Complete Join Reordering for Null-Intolerant Joins

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

Yunpeng Niu
miHoYo Inc.
niuyunpeng@u.nus.edu

Chee-Yong Chan
Department of Computer Science
National University of Singapore
chancy@comp.nus.edu.sg

***Abstract*—**
The join reordering problem is a core task in query optimization to find the most efficient evaluation order for join operations. The Enhanced Compensation-based Approach (ECA) is the state-of-the-art approach for this problem which is based on using new operators called compensation operators to enlarge the query plan search space with more join reorderings. However, ECA cannot provide complete join reorderability for queries involving one or more full outerjoins. In this paper, we present the first complete join reordering solution, named CJR. By introducing a new and more expressive compensation operator and an enhanced set of rewriting rules, CJR is able to provide complete join reorderability for all join queries with null-intolerant join predicates. Our experimental results on the Join Order Benchmark demonstrate that CJR can improve query performance by a factor of 12.32.

***Index Terms*—Query optimization, outerjoin, join reordering**

I. Introduction

Finding an efficient evaluation order for the join operations in a query is crucial in join query optimization. This is also known as the *join reordering* problem which has been studied extensively over the years (e.g., [1]–[13]). Join reordering for inner join queries is straightforward since inner joins are associative and commutative. However, it is significantly more difficult to reorder complex join queries involving outerjoins and/or antijoins.

Consider the query $Q = R_1 \overset{p_{12}}{\circ_x} (R_2 \overset{p_{23}}{\circ_y} R_3)$, where each $\overset{p_{ij}}{\circ_t}$ represents a join operation $\circ_t$ between R_i and R_j using the join predicate p_{ij}. Clearly, if both the join operations are inner joins, then the query Q is fully reorderable in the sense that the two join operations in Q can be evaluated in any order. This fully reorderability property is due to the commutativity and associativity of inner joins. However, if a query involves outerjoins, its join reorderability generally becomes more limited. As an example, consider the query $Q_1 = R_1 \overset{p_{12}}{⟕} (R_2 \overset{p_{23}}{⋈} R_3)$[1]. Note that Q_1 can not be reordered to an equivalent query that first join R_1 and R_2 followed by joining R_2 and R_3. That is, Q_1 is not equivalent to $Q_2 = (R_1 \overset{p_{12}}{\circ_{x'}} R_2) \overset{p_{23}}{\circ_{y'}} R_3$ for any combination of join operations $\circ_{x'}$ and $\circ_{y'}$; i.e., Q_2 is an invalid transformation of Q_1 that does not preserve equivalence. TBA [7] is the state-of-the-art transformation-based approach that generates join reorderings based on valid transformations using the associativity and commutativity properties of join operators.

To enable more join reorderings and hence enlarge the query plan space for optimization, [11] introduced the idea of query rewriting using additional operators called *compensation operators* to produce equivalent query plans with different join reorderings. This approach is referred to as CBA (for Compensation-Based Approach) in [14]. Continuing with the above example of Q_1, by using additional compensation operators β and λ, the join operations in Q_1 can be reordered to produce the following equivalent query $Q'_1 = \beta(\lambda_{p_{23},R_2}((R_1 \overset{p_{12}}{⟕} R_2) \overset{p_{23}}{⋈} R_3))$. CBA is able to provide complete join reorderability for queries involving inner joins, semijoins, and/or single-sided outerjoins.

The state-of-the-art approach for reordering joins is ECA (for Enhanced Compensation-based Approach) [14]. ECA has further enhanced CBA with two additional compensation operators as well as new rewriting rules that enable complete join reorderability for an even larger class of queries that include antijoins as well. For example, consider the query $Q_2 = R_1 \overset{p_{12}}{▷} (R_2 \overset{p_{23}}{⋈} R_3)$. Reordering of joins in Q_2 is not possible in CBA, but ECA can rewrite it to an equivalent query $Q'_2 = \pi_{R_1}(\gamma_{R_2}((R_1 \overset{p_{12}}{⟕} R_2) \overset{p_{23}}{⋈} R_3))$ using the new operator γ. Although ECA is an improvement over CBA, ECA still does not provide complete join reorderability for all join queries: specifically, not all join reorderings are possible for queries with full outerjoin.

In this paper, we present the first complete solution to the join reorderability problem for null-intolerant joins[2]. Specifically, we propose a new, comprehensive approach termed CJR (for Complete Join Reordering) that provides complete join reorderability for all join queries (i.e., queries with any combination of inner/semi/anti-joins and single-sided/full outerjoins). CJR is also a compensation-based approach that is based on a new compensation operator (τ) and the β and λ compensation operators from [11].

For example, consider the query $Q_3 = R_1 \overset{p_{12}}{⟗} (R_2 \overset{p_{23}}{⋈} R_3)$ where its join operations cannot be reordered using ECA

This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG-GC-2019-001-2A). Yunpeng Niu's work was done while he was a student at National University of Singapore.

[1]In this paper, we use $\overset{p_{ij}}{⟕}$, $\overset{p_{ij}}{⟗}$, $\overset{p_{ij}}{▷}$, and $\overset{p_{ij}}{◁}$ to denote respectively, the left outerjoin, right outerjoin, full outerjoin, left antijoin, and right antijoin between two join operands R_i and R_j using the the join predicate p_{ij}.

[2]Null-intolerant joins have null-intolerant join predicates. A predicate is classified as *null-intolerant* if it cannot evaluate to $true$ when referencing a null value; otherwise, it is considered to be *null-tolerant*.

or any existing approach due to non-associativity of full outerjoin. However, our approach CJR can reorder Q_3 to $\beta(\lambda_{R_3\ not\ null,R_2}((R_1 \overset{p_{12}}{⟗} R_2) \overset{p_{23}}{⟕} R_3))$. A fundamental difference between ECA and CJR is that ECA adopts a forward-based approach of rewriting complex join operators which first expresses a more complex join operator in terms of simpler join operators and performs reordering of the simpler operators. While such an approach works for reordering antijoins, it is not applicable for reordering full outerjoins. In contrast, CJR derives rewriting rules for full outerjoins via a backward-based approach; more details are discussed elsewhere [15]. Full outerjoin is an important class of join operations that is widely used in many traditional and modern applications to preserve information including data warehousing, data integration, and schema mapping [11], [16], as well as probabilistic databases [16]–[18] and graph data processing [19], [20].

We summarize our contributions as follows. First, we propose a compensation-based join reordering approach with a new compensation operator (Section III). We derive the complete set of rewriting rules for reordering full outerjoins and other join operators, and also rewriting rules for pulling up compensation operators above join operators. With these rewriting rules, our approach is the first to achieve complete join reorderability for all join queries (involving any combination of inner joins, one-sided/full outerjoins, and antijoins) with null-intolerant joins. Our approach strictly supersedes all existing approaches in terms of reorderability regardless of whether the join predicates are null-intolerant or null-tolerant.

Second, we extend CBA's SQL-level implementation techniques for the β operator to support queries involving full outerjoins, by redefining a key concept called nullification sets on a tuple basis, and providing a sound algorithm to compute the sorting order used for implementing β based on the new definition (Section IV).

Third, we demonstrate the effectiveness of the our complete join reordering rewriting rules with an experimental evaluation on the Join Order Benchmark and TPC-H Benchmark datasets. Our results show that with the enlarged query plan search space enabled by our approach, the query performance can be improved by up to a factor of 12.32 (Section V).

II. Background

A. Notations

We use $R_i \overset{p_{ij}}{\circ_a} R_j$ to denote a join operation with join operator $\circ_a$ and join predicate p_{ij} that refers to join operands R_i and R_j. Note that the subscript a and/or join predicate p_{ij} may be dropped from the join operator if not needed.

For notational convenience, we may use set operations on relations to specify a set/list of attributes. For example, the expression $R_1 \cap R_2$ in $\pi_{R_1 \cap R_2}(R_1 \overset{p_{12}}{\bowtie} R_2)$ denotes the list of attributes that are common to relations R_1 and R_2, and $A \subseteq R_1$ or $A \subseteq attr(R_1)$ indicates that A is a subset of the attributes of R_1.

1	null	3	4	5
1	null	3	null	5

Fig. 1: Example of dominated tuple

B. Compensation-based Approaches

In this section, we provide an overview of the two compensation-based approaches (i.e., CBA [11] and ECA [14]) for join reordering.

CBA uses two unary compensation operators $\lambda_{p,A}$ and β. The first operator, $\lambda_{p,A}(R)$, is known as the nullification operator. Given a selection predicate p on a relation R and a subset of attributes A of R, $\lambda_{p,A}(R)$ returns all the tuples in R such that for each tuple $t \in R$, if p evaluates to $true$ on t, then t remains unchanged; otherwise, all the values of attribute $A_i \in A$ of t will be set to $null$. That is,

$$\lambda_{p,A}(R) = \sigma_p(R) \cup \{t \in R - \sigma_p(R) |\ t.A_i = null\ \forall\ A_i \in A\}.$$

The predicate p is referred to as a *nullification predicate* and A is referred to as *nullification attributes*. We say that $\lambda_{p,A}(R)$ *nullifies* A if p does not evaluate to $true$.

The second operator, $\beta(R)$, which is known as the best-match operator, returns all the non-spurious tuples in R. A tuple $t \in R$ is a *spurious tuple* if t is a duplicated tuple or if t is *dominated* by some other tuple in R. Given two tuples $t, t' \in R$, t 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'. For example, in Figure 1, the first tuple dominates the second tuple. If we apply β to the table, the second tuple will be eliminated.

With the compensation operators, the join operators can be expressed in a canonical form that corresponds to computing a outer cartesian product[3] (denoted by $\overset{\circ}{\times}$), followed by a nullification operation λ and a best-match operation β:

$$R_1 \overset{p_{12}}{\bowtie} R_2 = \beta(\lambda_{p_{12},R_1 \cup R_2}(R_1 \overset{\circ}{\times} R_2))$$
$$R_1 \overset{p_{12}}{⟕} R_2 = \beta(\lambda_{p_{12},R_2}(R_1 \overset{\circ}{\times} R_2))$$

By deriving new rewriting rules to enable the compensation operators to be pushed down into (or pulled up from) relational expressions, CBA is able to reorder joins in a query by first rewriting the query into the above canonical form, reordering the cartesian product operations, and rewriting the reordered expression in terms of join operators whenever applicable. An example is illustrated in Section I with Q_1 being rewritten into an equivalent query Q_1' with the join operations reordered.

