Section 2.1 are the only possible basic transformations as pointed out by TBA. Second, for $C_J^{\gamma^*}$, our proposed approach has sound and complete rewriting rules for all three basic transformations w.r.t. all the join operators in $C_J^{\gamma^*}$. Hence, we have Theorem 3.2(a). Third, for $C_J$, while our proposed approach does not support complete join reorderability for queries with full outerjoins, our approach is able to support all the valid transformations supported by TBA, and also all the reorderings supported by CBA; hence we have Theorem 3.2(b).

### 4 OUR APPROACH

In this section, we present a new approach for the join reordering problem that achieves complete join reorderability for the class of queries $C_J^{\gamma^*}$.

As discussed in Section 3, TBA supports only join reorderings that are based on valid transformations (see Table 1). In particular, among the nine transformations related to antijoins, only three of them are valid. On the other hand, CBA focuses mainly on reordering inner joins and single-sided outerjoins, and it does not handle antijoins well. Thus, both the state-of-the-art approaches have very limited join reorderability for queries with antijoins.

#### 4.1 Overview

In order to enable join reorderings beyond the valid reorderings in TBA, a compensation-based approach (similar to CBA) is necessary.

Hence our proposed solution also adopts a compensation-based approach named Enhanced Compensation-based Approach (ECA).

A crucial design decision for our solution is the design of the set of compensation operators which is guided by four desiderata. First, the operators should maximize the join reorderability possibilities to address the limitations of existing approaches. Second, the approach should be supported by an efficient query plan enumeration algorithm with both cost-based pruning as well as reuse of optimal subplans. Third, an elegant solution should have a small set of operators to minimize implementation complexity. Finally, each of the operators should be amenable to efficient implementation preferably at both the SQL language level as well as natively at the system level. The former enables a less intrusive and easier implementation, and the latter is crucial for performance reasons.

Our approach uses four compensation operators: the $\beta$ and $\lambda$ operators from CBA, and two new operators $\gamma$ and $\gamma^*$. For both the classes of join queries $C_J^{\gamma}$ and $C_J$, our approach with $\gamma$ and $\gamma^*$ enables more join reorderings than both TBA and CBA. In particular, we achieve complete join reorderability for $C_J^{\gamma^*}$. Our approach also supports an efficient query plan enumeration algorithm. As our new operators are only mild extensions of existing relational operators, they are amenable to efficient native implementations. Moreover, the new operators are also implementable at the SQL level as will be discussed in Section 6.

It is important to note that although our solution is inspired by the compensation-based approach of CBA, our approach for query plan enumeration is totally different from CBA’s enumeration approach which is based on the concept of nullification sets.

#### 4.2 Compensation Operators

In addition to the two operators ($\lambda$ and $\beta$) introduced by CBA, our approach uses two new unary operators. The first operator, denoted by $\gamma_A(R_1)$, is a unary operator on $R_1$ with $A \subseteq \text{attr}(R_1)$ defined as follows:

$$\gamma_A(R_1) = \{r \in R_1 | \pi_A(r) \text{ is null}\} \quad (7)$$

The $\gamma_A(R_1)$ operator basically removes all tuples $r \in R_1$ where $\pi_A(r)$ is not null.

The second operator, denoted by $\gamma^+_A(B)(R_1)$, is a unary operator on relation $R_1$ with $A, B \subseteq \text{attr}(R_1)$ defined as follows:

$$\gamma^+_A(B)(R_1) = \beta(\gamma_A(R_1) \cup R') \quad (8)$$

where $R' = \lambda_{false, \text{attr}(R_1)-B}(R_1 - \gamma_A(R_1))$.

Unlike $\gamma_A(R_1)$ which simply removes all the tuples of $R_1 - \gamma_A(R_1)$ from $R_1$, $\gamma^+_A(B)(R_1)$ instead modifies the tuples in $R_1 - \gamma_A(R_1)$ by setting all the values of attributes in $R_1$ (excluding the attributes in $B$) to null.

**Example 4.1.** This example illustrates the $\gamma$ and $\gamma^*$ operators on the relation $R(A, B, C)$ shown below. $\gamma_A(R)$ selects the second tuple in $R$ with a null value for attribute $A$. For the remaining two tuples in $R$ that are not selected by $\gamma_A(R)$, their values for attributes $A$ and $C$ are set to null and this resultant set is represented by $R'$. Finally, $\gamma^+_A(B)(R)$ removes the spurious tuples in $\gamma_A(R) \cup R'$.
We propose to use the following rewriting rule for antijoins:

\[
\gamma A(R) \cup R' = \gamma A'(R')
\]

For notational convenience, if \( R_1, R_2, R_3, \) and \( R_4 \) are relations where \( \text{attr}(R_2), \text{attr}(R_3), \text{attr}(R_4) \subseteq \text{attr}(R_1) \), we use \( \gamma^*_{A(R_2)}(R_1) \) to denote \( \gamma^*_{\text{attr}(R_2) \cup \text{attr}(R_3) \cup \text{attr}(R_4)}(R_1) \) and use \( \gamma^*_{\text{attr}(R_3) \cup \text{attr}(R_4)}(R_1) \) to denote \( \gamma^*_{\text{attr}(R_3) \cup \text{attr}(R_4)}(R_1) \). Similarly, we use \( \gamma_{A(R)}(R_1) \) to denote \( \gamma_{\text{attr}(R) \cup \text{attr}(R_2)}(R_1) \), and \( \gamma_{R_2 \cup R_3}(R_1) \) to denote \( \gamma_{\text{attr}(R_2) \cup \text{attr}(R_3)}(R_1) \).

The rewriting rules for the new operators \( \gamma \) and \( \gamma^* \) are given in Table 2, and we have the following result.

**Theorem 4.2.** The 13 rewriting rules in Table 2 are sound.

A partial proof of Theorem 4.2 is given in Appendix A.

The rules in Table 2 are mainly for interchanging \( \gamma \gamma^* \) with conventional join operators which are essential for achieving join reordering. The two-step approach requires that \( \gamma \gamma^* \) actually not required for join reordering. For instance, Rule 2 reorders the rules that appear to have been omitted in Table 2. Hence, the attributes of \( \gamma \gamma^* \) come from an antijoin operation with \( \gamma \gamma^* \).

We conclude this section with a brief discussion on the completeness of the rewriting rules in Table 2. Table 2 contains all the rules needed for interchanging \( \gamma \gamma^* \) with conventional join operators. The rules that appear to have been omitted in Table 2 are actually not required for join reordering. For instance, Rule 2 requires that \( p_{12} \) does not reference \( R_3 \). It is actually impossible for \( p_{12} \) to reference attributes of \( R_3 \) in \( R_2 \) because \( R_3 \) must come from an antijoin operation with \( R_3 \). Hence, the attributes of \( R_3 \) do not appear in the join operands of \( p_{12} \) and will never be referenced by \( p_{12} \).

### 4.3 Rules for Join Reordering

Having introduced our proposed compensation operators and their properties, we now elaborate on how they are used in our approach. We propose to use the following rewriting rule for antijoins:

\[
R_1 p_{12} R_2 = \pi_{R_1}(\gamma_{R_2}(R_1 p_{12} R_2)) \tag{9}
\]

Semantically, the left antijoin of \( R_1 \) and \( R_2 \) can be computed by the following two steps: (1) compute the left-outerjoin between \( R_1 \) and \( R_2 \); and (2) prune the join results by removing those tuples in \( R_1 \) that can join with some tuple in \( R_2 \) and project out the attributes of \( R_1 \). Our proposed rewriting in Equation 9 is based on this idea and it is the crux behind our approach for reordering join queries involving antijoins. The reason why this two-step approach is effective is because it enables the pruning step to be postponed, which is often necessary when performing join reorderings. We illustrate this need for postponed pruning with the following example.

