Efficient Enumeration of the Complete Join Search Space

Altan Birler
altan.birler@tum.de
Technische Universität München
Garching, Germany

Thomas Neumann
neumann@in.tum.de
Technische Universität München
Garching, Germany

Abstract

Join plan enumeration is a critical step in query optimization, impacting the performance of queries by orders of magnitude. Many queries contain complex non-inner joins, such as outer joins and semi joins, which make the efficient enumeration of all valid join plans difficult. There are existing solutions, but they are either incomplete or expensive. We improve the state-of-the-art join plan enumeration, efficiently handling all cases, including outer joins. Our approach is both complete and efficient by exploiting the properties of relational join transformations.

CCS Concepts

• **Information systems → Query optimization**.

Keywords

query optimization, join enumeration, outer joins

ACM Reference Format:
Altan Birler and Thomas Neumann. 2025. Efficient Enumeration of the Complete Join Search Space. In *The 19th International Symposium on Database Programming Languages (DBPL '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3735106.3736536

1 Introduction

Query optimizers are responsible for generating efficient join plans given high-level queries. The performance of a query can vary by orders of magnitude depending on the join plan chosen [9]. To generate an efficient join plan, the optimizer searches through a large space of possible plans, all while ensuring that the generated plan is valid, i.e., produces the same result as the original query. This is made difficult by the interaction of various join types, such as inner joins, outer joins, and semi joins, which cannot be freely reordered [3] as shown in Figure 1. Additionally, since the search space is exponential in the size of the query definition, the efficient exploration of the search space is critical to finding a good plan in a reasonable time [8, 14]. Thus, the optimizer must be very efficient in detecting whether a plan is valid while iterating over the search space [10]. State-of-the-art approaches to this problem are either incomplete, i.e., they do not guarantee that all valid plans are generated, or they are based on expensive techniques. In this paper, we present a new approach to the problem of join plan generation that is both complete and efficient. Additionally, we support the decomposition of conjunctive predicates, which related work did not address. We achieve this result by pruning unnecessary reordering restrictions, i.e., if the join enumeration would never produce plans violating the restriction anyway, the restriction does not need to be considered. To achieve this result, we rely on a certain property of valid relational join operators we call the *hiding property*. We also show that existing techniques fail in the absence of the hiding property, implying that our approach is a strict improvement over existing techniques.

Figure 1: Invalid transformation in the presence of outer joins. An inner join and a full outer join are not associative in general due to nulls a full outer join may produce.

The main contributions of this paper are as follows:

- We introduce the *hiding property*, which constrains the interaction of commutativity and associativity. We show that existing approaches that provide completeness for relational joins do not generalize for operators that do not exhibit the hiding property.
- By exploiting the hiding property, we provide an efficient and complete validation algorithm for join plans. Our algorithm builds upon existing non-complete approaches which describe the query as a hypergraph [3, 10, 12, 16].
- We describe a graph algorithm for single decremental connectivity, which can be used to further speed up the construction step of our approach. Single decremental connectivity answers queries of the form: "Is there a path from u to v in the graph G after removing the edge (u, v)?"
- We describe a generalization of validation algorithms to support the decomposition of conjunctive predicates.
- We evaluate our approach against existing techniques and show that it is both complete and efficient.

The rest of the paper is organized as follows. In Section 2, we provide the necessary background on join plan enumeration and the properties of valid relational join transformations. We also introduce the hiding property, which complete enumeration techniques rely on. In Section 3, we discuss the state of the art approaches in the area of join plan enumeration for non-inner joins. We also discuss the limitations of existing approaches. Namely, they require the hiding property and do not support the decomposition of conjunctive predicates. In Section 4, we present our approach to join plan

DBPL '25, Berlin, Germany

ACM ISBN 979-8-4007-1919-6/2025/06
https://doi.org/10.1145/3735106.3736536

enumeration and show how it can be used to efficiently enumerate all valid join plans. In Section 5, we evaluate our approach against existing techniques and show that it is both complete and efficient. In Section 6, we discuss the limitations of existing approaches and outline future work, i.e., what plans do state of the art constructive enumeration techniques fail to generate and why. Finally, in Section 7, we summarize our findings.

2 Background

In this section, we describe state-of-the-art algorithms for enumerating the search space of join plans. These algorithms will provide the basis for more advanced algorithms that take complex join predicates and non-inner joins into account.

In Section 2.1, we introduce the notation used throughout this work. In Section 2.2, we describe join enumeration by the successive application of local transformation rules. In Section 2.3, we describe constructive plan generation, which generates plans in a bottom-up fashion, avoiding the need to enumerate the entire search space to find the optimal plan. In Section 2.4, we describe how to represent queries as graphs for more efficient enumeration when avoiding cross products. Finally, in Section 2.5, we describe how to represent queries as hypergraphs, which allows us to represent the constraints imposed on reordering by non-inner joins and complex join predicates.

2.1 Notation

This section describes the notation used throughout this work.

Operator $\circ^a$ denotes an arbitrary operator.

Subtrees $\mathcal{T}(\circ^a)$, left($\circ^a$), and right($\circ^a$) denote the relations in the subtree of the operator $\circ^a$, the left subtree of $\circ^a$, and the right subtree of $\circ^a$, respectively. If not otherwise specified, we consider the subtrees of $\circ^a$ within the initial query plan.

Subtree operators STO($\mathcal{R}$) denotes the set of operators that connect the set of relations $\mathcal{R}$. In other words, given any join plan formed by the relations in $\mathcal{R}$, STO($\mathcal{R}$) contains all operators that are part of the plan.

Syntactic eligibility set [12] SES($\circ^a$) denotes the set of tables referenced by the join operator $\circ^a$ in its predicates. SES-left($\circ^a$) and SES-right($\circ^a$) denote references to the left and right inputs of the join operator $\circ^a$, respectively.

Total eligibility set [12] TES($\circ^a$) denotes the extended set of tables required by the join operator $\circ^a$ in its input. TES-left($\circ^a$) and TES-right($\circ^a$) denote requirements to the left and right inputs of the join operator $\circ^a$, respectively. We will discuss the construction of the total eligibility sets in Section 2.5 and Section 4.

Null rejecting denotes a predicate that returns null or false on inputs that contain null values. Most SQL predicates such as $x = y$ are null rejecting, i.e., if either one of x or y is null, the predicate evaluates to null, which is interpreted as false, rejecting the input tuple. However, not all predicates are null rejecting, such as coalesce$(x, 0)$ = coalesce$(y, 0)$.

Join operators $\bowtie$, $\ltimes$, $⟕$, $⟗$, are used to denote the operators inner join, group join [13], left outer join, and full outer join, respectively. We group operators that share behaviors together. The $\bowtie$ operator is used to represent both inner joins $\bowtie$ and cross products $\times$. The $\ltimes$ operator is used to represent the left semi join, left anti join, and group join [13]. Additionally, we use the apostrophe notation to denote joins with null rejecting predicates. For instance, $'⟕$ denotes a full outer join with a null rejecting predicate on the left input and $'⟗'$ denotes a full outer join with null rejecting predicates on both inputs.

2.2 Enumeration of Join Plans

Given an initial join plan input by the user, a join optimization algorithm tries to find the optimal execution plan in terms of runtime cost. The execution plan must respect the semantics of the initial plan, meaning that the result of the query must not be changed by the optimization. The search space of valid plans can be defined through a set of local transformation rules. For instance, the cross product operator $\times$ is commutative $R_0 \times R_1 \equiv R_1 \times R_0$ and associative $R_0 \times (R_1 \times R_2) \equiv (R_0 \times R_1) \times R_2$. Given an initial plan of cross products, a transformative algorithm can apply these rules to generate all valid plans.