To support join reordering for queries with anti-join(s), ECA introduced two more compensation operators, γ and γ^*. $\gamma_A(R)$ is defined as $\gamma_A(R) = \{r \in R \mid r.A\ is\ null\}$, and $\gamma^*_{A,B}(R)$ is defined as $\gamma^*_{A,B}(R) = \beta(\gamma_A(R) \cup R')$, where $R' = \lambda_{false,attr(R)-B}(R - \gamma_A(R)))$. $\gamma^*_{A,B}(R)$ is similar to

[3]The outer cartesian product operator $\overset{\circ}{\times}$ preserves tuples from non-empty relations; i.e., $R \overset{\circ}{\times} S = R \overset{true}{⟗} S$.

$\gamma_A(R)$ and keeps all the R tuples where all attributes in A are null; for the R tuples that are not in $\gamma_A(R)$, $\gamma^*_{A,B}(R)$ nullifies all the attributes except for attributes in B. The two operators are used to derive rewriting rules for reordering antijoin queries.

Besides using additional compensation operators, ECA also differs from CBA in their approach for enumerating query plans. In CBA, the join operations in a query are conceptually reordered via its canonical form in terms of reordering cartesian product operations, which is achieved by enumerating query plans using a bottom-up approach based on the concept of nullification sets[4]. However, CBA's bottom-up enumeration approach does not permit cost-based pruning of query plans. In contrast, ECA enumerates query plans using a top-down, cost-based query rewriting approach.

C. Join Reordering Properties

To help detect invalid join reorderings, [7] has characterized valid and invalid join orderings using three properties (named assoc, l-asscom, and r-asscom) that are derived from the associativity and commutativity properties of join operators and are defined as follows:

- assoc property: $(e_1 \overset{p_{12}}{\circ_a} e_2) \overset{p_{23}}{\circ_b} e_3 = e_1 \overset{p_{12}}{\circ_a} (e_2 \overset{p_{23}}{\circ_b} e_3)$
- l-asscom property: $(e_1 \overset{p_{12}}{\circ_a} e_2) \overset{p_{13}}{\circ_b} e_3 = (e_1 \overset{p_{13}}{\circ_b} e_3) \overset{p_{12}}{\circ_a} e_2$
- r-asscom property: $e_1 \overset{p_{13}}{\circ_a} (e_2 \overset{p_{23}}{\circ_b} e_3) = e_2 \overset{p_{23}}{\circ_b} (e_1 \overset{p_{13}}{\circ_a} e_3)$

We denote these three join transformations by assoc($\circ_a, \circ_b$), l-assoc ($\circ_a, \circ_b$), and r-assoc($\circ_a, \circ_b$), respectively. We say that a specific transformation is *valid* if the corresponding property holds for the specified join operators; and *invalid* otherwise. For example, assoc($\bowtie$, $\rtimes$) is valid but r-assoc($\rtimes$, $\rtimes$) is invalid.

The goal of our compensation-based join reordering approach is to enable the join reordering in each invalid transformation to be possible by expressing the rewriting rule in a more general form (refer to Table I) where a join operator could be changed and/or compensation operator(s) could be introduced after the transformation is applied.

III. Our Approach

In this section, we present the first complete solution named CJR (for Complete Join Reorderability) for the join reorderability problem. Given a query Q that involves any combination of join operators, CJR is able to enumerate equivalent query plans for Q for any join ordering.

CJR is a compensation-based approach that is in the same spirit as the state-of-the-art ECA that uses additional compensation operators to reorder join operations. However, there are two key differences between CJR and ECA. First, while ECA introduces two new compensation operators (γ and γ^*) in addition to the two compensation operators (i.e., λ and β) from CBA, CJR introduces only one new compensation operator τ (Section III-A) in addition to also using λ and β. Thus, CJR is a more general solution than ECA and yet uses fewer compensation operators. Second, due to their different sets of compensation operators, CJR requires new rewriting rules (Sections III-A to III-D).

Similar to both CBA and ECA, our approach also requires that all join predicates are null-intolerant to have complete reorderability. If some join predicates are null-tolerant, our approach still strictly supersedes all existing approaches in terms of reorderability.

A. Compensation Operators

CJR uses three compensation operators, β, λ and τ. The first two operators (β and λ) are from CBA which have been introduced in Section II-B.

The third operator, τ, is a new unary operator that is a generalization of λ with two sets of nullification attributes (A and B):

$$\tau_{p,A,B}(R) = \lambda_{p,A}(R) \cup \lambda_{p,B}(R).$$

Semantically, for each tuple $t \in R$, if the nullification predicate p evaluates to true on t, t will remain unchanged in the output of $\tau_{p,A,B}(R)$; otherwise, two tuples, t_A and t_B, will be added to the output of $\tau_{p,A,B}$ where each t_S, $S \in \{A, B\}$, is derived from t by nullifying S. Note that $R_1 \overset{p_{12}}{\rtimes} R_2 = \beta(\tau_{p_{12},R_1,R_2}(R_1 \overset{\circ}{\times} R_2))$. We will illustrate in Section III-C how τ is used for reordering joins for full outerjoin queries in Example 4.

Although CJR uses the same λ operator as both CBA and ECA, a subtle difference is that CBA's and ECA's usage of λ is limited to the nullification predicates being null-intolerant, whereas CJR also uses λ with null-tolerant nullification predicates. Consequently, our approach can define the two new compensation operators in ECA more succinctly as $\gamma_A(R) = \beta(\lambda_{A \text{ is null},R}(R))$ and $\gamma^*_{A,B}(R) = \beta(\ \lambda_{A \text{ is null},R\setminus B}(R))$; thus, CJR can support all the rewriting rules in ECA (and hence also CBA).

B. Join Reordering Rules

In this section, we will look at the rewriting rules enabled by CJR for join reordering.

Example 1. *Consider the query $Q = R_1 \overset{p_{12}}{\rtimes} (R_2 \overset{p_{23}}{\bowtie} R_3)$ where we want to reorder the join operations to first join R_1 and R_2. Since $\rtimes$ and $\bowtie$ are not associative, $Q \neq Q_1$, where $Q_1 = (R_1 \overset{p_{12}}{\rtimes} R_2) \overset{p_{23}}{\bowtie} R_3$. Their non-equivalence is clear as Q preserves all the R_1 tuples, while in Q_1, some R_1 tuple could be eliminated after $\overset{p_{23}}{\bowtie}$.*

To preserve all the R_1 tuples, we could replace $\overset{p_{23}}{\bowtie}$ in Q_1 with $\overset{p_{23}}{\rtimes}$ to obtain $Q_2 = (R_1 \overset{p_{12}}{\rtimes} R_2) \overset{p_{23}}{\rtimes} R_3$. However, Q_2 is still not equivalent to Q, because in Q, some R_2 tuple could be eliminated by $\overset{p_{23}}{\bowtie}$, whereas Q_2 preserves all the R_2 tuples.

By using λ and β to compensate Q_2, it can be shown that $Q = \beta(\lambda_{R_3 \text{ not null},R_2}(Q_2))$. For each tuple t in the output of Q_2, the operator $\lambda_{R_3 \text{ not null},R_2}$ nullifies $t.R_2$ if "$t.R_3$ not

[4] A nullification set for a relation R is the set of all join predicates (including implied ones) that can nullify R (i.e., set all the attribute values of a tuple in R to null).