**Example 4.3.** Consider the query \( Q = R_1 p_{12} R_2 \). Suppose that we want to reorder the joins in \( Q \) so that the join between \( R_1 \) and \( R_2 \) is computed first. For the reordered join expression to be equivalent to \( Q \), the pruning step for the left-antijoin operation must be postponed after the completion of the second join with \( R_3 \). Thus, by expressing the antijoin \( p_{12} \) in terms of \( p_{12} \) and \( \gamma_{R_2} \), it becomes possible to reorder the joins \( \to R_1 \) and \( \to R_2 \), and to postpone \( \gamma_{R_2} \) till all the joins have been computed. In this example, \( Q \) can be rewritten into the equivalent query \( Q' = \pi_{R_2}(\gamma_{R_2}(R_1 p_{12} R_2)) \), as will become clear after we give the complete list of join reordering rules in Table 3. Observe that if \( R_1 p_{12} R_2 \) has a lower evaluation cost than that of \( R_2 \), then \( Q' \) is likely to have a lower evaluation cost than \( Q \).

Based on the rewriting rules in Table 2, Table 3 shows the new join reorderings, given by Rules 14 to 20, that are enabled by our approach. None of these seven join reorderings are possible with the existing approaches [8, 12]. Since our approach also uses the compensation operators \( \beta \) and \( \lambda \), the join reorderings given by Rules 21 to 25 (from [12]) are also supported by our approach. We have the following new result.

**Theorem 4.4.** Rules 14 to 20 in Table 3 are sound.

A partial proof of Theorem 4.4 is given in Appendix A.

![Figure 2: Query Rewriting for Example 4.5](image-url)

The following two examples serve to illustrate how we derive rules for join reordering using \( \gamma \gamma^* \). Example 4.5 shows how \( \gamma \) helps to derive a valid join reordering. The second example gives the derivation for Rule 18 in Table 3 with the help of \( \gamma \) and \( \gamma^* \). A formal proof of Rule 18 can be found in Appendix. Other rules in Table 3 can be proved similarly.

**Example 4.5.** Consider the reordering of the joins in the query \( Q = R_1 p_{12} R_2 R_3 \) such that the join between \( R_1 \) and \( R_2 \) is computed first. By applying Equation 9, \( Q \) is rewritten to \( Q_{a} \) as

![Figure 3: Query Rewriting for Example 4.6](image-url)
shown in Figure 2(a). To perform \( p_{12} \rightarrow \) ahead of \( p_{13} \), we need to push \( p_{12} \rightarrow \) in \( Q_{b} \) below both \( \pi_{R_{2}} \) and \( y_{R_{1}} \). The swap between \( \rightarrow \) and \( \pi_{R_{1}} \) is trivially accomplished by applying the following rewriting rule, where \( A \subseteq attr(R_{1}) \) and \( attr(p_{12}) \subseteq A \cup attr(R_{2}) \):

\[
\pi_{A}(R_{1} \circ_{\rightarrow} R_{2}) = \pi_{A}[\lambda_{R_{1},R_{2}}(R_{1} \circ_{\rightarrow} R_{2})]
\]

where \( \circ \in \{ \rightarrow, \leftarrow, \leftrightarrow \} \).

After this swapping, we obtain \( Q_{d} \) in Figure 2(b). Next, pushing \( p_{12} \rightarrow \) in \( Q_{b} \) below \( y_{R_{1}} \) is achieved by applying Rule 2 in Table 2 to produce \( Q_{c} \) in Figure 2(c). Now that \( p_{12} \rightarrow \) and \( \rightarrow \) are both adjacent operators in \( Q_{c} \), we apply the associativity property between \( \rightarrow \) and \( \circ \) (i.e., \( (R_{1} \circ_{\rightarrow} R_{2}) \rightarrow R_{3} = R_{1} \rightarrow (R_{2} \circ_{\rightarrow} R_{3}) \)) to swap these operators to obtain \( Q_{d} \) in Figure 2(d). Finally, we apply Equation (9) once more to rewrite \( Q_{d} \) to \( Q_{e} \) in Figure 2(e).

The next example illustrates the need for \( y^{\ast} \) operator in join reorderings.

Example 4.6. Consider the reordering of the joins in the query \( Q = R_{1} \circ_{\rightarrow} (R_{2} \circ_{\rightarrow} R_{3}) \) such that the join of \( R_{1} \) and \( R_{2} \) is computed first. By applying Equation (9), \( Q \) is rewritten to \( Q_{a} \) as depicted in Figure 3(a). We apply Equation (10) to get \( Q_{b} \). Then by pushing down the join \( \rightarrow \) below \( y_{R_{1}} \) in \( Q_{b} \), we get the query \( Q_{c} \). However, this rewriting actually does not preserve the equivalence of \( Q_{b} \) and \( Q_{c} \) because while all the tuples in \( R_{1} \) are preserved in the result of \( Q_{b} \), some of the tuples from \( R_{1} \) could be removed by \( y_{R_{1}} \) in the result of \( Q_{c} \).

Intuitively, to preserve equivalence for the rewriting that pushes \( p_{12} \rightarrow \) below \( y_{R_{1}} \) in \( Q_{b} \), we should restrict \( y_{R_{1}} \)'s power such that it cannot remove tuples from \( R_{1} \). To achieve this, we need to replace \( y_{R_{1}} \) by the more general form \( y^{\ast}_{R_{1}} \) after the rewriting to obtain the query \( Q_{d} \) shown in Figure 3(d). By the definition of \( y \) given in

\[
\begin{align*}
\text{Table 2: Rewriting rules for } \gamma \text{ and } \gamma^{\ast}. \text{ Note that } \text{attr}(R_{3}), \text{attr}(R_{4}) &\subseteq attr(R_{1}). \\
\text{Rule 14} &\quad \text{assoc}(\ast, \ast) \quad R_{1} \circ_{\rightarrow} (R_{2} \circ_{\rightarrow} R_{3}) = \pi_{R_{1}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2})) \\
\text{Rule 15} &\quad \text{assoc}(\ast, \ast) \quad R_{1} \circ_{\rightarrow} (R_{2} \circ_{\rightarrow} R_{3}) = \pi_{R_{1}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2})) \\
\text{Rule 16} &\quad \text{assoc}(\ast, \ast) \quad R_{1} \circ_{\rightarrow} (R_{2} \circ_{\rightarrow} R_{3}) = \pi_{R_{1}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2})) \\
\text{Rule 17} &\quad \text{assoc}(\ast, \ast) \quad (R_{1} \circ_{\rightarrow} R_{2}) \circ_{\rightarrow} R_{3} = \pi_{R_{1},R_{2}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2})) \\
\text{Rule 18} &\quad \text{assoc}(\ast, \ast) \quad R_{1} \circ_{\rightarrow} (R_{2} \circ_{\rightarrow} R_{3}) = \pi_{R_{1},R_{2}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2}))) \\
\text{Rule 19} &\quad \text{r-asscom}(\ast, \ast) \quad R_{2} \circ_{\rightarrow} (R_{1} \circ_{\rightarrow} R_{3}) = \pi_{R_{2}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2}))) \\
\text{Rule 20} &\quad \text{r-asscom}(\ast, \ast) \quad R_{2} \circ_{\rightarrow} (R_{1} \circ_{\rightarrow} R_{3}) = \pi_{R_{2}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2}))) \\
\text{Rule 21} &\quad \text{assoc}(\ast, \ast) \quad (R_{1} \circ_{\rightarrow} R_{2}) \circ_{\rightarrow} R_{3} = \pi_{R_{1}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2})) \\
\text{Rule 22} &\quad \text{assoc}(\ast, \ast) \quad R_{1} \circ_{\rightarrow} (R_{2} \circ_{\rightarrow} R_{3}) = \pi_{R_{1}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2}))) \\
\text{Rule 23} &\quad \text{r-asscom}(\ast, \ast) \quad R_{2} \circ_{\rightarrow} (R_{1} \circ_{\rightarrow} R_{3}) = \pi_{R_{2}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2}))) \\
\text{Rule 24} &\quad \text{r-asscom}(\ast, \ast) \quad R_{2} \circ_{\rightarrow} (R_{1} \circ_{\rightarrow} R_{3}) = \pi_{R_{2}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2}))) \\
\text{Rule 25} &\quad \text{r-asscom}(\ast, \ast) \quad R_{2} \circ_{\rightarrow} (R_{1} \circ_{\rightarrow} R_{3}) = \pi_{R_{2}}(y_{\lambda_{R_{1},R_{2}}}(R_{1} \circ_{\rightarrow} R_{2})))
\end{align*}
\]
We conclude this section with a brief discussion on the completeness of the rewriting rules in Table 3. In general, each assoc/lassom transformation for a given pair of join operators should be associated with two rewriting rules: one for rewriting from the LHS to RHS, and the other from rewriting from the RHS to LHS. For instance, Table 3 has two rewriting rules associated with the assoc(→, ∗) transformation. However, for some transformations such as assoc(→, →), the schema of its LHS, i.e., (R_1 ⊳ R_2) ⊳ R_3, is actually not meaningful: the output schema of (R_1 ⊳ R_2) is the schema of R_1 and it does not make sense to join (R_1 ⊳ R_2) with R_3 using the join predicate p_{23}. For such cases, there is only one rewriting rule associated with the transformation.