An inner join operator can additionally contain a selection predicate which references attributes of input relations. A selection predicate can only be applied when all the required attributes are available. For instance

$$R_0 \bowtie_{R_0.x=R_2.x} (R_1 \bowtie_{R_1.y=R_2.y} R_2) \not\equiv (R_0 \bowtie_{R_0.x=R_2.x} R_1) \bowtie_{R_1.y=R_2.y} R_2$$

because the selection predicate $R_0.x = R_2.x$ cannot be applied to the subtree[1]. Since we are only interested in the input relations referenced by the selection predicate, also known as the predicate's *syntactic eligibility set*, we will represent predicates with the shorthand notation $R_0 \bowtie_{02} (R_1 \bowtie_{12} R_2)$.

Certain non-inner join operators cannot be reordered in general, even if the syntactic eligibility sets are satisfied:

$$R_0 ⟕_{01} (R_1 ⟕_{12} R_2) \not\equiv (R_0 ⟕_{01} R_1) ⟕_{12} R_2$$

We refer to such constraints as *reordering restrictions*: Given the left initial plan, the join $⟕_{01}$ must succeed the join $⟕_{12}$ in the final execution plan. In this work, we focus on prohibiting invalid reorderings of outer joins. An alternative approach is to allow all reorderings of outer joins, but repair wrong query results by applying compensation operators [20, 21]. The drawback of this alternative approach is that the complex compensation operators can be costly to execute.

For most of this work, we consider predicates of joins as indivisible units of arbitrary complexity[2]. We will only differentiate whether a predicate is null rejecting for the left or right input of the join, i.e., whether the join eliminates tuples containing nulls in the left or right input. This property is important for precise reordering restrictions between outer joins and other join types. Note that more transformation rules can be defined by not considering the join predicates as indivisible units. For instance, the join $R_0 \bowtie_{R_0.x=R_2.x} (R_1 \bowtie_{R_1.x=R_2.x} R_2)$ can be transformed into $(R_0 \bowtie_{R_0.x=R_1.x} R_1) \bowtie_{R_1.x=R_2.x} R_2$ by rewriting the topmost join

[1]Note that we do not consider the equivalences that produce new cross products: $R_0 \bowtie_{R_0.x=R_2.x} (R_1 \bowtie_{R_1.y=R_2.y} R_2) \equiv (R_0 \times R_1) \bowtie_{R_0.x=R_2.x \wedge R_1.y=R_2.y} R_2$. Introducing cross products is a valid transformation but almost always results in plans with higher costs, especially considering errors in cardinality estimation. Not considering such transformations allows us to focus on reordering existing operators without introducing new ones. There are other valid use cases for the introduction of new operations such as semi join reduction, but they are out of scope for this work.

o^a \ o^b	$\bowtie$	$\bowtie'$	$\rtimes$	$'\rtimes$	$\ltimes$	$'\ltimes$	$\ltimes'$	$'\ltimes'$
$\bowtie$	+	+	+	+	−	−	−	−
$\bowtie'$	−	−	−	−	−	−	−	−
$\rtimes$	−	−	−	+	−	−	−	−
$'\rtimes$	−	−	−	+	−	−	−	−
$\ltimes$	−	−	−	+	−	−	−	−
$'\ltimes$	−	−	−	+	−	−	−	−
$\ltimes'$	−	−	−	+	−	+	−	+
$'\ltimes'$	−	−	−	+	−	+	−	+

Table 1: Associativity. Apostrophes indicate null rejecting sides.

o^a \ o^b	$\bowtie$	$\bowtie'$	$\rtimes$	$'\rtimes$	$\ltimes$	$'\ltimes$	$\ltimes'$	$'\ltimes'$
$\bowtie$	+	+	+	+	−	−	−	−
$\bowtie'$	+	+	+	+	−	−	−	−
$\rtimes$	+	+	+	+	−	−	−	−
$'\rtimes$	+	+	+	+	+	+	+	+
$\ltimes$	−	−	−	+	−	−	−	−
$'\ltimes$	−	−	−	+	−	+	−	+
$\ltimes'$	−	−	−	+	−	−	−	−
$'\ltimes'$	−	−	−	+	−	+	−	+

Table 2: L-asscom. Apostrophes indicate null rejecting sides.

predicate $R_1.x = R_2.x$ into $R_0.x = R_2.x$ using the equivalence of attributes. We will discuss how to decompose conjunctive predicates in Section 4.5. We will briefly discuss the limitations of current approaches in Section 6.

We have given examples for the transformation rules commutativity and associativity. However, the application of commutativity often results in redundant plans (in terms of mirror symmetry of individual joins). This further complicates join enumeration as commutativity of certain joins changes the type of the join: A left outer join becomes a right outer join and vice versa. To avoid these problems, Moerkotte et al. [10] propose the following three fundamental transformation rules:

(1) **associativity**: $R_0 \circ^a_{01} (R_1 \circ^b_{12} R_2) \equiv (R_0 \circ^a_{01} R_1) \circ^b_{12} R_2$
(2) **l-asscom**: $(R_0 \circ^a_{01} R_1) \circ^b_{02} R_2 \equiv (R_0 \circ^b_{02} R_2) \circ^a_{01} R_1$
(3) **r-asscom**: $R_0 \circ^a_{02} (R_1 \circ^b_{12} R_2) \equiv R_1 \circ^b_{12} (R_0 \circ^a_{02} R_2)$

These three rules are sufficient to generate all valid plans ignoring mirror symmetry of individual joins. For a pair of operators $\circ^a_{01}$ and $\circ^b_{12}$, a lookup table can be used to determine whether a rule can be applied. We provide these lookup tables in Tables 1 to 3. By successively applying the rules, all valid plans can be generated, assuming the predicates are indivisible.

Transformative approaches generate all possible plans, which can be expensive in terms of time and space [15]. As we only need the optimal plan according to a certain cost model, constructive approaches can be used to generate plans in a bottom-up fashion [6, 12].

[2]A predicate can contain arbitrary expressions such as $\text{length}(R_0.x||R_1.y) > R_0.z + R_1.u$ and may be impossible to simplify or decompose.

o^a \ o^b	$\bowtie$	$\bowtie'$	$\rtimes$	$'\rtimes$	$\ltimes$	$'\ltimes$	$\ltimes'$	$'\ltimes'$
$\bowtie$	+	−	−	−	−	−	−	−
$\bowtie'$	−	−	−	−	−	−	−	−
$\rtimes$	−	−	−	−	−	−	−	−
$'\rtimes$	−	−	−	−	−	−	−	−
$\ltimes$	−	−	−	−	−	−	−	−
$'\ltimes$	−	−	−	−	−	−	−	−
$\ltimes'$	−	−	−	−	−	−	+	+
$'\ltimes'$	−	−	−	−	−	−	+	+

Table 3: R-asscom. Apostrophes indicate null rejecting sides.

2.3 Constructive Plan Generation

Constructive plan generation is a bottom-up approach to generating plans based on the optimality principle [1]. Given a monotonous cost function (all practical cost functions are monotonous), an optimal plan exists where all its subplans are also optimal [12]. Thus, we can avoid generating *all* plans by only generating plans by combining smaller optimal subplans. However, not all combinations of optimal subplans are valid, as reordering restrictions may not be satisfied. Thus, an efficient check for plan validity is required, where we need to answer the question of whether a plan could be reached from the initial plan by applying valid transformation rules.

There are various approaches for checking plan validity, with different trade-offs between completeness (which percentage of valid plans are found) and efficiency (how fast can we enumerate plans). Our approach, described in Section 4, is both efficient and complete. We achieve this by relying on an enumeration algorithm that avoids cross products (see Section 2.4) and a hypergraph representation of the query (see Section 2.5).

2.4 Query as Graph

Cross products are often detrimental to the performance of a query [4, 11, 12, 14]. Thus, most query optimizers avoid cross products unless absolutely necessary, i.e., unless the original query contains a cross product. The bottom-up join enumeration while avoiding cross products can be reformulated as a graph problem and efficiently solved by connected subgraph enumeration algorithms [12].

