On the Correct and Complete Enumeration of the Core Search Space

Guido Moerkotte
University of Mannheim
Mannheim, Germany
moerkotte@informatik.uni-mannheim.de

Pit Fender
University of Mannheim
Mannheim, Germany
pfender@informatik.uni-mannheim.de

Marius Eich
University of Mannheim
Mannheim, Germany
meich@informatik.uni-mannheim.de

ABSTRACT

Reordering more than traditional joins (e.g. outerjoins, antijoins) requires some care, since not all reorderings are valid. To prevent invalid plans, two approaches have been described in the literature. We show that both approaches still produce invalid plans.

We present three conflict detectors. All of them are (1) correct, i.e., prevent invalid plans, (2) easier to understand and implement than the previous (buggy) approaches, (3) more flexible in the sense that the restriction that all predicates must reject nulls is no longer required, and (4) extensible in the sense that it is easy to add new operators. Further, the last of our three approaches is complete, i.e., it allows for the generation of all valid plans within the core search space.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: query processing, relational databases

Keywords

query optimization, join ordering, non-inner joins

1. INTRODUCTION

For a DBMS that provides support for SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows its translation into many equivalent plans. The process of choosing a low cost plan from the alternatives is known as query optimization or, more specifically, plan generation. Essential for the costs of a plan is its ordering of join operations, since the runtime of plans with different join orders can vary by several orders of magnitude.

When designing a plan generator, there are two approaches suitable to find an optimal join order: bottom-up join enumeration via dynamic programming (DP) and top-down join enumeration through memoization. Both approaches face the same challenge: the considered plans must be *valid*, i.e., produce the correct result. This is simple if only joins are considered, since they are commutative and associative. Thus, every plan is a valid plan.

If more operators like left outerjoins, full outerjoins, antijoins, semijoins, and groupjoins are considered then no longer are all plans valid. In fact, in the literature we find only two ways of preventing invalid plans in a DP-based plan generator. The first approach (NEL/EEL) is by Rao, Lindsay, Lohman, Pirahesh, and Simmen [20, 21]. Their conflict detector allows for joins, left outerjoins and antijoins. The second approach (SES/TES) is by Moerkotte and Neumann [15]. As we will show in Sections 7 and 8, both approaches generate INVALID PLANS. This leaves the implementer of a plan generator with zero (correct) choices for a DP-based plan generator (cf. Sec. 8).

We found this situation unbearable and decided to do some research on it. Here, we present our results. The highlight will be the conflict detector CD-C, which is

1. correct,
2. complete,
3. easy to understand and implement,
4. flexible, and
5. extensible.

Correct means that only valid plans are generated. Complete means that all valid plans in the core search space (defined in Sec. 3) are generated. As we will see, this is not easily achieved. Obviously, easy to understand and implement is a nice feature. CD-C is flexible in two respects. First, NEL/EEL and SES/TES both require that all join predicates reject nulls. In our approach, we eliminate this restriction. Thus, within a query some predicates may reject nulls, while others do not. This is important, since SQL allows predicates which are not null rejecting (e.g., IS NOT DISTINCT FROM). Second, we allow (as did NEL/EEL and SES/TES) for *complex join predicates* to reference more than two relations. Extensibility allows to extend the set of binary operators considered by a conflict detector. We achieve extensibility by a table-driven approach: several tables encode the properties of the operators, and CD-C simply explores these tables to detect conflicts and prevent invalid plans.

The rest of the paper is organized as follows. Sec. 2 defines some preliminaries. Sec. 3 defines the core search space. In order to do so, the essential properties of binary operators are defined. Sec. 4 clearly states the goal of our paper and uses the well-known algorithm DPsub to illustrate how a conflict detector is integrated into DP-based plan generators.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

Sec. 5 presents three conflict detectors. Each of them is correct, and the last one is complete. Sec. 7 contains the experimental results. Sec. 8 discusses related work and presents invalid plans generated by NEL/EEL and SES/TES. Sec. 9 concludes the paper.

2. PRELIMINARIES

This section contains basic definitions.

LOP denotes the set of logical binary operators we allow for in our plans: join ($\bowtie$), full outerjoin ($⟗$), left outerjoin ($⟕$), left antijoin ($\triangleright$), left semijoin ($\ltimes$), and groupjoin ($\bowtie\!\!\urcorner$) [16]. In Sec. 6, we also consider cross products ($\times$).

Null Rejecting for predicates is defined as follows [10]:

Definition 1. *A predicate is null rejecting for a set of attributes A if it evaluates to* FALSE *or* UNKNOWN *on every tuple in which all attributes in A are* NULL.