4.4 Pulling Up Compensation Operators

As illustrated by Examples 4.5 and 4.6, the compensation operators λ/γ/δ" generated during the query rewriting process could get sandwiched between other operators on top (e.g., π, other join operators) and a subtree of join operators below. In order to facilitate the reordering of the join operators, the compensation operators need to be pulled up to bypass other join operators above them.

Pulling up γ/δ" to bypass join operators can be achieved using rewriting rules in Table 2. For pulling up λ, we introduce two additional rewriting rules in Table 4. We have the following results.

Theorem 4.7. Rules 26 and 27 in Table 4 are sound.

<table>
<thead>
<tr>
<th>Rule</th>
<th>λ_{p_1,M}(λ_{p_2,N}(R)) = λ_{p_2,N}(λ_{p_1,M}(R)), if p_1 does not reference N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td>λ_{p_1,M}(λ_{p_2,N}(R)) = λ_{p_1,M,N}(λ_{p_1,M}(R)), if p_1 references N</td>
</tr>
</tbody>
</table>

Table 4: Rewriting rules for swapping λ. Note that λ_{p_1,M} comes from a join node above λ_{p_2,N}.

The next example shows how the rewriting rules help to pull up compensation operators in join reordering.

Example 4.8. Suppose that we want to reorder the join operations in query Q_4 in Figure 4(a) such that the join of R_1 and R_2 is computed first, followed by the join of R_1 and R_3. To achieve this, we first reorder " and " by Rule 14 in Table 3 to get Q_5. Next, we need to pull up the path of non-join operators (π_{p_1,y_{R_1}}, β, λ_{p_23}, R_5) in Q_5 so that p_{14} becomes adjacent to p_{23} to facilitate p_{23} to be reordered ahead of "→. The pulling up of r_{R_1} and y_{R_2} can be done by applying Equation (10) followed by Rule 4 in Table 2. The pulling up β is achieved by applying Equations (2), (4) and (6) in Section 2.2. Finally, the pulling up of λ_{p_23}, R_5 is achieved by applying Rule 30 from Table 5. The resultant query is Q_6 in Figure 4(c) which now enables "→ to be swapped with "→. □

5 TOP-DOWN PLAN ENUMERATION

In this section, we present a top-down plan enumeration strategy to compute an optimal query plan for an input query Q on a set of relations \( R = \{R_1, \cdots , R_n\} \).

In conventional top-down plan enumeration (e.g., [3]), an optimal query plan for Q is computed by enumerating all feasible decompositions of \( R \) into two non-empty, disjoint subsets S_1 and S_2 that could be joined together, and recursively finding optimal subplans for each of S_1 and S_2. Moreover, the best query subplan found for each subset of \( R \) is cached for reuse in subsequent enumerated query plans.

As our approach for reordering joins could introduce compensation operations which are generally different even for joining the same subset of relations under different query plans, this creates additional challenges for enumerating and reusing query subplans. The following example illustrates these additional complexities.

Example 5.1. Consider the query plan Q_4 for joining \( R = \{R_1, R_2, R_3, R_4, R_5\} \) shown in Figure 4. Q_3 and Q_6 are two query plans with different join re-orderings that are each equivalent to Q_4. In terms of top-down plan enumeration, if we use S to denote the set \{R_2, R_3, R_4\}, then Q_3 is formed by first decomposing \( R \) into (\( S \cup \{R_1\} \)) \( \circ \) (\( R_2 \)), followed by decomposing (\( S \cup \{R_1\} \)) into S \( \circ \) \( \{R_1\} \). In contrast, Q_6 is formed by first decomposing \( R \) into S \( \circ \) \( \{R_1, R_2\} \). Suppose that Q_3 is enumerated before Q_6 and that P_g = (R_3 \( \circ \) \( R_4 \)) \( \rightarrow \) R_5 is the best query subplan for joining S in Q_3. As part of the construction of Q_6, we need to determine the best plan for joining S. If we were to reuse the best plan found in Q_3 for S (i.e., replace the subplan P_h = (R_3 \( \circ \) \( R_4 \)) \( \rightarrow \) R_5 in Q_6 with P_g), the resultant query plan will not be equivalent to Q_6 because P_g and P_h are not equivalent.

The above example highlights two key complications with compensation operators. First, the enumeration of query subplans needs to keep track of any compensation operations generated. Second, the reuse of optimal query subplans requires more careful reasoning to ensure correctness.

To simplify the presentation, we first discuss our approach for the class of join queries without full outerjoins \( C_{f,j} \) in Sections 5.1 and 5.2, and then present an extension in Section 5.3 to handle all join queries including full outerjoins. Section 5.1 presents a basic approach that keeps track of compensation operations generated during plan enumeration without any reuse of optimal query subplans. Section 5.2 presents a refinement that also enables the reuse of optimal query subplans.
5.1 Basic Approach
To facilitate the tracking of compensation operators in query plans, our approach enumerates each query subplan $SP$ for joining $S \subseteq \mathcal{R}$ within the context of some complete query plan $P$ (i.e., query plan for the entire query). Given a query $Q$ to be optimized, the simplest choice of an initial complete query plan for $Q$ is the query plan that is directly translated from the specification of $Q$. We use $P_{\text{init}}$ to denote this initial query plan for $Q$. Note that our interest in $P_{\text{init}}$ is in both the logical join operators as well as join ordering specified in $P_{\text{init}}$, which are essential for generating appropriate compensation operators as the joins in $P_{\text{init}}$ are being reordered to enumerate query subplans. Thus, $P_{\text{init}}$ is a logical (and not a physical) query plan.