Assuming join predicates require attributes from exactly two input relations, we can represent the query as a simple undirected graph. The vertices of the graph represent the input relations and the edges represent the join predicates. The entire graph represents the entire query, while a subgraph represents a subset of the query with the relations and predicates contained therein. A bottom-up join enumeration algorithm computes optimal plans for all connected components of the query graph. To compute the optimal plan for a connected component, we consider all possible pairs of connected subcomponents that are connected by a join predicate. Note that, if a subgraph is connected, it can be evaluated without introducing new cross products. Thus, enumerating all optimal plans that do not introduce cross products can be done by enumerating all connected subgraph pairs of the query graph. Using the DPccp

algorithm [11], this enumeration can be done in constant time per connected subcomponent pair[3].

2.5 Query as Hypergraph

To support join predicates with more than two input relations, hypergraphs are used. A query hypergraph's edges are defined as a pair of sets of vertices. Note that we define the hyperedges as having left and right sets of vertices instead of the single syntactic eligibility set. This is advantageous for guaranteeing efficient executions for predicates of the form $p \equiv R_0.x+R_1.y = R_2.z$. While such a predicate could be applied on the inputs $\{R_0, R_2\}$ and $\{R_1\}$ as in $(R_0 \bowtie R_2) \bowtie_p R_1$, this would prevent us from using an efficient physical operator such as hash join. Thus, in edges, we explicitly define the left and right sets for predicates. To combine two plans for subgraphs using a join hyperedge, the left subgraph must contain the left set of the hyperedge and the right subgraph must contain the right set of the hyperedge. So the join edge $(\{R_0, R_2\}, \{R_1\})$ can join a plan for the left subgraph of relations R_0, R_2, R_3 and a plan for the right subgraph of relations R_1, R_5, as both subgraphs contain the required relations.

Hypergraphs can also be utilized to encode reordering restrictions [3]. For instance, given the initial plan $R_0 \bowtie_{01} (R_1 \bowtie_{12} R_2)$, the join $\bowtie_{01}$ *must* be applied after the join $\bowtie_{12}$. This information can be encoded using hyperedges. If we assign $\bowtie_{01}$ the hyperedge $(\{R_0\}, \{R_1, R_2\})$, we require that the right subgraph for this join must contain both R_1 and R_2. This implicitly enforces that $\bowtie_{12}$ must be applied first. The set of relations referred to by these extended hyperedges are called the *total eligibility set* of the join.

The DPccp algorithm for enumerating connected subcomponents of a graph has been extended to hypergraphs with the DPhyp algorithm [12]. Although the DPhyp algorithm does not guarantee constant time per connected hyper-subcomponent pair, it is still efficient in practice. Thus, encoding reordering restrictions as hyperedges allows us to efficiently enumerate all valid plans. However, to the best of our knowledge, no existing algorithm using hypergraph-based encoding of reordering restrictions is complete, i.e., they do not guarantee that all valid plans are found (see Section 3). And approaches that do guarantee completeness are not as efficient in practice. In Section 4, we will describe our novel hypergraph-based approach that guarantees completeness. We rely on the DPhyp algorithm that guarantees no cross products are introduced. When we can prove that the violation of a reordering restriction would necessitate the introduction of a new cross product, we can safely prune the restriction and not introduce hyperedges.

2.6 Cross Products

Most query optimizers reject plans containing cross products unless absolutely necessary. Unfortunately, some queries are disconnected. They must be evaluated using a cross product. In such a case, we introduce an additional edge to the query graph between the two disconnected components to hide the cross product from the rest of the query. Given $\mathcal{R} \times \mathcal{S}$, we reinterpret this query as $\mathcal{R} \bowtie_{RS} \mathcal{S}$, where R and S are arbitrary relations from $\mathcal{R}$ and $\mathcal{S}$, respectively.

[3]DPccp enumerates in constant time if the number of nodes fit into a machine word.

2.7 Selections and Maps

A complex join tree can contain various additional operators that break up the joins. Considering these additional operators within the reordering of the joins enables us to explore a larger search space. Selection and map operators can be interpreted as joins with faux tables. This allows them to be considered as a natural part of the query graph. We do not consider the integration of group by operators into the query graph [5] in this work.

2.8 Hiding Property

We want to exploit the properties of join operators to design efficient algorithms for join plan enumeration. The following basic implications can be derived from the definition of transformation rules in Section 2.2:

$$\text{l-asscom}(\circ^b, \circ^a) \implies \text{l-asscom}(\circ^a, \circ^b) \quad (1)$$

$$\text{r-asscom}(\circ^b, \circ^a) \implies \text{r-asscom}(\circ^a, \circ^b) \quad (2)$$

$$\text{assoc}(\circ^a, \circ^b) \wedge \text{assoc}(\circ^b, \circ^a) \wedge \text{l-asscom}(\circ^a, \circ^b) \implies \text{r-asscom}(\circ^a, \circ^b) \quad (3)$$

$$\text{assoc}(\circ^a, \circ^b) \wedge \text{assoc}(\circ^b, \circ^a) \wedge \text{r-asscom}(\circ^a, \circ^b) \implies \text{l-asscom}(\circ^a, \circ^b) \quad (4)$$

$$\text{assoc}(\circ^b, \circ^a) \wedge \text{l-asscom}(\circ^a, \circ^b) \wedge \text{r-asscom}(\circ^a, \circ^b) \implies \text{assoc}(\circ^a, \circ^b) \quad (5)$$

$$\text{assoc}(\circ^a, \circ^b) \wedge \text{l-asscom}(\circ^a, \circ^b) \wedge \text{r-asscom}(\circ^a, \circ^b) \implies \text{assoc}(\circ^b, \circ^a) \quad (6)$$

All relational join operators additionally exhibit the following four properties:

$$\text{assoc}(\circ^a, \circ^b) \wedge \text{assoc}(\circ^b, \circ^c) \implies \text{assoc}(\circ^a, \circ^c) \quad (7)$$

$$\text{l-asscom}(\circ^a, \circ^b) \wedge \text{assoc}(\circ^b, \circ^c) \implies \text{l-asscom}(\circ^a, \circ^c) \quad (8)$$

$$\text{assoc}(\circ^c, \circ^b) \wedge \text{r-asscom}(\circ^b, \circ^a) \implies \text{r-asscom}(\circ^c, \circ^a) \quad (9)$$

$$\text{r-asscom}(\circ^a, \circ^b) \wedge \text{l-asscom}(\circ^b, \circ^c) \implies \text{assoc}(\circ^a, \circ^c) \quad (10)$$

We call these four statements the *hiding property*. The hiding property states that if a join operator $\circ^b$ may be reordered below $\circ^a$, then all operators $\circ^c$ that are reorderable under the opposite side of $\circ^b$ must also be reorderable under $\circ^a$. In other terms, even if an operator $\circ^c$ *hides behind* $\circ^b$, its reordering restrictions regarding $\circ^a$ will be respected by $\circ^b$. This is shown in Figure 2 for Equation (8). If $\circ^b$ can be reordered under $\circ^a$, then $\circ^c$ must be reorderable under $\circ^a$ as well. In Section 3.4, we will show that the approach CD-C, an approach that claims completeness, fails to find valid plans in the absence of the hiding property. Furthermore, in Section 4.2, we use the hiding property to argue the completeness of our approach.

The individual transformation rules have local effects on the query plan. However, in order to do efficient bottom-up join enumeration, we need global properties that are easily checked. The hiding property (combined with the basic implications) extends the individual operator incompatibilities to global statements on reordering restrictions. Consider an initial join plan T transformed into a plan T' by applying a sequence of valid transformation rules.

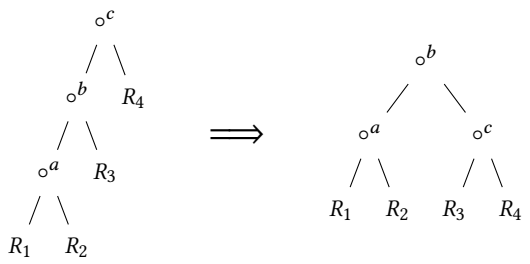

Figure 2: The hiding property. $\circ^c$ can *hide behind* $\circ^b$. Nonetheless, we know that if $\circ^c$ cannot be reordered under $\circ^a$, $\circ^b$ cannot be reordered under $\circ^a$ either.