Rule 1	assoc($\overleftarrow{\bowtie}$, $\bowtie$)	$(R_1 \overset{p^{12}}{\overleftarrow{\bowtie}} R_2) \overset{p^{23}}{\bowtie} R_3 = (R_2 \overset{p^{23}}{\bowtie} R_3) \overset{p^{12}}{\bowtie} R_1$
Rule 2	assoc($\overleftarrow{\bowtie}$, $\bowtie$)	$R_1 \overset{p^{12}}{\overleftarrow{\bowtie}} (R_2 \overset{p^{23}}{\bowtie} R_3) = \beta(\lambda_{R_3\ not\ null, R_2}((R_1 \overset{p^{12}}{\overleftarrow{\bowtie}} R_2) \overset{p^{23}}{\bowtie} R_3))$
Rule 3	assoc($\overleftarrow{\bowtie}$, $\triangleright$)	$(R_1 \overset{p^{12}}{\overleftarrow{\bowtie}} R_2) \overset{p^{23}}{\triangleright} R_3 = \beta(\pi_{R_1 \cup R_2}(\lambda_{R_2\ is\ null\ \vee\ R_3\ is\ null, R_1 \cup R_2 \cup R_3}(R_1 \overset{p^{12}}{\overleftarrow{\bowtie}} (R_2 \overset{p^{23}}{\bowtie} R_3))))$
Rule 4	assoc($\overleftarrow{\bowtie}$, $\triangleright$)	$R_1 \overset{p^{12}}{\overleftarrow{\bowtie}} (R_2 \overset{p^{23}}{\triangleright} R_3) = \beta(\pi_{R_1 \cup R_2}(\lambda_{R_2\ is\ null\ \vee\ R_3\ is\ null, R_2 \cup R_3}((R_1 \overset{p^{12}}{\overleftarrow{\bowtie}} R_2) \overset{p^{23}}{\bowtie} R_3)))$
Rule 5	assoc($\triangleright$, $\overleftarrow{\bowtie}$)	$R_1 \overset{p^{12}}{\triangleright} (R_2 \overset{p^{23}}{\overleftarrow{\bowtie}} R_3) = R_1 \overset{p^{12}}{\triangleright} R_2$
Rule 6	assoc($\overrightarrow{\bowtie}$, $\overleftarrow{\bowtie}$)	$(R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} R_2) \overset{p^{23}}{\overleftarrow{\bowtie}} R_3 = \beta(\lambda_{R_1\ not\ null, R_2}(R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} (R_2 \overset{p^{23}}{\overleftarrow{\bowtie}} R_3)))$
Rule 7	assoc($\overrightarrow{\bowtie}$, $\overleftarrow{\bowtie}$)	$R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} (R_2 \overset{p^{23}}{\overleftarrow{\bowtie}} R_3) = (R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} R_2) \overset{p^{23}}{\overrightarrow{\bowtie}} R_3$
Rule 8	assoc($\triangleright$, $\bowtie$)	$R_1 \overset{p^{12}}{\triangleright} (R_2 \overset{p^{23}}{\bowtie} R_3) = \beta(\pi_{R_1}(\lambda_{R_2\ is\ null, R_1 \cup R_2 \cup R_3}(\beta(\lambda_{p_{23}, R_2}((R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\bowtie} R_3)))))$
Rule 9	assoc($\triangleright$, $\triangleright$)	$R_1 \overset{p^{12}}{\triangleright} (R_2 \overset{p^{23}}{\triangleright} R_3) = \beta(\pi_{R_1}(\lambda_{R_2\ is\ null, R_1 \cup R_2 \cup R_3}(\beta(\lambda_{R_3\ is\ null, R_2 \cup R_3}((R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\bowtie} R_3)))))$
Rule 10	assoc($\triangleright$, $\overrightarrow{\bowtie}$)	$R_1 \overset{p^{12}}{\triangleright} (R_2 \overset{p^{23}}{\overrightarrow{\bowtie}} R_3) = \beta(\pi_{R_1}(\lambda_{R_2\ is\ null, R_1 \cup R_2 \cup R_3}((R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\overrightarrow{\bowtie}} R_3)))$
Rule 11	assoc($\overrightarrow{\bowtie}$, $\triangleright$)	$(R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} R_2) \overset{p^{23}}{\triangleright} R_3 = \beta(\pi_{R_1 \cup R_2}(\lambda_{R_3\ is\ null, R_1 \cup R_2 \cup R_3}(R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} (R_2 \overset{p^{23}}{\bowtie} R_3))))$
Rule 12	assoc($\overrightarrow{\bowtie}$, $\triangleright$)	$R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} (R_2 \overset{p^{23}}{\triangleright} R_3) = \beta(\pi_{R_1 \cup R_2}(\lambda_{R_3\ is\ null, R_2 \cup R_3}((R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\bowtie} R_3)))$
Rule 13	r-asscom($\triangleright$, $\bowtie$)	$R_2 \overset{p^{23}}{\triangleright} (R_1 \overset{p^{13}}{\bowtie} R_3) = \beta(\pi_{R_2}(\lambda_{R_3\ is\ null, R_1 \cup R_2 \cup R_3}(\beta(\lambda_{p_{13}, R_3}((R_2 \overset{p^{23}}{\bowtie} R_3) \overset{p^{13}}{\bowtie} R_1)))))$
Rule 14	r-asscom($\triangleright$, $\overrightarrow{\bowtie}$)	$R_1 \overset{p^{13}}{\triangleright} (R_2 \overset{p^{23}}{\overrightarrow{\bowtie}} R_3) = \beta(\pi_{R_1}(\lambda_{R_3\ is\ null, R_1 \cup R_2 \cup R_3}(\beta(\lambda_{p_{23}, R_3}((R_1 \overset{p^{13}}{\bowtie} R_3) \overset{p^{23}}{\bowtie} R_2)))))$
Rule 15	assoc($\overrightarrow{\bowtie}$, $\bowtie$)	$(R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} R_2) \overset{p^{23}}{\bowtie} R_3 = R_1 \overset{p^{12}}{\bowtie} (R_2 \overset{p^{23}}{\bowtie} R_3)$
Rule 16	assoc($\overrightarrow{\bowtie}$, $\bowtie$)	$R_1 \overset{p^{12}}{\overrightarrow{\bowtie}} (R_2 \overset{p^{23}}{\bowtie} R_3) = \beta(\lambda_{p_{23}, R_2}((R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\bowtie} R_3))$
Rule 17	r-asscom($\bowtie$, $\overrightarrow{\bowtie}$)	$R_1 \overset{p^{13}}{\bowtie} (R_2 \overset{p^{23}}{\overrightarrow{\bowtie}} R_3) = R_2 \overset{p^{23}}{\bowtie} (R_1 \overset{p^{13}}{\bowtie} R_3)$
Rule 18	r-asscom($\bowtie$, $\overrightarrow{\bowtie}$)	$R_2 \overset{p^{23}}{\overrightarrow{\bowtie}} (R_1 \overset{p^{13}}{\bowtie} R_3) = \beta(\lambda_{p_{13}, R_3}((R_2 \overset{p^{23}}{\bowtie} R_3) \overset{p^{13}}{\bowtie} R_1))$
Rule 19	r-asscom($\overrightarrow{\bowtie}$, $\overrightarrow{\bowtie}$)	$R_1 \overset{p^{13}}{\overrightarrow{\bowtie}} (R_2 \overset{p^{23}}{\overrightarrow{\bowtie}} R_3) = \beta(\lambda_{p_{23}, R_3}((R_1 \overset{p^{13}}{\bowtie} R_3) \overset{p^{23}}{\overrightarrow{\bowtie}} R_2))$

TABLE I: Join reorderings enabled by CJR. Rules 8-14 and Rules 15-19 are translated from ECA [14] and CBA [11], respectively, using CJR's compensation operators. Rules 1-7 are for reordering full outerjoins. Rules 2, 3, 4 & 6 are the new rules with compensation operators. Rules 1, 5 & 7 [3]–[5] compensate by changing the join type and do not require compensation operators.

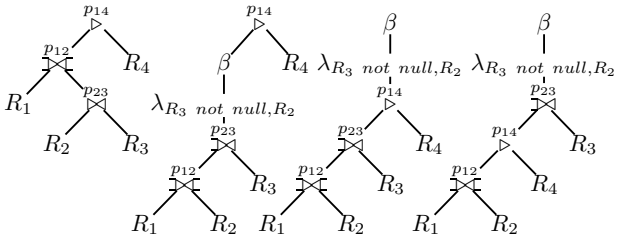
null" evaluates to not true (i.e., if $t.R_3$ is null) [5]*. This is because if $t.R_3$ is null, it means that the $t.R_2$ value cannot join will any tuple from R_3. Therefore, $t.R_2$ should be nullified as in Q. The β operator eliminates spurious tuples from the nullification of Q_2.* □

Table I shows the rewriting rules that are enabled by CJR; note that as semijoins are rewritten in terms of inner joins (i.e., $R_1 \ltimes R_2 = \pi_{R_1}(R_1 \bowtie R_2)$), Table I does not have explicit rewriting rules for semijoins. Each of the rewriting rules enables a join reordering that is considered to be an invalid transformation (as defined in Section II-C without using any compensation operator and without any join operator transformation). Rules 1-7 are the rules for reordering full outerjoin queries, rules 8-14 are translated from [14], and rules 15-19 are from [11]. Thus, CJR is able to support the reordering of any pair of join operators, which is strictly more expressive than the join orderings supported by ECA. We include the proofs for some of the rules in our technical report [15].

C. Pulling up Compensation Operators

To achieve complete join reorderability, besides the join reordering rules in Section III-B, we also need additional rewriting rules for pulling compensation operators above join operators. The following example illustrates the need for these additional rewriting rules.

[5]Note that "A is null" evaluates to $true$ if all attributes in A have null values, and "A not null" evaluates to $true$ if some attribute in A has a non-null value.



(a) Q_a (b) Q_b (c) Q_c (d) Q_d

Fig. 2: Query plans for Examples 2

Example 2. *Consider the query Q_a in Figure 2(a). By applying Rule 2 in Table I, we obtain Q_b in Figure 2(b). Now suppose that we want to reorder $\overset{p^{14}}{\triangleright}$ and $\overset{p^{23}}{\overrightarrow{\bowtie}}$. However, since the compensation operators β and $\lambda_{R_3\ not\ null, R_2}$ are sandwiched in between $\overset{p^{14}}{\triangleright}$ and $\overset{p^{23}}{\overrightarrow{\bowtie}}$, we could not freely reorder the joins. Therefore, we need to first pull β and $\lambda_{R_3\ not\ null, R_2}$ above $\overset{p^{14}}{\triangleright}$ to make $\overset{p^{14}}{\triangleright}$ and $\overset{p^{23}}{\overrightarrow{\bowtie}}$ adjacent, before we could reorder the two joins. In this case, the compensation operators can be trivially pulled up using Rule 9 in Table II. After we pull up the compensation operators, we obtain Q_c, where $\overset{p^{14}}{\triangleright}$ and $\overset{p^{23}}{\overrightarrow{\bowtie}}$ are adjacent. As l-asscom($\overrightarrow{\bowtie}$,$\triangleright$) is a valid transformation [7],*