Algorithm 1 shows the main driver of our top-down enumeration approach. An array $\text{bestAccess}[R_i]$ is used to store the best access plan for each relation $R_i \in \mathcal{R}$; and another array $\text{bestPlan}[S]$ is used to store the query plan $P$ that contains a best subplan $SP$ for joining $S \subseteq \mathcal{R}$.3

Algorithm 2: GenerateSubplan($P$, $i$, $S$)

Input: $P$ is a query plan, $i$ is either null or a join node in $P$, $S$ is a subset of relations in $P$
Output: A query plan $P'$ that is derived from $P$ that contains an optimal subplan $SP$ for joining $S$. If $i \neq \text{null}$, then $SP$ is a child subtree of $i$; otherwise, $P' = SP$

1 if $S = \{ R_i \}$ then
2 return $P$ with $R_i$ replaced by $\text{bestAccess}[R_i]$
3 foreach joinable pair $(S_1, S_2)$ of $S$ do
4 initialize $P' = P$
5 let $j$ be the join node in $P'$ that joins $S_1$ and $S_2$
6 while $\text{par}(j) \neq i$ do
7 $P' = \text{GenerateSubplan}(P', j, S_j)\$
8 $P' = \text{GenerateSubplan}(P', j, S_j)\$
9 if $\text{cost(subtree}(P', S_j)) < \text{cost(subtree}(\text{bestPlan}[S], S))$ then
10 bestPlan$[S] = P'$
11 return bestPlan$[S]$

Algorithm 3: Swap($P$, $i$)

Input: $P$ is a query plan where $i$ is an internal non-root join node in $P$
Output: A modified query plan $P$ with $i$ swapped with its parent join node

1 let $j$ be the parent join node of $i$ in $P$
2 if there are compensation operators between $i$ and $j$ then
3 apply appropriate rewriting rule to swap all compensation operators above $j$
4 let $T_\ell$ and $T_r$ denote the left and right child subtrees of $i$ in $P$
5 if $j$ refers to $T_\ell$ and not to $T_r$ then
6 $P = \text{apply appropriate l-asscom rewriting rule to swap } i$ and $j$
7 else
8 if $j$ refers to $T_r$ and not to $T_\ell$ then
9 if $i$ is the left child of $j$ then
10 $P = \text{apply appropriate l-asscom rewriting rule to swap } i$ and $j$
11 else
12 $P = \text{apply appropriate r-asscom rewriting rule to swap } i$ and $j$
13 return $P$

The for-loop (step 4) iterates all feasible query subplans to join $S$. We define $(S_2, S_2)$ to be a joinable pair of $S$ if there exists a join ordering of $Q$ with an internal node such that $S_1$ and $S_2$ are precisely the set of relations in the child subtrees of that node and $S = S_1 \cup S_2$. Both the checking of whether $(S_1, S_2)$ is a joinable pair of $S$ (step 4) as well as identifying the join node $j$ for this pair (step 5) can be

3 Although our basic approach does not reuse optimal query subplans and therefore does not really need to use bestPlan[], we maintain this structure for ease of transition when we present our enhanced algorithms later.
efficiently performed based on the following property: \((S_1, S_2)\) is a joinable pair of \(S\) with join node \(j\) if \(j\) is the only join node in \(P\) that refers to some relation in each of \(S_1\) and \(S_2\). As an example, consider the query plan \(P = Q_a\) in Figure 4. If \(S = \{(R_1, R_2, R_3, R_4)\}\), then \((\{R_1, R_2\}, \{R_3, R_4\}\) is not a joinable pair of \(S\) because there is more than one join node in \(P\) (i.e., \(p_{14}\) and \(p_{23}\)) that refers to each of \(\{R_1, R_2\}\) and \(\{R_3, R_4\}\). On the other hand, \((\{R_1, R_2, R_3\}, \{R_4\}\) is a joinable pair of \(S\) as there is exactly one join node \(p_{14}\) in \(P\) that refers to each of \(\{R_1, R_2, R_3\}\) and \(\{R_4\}\).

Given the join node \(j\) for a joinable pair \((S_1, S_2)\) of \(S\), \textbf{GenerateSubplan} first moves \(j\) so that \(i\) becomes the parent join node of \(j\) in \(P\) (steps 6 and 7); if \(j = \text{null}\), then \(j\) becomes the root join node in \(P\). This transformation of \(P\) is performed by calling the Swap function (Algorithm 3) iteratively to swap \(j\) with its parent node in \(P\). In step 6, \(\text{par}(j)\) denotes the parent join node of \(j\) in \(P\) where \(\text{par}(j) = \text{null}\) if \(j\) is the root join node in \(P\).

After the relocation of \(j\) in \(P\), \textbf{GenerateSubplan} recursively generates an optimal subplan for each of \(S_1\) and \(S_2\) (steps 8-9).

Steps 10 uses the best plan found so far to eliminate non-optimal subplans. Step 11 stores the cheapest query plan \(P\) for joining \(S\) in bestPlan[\(S\)].

The evaluation cost of a query plan \(P\) is given by \(\text{cost}(P)\); if \(P = \text{null}\), then \(\text{cost}(P) = \infty\). The cost model for estimating the cost of query plans is discussed in Section 6.2.

Since the query subplan \(SP\) for joining \(S\) is embedded within a complete query plan \(P\), we use \textbf{subtree}(\(P, S\)) to denote \(SP\). More precisely, \textbf{subtree}(\(P, S\)) denotes the smallest subtree \(SP\) in \(P\) (in terms of the number of relations in \(S\)) that satisfies the following two properties: (1) \(SP\) contains all the relations in \(S\), and (2) \(SP\) contains all the compensation operators that are between the root join node \(r\) in \(SP\) and the closest ancestor join node \(r'\) (if \(r'\) does not exist, then \(SP = P\)). As an example, in Figure 4(b), \textbf{subtree}(\(Q_b, \{R_1, R_2, R_3\}\)) is the subplan of \(Q_b\) rooted at the node \(\pi_{R_1}\).

Given a query plan \(P\) and an internal non-root join node \(i\) in \(P\), the function \textbf{Swap} (Algorithm 3) transforms \(P\) by swapping \(i\) with its parent join node \(j\) in \(P\). If there are compensation operators between \(i\) and \(j\), we first pull these compensation axes above \(b\) by applying appropriate rewriting rules from Table 2, 4 or 5, to make \(i\) and \(j\) adjacent. Then we swap \(i\) and \(j\) by applying an appropriate assoc/l-awcm/assoc rewriting rule. The structural transformation for this kind of swap is illustrated in Figure 1. Since our approach provides complete join reordering for \(C_{\text{join}}\), it is always possible to find appropriate rewriting rules to perform the swap operation.

5.2 Enabling Reuse of Query Subplans

In this section, we present a refinement of the basic approach presented in the previous section to enable the reuse of optimal query subplans whenever possible.

As illustrated by Example 5.1, the compensation operators generated from an invalid transformation could affect the reusability of query subplans. The following example provides the intuition for the condition under which it is correct to reuse the best query subplan for joining some subset of relations \(S\) (w.r.t. some complete query plan) in a different complete query plan for joining \(S\).

Example 5.2. Consider the three query subplans for joining \(S = \{R_1, R_2, R_3\}\) in Figures 4(a), (b), and (c). Note that \textbf{subtree}(\(Q_b, S\)) rooted at \(\pi_{R_1}\) is equivalent to \textbf{subtree}(\(Q_a, S\)) because the compensation operators \((\pi_{R_1}, \beta_{R_2}, \beta_{R_3})\) generated by swapping the two join operators \(p_{12}\) and \(p_{23}\) in \(Q_a\) to derive \(Q_b\) are all within \textbf{subtree}(\(Q_a, S\)). On the other hand, \textbf{subtree}(\(Q_c, S\)) rooted at \(p_{23}\) is not equivalent to \textbf{subtree}(\(Q_a, S\)) rooted at \(p_{23}\) because all the compensation operators associated with the swap of the join operators are outside of \textbf{subtree}(\(Q_a, S\)).

The above example illustrates that query subplan equivalence needs to take into account of the compensation operators that are associated with the join operators within the subplans. Specifically, a join operator depends on a compensation operator in two scenarios. First, if a sequence of compensation operators \(C\) is generated when a join operator \(o_1\) is swapped above another join operator \(o_2\), then both \(o_1\) and \(o_2\) depend on each operator in \(C\). Second, if a compensation operator \(c\) is swapped above a join operation \(a\) and it causes the join type of \(a\) to be changed, then \(a\) depends on \(c\).