As joins satisfy the hiding property, the following statements hold for any pair of operators $\circ^a$ and $\circ^b$ in T and T':

- If $\neg\text{assoc}(\circ^a, \circ^b)$, and $\circ^a$ was in the left subtree of $\circ^b$ in T, then $\circ^b$ cannot be in the right subtree of $\circ^a$ in T'.
- If $\neg\text{assoc}(\circ^b, \circ^a)$, and $\circ^a$ was in the right subtree of $\circ^b$ in T, then $\circ^b$ cannot be in the left subtree of $\circ^a$ in T'.
- If $\neg\text{l-asscom}(\circ^a, \circ^b)$, and $\circ^a$ was in the left subtree of $\circ^b$ in T, then $\circ^b$ cannot be in the left subtree of $\circ^a$ in T'.
- If $\neg\text{r-asscom}(\circ^a, \circ^b)$, and $\circ^a$ was in the right subtree of $\circ^b$ in T, then $\circ^b$ cannot be in the right subtree of $\circ^a$ in T'.

These statements generalize the local incompatibilities of individual operators to global statements of the query plan. The dynamic programming approaches described in Section 3 will exploit these global statements to efficiently verify the validity of plan without having to trace an entire sequence of valid transformations.

We could not yet formally prove that the hiding property implies the global statements. However, we experimentally verified with random queries and transformation sequences that the hiding property can be used to deduce all the global incompatibilities that arise. We start with a random tree, apply a sequence of transformations, and use Datalog to extend the known operator compatibilities. We then check whether all global statements are satisfied for all operator pairs. For example, if an operator $\circ^a$ was in the left subtree of $\circ^b$ in the initial plan, and $\circ^b$ is in the right subtree of $\circ^a$ in the final plan, then we check whether $\text{assoc}(\circ^a, \circ^b)$ can be derived using the applied transformations, basic implications, and the hiding property.

3 Related Work

There are different approaches for checking the validity of a plan, i.e., determining whether an optimized plan can be reached from the initial by only applying valid transformations. Moerkotte et al. [10] propose three approaches for computing validity: CD-A, CD-B, and CD-C. We only consider CD-A and CD-C as they supersede CD-B in terms of either efficiency or completeness. CD-A exclusively relies on the hypergraph representation of the query (see Section 2.5) and is thus efficient. However, CD-A is not complete: It is not able to generate all valid plans reachable from the initial plan via valid transformations. Moerkotte et al. [10] claim completeness for CD-C when it is disallowed to decompose conjunctive predicates. However, CD-C relies on *conflict rules* (CRs) instead of hyperedges, which makes the enumeration of join plans more costly.

In Section 3.1, we give an overview of how the simpler CD-A algorithm works in checking plan validity. In Section 3.2, we describe how CRs work, which are utilized by CD-C. In Section 3.3, we build on Section 3.2 to describe the full CD-C algorithm. Finally, in Section 3.4, we show how CD-C is not complete for arbitrary operators. We describe the implicit assumptions made by CD-C to ensure completeness, and argue that these assumptions hold with relational joins. This analysis will form the basis for why our approach based on the weaker hypergraph construct can be complete.

3.1 CD-A

In Section 2.5, we described how the hypergraph representation of a query can be used to encode reordering restrictions. The CD-A algorithm builds such a hypergraph representation by first determining total eligibility sets (TES) for each join operator in the query. The TES start with the syntactic eligibility sets (SES), which are the input relations referenced by the predicate of the join. Then, the algorithm iterates bottom up over the query tree, finding conflicts between operators. If an operator pair is found to be conflicting, the upper operator's total eligibility set is extended with the lower operator's inputs, ensuring that the lower operator is executed before the upper operator. Pseudocode for the algorithm is given in Algorithm 1.

```
Input: Join operator ∘^b
Output: Total eligibility set TES(∘^b)
for ∘^a ∈ STO(left(∘^b)) do
    if ¬assoc(∘^a, ∘^b) then
        TES(∘^b) += left(∘^a);
    end
    if ¬l-asscom(∘^a, ∘^b) then
        TES(∘^b) += right(∘^a);
    end
end
for ∘^a ∈ STO(right(∘^b)) do
    if ¬assoc(∘^a, ∘^b)) then
        TES(∘^b) += right(∘^a);
    end
    if ¬r-asscom(∘^a, ∘^b) then
        TES(∘^b) += left(∘^a);
    end
end
```

Algorithm 1: CD-A without degenerate predicates. This algorithm is invoked for each join operator $\circ^b$ in the initial plan, bottom up.

After the TES are computed, those sets can be split to left and right sets based on the input relations of the join operator to form hyperedges. These hyperedges are then used in the enumeration algorithm to guarantee no invalid plans are generated.

Given the left example in Figure 1 and Table 1, the algorithm will produce $\text{TES}(\mathbb{X}_{12}) = \{R_1, R_2\}$ and $\text{TES}(\bowtie_{23}) = \{R_1, R_2, R_3\}$. These

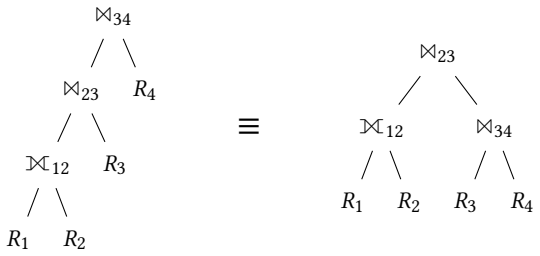

Figure 3: Valid transformation in the presence of outer joins. The inner joins can be reordered amongst themselves. But an imprecise reordering restriction may prevent the valid transformation.

correspond to the hyperedges $(\{R_1\}, \{R_2\})$ and $(\{R_1, R_2\}, \{R_3\})$, respectively. Given these hyperedges, the right tree in Figure 1 is not valid, as the hyperedge $(\{R_1, R_2\}, \{R_3\})$ for the inner join is not satisfied as its left side does not contain R_1. CD-A thus prevents the invalid plan.

Nonetheless, CD-A is not complete. Given the left example in Figure 3, the algorithm will produce $\text{TES}(\bowtie_{34}) = \{R_1, R_3, R_4\}$ and the corresponding hyperedge $(\{R_1, R_3\}, \{R_4\})$, to prevent the inner join from being executed before the outer join. However, this hyperedge is too restrictive, as it prevents the valid transformation to the right tree in Figure 3. Note also that this restriction is unnecessary. Reordering $\bowtie_{34}$ below $⟗_{12}$ would result in a cross product, which we explicitly disallow. We exploit this observation to eliminate redundant hyperedges in our approach (see Section 4).

3.2 Conflict Rules (CRs)

A hyperedge $(\mathcal{R}_{\text{left}}, \mathcal{R}_{\text{right}})$ corresponding to a join $\circ^a$ requires the set of relations $\mathcal{R}_{\text{left}}$ to be present in the left subtree of o^a and $\mathcal{R}_{\text{right}}$ to be present in the right subtree of o^a. A CR, similar to a hyperedge, is a pair of sets of relations: $\mathcal{S}_{\text{left}} \rightarrow \mathcal{S}_{\text{right}}$. However, the requirement is more relaxed compared to hyperedges. If any relation from $\mathcal{S}_{\text{left}}$ is present in the input of the join, all relations from $\mathcal{S}_{\text{right}}$ must also be present in the input. So, if none of the relations from $\mathcal{S}_{\text{left}}$ are present, the relations from $\mathcal{S}_{\text{right}}$ are not required.

The DPccp algorithm can be extended to support CRs, albeit with high costs. For each connected subcomponent pair that DPccp finds, all the CRs corresponding to the connecting edge are validated. If any CR is violated, the subcomponent pair is not valid and is not considered for an optimal plan.