Rule 1	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\bowtie} R_2 = \beta(\lambda_{p,R_3}(R_1 \overset{p_{12}}{\bowtie} R_2))$, where p_{12} does not reference R_3
Rule 2	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\bowtie} R_2 = \beta(\lambda_{p,R_1 \cup R_2}(R_1 \overset{p_{12}}{\bowtie} R_2))$, where p_{12} references R_3
Rule 3	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\rtimes} R_2 = \beta(\lambda_{p,R_3}(R_1 \overset{p_{12}}{\rtimes} R_2))$, where p_{12} does not reference R_3
Rule 4	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\rtimes} R_2 = \beta(\lambda_{p,R_3 \cup R_2}(R_1 \overset{p_{12}}{\rtimes} R_2))$, where p_{12} references R_3
Rule 5	$R_2 \overset{p_{12}}{\ltimes} \beta(\lambda_{p,R_3}(R_1)) = \beta(\lambda_{p,R_3}(R_2 \overset{p_{12}}{\ltimes} R_1))$, where p_{12} does not reference R_3
Rule 6	$R_2 \overset{p_{12}}{\ltimes} \beta(\lambda_{p,R_3}(R_1)) = \beta(\lambda_{p,R_1}(R_2 \overset{p_{12}}{\ltimes} R_1))$, where p_{12} references R_3
Rule 7	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\bowtie\!\!\!\!-} R_2 = \beta(\lambda_{p,R_3}(R_1 \overset{p_{12}}{\bowtie\!\!\!\!-} R_2))$, where p_{12} does not reference R_3
Rule 8	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\bowtie\!\!\!\!-} R_2 = \beta(\tau_{p,R_1,R_3 \cup R_2}(R_1 \overset{p_{12}}{\bowtie\!\!\!\!-} R_2))$, where p_{12} references R_3
Rule 9	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\triangleright} R_2 = \beta(\lambda_{p,R_3}(R_1 \overset{p_{12}}{\triangleright} R_2))$, where p_{12} does not reference R_3
Rule 10	$\beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\triangleright} R_2 = \beta(\pi_{R_1}(\lambda_{R_2 \ is \ null, R_1 \cup R_2}(\beta(\lambda_{p,R_3 \cup R_2}(R_1 \overset{p_{12}}{\bowtie} R_2)))))$, where p_{12} references R_3
Rule 11	$R_2 \overset{p_{12}}{\triangleright} \beta(\lambda_{p,R_3}(R_1)) = R_2 \overset{p_{12}}{\triangleright} R_1$, where p_{12} does not reference R_3
Rule 12	$R_2 \overset{p_{12}}{\triangleright} \beta(\lambda_{p,R_3}(R_1)) = \beta(\pi_{R_2}(\lambda_{R_1 \ is \ null, R_2 \cup R_1}(\beta(\lambda_{p,R_1}(R_2 \overset{p_{12}}{\bowtie} R_1)))))$, where p_{12} references R_3
Rule 13	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\bowtie} R_2 = \beta(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\bowtie} R_2))$, where p_{12} does not reference $R_3 \cup R_4$
Rule 14	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\bowtie} R_2 = \beta(\lambda_{p_{12},R_1 \cup R_2}(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\bowtie} R_2)))$, where p_{12} references $R_3 \cup R_4$
Rule 15	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\rtimes} R_2 = \beta(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\rtimes} R_2))$, where p_{12} does not reference $R_3 \cup R_4$
Rule 16	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\rtimes} R_2 = \beta(\lambda_{p_{12},R_2}(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\rtimes} R_2)))$, where p_{12} references $R_3 \cup R_4$
Rule 17	$R_2 \overset{p_{12}}{\ltimes} \beta(\tau_{p,R_3,R_4}(R_1)) = \beta(\tau_{p,R_3,R_4}(R_2 \overset{p_{12}}{\ltimes} R_1))$, where p_{12} does not reference $R_3 \cup R_4$
Rule 18	$R_2 \overset{p_{12}}{\ltimes} \beta(\tau_{p,R_3,R_4}(R_1)) = \beta(\lambda_{p_{12},R_1}(\tau_{p,R_3,R_4}(R_2 \overset{p_{12}}{\ltimes} R_1)))$, where p_{12} references $R_3 \cup R_4$
Rule 19	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\bowtie\!\!\!\!-} R_2 = \beta(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\bowtie\!\!\!\!-} R_2))$, where p_{12} does not reference $R_3 \cup R_4$
Rule 20	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\bowtie\!\!\!\!-} R_2 = \beta(\tau_{p_{12},R_1,R_2}(\beta(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\bowtie\!\!\!\!-} R_2))))$, where p_{12} references $R_3 \cup R_4$
Rule 21	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\triangleright} R_2 = \beta(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\triangleright} R_2))$, where p_{12} does not reference $R_3 \cup R_4$
Rule 22	$\beta(\tau_{p,R_3,R_4}(R_1)) \overset{p_{12}}{\triangleright} R_2 = \beta(\pi_{R_1}(\lambda_{R_2 \ is \ null, R_1 \cup R_2}(\beta(\lambda_{p_{12},R_2}(\tau_{p,R_3,R_4}(R_1 \overset{p_{12}}{\bowtie} R_2))))))$, where p_{12} references $R_3 \cup R_4$
Rule 23	$R_2 \overset{p_{12}}{\triangleright} \beta(\tau_{p,R_3,R_4}(R_1)) = R_2 \overset{p_{12}}{\triangleright} R_1$, where p_{12} does not reference $R_3 \cup R_4$
Rule 24	$R_2 \overset{p_{12}}{\triangleright} \beta(\tau_{p,R_3,R_4}(R_1)) = \beta(\pi_{R_2}(\lambda_{R_1 \ is \ null, R_2 \cup R_1}(\beta(\lambda_{p_{12},R_1}(\tau_{p,R_3,R_4}(R_2 \overset{p_{12}}{\bowtie} R_1))))))$, where p_{12} references $R_3 \cup R_4$

TABLE II: Pulling-up rules for λ and τ. Note that $R_3, R_4 \subseteq R_1$. The join predicates are null-intolerant, and p can be null-tolerant or null-intolerant. Rules 1-12 are for pulling up λ, and Rules 13-24 are for pulling up τ. Rules 7-8 & 13-24 are new rules. Rules 1-6 & 9-12 are from ECA [14]; these rules still hold when p is null-tolerant.

$\overset{p_{23}}{\rtimes}$ *and* $\overset{p_{14}}{\triangleright}$ *can be reordered to obtain* Q_d. □

In Example 2, the compensation operators can be simply pulled up without changing their forms. However, in Examples 3 and 4, we will look at two other cases where the compensation operator needs to be changed after being pulled up. Specifically, Example 3 changes the compensation operator due to an effect known as the *rippling effect*, and Example 4 illustrates the need for our new compensation operator τ.

Example 3. *Consider the query* $Q = \beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\bowtie} R_2$, *where* $R_3 \subseteq R_1$. *Suppose that we want to pull* β *and* $\lambda_{p,R_3}(R_1)$ *above* $\overset{p_{12}}{\bowtie}$. *If* p_{12} *does not reference any attribute in* R_3, *then we can simply pull up* β *and* $\lambda_{p,R_3}(R_1)$ *to obtain* $Q = \beta(\lambda_{p,R_3}(R_1 \overset{p_{12}}{\bowtie} R_2))$. *This is because whether a tuple is nullified by* $\lambda_{p,R_3}(R_1)$ *or not, it will not affect the tuple's joinability with* R_2. *Therefore, performing the nullification before or after the join does not make any difference.*

However, if p_{12} *references some attribute in* R_3, *then the nullified tuples in* R_1 *will not join with any tuple in* R_2 *since join predicates are assumed to be null-intolerant. Let* $t \in R_1$ *be a tuple that is nullified in* $\lambda_{p,R_3}(R_1)$. *If we pull* λ_{p,R_3} *above* $\overset{p_{12}}{\bowtie}$ *to obtain* $Q' = \beta(\lambda_{p,R_3}(R_1 \overset{p_{12}}{\bowtie} R_2))$, *the problem is that* p_{12} *will not see the new null value in* t *introduced by* λ_{p,R_3} *since* λ_{p,R_3} *is now performed after the join. As a result,* t *may appear in the inner join result of* Q' *while it is not in the join result of* Q.

To fix this, we replace the nullification attribute R_3 *in* Q' *with* $R_1 \cup R_2$ *to obtain* $Q'' = \beta(\lambda_{p,R_1 \cup R_2}(R_1 \overset{p_{12}}{\bowtie} R_2))$. *Now we have* $Q'' = Q$. *The difference between* Q'' *and* Q' *is that if* p *evaluates to not true for* $t \in R_1$, *instead of nullifying only* R_3, Q'' *would nullify all the attributes in* R_1 *and* R_2. *This is because if* R_3 *is nullified in* t, *then* t *will not appear in the join result of* Q; *if it appears in the join result of* Q'', *we should remove the result tuple by nullifying all the attributes. This is referred to as the rippling effect in [11].*

□

The next example shows a case where a λ operator needs to be changed to τ when being pulled up.

Example 4. *Consider* $Q = \beta(\lambda_{p,R_3}(R_1)) \overset{p_{12}}{\bowtie\!\!\!\!-} R_2$, *where* $R_3 \subseteq R_1$ *and* p_{12} *references* R_3. *Suppose that we want to pull* β *and* λ_{p,R_3} *above* $\overset{p_{12}}{\bowtie\!\!\!\!-}$. *If we simply pull up both operators, we would obtain* $Q' = \beta(\lambda_{p,R_3}(R_1 \overset{p_{12}}{\bowtie\!\!\!\!-} R_2))$, *To see that* $Q \neq Q'$, *consider* $t_1 \in R_1$ *to be a tuple that is nullified by* λ_{p,R_3}. *In* Q, t_1 *will not be able to join with any tuple in* R_2 *after the nullification, but in* Q', t_1 *may join with some tuple* $t_2 \in$

R_2 and produce a tuple (t_1, t_2) in the output of Q' with R_3 nullified in t_1.

To fix this, we replace λ_{p,R_3} in Q' with $\tau_{p,R_1,R_3 \cup R_2}$ to obtain $Q'' = \beta(\tau_{p,R_1,R_3 \cup R_2}(R_1 \overset{p_{12}}{\rightleftarrows} R_2))$. Now $Q = Q''$. Recall that the problem with Q' is that, if p evaluates to not true on a tuple $t_1 \in R_1$, we should not let t_1 to join with any tuple in R_2; but if we compute $R_1 \overset{p_{12}}{\rightleftarrows} R_2$ before λ_{p,R_3}, t_1 may join with some $t_2 \in R_2$ and produce (t_1, t_2) in the full outerjoin result. Now in Q'', $\tau_{p,R_1,R_3 \cup R_2}$ will duplicate (t_1, t_2) in the full outerjoin result, and nullify R_1 for one copy to get $(null, t_2)$, and nullify $R_3 \cup R_2$ for the other copy to get $(t_1, null)$, with R_3 nullified in t_1. This produces the same result as if t_1 does not join t_2, like in Q. Thus, we have $Q = Q''$. □

To achieve complete join reorderability, we need rules to pull up λ and τ above all types of join operators. The complete set of such rewriting rules are given in Table II.

D. Rules for Reordering Compensation and Unary Operators

Table III shows the rewriting rules for reordering the compensation operators and unary relational operators (selection/projection/top-k).

In Rule 7, $t_{k,L}(R)$ denotes the top-k operator corresponding to the SQL query `select * from R order by L limit k`, where k is an integer and L is the order-by list. Reordering rules between other unary operators (e.g., group by with aggregation) and compensation operators is non-trivial and requires further research.