To capture the dependencies among join and compensation operators, we introduce a new labeled edge type in query plans termed dependency edge (d-edge). For the first scenario, for each \(o_1 \in [1, 2]\), we add a d-edge from \(o_1\) to a virtual node \(v\) that represents \(C\) with label \((\alpha_1, \alpha_2)\) to indicate that \(C\) is generated when \(\alpha_2\) is swapped above \(\alpha_1\). For the second scenario, we add a d-edge from \(c\) to \(v\) with label \((\alpha, \beta)\) to indicate that the join type of \(a\) was changed from \(c\) is swapped above \(\alpha\). We denote a d-edge from node \(c\) to \(v\) with label \(\ell\) by \(c \xrightarrow{\ell} v\).

Example 5.3. Referring once more to Figure 4, when the join operators \(p_{12}\) and \(p_{23}\) in \(Q_a\) are swapped to derive \(Q_b\), it generates the sequence of compensations \(C = (\pi_{R_1}, \beta_{R_2}, \beta_{R_3})\). To represent the dependency of the join operators \(C\) on, our approach would add two d-edges: \(e_1 = o_1 \xrightarrow{\ell} v_1\) and \(e_2 = o_2 \xrightarrow{\ell} v_2\), where \(v\) is a virtual node that represents \(C\).

We now state the precise conditions for two query subplans to be considered equivalent. Given two d-edges, \(e_1\) and \(e_2\), where each \(e_j = o_j \xrightarrow{\ell} v_1, e_2\) defined to be equivalent if and only if \(p_1 = p_2\) and \(label_1 = label_2\). Note that the definition does not require that \(v_1 = v_2\) because compensation operators could change form when they are swapped (e.g., Rules 3 and 8 in Table 2). Given a query subplan \(SP\), we use \textbf{ExtDEdge}(\(SP\)) to denote the set of d-edges whose source nodes are within \(SP\) but their destination nodes are outside of \(SP\). We use \textbf{ExtDEdge}(\(SP_1\)) \(\equiv\) \textbf{ExtDEdge}(\(SP_2\)) to denote that the d-edges in both \textbf{ExtDEdge}(\(SP_1\)) and \textbf{ExtDEdge}(\(SP_2\)) are equivalent. We have the following result.

Theorem 5.4. Given two query plans \(P_1\) and \(P_2\) where each \(P_i\) contains a subplan \(SP_i\) for joining a set of relations \(S\), \(SP_1\) and \(SP_2\) are equivalent subplans if \textbf{ExtDEdge}(\(SP_1\)) \(\equiv\) \textbf{ExtDEdge}(\(SP_2\)).
Example 5.5. Consider the two query subplans for joining \( S = (R_1, R_2, R_3) \) in Figures 4(g) and (h). Note that there are three d-edges in \( Q_d \): \( e_1 : p_{13} \circ (p_{11} \circ (p_{12} \circ (\pi_{R_1} \gamma_{R_1} \gamma_{R_2} \gamma_{R_3} \sigma_{R_1}))) \) and \( e_2 : p_{13} \circ (\pi_{R_1} \gamma_{R_1} \gamma_{R_2} \gamma_{R_3} \sigma_{R_1}) \). Both \( e_1 \) and \( e_2 \) are added from the swapping of \( p_{11} \) and \( p_{12} \) in \( Q_d \) to derive \( Q_a \), while \( e_3 \) is added from the pulling up of compensation operators \( C_1 \) above \( \sigma_{p_{0}} \) to derive \( Q_f \). Since \( \text{ExtEdge}(\text{subtree}(Q_g, S)) = \{e_1\} \) and \( \text{ExtEdge}(\text{subtree}(Q_h, S)) = \text{null} \), \( \text{subtree}(Q_h, S) \) and \( \text{subtree}(Q_g, S) \) are not equivalent query subplans.

5.3 Handling Full-Outerjoins

In this section, we explain how our join enumeration approach can be easily extended to handle the class of all join queries \( C_f \) including those queries with full outerjoins. As discussed in Section 3, although our approach does not provide complete join reorderability for \( C_f \), any join reordering supported by TBA is also supported by our approach.

Since our approach provides only partial reorderability for \( C_f \), it might not be always possible for Algorithm 3 to find an applicable rewriting rule to perform a swap operation involving a full outerjoin. For such cases, Swap simply returns null to indicate that the swap cannot be performed. If a call to Swap returns null, then GenerateSubplan would abandon the construction of that query subplan to join \( S_1 \) and \( S_2 \).

5.4 Discussion

To the best of our knowledge, our approach is the first top-down plan enumeration for query plans with compensation operators. CBA [12] described a bottom-up plan enumeration algorithm (based on the concept of Nullification Sets) for query plans containing \( \lambda \) and \( \beta \) compensation operators. However, their algorithm simply enumerates all possible join plans without any pruning or reusing of query subplans.

Incorporating cost-based pruning and dynamic programming-style reusing of query subplans is difficult for bottom-up plan enumeration because of the presence of compensation operators in query plans. As we have explained in Example 5.1, compensation operators complicate the reasoning about query subplan equivalence. Therefore, we have chosen to design a top-down approach for plan enumeration in this paper that is able to incorporate both cost-based pruning as well as reusing of optimal subplans.

6 IMPLEMENTATION ISSUES

In this section, we discuss how the compensation operators in our approach can be implemented. In general, there are two ways to implement new query operators. The first is a language-based approach that implements the new operators at the SQL language level. Given a query plan \( P \) produced by the optimizer, if \( P \) contains any compensation operator, \( P \) will be rewritten into a SQL query that enforces the join ordering in \( P \) for execution. The second approach is a native approach that implements at least one evaluation algorithm for each compensation operator.

In this paper, we focus on the first approach as it is a less intrusive approach that can be more easily implemented.

6.1 SQL-level Implementation

Among the four compensation operators (\( \lambda, \beta, \gamma, \gamma^* \)) used in our approach, there already exists SQL-level implementations for both \( \lambda \) and \( \beta \) operators, which were introduced by CBA [12]. We first overview how \( \lambda \) and \( \beta \) are implemented in CBA, and explain the implementation of the operators \( \gamma \) and \( \gamma^* \) introduced by our approach. A complete example showing the SQL implementation of a query plan with compensation operators is given in Appendix E.

The nullification operator \( \lambda_{p_{11}}A_{i} \) (see Section 2.5) is easily implemented using SQL’s case expression for each attribute \( A_{i,j} \) as follows: “CASE WHEN \( p_{11} \) THEN \( A_{i,j} \) END AS \( B_{i,j} \)”. Here, \( B_{i,j} \) is set to \( A_{i,j} \) if \( p_{11} \) is true; and null, otherwise.

For the best-match operation \( \beta(R) \), the key idea behind CBA’s implementation is to sort \( R \) to generate \( R' \) in such a way that for each spurious tuple \( t \in R' \), \( t \) is immediately preceded by a tuple in \( R' \) that dominates or duplicates \( t \). In this way, the spurious tuples in \( R' \) can be easily eliminated by a scan of \( R' \). Depending on the \( \lambda \) operators in a query plan, the implementation of \( \beta \) in general might require more than one sorting of \( R \) (with different orderings) to completely eliminate all its spurious tuples. The sorting and spurious tuple elimination operations in \( \beta \) are implemented using SQL’s window function construct.

We now discuss how the two new operators introduced by our approach can be easily implemented following their definitions in Section 4.2. The \( \gamma_{A}(R) \) operation is simply implemented using a SELECT subquery to eliminate all tuples in \( R \) with a non-null value for some attribute in \( A \).

The \( \gamma_{\lambda,\beta}(R) \) operation is implemented directly following its definition using a combination of SELECT subquery, \( \lambda \) operation, and \( \beta \) operation. A SELECT subquery is first used to find the tuples in \( R \) with non-null values for all attributes in \( A \), and a \( \lambda \) operation is applied on these tuples to nullify all its attribute values except for the attributes \( B \). Finally, we apply the \( \beta \) operation to eliminate all the spurious tuples.

In summary, all the compensation operators used in our approach can be easily implemented at the SQL level.

6.2 Cost Model

In this section, we discuss the cost models for the four compensation operations, \( \lambda, \beta, \gamma, \gamma^* \).