For examples, we refer to the introduction and [10]. In the literature, some synonyms for null rejecting are used: null intolerant, strong, and strict.

Free Attributes and Tables $\mathcal{F}(\cdot), \mathcal{F}_{\mathbf{T}}(\cdot)$. As usual, we denote by $\mathcal{A}(e)$ the set of attributes/variables provided by some expression e and by $\mathcal{F}(e)$ the set of free attributes/variables in some expression e. For example, if $p \equiv R.a + S.b = S.c + T.d$, then $\mathcal{F}(p) = \{R.a, S.b, S.c, T.d\}$.

Set of tables ($\mathcal{T}$), and subtree operators (STO). For a set of attributes A, $\mathcal{T}(A)$ denotes the set of tables to which these attributes belong. We abbreviate $\mathcal{T}(\mathcal{F}(e))$ by $\mathcal{F}_{\mathrm{T}}(e)$. For p we have $\mathcal{T}(\mathcal{F}(e)) = \{R, S, T\}$. Let $\circ$ be an operator in the initial operator tree. We denote by left($\circ$) (right($\circ$)) its left (right) child. STO($\circ$) denotes the operators contained in the operator subtree rooted at $\circ$. $\mathcal{T}(\circ)$ denotes the set of tables contained in the subtree rooted at $\circ$.

NEL/SES model the producer/consumer constraints. [15] introduced the notion of the *syntactic eligibility sets* (SES for short). The SES are attached to operators and contain the set of tables that must be present before the operator can be applied. The producer/consumer constraints for a plan of the form plan(S_1) $\circ$ plan(S_2) are satisfied if SES($\circ$) $\subseteq S_1 \cup S_2$ holds. SES is also called NEL [21]. For non-dependent operators, their SES is equal to the set of attributes referenced in their predicate. For p as above, we have SES($\circ_p$) $= \{R, S, T\}$.

Degenerate Predicates contained in binary operators are those that do not reference tables from both of their inputs:

Definition 2. *Let p be a predicate associated with a binary operator $\circ$ and $\mathcal{F}_T(p)$ the tables referenced by p. Then, p is called* degenerate *if $\mathcal{T}(left(\circ)) \cap \mathcal{F}_T(p) = \emptyset \vee \mathcal{T}(right(\circ)) \cap \mathcal{F}_T(p) = \emptyset$ holds.*

For example, in $\bowtie_{true}$ the predicate *true* is degenerate. Further, the expression is equivalent to a cross product. Since degenerate predicates are troublesome, we assume until Sec. 6 that no degenerate predicates (and, hence, no cross products) occur. In Sec. 6, we relax this assumption. Further, while presenting Calc$_{\texttt{SES}}$ and CD-C, we will already take some care of degenerate predicates and cross products.

3. CORE SEARCH SPACE

This section defines the core search space. It is defined by a set of transformation rules exploring all valid alternatives to a given initial plan. Sec. 3.1 introduces these transformation rules and Sec. 3.2 defines the core search space.

3.1 Reorderability

Traditional join ordering approaches just reorder joins and no other binary operators. Since the join is commutative and associative, all plans are valid and there is no danger of generating invalid plans. Real plan generators must reorder more than just plain joins (e.g., $\ltimes$, $\triangleright$, $⟕$, $⟗$, $\bowtie\!\!\urcorner$). In order to describe the reorderability properties of these operators, we need to carry over the notions of commutativity and associativity to pairs of operators. It is easy to see that some of these operators are commutative while others are not (see Table 1). If some binary operator $\circ$ is commutative, we denote this by comm($\circ$).

$\circ$	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	$⟕$	$⟗$	$\bowtie\!\!\urcorner$
comm($\circ$)	+	+	-	-	-	+	-

Table 1: The comm($\circ$)-property

Associativity is just a little more complex. We say that two not necessarily distinct operators $\circ^a$ and $\circ^b$ are *associative* if the following equivalence holds:

$$(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 \equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3). \quad (1)$$