Rule 1	$\sigma_p(\beta(R_1)) = \beta(\sigma_p(R_1))$, where p is null-intolerant
Rule 2	$\sigma_{p_1}(\lambda_{p_2,R_2}(R_1)) = \lambda_{p_2,R_2}(\sigma_{p_1}(R_1))$, where p_1 does not reference R_2
Rule 3	$\sigma_{p_1}(\tau_{p_2,R_2,R_3}(R_1)) = \tau_{p_2,R_2,R_3}(\sigma_{p_1}(R_1))$, where p_1 does not reference $R_2 \cup R_3$
Rule 4	$\beta(\pi_{R_2}(\beta(R_1))) = \beta(\pi_{R_2}(R_1))$
Rule 5	$\beta(\lambda_{p,R_2}(\pi_{R_3}(R_1))) = \beta(\pi_{R_3}(\lambda_{p,R_2}(R_1)))$
Rule 6	$\beta(\tau_{p,R_2,R_3}(\pi_{R_4}(R_1))) = \beta(\pi_{R_4}(\tau_{p,R_2,R_3}(R_1)))$
Rule 7	$t_{k,L}(\lambda_{p,R_2}(R_1)) = \lambda_{p,R_2}(t_{k,L}(R_1))$, where L does not involve attributes in R_2

TABLE III: Rules for reordering compensation and unary relational operators

E. Query Plan Enumeration

As observed by ECA [14], the query plan enumeration problem becomes more intricate in the presence of compensation operators due to two complexities: keeping tracking of compensation operators generated and reasoning about the equivalence of query subplans with compensation operators. Since ECA is driven by query rewriting rules, ECA adopted a cost-based, top-down enumeration approach. This top-down approach can be adapted for CJR as well by using our compensation operators and rewriting rules.

F. Handling Duplicates and All-null Tuples

Same as CBA [11], we assume that each base table has a key column, and the key IDs will be carried along to the root of the join tree, so there will be no duplicates in the base tables since tuples have unique key IDs. In practice, if the base table does not have a key column, we can add a virtual key column to the table during query optimization (i.e., to use the tuple ID as key ID as noted by CBA). Then duplicate tuples in the base tables will not be eliminated by the compensation operators because they have unique virtual keys during optimization. Furthermore, if a base table has tuples whose attribute values are all nulls, such tuples can be prevented from being eliminated by marking the null values in these tuples as special null values which are treated as non-null values by the the compensation operators.

IV. Implementation of Compensations

This section discusses the implementation of the compensation operators (i.e., β, λ, and τ). Similar to CBA and ECA, we shall focus on a SQL-level implementation, where the query plan (with compensation operators) chosen by the query optimizer is translated back into an SQL query that implements that plan. This approach is easier to implement and is less intrusive than a native approach to implement the operators.

A. Implementation of λ and τ

The λ operator is implemented following CBA's approach, which uses SQL's case expression to nullify the nullification attributes. Specifically, for each $A_i \in A$, we use a case expression "CASE WHEN p THEN A_i END AS B_i", where B_i will be A_i if p evaluates to $true$; and $null$, otherwise.

For the new compensation operator, $\tau_{p,A,B}(R)$, its implementation is straightforward using SQL's WITH construct to first define a temporary table for the tuples in R that do not evaluate to $true$ for p, and the main query unions the results of the temporary table with two other subqueries that each nullifies either attributes A or B (via a case expression) for the tuples in R that evaluates to $true$ for p.

For cost modeling, the cost of both λ and τ is the cost for doing one sequential scan of the input operand to evaluate the predicate p.

B. Implementation of β

In this section, we discuss the implementation of the most complex operator β which is used to eliminate the spurious tuples from its relation operand R.

Recall that $\beta(R)$ is used to remove the duplicated or dominated tuples in R. The existing implementation approach for β is based on sorting [11], where the idea is to find a *favourable ordering* such that when R is sorted using this ordering, each spurious tuple in the sorted R will be immediately preceded by a tuple that duplicates or dominates it. With this property, the spurious tuples in R can be eliminated by using a window function in a SQL query. In general, it may not be always possible to eliminate all spurious tuples in R using a single favourable ordering.

However, the algorithm for computing the favourable ordering in [11] is only applicable for queries without any full outerjoin. This is because the approach is based on a concept called *nullification sets*, which are defined for queries involving only inner joins and left outerjoins. Using a similar sorting-based approach for full outerjoin queries requires a redefinition of nullification sets and the design of a sound algorithm for computing the favourable ordering based on this new definition.

We first give an overview of the algorithm in [11] to compute the favourable ordering. Let $NS(Q)$ denote the nullification sets computed for a query Q, and R_i denote some base relation.

In the left outerjoin result of $Q_1 = R_1 \overset{p_{12}}{⟕} R_2$, we have $NS(Q_1) = \{R_1 : \emptyset, R_2 : \{p_{12}\}\}$, which means that the predicate p_{12} nullifies R_2 and R_1 is not nullified by p_{12}. Similarly, in the inner join result of $Q_2 = R_1 \overset{p_{12}}{\bowtie} R_2$, we have $NS(Q_1) = \{R_1 : \{p_{12}\}, R_2 : \{p_{12}\}\}$. Essentially, $NS(Q)$ gives a mapping between the base relations and their respective nullifying predicates in the query Q. For $Q_3 = \beta(\lambda_{p,R_1}(R))$, we have $NS(Q_3) = \{R_1 : \{p\}\}$. We use $NS_{R_i}(Q)$ to denote the set of nullifying predicates for R_i in Q.

One important property of nullification sets is that if $NS_{R_i}(Q) \subseteq NS_{R_j}(Q)$, then for any tuple t in the query result of Q, $t.R_i$ is null implies that $t.R_j$ must also be null, where $t.R$ denotes the value of R in t. It can be shown that if the relations that are nullified by compensation operators in Q are linearly ordered based on the inclusion property of their nullification sets, then sorting the result of Q by this ordering can guarantee that a dominated tuple will always be preceded by its dominating tuple [11]. Another important observation is that, given $\beta(\lambda(J))$ where J is a join result, if a tuple $t \in J$ is not nullified by λ, then t will not be dominated by any tuple in $\lambda(J)$ [11]. With these properties and observations, [11] proposes an algorithm to construct a directed acyclic graph (DAG) based on the inclusion property of the nullification sets in $NS(Q)$, which is used to derive a favourable ordering for Q. The algorithm for constructing the DAG and computing the favourable ordering is given in [11].

Next, we explain that to adapt this algorithm for full outerjoin queries, it is necessary to define nullification sets on a tuple basis instead of on a query basis. Given a query Q, the computation of nullification sets in [11] assumes that the nullification sets are the same for each result tuple $t \in Q$. However, this is no longer true if Q involves full outerjoins. For $Q_3 = R_1 \overset{p_{12}}{⟗} R_2$, for some result tuple $t \in Q_3$, p_{12} nullifies R_1, but for some other result tuple $t' \in Q_3$, p_{12} nullifies R_2, based on the definition of full outerjoin. Thus, we need to find a favourable ordering that is sound for both types of tuples.

To achieve this, we extend the definition of nullification sets on queries to tuples. More formally, given a query Q which may involve full outerjoins, let $NS(t)$ represent the nullification sets for a result tuple $t \in Q$, which is defined as the mapping between the base relations and their nullifying predicates for the tuple t. Then we use $\mathcal{NS}(Q)$ to denote the set $\{NS(t)|t \in Q\}$. Thus, given a query Q, if Q involves m full outerjoins, $\mathcal{NS}(Q)$ may contain up to 2^m different versions of nullification sets. In other words, the result tuples of Q are conceptually partitioned into 2^m partitions, and tuples from different partitions could have different nullification sets.

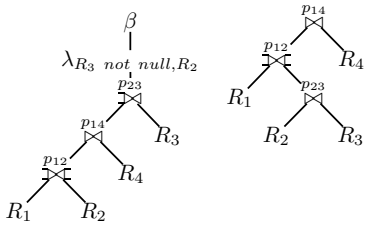
Algorithm 1: Algorithm for computing $\mathcal{NS}(Q)$

Input: A query Q
Output: The nullification sets $\mathcal{NS}(Q)$ for Q

Let T be the query tree for Q
return ComputeNS(T)
Function `ComputeNS(`T`)`**:**
  Let n be the root node of T
  if n *is a base relation* R **then**
    Initialize a new $NS_1 = \{R : \emptyset\}$
    Initialize the list $list\_NS = [NS_1]$
    return $list\_NS$
  else if n *is a join node* **then**
    $left\_list\_NS$ = ComputeNS($n.left$)
    $right\_list\_NS$ = ComputeNS($n.right$)
    Let $pred$ be the join predicate of n
    if n *is a full outerjoin node* **then**
      Let $rels\_ref$ be the set of relations referenced by $pred$
      Initialize an empty list $list\_NS$
      foreach NS_l *in* $left\_list\_NS$, NS_r *in* $right\_list\_NS$ **do**
        Create a copy NS'_l of NS_l, and a copy NS'_r of NS_r
        foreach *relation* r *in* $n.left$ **do**
          $NS'_l[r]$ = $NS'_l[r] \cup \{pred\}$ $\cup\{p \in NS'_l[t] \cup NS'_r[t] | t \in rels\_ref\}$
        Merge NS'_l and NS'_r to get NS'
        Create a copy NS''_l of NS_l, and a copy NS''_r of NS_r
        foreach *relation* r *in* $n.right$ **do**
          $NS''_r[r]$ = $NS''_r[r] \cup \{pred\} \cup \{p \in NS''_l[t] \cup NS''_r[t] | t \in rels\_ref\}$
        Merge NS''_l and NS''_r to get NS''
        $list\_NS$ += $[NS', NS'']$
      return $list\_NS$
    else if n *is a inner join/left outerjoin/antijoin node* **then**
      /* omitted */
  else if n *is a compensation node* **then**
    /* omitted */
End Function

The algorithm for computing $\mathcal{NS}(Q)$ is shown in Algorithm 1. We show only the computation for full outerjoin nodes in an input query plan tree and omit the details for other nodes due to space constraints. The computation for left outerjoin nodes and inner join nodes remains the same as in [11]. The computation for antijoin nodes is trivial: since all the relations in the right operand are removed and no null values are introduced for the relations in the left operand, we simply return `ComputeNS(n.left)` for an antijoin node n.