For \( \lambda \) and \( \beta \), which are essentially selection operations, the cost for them is simply the cost of a scan through their operand.

As the evaluation of \( \gamma \) is dominated by the cost of sorting, the cost for \( \beta \) is \( O(n \log n) \), where \( n \) is the size of its operand. For \( \gamma^* \), since it is semantically equivalent to a \( \lambda \) operation followed by a \( \beta \) operation, the cost for \( \gamma^* \) is also \( O(n \log n) \), where \( n \) is the size of its operand.

7 EXPERIMENTAL RESULTS

In this section, we present an experimental evaluation to demonstrate the performance improvements that can be achieved by our approach of supporting more join reorderings. Our results show that our approach can improve the running time performance by up to a factor of 2.84 for a 5-table join query in PostgreSQL. Additional performance results using a commercial database system and discussions are presented in Appendix F.
Our experiments were conducted on an Intel Xeon Processor E5-2603 v2 server running Ubuntu Linux 14.04 with 32GB memory and two 1TB SATA disks (one for the OS and database server installation and the other for database files storage). The queries were run on PostgreSQL 9.6.1 database server with shared_buffers = 5GB and work_mem = 1GB.

We evaluated the following three queries (with increasing complexity) on three TPC-H benchmark datasets (with scaling factors 1, 10, and 100, respectively):

**Query Q1:**

1. \( Q_1 \): \( R_1.\ p_{12} \land (R_2. \ p_{23} \land R_3) \)
2. \( Q_2 \): \( R_1.\ p_{12} \land ((R_2. \ p_{24} \land R_1) \ p_{25} \land R_1) \)
3. \( Q_3 \): \( R_1.\ p_{12} \land ((R_2. \ p_{24} \land (R_3. \ p_{25} \land R_3)) \ p_{26} \land R_3) \)

where \( R_1 = \text{Supplier}, R_2 = \text{Partsupp}, R_3 = \sigma_{p\text{.name}=c_1}(\text{Part}), R_4 = \text{Lineitem}, R_5 = \sigma_{c.\text{totalprice}>c_2}(\text{Orders}), p_{12} \triangleq \{ \text{s_suppkey} = \text{ps_suppkey} \land (\text{s_acctbal} > \text{ps_supplycost}) \}, p_{23} \triangleq \{ \text{ps_partkey} = \text{p_partkey} \}, p_{24} = \{ \text{ps_suppkey} = \text{l_suppkey} \} \land \{ \text{ps_partkey} = \text{l_partkey} \}, p_{25} = \{ \text{l_orderkey} = \text{o_orderkey} \}. \)

where \( c_1 \) and \( c_2 \) are some constant values, and \( \nu \) is a parameter that is used to vary the selectivity factor of \( R_1.\ p_{12} \to R_2 \) (denoted by \( f_{12} \)); i.e., \( f_{12} = \frac{|R_1.\ p_{12}|}{|R_1|} \).

As it will become clear later, \( f_{12} \) affects the relative performance of the compared query plans.

For each query \( Q_1 \), we compared its running time for two query plans: the query plan produced by PostgreSQL’s query optimizer (denoted by \( p_{ECA}^1 \)) and the query plan produced by our approach (denoted by \( p_{p^g}^1 \)). Each reported timing for a query plan is the average of three executions of the plan with a cold cache.

**Query Q1.** The two query plans for \( Q_1 \) are shown in Figures 5(a) and (b). Since \( Q_1 \) has two antijoins and assoc(\( \bowtie \), \( \bowtie \)) is an invalid transformation, it is not possible to reorder these two joins using a conventional optimizer. Hence, the only possible join ordering considered by PostgreSQL is \( p_{p^g}^1 \). In contrast, our approach could reorder the joins in \( Q_1 \) to produce the plan \( p_{ECA}^1 \) by applying Rule 15 in Table 3. We expect \( p_{ECA}^1 \) to outperform \( p_{p^g}^1 \) when \( f_{12} \) is large (i.e., a large proportion of tuples in \( R_1 \) do not join with any tuple in \( R_2 \)) as this would reduce the cost of joining the \( R_1 \) tuples from the first join with \( R_3 \) in \( p_{ECA}^1 \). The SQL queries corresponding to these two query plans are given in Appendix E.

The performance comparisons for \( Q_1 \) are shown in Figures 6(a) to (c) for the 1GB, 10GB, and 100GB TPC-H databases with the selectivity factor of \( R_1.\ p_{12} \to R_2 \) (i.e., \( f_{12} \)) being varied. The performance results follow our expectation: \( p_{ECA}^1 \) outperforms \( p_{p^g}^1 \) except for a few cases when \( f_{12} \) is small. Moreover, the performance improvement of \( p_{ECA}^1 \) over \( p_{p^g}^1 \) also increases with the database size: our approach wins by up to a factor of 1.36, 1.47, and 1.65 for the 1GB, 10GB, and 100GB databases, respectively.

**Query Q2.** A straightforward query plan obtained from a direct translation of \( Q_2 \) is \( p_{direct}^2 \) shown in Figure 5(c). PostgreSQL is able to reorder the joins in \( p_{direct}^2 \) to the plan \( p_{p^g}^2 \) in Figure 5(d) by applying the l-asscom(\( \bowtie \), \( \bowtie \)) rule (which is a valid transformation). PostgreSQL chooses \( p_{p^g}^2 \) over \( p_{direct}^2 \) because performing \( R_2 \ p_{23} \to R_1 \) first would reduce the size of \( R_2 \) and hence reduce the cost of \( R_2 \ p_{25} \to R_4 \).

In contrast, our approach is able to first apply the assoc(\( \bowtie \), \( \bowtie \)) rule (i.e., Rule 14 in Table 3) to obtain the intermediate plan \( p_{int}^2 \) (Figure 5(e)), and then apply the assoc(\( \bowtie \), \( \bowtie \)) rule (Rule 22 in Table 3) and l-asscom(\( \bowtie \), \( \bowtie \)) rule to derive \( p_{ECA}^2 \) in Figure 5(f). Similarly as for \( Q_1 \), we expect \( p_{ECA}^2 \) to outperform \( p_{p^g}^2 \) when the join selectivity \( f_{12} \) is large, because a larger \( f_{12} \) means that there would be fewer \( R_2 \) tuples to join with \( R_1 \) and \( R_3 \), and thus reducing the join cost.

The performance results comparing \( p_{p^g}^2 \) and \( p_{ECA}^2 \) are shown in Figures 6(d) to (f) for the 1GB, 10GB, and 100GB TPC-H databases. The results show that our approach wins by up to a factor of 2.20, 2.17, 2.35 for the 1GB, 10GB, and 100GB databases, respectively.
Query Q3. Q3 has one additional join with R5 compared to Q2. The two query plans obtained are shown in Figure 5(g) and (h). Their performance comparison in Figures 6(g) to (i) shows that our approach wins by a factor of 2.20, 2.45, 2.84 for the 1GB, 10GB, 100GB databases, respectively. The explanation for this performance trend is similar to that of Q2 and we omit a detailed discussion.

8 CONCLUSIONS
The join reordering problem is a fundamental task in query optimization. While this is a well studied problem for inner-join queries, there is currently no single best solution for reordering joins in complex queries involving non-inner joins. In this paper, we have formalized the notion of complete join reorderability and developed a new approach for join reordering that enables more join reorderings compared to the state-of-the-art solutions. Our performance study has demonstrated that enabling more join reorderings can improve query execution time by up to a factor of 2.84 for a 5-table join query. As part of our future work, we plan to investigate enabling complete join reorderability for queries with full outer joins.

Acknowledgements We would like to thank the reviewers for their constructive comments. This research is supported in part by MOE Grant R-252-000-665-114.
REFERENCES


A PROOFS FOR REWRITING RULES 3 & 18

In this section, we present partial proofs of Theorems 4.2 and 4.4 by presenting the proofs of two rewriting rules: Rule 3 in Table 2 and Rule 18 in Table 3. The other rewriting rules can be proved similarly.