CRs can be used to encode more complex relationships compared to hyperedges. However, surprisingly, our approach is able to find a hypergraph that is also complete (see Section 4). This means that the additional power provided by CRs, and the high validation costs, are not necessary for completeness.

3.3 CD-C

The CD-C algorithm works similarly to CD-A, iterating over the query tree and checking for conflicts. However, instead of producing hyperedges, it uses CRs to check for conflicts. The simplified pseudocode for CD-C is given in Algorithm 2, ignoring degenerate predicates[4].

Input: Join operator $\circ^b$
Output: Conflict rules $\text{CR}(\circ^b)$
for $\circ^a \in \mathit{STO}(\mathit{left}(\circ^b))$ **do**
  if $\neg \mathit{assoc}(\circ^a, \circ^b)$ **then**
    $\text{CR}(\circ^b)$ += $\text{right}(\circ^a) \rightarrow \text{SES-left}(\circ^a)$
  end
  if $\neg \mathit{l\text{-}asscom}(\circ^a, \circ^b)$ **then**
    $\text{CR}(\circ^b)$ += $\text{left}(\circ^a) \rightarrow \text{SES-right}(\circ^a)$
  end
end
for $\circ^a \in \mathit{STO}(\mathit{right}(\circ^b))$ **do**
  if $\neg \mathit{assoc}(\circ^a, \circ^b))$ **then**
    $\text{CR}(\circ^b)$ += $\text{left}(\circ^a) \rightarrow \text{SES-right}(\circ^a)$
  end
  if $\neg \mathit{r\text{-}asscom}(\circ^a, \circ^b)$ **then**
    $\text{CR}(\circ^b)$ += $\text{right}(\circ^a) \rightarrow \text{SES-left}(\circ^a)$
  end
end

Algorithm 2: CD-C without degenerate predicates. This algorithm is invoked for each join operator $\circ^b$ in the initial plan, bottom up.

CD-C is complete when predicates may not be decomposed. Thus, it can find valid plans that CD-A cannot. For example, in Figure 3, CD-C will produce the CR $\{R_2\} \rightarrow \{R_1\}$ for the join $\bowtie_{34}$. This CR is satisfied by the right tree in Figure 3, as $\bowtie_{34}$ does not contain R_2 in its input.

CD-C is not as efficient as CD-A, as it has to potentially loop over and check multiple CRs for each operator instead of just the one hyperedge. While the number of CRs can be reduced using simplification rules [10], the number of CRs per operator is upper-bounded by twice the number of operators in its subtree. Regardless, since the check whether an operator is applicable is executed for each connected component pair, any slowdown in the validation results in a significant slowdown in the overall algorithm.

3.4 CD-C Is Not Complete for Arbitrary Operators

CD-C is complete for relational joins. For completeness, it relies on the hiding property as described in Section 2.8. We can, however, show that CD-C is not complete for arbitrary hypothetical operators. Here, we will provide an example for operators that violate the transitivity of associativity as in Equation (7). If there is an operator triplet where associativity is not transitive:

$$\exists \circ^a, \circ^b, \circ^c : \mathit{assoc}(\circ^a, \circ^b) \wedge \mathit{assoc}(\circ^b, \circ^c) \wedge \neg \mathit{assoc}(\circ^a, \circ^c)$$

we can give an initial plan and a final plan where CD-C will not recognize the validity of the final plan, even though the final plan can be reached through valid transformations. In Figure 4, we show an example of this. In the initial tree, the CR $\{R_2, R_3\} \rightarrow \{R_1\}$ is

[4]Degenerate predicates are predicates that do not refer to at least one relation from each side of the join operator.

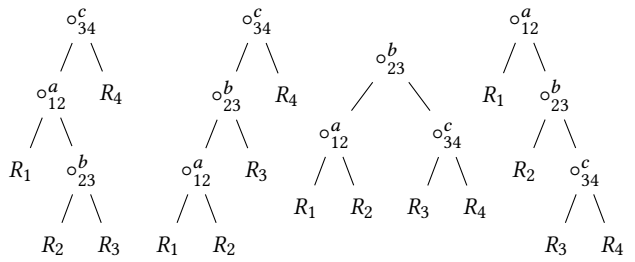

Figure 4: Valid transformations leading to a final state that is rejected by CD-C. The transformations applied are left to right in order $assoc(\circ^a, \circ^b)$, $assoc(\circ^b, \circ^c)$, and $assoc(\circ^a, \circ^b)$.

produced for the operator $\circ^c$ as $\neg assoc(\circ^a, \circ^c)$. Since the input of $\circ^c$ contains R_3 but not R_1 in the final plan, the CR is violated.

For the relational join operators that we consider, associativity is transitive. However, it is possible to construct a set of hypothetical operators where associativity is not transitive. As a simple example for binary operators, we can use ternary logic operators that input and output 3 possible values: 0, 1, and 2. The following is an example of three ternary logic operators that violate transitivity of associativity:

- $\circ^a(x, y)$: if $x = 0 \wedge y = 0$ then 1, else 0.
- $\circ^b(x, y)$: 0.
- $\circ^c(x, y)$: if $x = 1 \wedge y = 1$ then 2, else 0.

Since transitivity is required for the completeness of CD-C, we can explicitly exploit transitivity to develop a hypergraph representation of the query that is also complete. We demonstrate this in Section 4.

4 Approach

In Section 3.1, we provided an example of a redundant reordering restriction. Even though the rules of CD-A require us to create a hyperedge for $\bowtie_{34}$ in Figure 1 to prevent that inner join from being placed below the outer join, that hyperedge prevents us from reordering the inner joins amongst each other. Such restrictions have a drastic impact on the search space: A single outer join below a complex plan can end up preventing a significant number of reorderings (see Section 5). Surprisingly, in this example, considering the outer join as a base table for the rest of the plan improves the situation. This is counterintuitive, as considering more joins for reordering should not reduce our search space.

Nonetheless, that hyperedge is not needed. As we prevent cross products, $\bowtie_{34}$ could never go below $⟗_{12}$. For $\bowtie_{34}$ to be placed below $⟗_{12}$, $\bowtie_{23}$ would have to be placed below $⟗_{12}$, which we also prevent with a hyperedge. So a single hyperedge for the middle join $\bowtie_{23}$ is sufficient to prevent invalid reordering. We generalize this technique and present an algorithm to avoid introducing redundant hyperedges.

4.1 Redundant Reordering Restrictions

Assume we have two operators $\circ^a$ and $\circ^b$ where $\neg assoc(\circ^a, \circ^b)$ and $\circ^a$ is in the right subtree of $\circ^b$. We need to prevent plans where $\circ^b$ is in the right subtree of $\circ^a$. We only need to prevent this if it is possible for $\circ^b$ to be in the right subtree of $\circ^a$. To check this, we need to ask the question: Is it possible for $\circ^b$ to be placed below $\circ^a$ without resulting in cross products? In essence, we remove $\circ^a$ from the query graph and then check whether we can connect the corresponding relations. We describe a connectivity check algorithm in Section 4.3. Given a connectivity check operation, we amend CD-A as follows in Algorithm 3, calling it CD-E.

Input: Join operator $\circ^b$
Output: Total eligibility set TES($\circ^b$)
for $\circ^a \in STO(left(\circ^b))$ **do**
  if $\neg assoc(\circ^a, \circ^b) \wedge connected(right(\circ^a), right(\circ^b), \circ^a)$ **then**
    TES($\circ^b$) += TES-left($\circ^a$);
  end
  if $\neg l\text{-}asscom(\circ^a, \circ^b) \wedge connected(left(\circ^a), right(\circ^b), \circ^a)$ **then**
    TES($\circ^b$) += TES-right($\circ^a$);
  end
end
for $\circ^a \in STO(right(\circ^b))$ **do**
  if $\neg assoc(\circ^a, \circ^b) \wedge connected(left(\circ^a), left(\circ^b), \circ^a)$ **then**
    TES($\circ^b$) += TES-right($\circ^a$);
  end
  if $\neg r\text{-}asscom(\circ^a, \circ^b) \wedge connected(right(\circ^a), left(\circ^b), \circ^a)$ **then**
    TES($\circ^b$) += TES-left($\circ^a$);
  end
end

Algorithm 3: CD-E. CD-E is based on CD-A in Algorithm 1, but with two important changes. First, instead of amending TES($\circ^b$) with the entire $\mathcal{T}(\circ^a)$, we only add from TES($\circ^a$). Second, we check for connectivity before adding the TES. This algorithm is invoked for each join operator $\circ^b$ in the initial plan, bottom up.