To explain the computation of nullification sets for a full outerjoin node, we use the query example shown in Figure 3. We use the compensated query Q_4 as the example input to Algorithm 1. Note that Q_4 could be obtained by transforming



(a) Q_4 (b) Q_4'

Fig. 3: An example query used in Section IV-B Implementation of β.

a conventional query Q_4' (shown in Figure 3(b)), that is, by applying Rule 2 in Table I to swap $\overset{p_{12}}{\supset\!\!\bowtie}$ and $\overset{p_{12}}{\bowtie}$, and then applying l-asscom($\supset\!\!\bowtie$, $\bowtie$) to swap $\overset{p_{14}}{\bowtie}$ down.

Algorithm 1 will perform a post-order traversal on the query tree of Q_4 (denoted by T) by calling the function `ComputeNS(T)`. The function `ComputeNS(T)` returns the computed nullification sets for an input query tree T. With Q_4 in Figure 3(a), the post-order traversal will first compute the nullification sets for the subtree rooted at $\overset{p_{12}}{\supset\!\!\bowtie}$. Nullification sets are first initialized for the base relations R_1 and R_2 (lines 5-8 in Algorithm 1), and on lines 9-11, we get $left\_list\_NS = [\{R_1 : \emptyset\}]$ and $right\_list\_NS = [\{R_2 : \emptyset\}]$. From line 13 to 26, the tuples in the full outerjoin result are conceptually partitioned into two groups, and we compute NS' and NS'' respectively for the two groups. For the first group of tuples (lines 17-20), the left operand gets nullified, so we add the join predicate p_{12} into the nullification sets of each relation in $n.left$ (line 19), to get $NS_l' = \{R_1 : \{p_{12}\}\}$. Note that line 19 (and 23) applies the rippling effect of null-intolerant join predicates [11], which says if a null-intolerant predicate p_1 nullifies R_1, and p_1 references R_2, then all predicates that nullify R_2 also (indirectly) nullifies R_1. Then NS_l' is merged with $NS_r' = \{R_2 : \emptyset\}$ to get $NS' = \{R_1 : \{p_{12}\}, R_2 : \emptyset\}$. Lines 21-24 do the same for the second group of tuples where the right operand of the full outerjoin gets nullified, and we get $NS'' = \{\{R_1 : \emptyset\}, R_2 : \{p_{12}\}\}$. Both NS' and NS'' are returned as nullification sets on this subtree (lines 25-26).

Next, when the algorithm traverses the node $\overset{p_{14}}{\bowtie}$ in Q_4, we have $left\_list\_NS = [\{R_1 : \{p_{12}\}, R_2 : \emptyset\}, \{R_1 : \emptyset, R_2 : \{p_{12}\}\}]$ and $right\_list\_NS = [\{R_4 : \emptyset\}]$ (similar to lines 10-11 for full outerjoins). Then for each pair (NS_l, NS_r) where $NS_l \in left\_list\_NS$ and $NS_r \in right\_list\_NS$, the join predicate p_{14} is added to nullification sets of relations in both operands of the inner join, and no conceptual partitioning of result tuples is needed for an inner join. Note that here since p_{14} references R_1, predicates that nullify R_1 should also be added to the nullification sets of all the relations due to the ripping effect. The returned nullification sets will be $[\{R_1 : \{p_{12}, p_{14}\}, R_2 : \{p_{12}, p_{14}\}, R_4 : \{p_{12}, p_{14}\}\}, \{R_1 : \{p_{14}\}, R_2 : \{p_{12}, p_{14}\}, R_4 : \{p_{14}\}\}]$. We continue to run Algorithm 1 to traverse every node in Q_4 to get the eventual $\mathcal{NS}(Q_4)$.

In the following, we describe how to compute a favourable ordering for a query Q that is compatible to all versions of nullification sets in $\mathcal{NS}(Q)$, and thus is guaranteed to be a favourable ordering for all the result tuples of Q.

After we obtain $\mathcal{NS}(Q)$ from Algorithm 1, for each $NS_i \in \mathcal{NS}(Q)$, we construct a DAG named D_i as described in [11]. A node in D_i represents a set of base relations. Essentially, each D_i constitutes a partial order among the base relations. Note that in one D_i, for relations represented by the same node, the ordering between them does not matter. Our goal is to find a total order that is compatible to all the partial orders in all D_i.

To find the total order, we first convert each D_i to a set of directed edges among base relations , denoted as G_i. For example, if there is an edge from $\{R_1\}$ to $\{R_2, R_3\}$ in D_i, we will have two directed edges $R_1 \rightarrow R_2$ and $R_1 \rightarrow R_3$ in G_i. After the conversion, we union all the edges in all G_i to obtain a new graph G, in which each node corresponds to exactly one base relation. Finally, we run topological sort on G to find the favourable ordering.

Note that it is in general possible for G to be cyclic, which implies that there are conflicts among different G_i. In this case, a favourable ordering does not exist. [11] noted that a favourable ordering may not always exist even for queries without full outerjoins. For such queries, we use a more direct approach to implement β as described below.

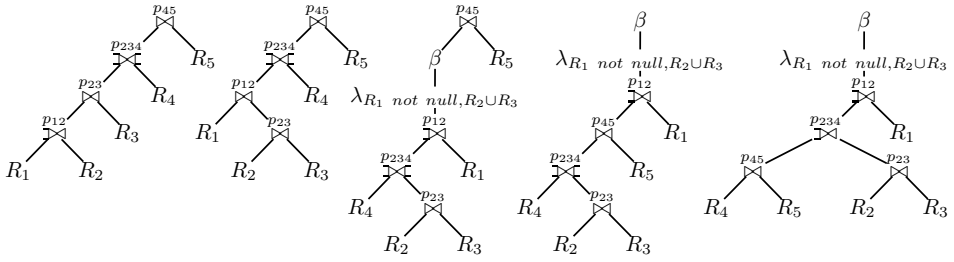
When a favourable ordering does not exist, the most direct way to implement $\beta(R)$ is to do a self-antijoin of R to find tuples that are not dominated or duplicated by any other tuples in R. However, as mentioned before, an important observation is, spurious tuples must have been further nullified by some compensation operators. Therefore, we need not consider the tuples that are not nullified by compensation operators. Thus, when we evaluate λ and τ, we add a column in the result to indicate whether the result tuple has been further nullified by some λ or τ operators. Then when we use a self-antijoin to compute the β result, we consider only the further nullified tuples to be eliminated by β.

Our technical report [15] includes an example β implementation with detailed SQL queries using the sorting based method and the self-antijoin based method.

V. Experiments

In this section, we present experimental results to demonstrate the benefits of the enlarged query plan search space for full outerjoin queries that is enabled by our proposed `CJR`.

As there are no database benchmarks that are focused on full outerjoin queries, we generated our test SQL queries with full outerjoins from two well-known benchmarks: Join Order Benchmark (Section V-A) and TPC-H (Section V-B). For each generated test query Q, we compared the performance of two query plans. The first query plan, referred to as a conventional query plan, is the best query plan for Q selected by PostgreSQL database server. The second query plan, referred to as compensated query plan, is the query plan for Q generated using our approach as follows. The SQL query is



(a) P_0 (b) P_1 (c) P_2 (d) P_3 (e) P_4

Fig. 4: Query plans for 2d_19 on JOB dataset. R_1 to R_5 denote the tables company_name, movie_companies, title, movie_keyword, and keyword, respectively. P_0 is the canonical plan directly translated from the SQL query. P_1 is the best conventional plan, and P_4 is the best compensated plan.

first translated into a canonical logical query plan from which all possible join reorderings for that query are enumerated by applying our query rewriting rules (Section III). The rewritten query plans (with at least one compensation operator) are translated into SQL queries (Section IV) and their estimated query plan costs are estimated by PostgreSQL database server. From among the rewritten query plans involving compensation operators, the one with the minimum estimated query plan cost is the compensated query plan. The performance of both the conventional and compensated query plans are compared by executing their corresponding SQL queries on PostgreSQL database server.

Our experiments were conducted on an Intel Xeon Processor E5-2603 v2 server running Ubuntu Linux 16.04 with 32GB main 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 13.2 database server with shared_buffers = 5GB and work_mem = 1GB. The experimental results on Join Order Benchmark and TPC-H show that `CJR` improves query performance by up to a factor of 12.32.

A. Experiment 1: JOB Benchmark

The Join Order Benchmark (JOB) [21] is designed for evaluating the performance of join order optimization. However, JOB focuses only on inner join queries without any outerjoin queries. We describe next our approach to generate outerjoin test queries from JOB's inner join queries.

Given an inner join query in JOB, we derive a mutated query from it by randomly mutating each inner join in the query into either left outerjoin, full outerjoin, antijoin, or inner join (each with a probability 0.25). Mutating an inner join to inner join means that that join operation is unchanged. Thus, the mutated and original queries are the same except for the mutated join operations. A mutated query is discarded if it is invalid (e.g., the mutated query $\pi_{S.a,R.b}(R \overset{p}{\triangleright} S)$ is invalid as the projected attribute S.a does not exist in the schema of the join result).

JOB is based on the IMDB movie dataset of size 3.7GB and consists of 113 inner join queries derived from 33 join graphs. We used all the 17 inner join queries from the first 5 join graphs in JOB to generate 20 mutated queries from each inner join query as described above. After discarding the invalid mutated queries from the 340 mutated queries, we have a total of 158 valid test queries.

Among the 158 test queries, there are 14 queries (8.8%) where the best query plan is a compensated query plan. The performance improvement factors (ratio of execution time for conventional query plan to execution time for compensated query plan) for these 14 queries are shown in Table IV[6]. The numbers reported are the average among three runs. Note that all the test queries in Table IV contain at least one full outerjoin. The detailed SQL of the queries can be found in our technical report [15]. Also note that for all queries in Table IV, the best `ECA` plan is the same as the best conventional plan. This is because the rewriting rules in `ECA` are designed to reorder antijoins, and only two queries, 5b_13 and 5c_8 in Table IV, involve antijoins; however, the compensated query plans for these two queries are estimated to be more costly than the corresponding conventional query plans. From the results in Table IV, we observe that for 7 of the 14 queries, the best compensated query plan improves over the best conventional query plan by more than 3x speedup.

Note that a compensated query plan does not always outperform a conventional query plan as the benefit of the former depends on whether a reordered join operation could produce a small intermediate join result that helps reduce the cost of the subsequent operations in the compensated query plan. Thus the benefit of a compensated query plan depends on the selectivity of the join/select operations.