Rule 3: $R_2 \xrightarrow{p_{12}} y(R_1) \Rightarrow y^*_w(R_1, R_2) \xrightarrow{p_{12}} R_1$, where $p_{12}$ does not reference $R_3$.

Proof. Let LHS denote $R_2 \xrightarrow{p_{12}} y(R_1)$ and RHS denote $y^*_w(R_1, R_2) \xrightarrow{p_{12}} R_1$. By the definition of $y$, $\text{attr}(R_3) \subseteq \text{attr}(R_1)$. LHS removes all tuples in $R_1$ with non-null values for $\text{attr}(R_3)$ (by $y_w$) and computes a left outerjoin between $R_2$ and $y(R_1)$. RHS first computes a left outerjoin between $R_2$ and $R_1$, and for the tuples in the join result with non-null values for $\text{attr}(R_3)$, their values for attributes in $\text{attr}(R_1)$ are set to null (by $y^*$). Thus, each tuple in LHS and RHS is either of the form $(r_2, \text{null})$ or $(r_2, r_1)$, where $r_2 \in R_2$ and $r_1 \in R_1$.

First, we show that LHS $\subseteq$ RHS. For a tuple $t$ of the form $(r_2, \text{null})$ in LHS, we know that the values of $\text{attr}(R_3)$ for $t$ are null; otherwise, $t$ would have been removed by $y_w$. Moreover, the join predicate $p_{12}$ must evaluate to true for $t$. Therefore, $t \in R_2 \xrightarrow{p_{12}} R_1$, and $t$ is not removed or modified by $y^*_w$, which implies that $t \in \text{RHS}$.

For a tuple $t$ of the form $(r_2, \text{null})$ in LHS, this implies that either (a) $r_2$ does not join (w.r.t. $p_{12}$) with any tuple in $R_1$ or (b) $r_2$ joins (w.r.t. $p_{12}$) only with tuples in $R_1$ that have non-null values for $\text{attr}(R_3)$. For case (a), $t \in R_2 \xrightarrow{p_{12}} R_1$ and $t$ is not removed or modified by $y^*_w$; thus, $t$ is also in RHS. For case (b), $y^*_w R_1, R_2$ will set the values for $\text{attr}(R_1)$ to null, and therefore $(r_2, \text{null})$ is in RHS. Therefore, LHS $\subseteq$ RHS.

Next, we show that RHS $\subseteq$ LHS. For a tuple $t$ of the form $(r_2, r_1)$ in RHS, we know that join predicate $p_{12}$ evaluates to true for $r_2$ and $r_1$, and that the $t$’s values for $\text{attr}(R_3)$ are null. Therefore, $t$ is also in LHS. For a tuple $t$ of the form $(r_2, \text{null})$ in RHS, either $r_2$ does not join (w.r.t. $p_{12}$) with any tuple from $R_1$, or $r_2$ joins (w.r.t. $p_{12}$) only with tuples in $R_1$ that have non-null values for $\text{attr}(R_3)$. By the definition of left outerjoin, $t$ is also in LHS. Thus, RHS $\subseteq$ LHS.

Rule 18: $R_1 \xrightarrow{p_{21}} (R_2 \xrightarrow{p_{23}} R_3) \Rightarrow \pi_{1}(R_1, R_2) \xrightarrow{p_{21}} (R_1 \xrightarrow{p_{23}} R_3)$.

Proof.

$R_1 \xrightarrow{p_{21}} (R_2 \xrightarrow{p_{23}} R_3) \Rightarrow R_1 \xrightarrow{p_{21}} \pi_{1}(R_1, R_2) \xrightarrow{p_{21}} (R_1 \xrightarrow{p_{23}} R_3)$, by Equation 9 in Section 4.3.

$\Rightarrow R_1 \xrightarrow{p_{21}} \pi_{1}(R_1, R_2) \xrightarrow{p_{23}} (R_1 \xrightarrow{p_{23}} R_3)$, by Equation 10 in Section 4.3.

$\Rightarrow \pi_{1}(R_1, R_2) \xrightarrow{p_{21}} (R_1 \xrightarrow{p_{23}} R_3)$, by Rule 3 in Table 2.

B RULES FOR PULLING $\lambda$

Based on Rules 26 and 27 discussed in Section 4.4, Table 5 shows the rewriting rules for pulling $\lambda$ above join operators.

C ENHANCED APPROACH

This section presents the details of our enhanced approach introduced in Section 5.2 to enable the reuse of optimal query subplans.

The enhanced approach consists of the same main driver as the basic approach (Algorithm 1 in Section 5.1) and modified versions of GenerateSubPlan and Swap (shown as Algorithms 4 and 5). GenerateSubPlan uses two additional functions, GetBestPlan (shown as Algorithm 6) and UpdateBestPlan (details are not shown) to reuse optimal subplans when applicable.

Recall from the discussion in Section 5.2 that for a set of relations $S$ to be joined, there could be multiple query plans each consisting of a different subplan for joining $S$, and these subplans are not necessarily equivalent. Our enhanced approach uses bestPlan[S] to store these incomparable optimal subplans for joining $S$.

Specifically, in GenerateSubPlan, if there is no reusable optimal subplan for joining a set of relations $S$, steps 7 to 15 will compute an optimal subplan $P'$ for $S$ and invoke UpdateBestPlan($S, P'$) to insert $P'$ into bestPlan[S]. On the other hand, if an applicable optimal subplan for $S$ is available (retrieved by GetBestPlan in step 3), it will be reused (step 5). The correctness of GetBestPlan is based on Theorem 5.4.

Steps 5 and 17 in Swap function generate appropriate d-edges in query plans when operators are swapped to enable reasoning about query subplan equivalence.

D HANDLING NULL-TOLERANT JOIN PREDICATES

Although we have so far restricted the discussed query classes to contain only null-intolerant join predicates, our approach can be
Algorithm 5: Swap$(P, i)$

**Input:** $P$ is a query plan where $i$ is an internal non-root join node in $P$

**Output:** A modified query plan $P'$ with $i$ swapped with its parent join node

1. let $j$ be the parent join node of $i$ in $P$
2. if there are compensation operators between $i$ and $j$ then
   3. apply appropriate rewriting rules in Table 2, 4 or 5 to pull these compensation operators above $j$
   4. if swapping the $f$ and some compensation operator $c$ has changed the joint type of $j$ then
      5. add a d-edge $j \xrightarrow{(c,i)} c$ to $P$
3. let $T_r$ and $T_l$ denote the left and right child subtrees of $i$ in $P$
4. if $j$ refers to $T_r$ and not to $T_l$ then
   5. $P' = \text{apply appropriate assoc rewriting rule to swap } i \text{ and } j$
else
5. if $j$ refers to $T_r$ and not to $T_l$ then
   6. if $i$ is the left child of $j$ then
      7. $P' = \text{apply appropriate l-asscom rewriting rule to swap } i \text{ and } j$
   else
      8. $P' = \text{apply appropriate r-asscom rewriting rule to swap } i \text{ and } j$
   9. if the swapping of $i$ and $j$ has produced a set of compensation operators $C$ then
      10. let $v$ be a virtual node representing $C$
      11. add d-edges $i \xleftarrow{(j,i)} v$ and $j \xrightarrow{(j,i)} v$ to $P$
12. return $P$

Algorithm 4: GenerateSubplan($P, i, S$)

**Input:** $P$ is a query plan, $i$ is either null or a join node in $P$, $S$ is a subset of relations in $P$

**Output:** A query plan $P'$ that is derived from $P$ that contains an optimal subplan $SP$ for joining $S$. If $i \neq \text{null}$, then $SP$ is a child subtrees of $i$; otherwise, $P' = SP$