4.2 Completeness of CD-E

In this section, we briefly argue for the completeness of CD-E if predicate decomposition is not allowed. We additionally empirically verify this in Section 5.

We want to look at cases where CD-C allows a plan where CD-A prevents it. Assume, for the sake of argument, we have two operators $\circ^a$ and $\circ^c$ where $\neg assoc(\circ^a, \circ^c)$ and $\circ^a$ is in the left subtree of $\circ^c$. In Section 3, we saw that CD-A could prevent valid reorderings while CD-C does not. We know that both CD-A and CD-C will prevent $\circ^c$ from being reordered in the right subtree of $\circ^a$. But CD-C may allow $\circ^c$ to be reordered below a third operator $\circ^b$ while CD-A may prevent it. So, for there to be a plan that is valid under CD-C but not under CD-A, there needs to be a third operator $\circ^b$ on the path from $\circ^a$ to $\circ^c$.

An example is given in Figure 3, with three operators $\circ^a$, $\circ^b$, and $\circ^c$ corresponding to $\mathbb{K}_{12}$, $\bowtie_{23}$, and $\bowtie_{34}$, respectively. If a reordering of $\circ^c$ is allowed by CD-C, but prevented by CD-A, this implies that in the resulting plan $\circ^c$ does not contain $\circ^a$ in its input, as in the right side of Figure 3. Given that $\circ^a$ was in the left subtree of $\circ^c$, this implies that $\circ^c$ has moved to the right subtree of an operator $\circ^b$ on the path from $\circ^a$ to $\circ^c$. Hence, assoc($\circ^b$, $\circ^c$) must hold. We assumed initially that $\neg assoc(\circ^a, \circ^c)$. Since, for join operators, associativity is transitive, this implies that $\neg assoc(\circ^a, \circ^b)$ must also hold. This implies that $\circ^b$ cannot be reordered to the right of $\circ^a$. Algorithm 3 will have introduced a hyperedge for $\circ^b$ to prevent this. We can also infer that a path on the query graph from the right side of $\circ^c$ to the right side of $\circ^a$ must go over $\circ^b$, as we know predicates cannot be decomposed and $\circ^c$ can be reordered to the right side of $\circ^b$. This means that the hyperedge for $\circ^b$ will also prevent $\circ^c$ from being reordered to the right side of $\circ^a$. Thus, the connectivity check will fail, and CD-E will not add the hyperedge for $\circ^c$. Hence, we have argued that CD-E also allows all plans that CD-C allows.

4.3 Proving Redundancy via Connectivity Checks

As we have seen in Section 4.1, we can use connectivity checks to prove that a reordering restriction is redundant. In this section, we describe a generic algorithm for connectivity checks within hypergraphs of queries. The algorithm potentially has quadratic runtime in the size of the query per connectivity check. While this is not ideal, we can check connectivity exactly and allow for a large number of reorderings by preventing all redundant hyperedges. Still, we can improve on the performance of the algorithm with a second algorithm that takes constant time but can return false positives. We describe this algorithm in Section 4.4.

Our basic algorithm is based on the union find data structure. The union find data structure efficiently supports merging of two sets merge($\mathcal{R}, \mathcal{S}$) and checking whether two elements are in the same set find(R) = find(S). We extend this with the isMerged($\mathcal{R}$) operation in Algorithm 4, which checks whether all relations in $\mathcal{R}$ are part of the same set, i.e., whether they all have been merged together.

Input: Relations $\mathcal{R}$
Output: isMerged($\mathcal{R}$): Whether the relations have been made into a union
root = find(r) for arbitrary $r \in \mathcal{R}$;
for $r \in \mathcal{R}$ **do**
  if *find*(r) $\neq$ *root* **then**
    return false;
  end
end
return true;

Algorithm 4: Algorithm for checking whether all relations in a set are part of the same component within the union find data structure.

Our algorithm for checking connectivity is as follows. Whenever a hyperedge can be applied, i.e., the left and right sides of the hyperedge are connected, we merge the two sides of the hyperedge. We repeat this as long as we can find an edge to apply. In the end, we have maximal connected components of our hypergraph. We can finally check whether the two relation sets we are interested in are in the same component. This is illustrated in Algorithm 5, where we further increase the efficiency by assuming the inputs sets are already connected and exiting early whenever we can determine connectedness.

Input: Relations $\mathcal{R}_1, \mathcal{R}_2$, excluded edge E, query hyperedges $\mathcal{E}$
Output: connected($\mathcal{R}_1, \mathcal{R}_2, E$): Whether the relations from $\mathcal{R}_1$ and $\mathcal{R}_2$ are connected in the hypergraph without E
Let r_1, r_2 be arbitrary relations from $\mathcal{R}_1, \mathcal{R}_2$;
for $r \in \mathcal{R}_1$ **do**
  merge(r_1, r);
end
for $r \in \mathcal{R}_2$ **do**
  merge(r_2, r);
end
while *there is an edge that can be applied* **do**
  for $(\mathcal{S}_1, \mathcal{S}_2) \in \mathcal{E} \setminus \{E\}$ **do**
    if *isMerged*($\mathcal{S}_1$) $\wedge$ *isMerged*($\mathcal{S}_2$) **then**
      merge($\mathcal{S}_2, \mathcal{S}_2$);
      if *find*(r_1) = *find*(r_2) **then**
        return true;
      end
    end
  end
end
return false;

Algorithm 5: Algorithm for checking connectivity in a query hypergraph. We repeatedly apply edges until no new edges can be applied.

4.4 Fast Path for Connectivity Checks

To quickly verify whether an operator $\circ^b$ can be placed below $\circ^a$, we can quickly check whether the input relations of these operators are still connected in the query graph, if we were to remove the edge corresponding to $\circ^a$. If they are connected, we can run the more precise check in Algorithm 5. If they are not connected, we can immediately return false. We can make this check very efficient by building index structures on the initial state of the query graph, ignoring updates. If we answer connectivity queries on the initial query graph based on SES, we will have false positives but no false negatives. We may report that a pair of relations is connected when they are not but never the other way around, as the newly introduced hyperedges only reduce connectivity in the graph.

We call this problem *single decremental connectivity* as a special case of decremental dynamic connectivity [19] where only a single edge is removed before a connectivity query. To answer queries efficiently, we preprocess the query graph in multiple steps as shown in Figure 5:

- We start with the initial query graph where nodes are relations and edges are joins.

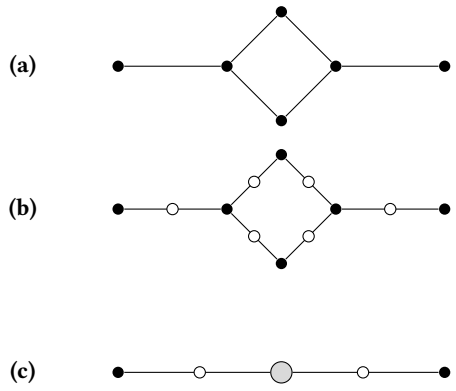

Figure 5: Preprocessing the query graph for fast connectivity checks. (a) The initial query graph with relations and joins. (b) The graph with midpoints added, representing the joins. (c) The bridge tree, a spanning tree of the query graph where each edge is a bridge.