To further investigate the effect of a subquery selectivity on the performance improvement, we take a closer look at Query 2d_19 in Table IV which has an improvement factor 1.52. As we will show, the improvement factor for this query can be increased by increasing its selectivity.

The SQL query of 2d_19 is shown in Figure 5, and Figure 4(a) shows the canonical query plan P_0 directly translated from this SQL query. The best conventional query plan P_1 for this

[6]Each test query is labeled as m_n where m denotes the query number of the JOB query that the test query is derived from and n denotes the n^{th} mutated query generated from that JOB query.

Query No.	Best conventional plan	Best compensated plan	Improvement factor
1b_3	25.41s	12.81s	1.98
1b_14	25.14s	12.76s	1.96
1d_2	24.69s	5.03s	4.90
1d_9	5.74s	1.50s	3.81
1d_19	5.57s	1.53s	3.62
2d_2	46.36s	33.63s	1.37
2d_14	44.47s	32.53s	1.36
2d_19	42.67s	28.05s	1.52
4c_14	13.31s	7.81s	1.70
5b_2	$>$ 60.00s	4.87s	$>$ 12.32
5b_5	$>$ 60.00s	5.21s	$>$ 11.51
5b_13	$>$ 60.00s	5.24s	$>$ 11.45
5b_16	$>$ 60.00s	5.10s	$>$ 11.76
5c_8	23.74s	23.68s	1.002

TABLE IV: Performance comparison for queries where the compensated query plan is the best plan. Query plans that executed for more than 1 minute were terminated before completion.

query is shown in Figure 4(b), and the best compensated query plan P_4 is shown in Figure 4(e). The query plan P_4 is obtained from P_0 by the following sequence of query rewritings. First, apply Rule 15 in Table I on P_0 to obtain P_1, and apply Rule 2 in Table I on P_1 to obtain P_2. Next, apply Rule 1 in Table II and l-asscom($\mathbb{1}\!\!\bowtie$, $\bowtie$) on P_2 to obtain P_3. Note that l-asscom($\mathbb{1}\!\!\bowtie$, $\bowtie$) is a valid transformation and does not require a compensation rule. Finally, apply Rule 1 in Table I on P_3 to obtain P_4. Note that without the compensation rule Rule 2 in Table I and the pulling-up rule Rule 1 in Table II, it is not possible to derive the query plan P_4 using a conventional query optimizer.

```
SELECT * FROM
  (((((SELECT * FROM company_name
        WHERE cn_country_code ='[us]') AS company_name
    LEFT JOIN movie_companies ON cn_id=mc_company_id)
      INNER JOIN title ON mc_movie_id=t_id)
        FULL JOIN movie_keyword ON t_id=mk_movie_id
            AND mc_movie_id=mk_movie_id)
          INNER JOIN (SELECT * FROM keyword WHERE
              k_keyword ='character-name-in-title')
                AS keyword
            ON mk_keyword_id=k_id
  ) AS joined;
```

Fig. 5: SQL query for 2d_19

For this query, P_4 has a lower cost than P_1 because the `keyword` table (R_5) has a selection predicate `k_keyword = 'character-name-in-title'`, and by joining `movie_keyword` (R_4) and `keyword` (R_5) early in P_4, the small intermediate result produced by this join operation helps to reduce the costs of the subsequent join operations. However, the keyword `'character-name-in-title'` used in this query turns out to be the top-3 most frequent keyword among all the 134,170 keywords in all the movies with a resultant selectivity factor of 0.9248% on the `movie_keyword` table.

Table V shows the effect of increasing the selectivity of this query's selection predicate by varying the keyword constant used in the query. The results show that as a less frequent keyword is used in the selection predicate, the reduction in the predicate selectivity factor from 0.9248% to 0.00002% results in the improvement factor increasing from 1.52x to 21.05x.

Top-n keyword	Selectivity factor on movie_keyword table	Improvement factor
3	0.9248%	1.52
13	0.2330%	2.77
23	0.1740%	2.80
33	0.1500%	2.93
100	0.0889%	3.03
300	0.0433%	4.46
1000	0.0154%	5.08
2000	0.0077%	5.70
10000	0.0012%	9.72
50000	0.00011%	18.47
134170	0.00002%	21.05

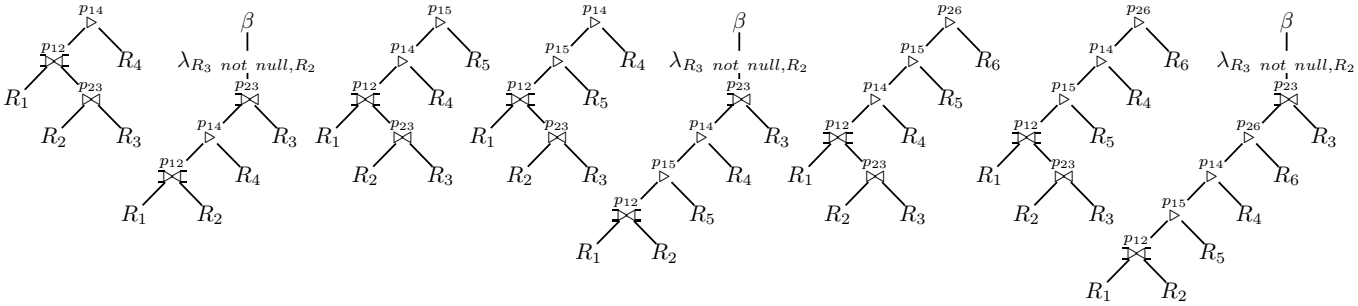
TABLE V: Effect of selectivity of Query 2d_19's selection predicate on performance improvement

1) Experiments with DuckDB: We have also repeated our experiments on JOB using DuckDB[7], which is an advanced, open-source high-performance analytical database system [22]. This is to show evidence that a modern database like DuckDB cannot produce more reorderings than PostgreSQL, because all the popular/modern database engines are based on traditional relational operators and are therefore limited to the valid join reorderings in `TBA` [7]. Specifically, we have run the queries shown in Table IV using DuckDB. For 9 of the 14 queries, the compensated plans are faster than the conventional plans; the execution time and improvement factors are given in Table VI. For the remaining 5 queries, the conventional plans run faster than the compensated plans on DuckDB. As our approach will choose the best conventional/compensated query plan for a query, the performance improvement factor is 1 for these 5 queries and their results are omitted from Table VI. For the 5 queries where the conventional plans are faster in DuckDB, we have verified that these DuckDB's conventional query plans use the same join orderings as the corresponding PostgreSQL's conventional query plans. The reason why the conventional plans for those 5 queries are faster on DuckDB compared to PostgreSQL is because DuckDB has chosen more efficient join algorithms for some of the join operations (e.g., hash join vs. sort-merge join), due to more accurate cost estimation for the join algorithms.

B. Experiment 2: TPC-H Benchmark

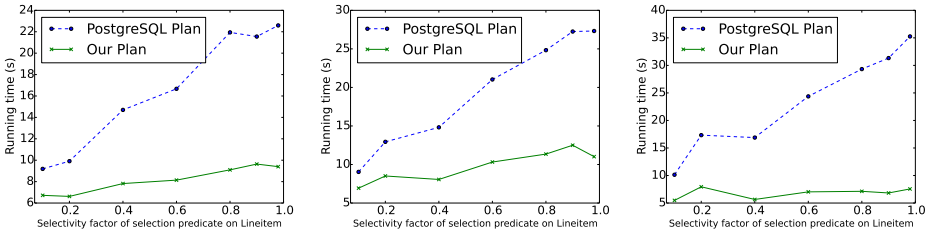
In this experiment, we run test queries on the TPC-H benchmark dataset with a scaling factor 1. Similar to JOB, the TPC-H benchmark is not focused on full outerjoin queries. Unlike JOB, the TPC-H benchmark does not have complex multi-table join queries that are amenable to the mutation

[7]https://duckdb.org/



(a) P_1^{pg} (b) P_1^{CJR} (c) P_2^{direct} (d) P_2^{pg} (e) P_2^{CJR} (f) P_3^{direct} (g) P_3^{pg} (h) P_3^{CJR}

Fig. 6: Query plans for Experiment 2 on TPC-H dataset



(a) Q_1 (b) Q_2 (c) Q_3

Fig. 7: Performance results for Experiment 2 with queries Q_1, Q_2 and Q_3

Query No.	Best conventional plan	Best compensated plan	Improvement factor
1d_2	1.12s	0.62s	1.80
1d_9	7.84s	0.52s	15.07
1d_19	2.67s	0.43s	6.20
2d_2	56.46s	8.17s	6.91
2d_19	70.94s	9.36s	7.57
5b_2	11.33s	0.38s	29.81
5b_5	0.99s	0.38s	2.60
5b_16	0.80s	0.38s	2.10
5c_8	1.08s	0.47s	2.29

TABLE VI: Performance results on DuckDB

approach that we had used to generate test queries from the queries in JOB. Therefore, for the test queries in this experiment, we designed the following three outerjoin queries (with increasing complexity).

$$Q_1: \quad (R_1 \overset{p_{12}}{\bowtie\!\!\!\!\subset} (R_2 \overset{p_{23}}{\bowtie} R_3)) \overset{p_{14}}{\triangleright} R_4$$
$$Q_2: \quad ((R_1 \overset{p_{12}}{\bowtie\!\!\!\!\subset} (R_2 \overset{p_{23}}{\bowtie} R_3)) \overset{p_{14}}{\triangleright} R_4) \overset{p_{15}}{\triangleright} R_5$$
$$Q_3: \quad (((R_1 \overset{p_{12}}{\bowtie\!\!\!\!\subset} (R_2 \overset{p_{23}}{\bowtie} R_3)) \overset{p_{14}}{\triangleright} R_4) \overset{p_{15}}{\triangleright} R_5) \overset{p_{26}}{\triangleright} R_6$$

Here, $R_1 = \sigma_{c\_acctbal>c_1}(Customer)$, $R_2 = \sigma_{o\_totalprice>c_2}(Orders)$, $R_3 = \sigma_{l\_quantity>v}(Lineitem)$, $R_4 = \sigma_{s\_acctbal>c_4}(Supplier)$, $R_5 = \sigma_{n\_name>c_5}(Nation)$, $R_6 = \sigma_{c\_address<c_6}(Customer)$; and p_{12} is "$c\_custkey = o\_custkey$", p_{23} is "$(o\_orderkey = l\_orderkey) \wedge (l\_extendedprice > c_3 \times o\_totalprice)$", p_{14} is "$s\_nationkey = c\_nationkey$", p_{15} is "$n\_nationkey = c\_nationkey$", p_{26} is "$o\_custkey = c\_custkey$". $c_1, c_2, ..., c_6$ are some constant values, and v is a parameter that is used to vary the selectivity factor of the selection predicate on *Lineitem* table (i.e., the ratio of the cardinality of R_3 to the cardinality of *Lineitem*).