1. if $S = \{R_i\}$ then
   2. return $P$ with $R_i$ replaced by $\text{bestAccess}[R_i]$
   3. $P_{opt} = \text{GetBestPlan}(SP)$
   4. if $P_{opt} \neq \text{null}$ then
      5. replace subtree($S, P$) in $P$ by subtree($P_{opt}, S$)
   6. return $P$
   7. foreach joinable pair ($S_1, S_2$) of $S$ do
      8. initialize $P' = P$
      9. let $j$ be the join node in $P'$ that joins $S_1$ and $S_2$
   while $par(j) \neq i$ do
      11. $P' = \text{Swap}(P', j)$
      12. $P' = \text{GenerateSubplan}(P', j, S_1)$
      13. $P' = \text{GenerateSubplan}(P', j, S_2)$
      14. if $\text{cost}(\text{subtree}(P', S)) < \text{cost}(\text{subtree}(\text{GetBestPlan}(S, P', S)))$ then
         15. update $P_{opt} = \text{UpdateBestPlan}(P', S)$
   16. return $\text{GetBestPlan}(S, P)$

Algorithm 6: GetBestPlan($P, S$)

**Input:** $P$ is a query plan, $S$ is a subset of relations in $P$

**Output:** A query plan $P_{opt}$ in bestPlan[S] where subtree($P_{opt}, S$) is equivalent to subtree($P, S$)

1. $SP_1 = \text{subtree}(P, S)$
2. foreach query plan $P_{opt}$ in bestPlan[S] do
   3. $SP_2 = \text{subtree}(P_{opt}, S)$
   4. if $\text{ExtDEdge}(SP_1) \equiv \text{ExtDEdge}(SP_2)$ then
      5. return $P_{opt}$
6. return null

easily extended to handle queries with null-tolerant join predicates. As can be expected, we will only have partial join reorderability for queries with null-tolerant predicates.

Indeed, most of the valid transformations summarized in Table 1 do not require the join predicates to be null-intolerant [8]. To handle null-tolerant join predicates, the key modification required to our approach is in query plan enumeration (Section 5): whenever a swap is being considered between a pair of join nodes that involves some null-tolerant join predicate, the Swap function permits the swap if it is allowed by the corresponding valid transformation [8]; otherwise, the Swap function returns null to indicate that the reordering is infeasible.

Let $C_j$ denote the class of join queries with join operators from the set $\{\bowtie\prec, \bowtie\succ, \bowtie\prec\succ, \bowtie\prec\succ\}$ where the join predicates could be null-tolerant or null-intolerant. We have the following result.

Table 5: Rewriting rules for pulling $\lambda$ above join operators

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{in+1}$ does not reference $R_{in+1}$</td>
</tr>
<tr>
<td>29</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ references $R_{i1}$</td>
</tr>
<tr>
<td>30</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ does not reference $R_{i1}$</td>
</tr>
<tr>
<td>31</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ references $R_{i1}$</td>
</tr>
<tr>
<td>32</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ does not reference $R_{i1}$</td>
</tr>
<tr>
<td>33</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ references $R_{i1}$</td>
</tr>
<tr>
<td>34</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ does not reference $R_{i1}$</td>
</tr>
<tr>
<td>35</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ references $R_{i1}$</td>
</tr>
<tr>
<td>36</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ does not reference $R_{i1}$</td>
</tr>
<tr>
<td>37</td>
<td>$\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}} = \beta_{\lambda_{P_iR_j}([R_{i1}, \ldots, R_{in+1}])<em>{\pi</em>{R_{i1}}^{p_{i1}} \cdots \pi_{R_{in+1}}^{p_{in+1}}}}$, where $p_{i1}$ references $R_{i1}$</td>
</tr>
</tbody>
</table>
Theorem D.1. For $C_j$, our approach strictly supersedes both TBA and CBA.

The soundness of the above theorem follows from the fact that our approach allows all valid transformations involving null-tolerant join predicates that are supported in TBA, and CBA does not support join reordering with null-tolerant join predicates.

E SQL QUERIES FOR $Q_1$ IN SECTION 7

The SQL queries executed by PostgreSQL and our approach for $Q_1$ in Section 7 are shown in Figure 7(a) and (b), respectively.

F ADDITIONAL PERFORMANCE RESULTS & DISCUSSIONS

F.1 Experiments with a Commercial DBMS

In this section, we present additional performance results with PostgreSQL that were presented in Section 7. The results for PostgreSQL, our approach for both $Q_2$ and $Q_3$ grow with the data size, this is not the case for $Q_2$. The reason is as follows. We observe that for each of the three databases $D$, the running time for $P^{BG}$ on $D$ is about 9-12 times longer than that for $P^{ECA}$ on $D$. This behaviour also holds for our query plans but only on the 10GB and 100GB databases; for the 1GB database, the running time for $P^{ECA}$ is only 5 times longer than that for $P^{ECA}$. This is because for the 1GB database, $R_4$ fits in main memory, and since $P^{BG}$ is performed first in $P^{ECA}$ to obtain a small intermediate result, joining this intermediate result with $R_4$ also produces a small result. Thus, having an additional table $R_4$ in $Q_2$ does not significantly increase the running time of $P^{ECA}$ over $P^{ECA}$. In contrast, for $P^{BG}$ on the 1GB database, although $R_4$ also fits in main memory, joining $R_4$ in $P^{BG}$ produces a large intermediate result which does not fit in main memory, so the running time for $P^{BG}$ is about 9 times longer than that for $P^{ECA}$. For the 10GB and 100GB databases, since neither $R_4$ nor the intermediate result after joining with $R_4$ can fit completely in main memory, the running time for $Q_2$ is 9-12 times longer than that for $Q_1$ using each of the approaches. For $Q_3$, as the additional $R_5$ is a large table, the performance is affected more by the database size, and the improvement factor of our approach increases with database size.
SELECT s_suppkey
FROM supplier
WHERE NOT EXISTS
(
  SELECT NULL
  FROM (SELECT ps_suppkey, ps_supplycost
        FROM partsupp
        WHERE NOT EXISTS
          (SELECT NULL
           FROM part
           WHERE ps_partkey = p_partkey
           AND p_name = 'cyan orchid indian cornflower saddle'))
  WHERE ps_suppkey = s_suppkey
  AND s_acctbal > :v * ps_supplycost
)

(a) SQL query executed by PostgreSQL for Q1

SELECT s_suppkey
FROM (SELECT s_suppkey, ps_suppkey, ps_partkey, p_partkey,
        max(s_suppkey) OVER (ORDER BY s_suppkey, ps_suppkey, ps_partkey, p_partkey
        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS s_suppkey_p,
        max(ps_suppkey) OVER (ORDER BY s_suppkey, ps_suppkey, ps_partkey, p_partkey
        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS ps_suppkey_p,
        max(ps_partkey) OVER (ORDER BY s_suppkey, ps_suppkey, ps_partkey, p_partkey
        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS ps_partkey_p,
        max(p_partkey) OVER (ORDER BY s_suppkey, ps_suppkey, ps_partkey, p_partkey
        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS p_partkey_p,
        row_number() OVER (ORDER BY s_suppkey, ps_suppkey, ps_partkey, p_partkey
        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) AS row_num
        FROM (supplier LEFT JOIN partsupp
              ON s_suppkey = ps_suppkey AND s_acctbal > :v * ps_supplycost)
            LEFT JOIN (SELECT p_partkey FROM part WHERE p_name = 'cyan orchid indian cornflower saddle')
            AS part ON p_partkey = ps_partkey)
WHERE (s_suppkey <> s_suppkey_p OR
        ps_suppkey <> ps_suppkey_p OR
        ps_partkey <> ps_partkey_p OR
        p_partkey <> p_partkey_p OR
        row_num = 1)
) AS window_table

AND

ps_suppkey IS NULL

(b) Rewritten SQL query executed by our approach for Q1

Figure 7: SQL Queries for Q1 (Section 7)