- We add midpoints to the edges graph, shown as white nodes in Figure 5(b). These white nodes represent the joins.
- We build a bridge tree on the graph, shown in Figure 5(c). The bridge tree is a spanning tree of the query graph where each edge is a bridge. The bridge tree can be built using Tarjan's algorithm [7, 17, 18] that computes and merges bridge-connected components [22]. The merged components shown as shaded nodes in Figure 5(c).
- Given this spanning tree, we can answer connectivity queries in constant time by querying the lowest common ancestors of nodes[5]. Given two relations and a join edge, the relations are disconnected if the two relations are in different components and the node corresponding to the join predicate is not on the path between the two relations in the bridge tree.

If performance for the TES computation is critical, one could use the fast connectivity check to completely replace the expensive canPlaceBelow check. However, this would result in false negatives, meaning that the algorithm would not be complete. We name this alternative approach CD-D and evaluate it in Section 5.

4.5 Decomposing Predicates

Many queries contain inner joins with conjunctive predicates such as $a = b \wedge b = c$. Such predicates can be decomposed and the terms can be reordered independently. For example, the plan

$$(R \bowtie_{R.a=S.a} S) \bowtie_{S.b=T.b \wedge R.c=T.c} T$$

can be reordered to the following:

$$R \bowtie_{R.a=S.a \wedge R.c=T.c} (S \bowtie_{S.b=T.b} T)$$

Such predicates are especially common when many relations join on the same attribute such as $R.a = S.a = T.a = U.a$, where the transitive closure would result in a clique query graph. Disabling the decomposition of such predicates would result in the planner missing out on many valid reorderings.

Decomposing precidates is the same as interpreting each term in the conjunctive predicate as a separate inner join. Unfortunately, CD-C, as described in Section 3.1, may generate erroneous plans if all terms are considered as separate joins. For the query

$$((R_0 \mathbin{⟗}_{01} R_1)' \mathbin{⟗}_{12} R_2) \bowtie_{03,23} R_3$$

where the topmost inner join contains two conjunctive terms, CD-C does not explicitly disallow the plan

$$(R_1' \mathbin{⟗}_{12} R_2) \bowtie_{23} (R_0 \bowtie_{03} R_3)$$

where the $\mathbin{⟗}_{01}$ disappears[6].

The hypergraph based approaches that we have discussed, CD-A, CD-D, and CD-E can be adapted to allow for decomposing predicates, by simply considering each term in the conjunctive predicate as a separate join. One important caveat is that the transformations assoc, l-asscom, and r-asscom may lead to the sides of a join predicate being swapped, i.e., the same effect as applying commutativity. Thus, the checks for whether a join may be applied need to consider that the predicates may have been flipped for inner joins.

We evaluate the effectiveness of the hypergraph based algorithms in Section 5. None of them are complete in the presence of decomposable predicates, but CD-E finds the most plans.

5 Evaluation

In this section, we evaluate our approach CD-E to experimentally validate its correctness and completeness. We also compare it to other approaches, CD-A, CD-C, and CD-D, to show that CD-E is strictly better in the number of enumerated plans.

In Section 5.1, we describe the experimental setup, including the query generation methodology. In Section 5.2, we provide a short summary of the algorithms we test. In Section 5.3, we evaluate the algorithms with non-decomposable predicates. In Section 5.4, we evaluate the algorithms with decomposable predicates. Finally, in Section 5.5, we show some examples where CD-E does not find all valid plans with decomposable predicates.

5.1 Experimental Setup

To evaluate our approach, we have implemented a transformation-based join plan enumeration based on memoization [15]. This implementation is used to verify all other approaches. Every other algorithm must produce plans that are a subset of the plans produced by the transformation-based approach. All algorithms we show in this section correctly produce only a subset of the plans produced by the transformation-based approach.

We generate all possible queries with single binary predicates joining up to 7 relations similar to Moerkotte et al. [10]. Our query generator iterates over all possible tree shapes, then all possible join operators, and finally all possible binary predicates with one input from the left side and one input from the right side (we do not consider degenerate predicates). The join operators we iterate over are listed in Tables 1 to 3.

[5] A static tree can be preprocessed in $O(n)$ time, where n is the number of nodes, to answer lowest common ancestor queries in $O(1)$ time [2]. This can be done with an infix traversal of the tree followed by the construction of a range minimum query structure.

[6] More precisely, CD-C allows for the connected component pair $(\{1,2\},\{0,3\})$, which should not be allowed.

	CD-A	CD-H	CD-D	CD-C	CD-E	Total
Complete Queries	63.8%	65.9%	80.7%	100.0%	100.0%	12,659,013,896
Found Plans	89.2%	90.0%	96.0%	100.0%	100.0%	225,829,100,155

Table 4: Completeness with non-decomposable predicates. We show both the percentage of queries for which the algorithms found all plans, and the percentage of all plans that were found across all queries.

	CD-A	CD-H	CD-D	CD-C	CD-E	Total
Complete Queries	54.3%	55.2%	67.2%	-	85.5%	23,666,545,952
Found Plans	81.3%	81.7%	89.7%	-	96.3%	575,494,740,652

Table 5: Completeness with decomposable predicates. We show both the percentage of queries for which the algorithms found all plans, and the percentage of all plans that were found across all queries. CD-C produces incorrect plans and is thus not shown.

5.2 Algorithms

We compare the following algorithms:

- **CD-A**: The original algorithm due to Moerkotte et al. [10] that builds a hypergraph based on operator reordering restrictions.
- **CD-C**: The algorithm due to Moerkotte et al. [10] that builds up conflict rules instead of a hypergraph. This algorithm is complete when predicates are not decomposable. This algorithm does not support decomposable predicates.
- **CD-H**: This algorithm is a mixture of CD-C and CD-A. Moerkotte et al. [10] describe an algorithm to prune conflict rules. CD-H starts with CD-C and tries to prune the conflict rules. If it is not able to prune all conflict rules, it switches to CD-A. This algorithm always ends up with a hypergraph with no conflict rules. This makes it very efficient. Also, as it is based on CD-C, it has fewer false negatives than CD-A.
- **CD-D**: Our algorithm from Section 4.4 that relies on the fast connectivity check on the original query graph to prune reordering restrictions.
- **CD-E**: Our algorithm from Section 4.3 that applies a more connectivity check and is complete when predicates are not decomposable. This algorithm supports decomposable predicates.

5.3 Non-Decomposable Predicates

Given only one predicate per operator (which are not decomposable), we have over 10^{10} initial query plans for up to 7 relations. We show the number of plans found by each algorithm in Table 4. The results show that CD-C and CD-E are complete, while CD-A, CD-D, and CD-H are not. CD-D is not complete, but it is able to find 96% of all plans, which is a significant improvement over CD-A and CD-H, which find around 90% of all plans. CD-H improves on CD-A, but not significantly. CD-D represents a middle ground between CD-A and CD-E.

5.4 Decomposable Predicates

To test decomposable predicates, we take all queries we generated for Section 5.3, and add an additional binary predicate to an inner join within the query. We try all inner joins and all possible input relations for the additional binary predicate (we do not consider degenerate predicates). Additionally, we consider commutativity for the inner joins. We show the number of plans found by each algorithm in Table 5. As described in Section 4.5, CD-C produces incorrect plans that do not respect reordering restrictions. The other algorithms are correct, but not complete, with CD-E being the most exhaustive, being able to find 96% of all possible plans, compared to CD-A, which finds 81% of all possible plans, meaning that the number of missed plans is reduced by a factor of over 4. The ranking of the other approaches is similar to the non-decomposable case in Table 4.

5.5 Missing Plans with Decomposable Predicates

In this section, we want to discuss when CD-E is not able to find all plans.