For each of these queries, we compare the performance of its conventional query plan produced by PostgreSQL against the best compensated query plan produced by our approach. The running times of the queries are compared in Figure 7 as the selectivity factor of the selection predicate on *Lineitem* table (i.e., R_3) is varied.

For query Q_1, PostgreSQL is unable to reorder the joins as shown by its query plan P_1^{pg} in Figure 6(a). In contrast, our approach `CJR` could reorder the joins to get the rewritten query plan P_1^{CJR} shown in Figure 6(b).

From the results in Figure 7(a), we observe that the compensated query plan P_1^{CJR} has better performance than the conventional query plan P_1^{pg}, especially when R_3 is large. This is because the compensated query plan P_1^{CJR} postpones joining with R_3 to avoid generating a large intermediate result table early as in P_1^{pg}. In contrast, as the joins are not reordered in P_1^{pg}, the large intermediate result produced makes the subsequent joins expensive. The runtime improvement factor of the compensated query plan over the traditional query plan for Q_1 is up to 2.41.

For Q_2, its canonical query plan is shown in Figure 6(c). The conventional query plan P_2^{pg} generated by PostgreSQL is shown in Figure 6(d) which is able to reorder the two antijoins in the query. This is because l-asscom($\triangleright$, $\triangleright$) is a valid transformation [7] and therefore does not require any compensation. `CJR`'s compensated query plan P_2^{CJR} is shown in Figure 6(e)

which also postpones the joining of table R_3. As Figure 7(b) shows, the compensated query plan P_2^{CJR} outperforms the conventional query plan P_2^{pg} especially when R_3 is large. The runtime improvement factor of the compensation query plan over the traditional query plan for Q_2 is up to 2.48.

For Q_3, the canonical query plan, PostgreSQL's query plan P_3^{pg}, and CJR's compensated query plan P_3^{CJR} are shown in Figure 6(f)-(h). The performance comparison in Figure 7(c) shows that the compensated query plan P_3^{CJR} outperforms the conventional query plan P_3^{pg}. The reason is similar to that for Q_1 and Q_2. The runtime improvement factor of the compensation query plan over the traditional query plan for Q_3 is up to 4.66.

Query No.	Best conventional plan	Best compensated plan	Improvement factor
1	10.46s	10.06s	1.03
5	13.90s	10.59s	1.31
8	6.94s	2.81s	2.46
22	12.39s	2.42s	5.12
37	11.58s	11.27s	1.02
50	12.19s	3.06s	3.98
60	13.50s	12.25s	1.10
70	12.87s	2.67s	4.81
88	7.83s	2.81s	2.78
97	12.13s	2.61s	4.64

TABLE VII: Performance results for mutated TPC-H queries

In addition to the three discussed queries, we have also applied the mutation method described in Section V-A on queries Q_1, Q_2, and Q_3 to generate 100 additional outerjoin queries for each of Q_1, Q_2, and Q_3. Using a selectivity factor of 0.8 for the Lineitem table, the improvement factors for these three groups of mutated queries are up to 5.35, 5.12, and 7.14, respectively. Due to space constraint, we discuss only the detailed results for Q_2. After discarding the invalid queries from the 100 mutated Q_2 queries, we obtain 48 valid outerjoin queries, and run these queries on the TPC-H data. For 10 of the 48 outerjoin queries, the best compensated plan runs faster than the best conventional plan with a performance improvement factor of up to 5.12. The detailed results are shown in Table VII.

VI. Related Work

We have already introduced the closely related works TBA [7], CBA [11], and ECA [14] in Section I.

A rather different approach from these query rewriting approaches is RO [6] which evaluates a query involving some outerjoin by first computing a derived relation for each join operand, and then computing the inner join of the derived relations. Although the approach is interesting, the cost of computing the derived relations can be as high as computing the original outerjoin query [14].

Table VIII summarizes the join reorderability comparison of our approach CJR against the four existing works. For clarity, the comparison is shown in terms of two scenarios depending on whether there exists some null-tolerant join predicates in the queries: Table VIII(a) is for queries with only null-intolerant join predicates, and Table VIII(b) is for queries with some null-tolerant join predicates. For each approach, we indicate which of the five types of join reordering transformations are supported by that approach. For example, Table VIII(a) shows that TBA supports only the two types of valid transformations and none of the three types of invalid transformations.

For queries with only null-intolerant join predicates, the compensation-based approaches (i.e., CBA, ECA, CJR) all supersede TBA; in particular, our approach CJR is the only approach that supports complete join reorderability.

For queries with some null-tolerant join predicates, the supported join reorderability becomes more limited as none of the approaches can support complete join reorderability. Specifically, even the compensation-based approaches can not support these invalid join transformations, and our approach CJR supports all the valid join transformations similar to TBA, CBA, and ECA.

	valid transformation not involving $\triangleright$	valid transformation involving $\triangleright$	invalid transformation involving $\bowtie, \mathbb{J}\bowtie$	invalid transformation involving $\triangleright$	invalid transformation involving $\mathbb{J}\subset$
TBA [7]	✓	✓	✗	✗	✗
RO [6]	✓	✗	✓	✗	✓
CBA [11]	✓	✓	✓	✗	✗
ECA [14]	✓	✓	✓	✓	✗
CJR	✓	✓	✓	✓	✓

(a) Queries with only null-intolerant (i.e. null-rejecting) join predicates

	valid transformation not involving $\triangleright$	valid transformation involving $\triangleright$	invalid transformation involving $\bowtie, \mathbb{J}\bowtie$	invalid transformation involving $\triangleright$	invalid transformation involving $\mathbb{J}\subset$
TBA [7]	✓	✓	✗	✗	✗
RO [6]	✗	✗	✗	✗	✗
CBA [11]	✓	✓	✗	✗	✗
ECA [14]	✓	✓	✗	✗	✗
CJR	✓	✓	✗	✗	✗

(b) Queries with some null-tolerant join predicates

TABLE VIII: Comparison of join reorderability

VII. Conclusion

In this paper, we have presented the first complete solution to the join reorderability problem for null-intolerant joins. Our approach, which consists of a new compensation operator and an enhanced set of rewriting rules, provides complete join reorderability for queries with null-intolerant join predicates. Our experimental results on the Join Order Benchmark and TPC-H Benchmark have demonstrated that with the enlarged query plan search space, query performance can be improved by up to a factor of 12.32. As part of our future work, we plan to work on efficient native implementation of compensation operators.

REFERENCES

[1] G. Bhargava, P. Goel, and B. Iyer, "Hypergraph based reorderings of outer join queries with complex predicates," in *ACM SIGMOD*, 1995, pp. 304–315.

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

[3] C. Galindo-Legaria and A. Rosenthal, "How to extend a conventional optimizer to handle one- and two-sided outerjoin," in *IEEE ICDE*, 1992, pp. 402–409.

[4] ——, "Outerjoin simplification and reordering for query optimization," *ACM TODS*, vol. 22, no. 1, pp. 43–74, Mar. 1997.

[5] C. A. Galindo-Legaria, "Algebraic optimization of outerjoin queries," Ph.D. dissertation, Harvard University, 1992.

[6] G. Hill and A. Ross, "Reducing outer joins," *VLDB Journal*, vol. 18, no. 3, pp. 599–610, Jun. 2009.

[7] G. Moerkotte, P. Fender, and M. Eich, "On the correct and complete enumeration of the core search space," in *ACM SIGMOD*, 2013, pp. 493–504.

[8] G. Moerkotte and T. Neumann, "Dynamic programming strikes back," in *ACM SIGMOD*, 2008, pp. 539–552.

[9] J. Rao, B. G. Lindsay, G. M. Lohman, H. Pirahesh, and D. E. Simmen, "Using EELs, a practical approach to outerjoin and antijoin reordering," IBM Research Division, Tech. Rep. RJ 10203, December 2000.

[10] ——, "Using EELs, a practical approach to outerjoin and antijoin reordering," in *IEEE ICDE*, 2001, pp. 585–594.

[11] J. Rao, H. Pirahesh, and C. Zuzarte, "Canonical abstraction for outerjoin optimization," in *ACM SIGMOD*, 2004, pp. 671–682.

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

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

[14] T. Wang and C. Chan, "Improving join reorderability with compensation operators," in *ACM SIGMOD*, 2018, pp. 693–708.

[15] T. Wang and C.-Y. Chan, "Complete Join Reordering for Null-Intolerant Joins," National University of Singapore, Tech. Rep., 2022, https://bitbucket.org/taining/foj-enum/downloads/CJR-report.pdf.

[16] A. Nica, "Incremental maintenance of materialized views with outer-joins," *Inf. Syst.*, vol. 37, no. 5, pp. 430–442, 2012.

[17] R. Ikeda and J. Widom, "Outerjoins in uncertain databases," in *MUD*, ser. CTIT Workshop Proceedings Series, vol. WP09-14, 2009, pp. 33–46.

[18] K. Papaioannou, M. Theobald, and M. H. Böhlen, "Outer and anti joins in temporal-probabilistic databases," in *ICDE*. IEEE, 2019, pp. 1742–1745.

[19] K. Zhao and J. X. Yu, "All-in-one: Graph processing in rdbmss revisited," in *ACM SIGMOD*, 2017, pp. 1165–1180.

[20] ——, "Graph processing in rdbmss," *IEEE Data Eng. Bull.*, vol. 40, no. 3, pp. 6–17, 2017.

[21] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "Query optimization through the looking glass, and what we found running the join order benchmark," *VLDB Journal*, vol. 27, no. 5, pp. 643–668, 2018.

[22] M. Raasveldt and H. Mühleisen, "Data management for data science - towards embedded analytics," in *CIDR*, 2020.