5.5.1 Bottom-Up Iteration. The bottom-up iteration of CD-E is not always optimal, as a hyperedge introduced later on can make an earlier hyperedge redundant. A different order of the iterations, especially one based on the structure of the query graph, can lead to a significant reduction in missed plans. Nonetheless, the complexity of the algorithm increases significantly, thus we leave such considerations for future work. An example is the following query:

$$((R_0 \bowtie_{01} R_1) \rtimes_{02} R_2) \bowtie_{13,23} R_3$$

Do we need to prevent $\rtimes_{02}$ from being moved to the right of $\bowtie_{01}$? If $\bowtie_{01}$ is removed, based on the connectedness in the initial query graph, R_1 and R_2 are connected due to the inner joins on the top. Nonetheless, after we determine all reordering restrictions, we recognize that the inner joins $\bowtie_{23}$ must succeed the left outer join $\rtimes_{02}$. Thus, we do not need to explicitly prevent $\rtimes_{02}$ from being moved to the right of $\bowtie_{01}$. However, in the bottom-up iteration, we do not recognize this redundancy.

5.5.2 Imprecise Connectedness. We use connectedness as an indicator for whether an operator $\circ^b$ can be moved to the left or right of an operator $\circ^a$. Nonetheless, this is just an approximation. Consider

the following query:

$$((R_0 \bowtie_{01} R_1) \bowtie_{02,12} R_2) \mathbb{K}_{03} R_3$$

If the $\mathbb{K}_{03}$ is assigned the hyperedge $(\{R_0, R_1, R_2\}, \{R_3\})$, then, it is guaranteed that the all inner joins must preceed the full outer join. Nonetheless, if we remove $\bowtie_{01}$, all relations remain connected, meaning that the connectedness check cannot detect that $\bowtie_{01}$ must preceed $\mathbb{K}_{03}$.

6 Limitations and Future Work

In this section, we discuss the limitations of our approach and outline future work. While our approach, CD-E, is complete and efficient for enumerating join plans with non-decomposable predicates, it does not guarantee completeness for decomposable predicates. In Section 5, we quantify the completeness of CD-E with respect to the other approaches, and discuss the specific cases where CD-E fails to find valid plans with decomposable predicates. We believe it is promising to address the limitations of the bottom-up iteration order of CD-E to improve the percantage of plans it finds.

Another significant limitation of all the discussed approaches is that, none of them consider the possibility of rewriting predicates. Rewrites can be algebraic transformations $a - b = c \equiv a = b + c$. Rewrites can also exploit equivalences of attributes. Consider the following query:

$$(R_1 ' \mathbb{K}_{R_1.x=R_2.x} R_2) \bowtie_{R_1.x=R_3.x} R_3$$

In the result of the outer join, the attribute $R_1.x$ is not guaranteed to be equal to $R_2.x$. Nonetheless, we can use l-asscom to reorder the joins:

$$(R_1 \bowtie_{R_1.x=R_3.x} R_3)' \mathbb{K}_{R_1.x=R_2.x} R_2$$

Now, we can rewrite the predicate of the left outer join based on the equivalence the inner join provides:

$$(R_1 \bowtie_{R_1.x=R_3.x} R_3)' \mathbb{K}_{R_3.x=R_2.x} R_2$$

After this rewrite, we can apply assoc to reorder the joins:

$$R_1 \bowtie_{R_1.x=R_3.x} (R_3 ' \mathbb{K}_{R_3.x=R_2.x} R_2)$$

None of the discussed approaches are able to find this plan, as they do not consider the possibility of rewriting predicates. We see such rewrites as an avenue for future work.

7 Conclusion

In this paper, we presented new approaches to join plan enumeration based on representing the query as a hypergraph. Our algorithm CD-E is both complete and efficient. It demonstrates significant improvements over existing approaches. We also presented CD-D, which is even more efficient than CD-E and is close to being complete, i.e., it finds almost all possible plans.

Classical techniques such as CD-A can have unintuitive behavior, such as the search space being reduced with the introduction of additional operators. CD-C avoids such behaviors but significantly slows down join enumeration and does not support the decomposition of conjunctive predicates. Such surprising behaviors lead to difficult-to-diagnose issues for users. Thus, we believe all systems should adopt more robust approaches such as CD-E or CD-D to improve user experience and query performance.

References

[1] Richard Bellman. 1952. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences of the United States of America* 38, 8 (1952), 716–719. http://www.jstor.org/stable/88493

[2] Michael A. Bender and Martin Farach-Colton. 2000. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1776)*, Gaston H. Gonnet, Daniel Panario, and Alfredo Viola (Eds.). Springer, 88–94. doi:10.1007/10719839_9

[3] Gautam Bhargava, Piyush Goel, and Balakrishna R. Iyer. 1995. Hypergraph Based Reorderings of Outer Join Queries with Complex Predicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 304–315. doi:10.1145/223784.223847

[4] Altan Birler, Mihail Stoian, and Thomas Neumann. 2025. Optimizing Linearized Join Enumeration by Adapting to the Query Structure. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025), 21. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 03.-07, März 2025, Bamberg, Germany, Proceedings (LNI, Vol. P-361)*, Meike Klettke, Ralf Schenkel, Andreas Henrich, Daniela Nicklas, Maximilian E. Schüle, and Klaus Meyer-Wegener (Eds.). Gesellschaft für Informatik e.V., 193–216. doi:10.18420/BTW2025-09

[5] Philipp Fent, Altan Birler, and Thomas Neumann. 2023. Practical planning and execution of groupjoin and nested aggregates. *VLDB J.* 32, 6 (2023), 1165–1190. doi:10.1007/S00778-022-00765-X

[6] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. 1993. Query Optimization in the IBM DB2 Family. *IEEE Data Eng. Bull.* 16, 4 (1993), 4–18. http://sites.computer.org/debull/93DEC-CD.pdf

[7] John E. Hopcroft and Robert Endre Tarjan. 1973. Efficient Algorithms for Graph Manipulation [H] (Algorithm 447). *Commun. ACM* 16, 6 (1973), 372–378. doi:10.1145/362248.362272

[8] Toshihide Ibaraki and Tiko Kameda. 1984. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Trans. Database Syst.* 9, 3 (1984), 482–502. doi:10.1145/1270.1498

[9] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. doi:10.1007/S00778-017-0480-7

[10] Guido Moerkotte, Pit Fender, and Marius Eich. 2013. On the correct and complete enumeration of the core search space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 493–504. doi:10.1145/2463676.2465314

[11] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 930–941. http://dl.acm.org/citation.cfm?id=1164207

[12] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 539–552. doi:10.1145/1376616.1376672

[13] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *Proc. VLDB Endow.* 4, 11 (2011), 843–851. http://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf

[14] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 677–692. doi:10.1145/3183713.3183733

[15] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. 1997. The Complexity of Transformation-Based Join Enumeration. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 306–315.

[16] Jun Rao, Bruce G. Lindsay, Guy M. Lohman, Hamid Pirahesh, and David E. Simmen. 2001. Using EELs, a Practical Approach to Outerjoin and Antijoin Reordering. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 585–594. doi:10.1109/ICDE.2001.914873

[17] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. doi:10.1137/0201010

[18] Robert Endre Tarjan. 1974. A Note on Finding the Bridges of a Graph. *Inf. Process. Lett.* 2, 6 (1974), 160–161. doi:10.1016/0020-0190(74)90003-9

[19] Mikkel Thorup. 1997. Decremental Dynamic Connectivity. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*, Michael E. Saks (Ed.). ACM/SIAM, 305–313.

http://dl.acm.org/citation.cfm?id=314161.314313

[20] TaiNing Wang and Chee-Yong Chan. 2018. Improving Join Reorderability with Compensation Operators. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 693–708. doi:10.1145/3183713.3183731

[21] TaiNing Wang, Yunpeng Niu, and Chee-Yong Chan. 2023. Complete Join Reordering for Null-Intolerant Joins. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 1734–1746. doi:10.1109/ICDE55515.2023.00136

[22] Jeffery R. Westbrook and Robert Endre Tarjan. 1992. Maintaining Bridge-Connected and Biconnected Components On-Line. *Algorithmica* 7, 5&6 (1992), 433–464. doi:10.1007/BF01758773