Here, we use the following convention. If operators do not carry a predicate or other expressions, their subscripts are immaterial and can be ignored. If an operator has a predicate, then ij indicates that it references attributes (and, thus, relations) from at most e_i and e_j. Thus, (for $1 \leq i, j \leq 3$, $i \neq j$) this also indicates that $\mathcal{F}(e) \cap e_k = \emptyset$ for $1 \leq k \leq 3$ and $k \notin \{i, j\}$. This ensures that the equivalence is correctly typed on both sides of the equivalence sign. For example, the predicate of $\circ^a_{12}$ accesses tables from e_1 and e_2, but not e_3. Note that $\circ^a_{12}$ may carry a complex predicate referencing more than two tables from e_1 and e_2. We will see an example in the next subsection. If some $\circ^a_{123}$ referenced tables in all three expression e_i, the expression on the left-hand side of Eqv. 1 would be invalid and the right-hand side would be valid, but could not be transformed into the left-hand side. For the purpose of conflict detection, complex predicates accessing more than two relations are no challenge, they just enlarge the set of tables that must be present before the complex predicate can be evaluated. The real challenge with complex predicates is to efficiently enumerate the now more restricted search space (cf. Sec. 6.1).

If for two operators $\circ^a$ and $\circ^b$ Eqv. 1 holds, we denote this by assoc($\circ^a, \circ^b$). It is important to note that assoc is not symmetric. Thus, the order of the operators (i.e., ($\circ^a, \circ^b$) vs. ($\circ^b, \circ^a$)) is important. Therefore, we tie the order in assoc to the syntactic pattern of Eqv. 1. It has to be the same order as on the left-hand side of the equivalence. This means that the left association has to be on the left-hand side and, consequently, the right association on the right-hand side of the equivalence.

If comm($\circ^a$) and comm($\circ^b$) holds, then assoc($\circ^a, \circ^b$) implies assoc($\circ^b, \circ^a$) and vice versa, as can be seen from

$$\begin{aligned}
(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 &\equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3) & \mathrm{assoc}(\circ^a, \circ^b)\\
&\equiv (e_2 \circ^b_{23} e_3) \circ^a_{12} e_1 & \mathrm{comm}(\circ^a)\\
&\equiv (e_3 \circ^b_{23} e_2) \circ^a_{12} e_1 & \mathrm{comm}(\circ^b)\\
&\equiv e_3 \circ^b_{23} (e_2 \circ^a_{12} e_1) & \mathrm{assoc}(\circ^b, \circ^a)\\
&\equiv (e_2 \circ^a_{12} e_1) \circ^b_{23} e_3 & \mathrm{comm}(\circ^b)\\
&\equiv (e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 & \mathrm{comm}(\circ^a).
\end{aligned}$$

Table 2 summarizes the associativity properties. Be careful, since assoc is not symmetric, $\circ^a$ must be looked up within a row and $\circ^b$ within a column, *not* vice versa. Entries with a footnote in Table 2 denote that assoc$(\circ^a, \circ^b)$ only holds if the predicates reject nulls (see Def. 1). For more details, see the corresponding footnotes at the bottom of Table 2.

$\circ^a$	$\circ^b$						
	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	⟕	⟗	$\bar{\bowtie}$
$\times$	+	+	+	+	+	-	+
$\bowtie$	+	+	+	+	+	-	+
$\ltimes$	-	-	-	-	-	-	-
$\triangleright$	-	-	-	-	-	-	-
⟕	-	-	-	-	$+^1$	-	-
⟗	-	-	-	-	$+^1$	$+^2$	-
$\bar{\bowtie}$	-	-	-	-	-	-	-

[1] if p_{23} rejects nulls on $\mathcal{A}(e_2)$ (Eqv. 1)
[2] if p_{12} and p_{23} reject nulls on $\mathcal{A}(e_2)$ (Eqv. 1)

Table 2: The assoc$(\circ^a, \circ^b)$-property

Are we done? No! Consider the following well-known equivalence for the semijoin:

$$(e_1 \ltimes_{12} e_2) \ltimes_{13} e_3 \equiv (e_1 \ltimes_{13} e_3) \ltimes_{12} e_2.$$

It is easy to see that we cannot derive the plan on the right-hand side from the plan on the left-hand side using associativity and commutativity of $\ltimes$: neither holds. Thus, we need something new.

We define the *left asscom property* (l-asscom for short) as follows:

$$(e_1 \circ^a_{12} e_2) \circ^b_{13} e_3 \equiv (e_1 \circ^b_{13} e_3) \circ^a_{12} e_2. \tag{2}$$

We denote by l-asscom$(\circ^a, \circ^b)$ the fact that Eqv. 2 holds for $\circ^a$ and $\circ^b$.

Analogously, we can define a *right asscom property* (r-asscom):

$$e_1 \circ^a_{13} (e_2 \circ^b_{23} e_3) \equiv e_2 \circ^b_{23} (e_1 \circ^a_{13} e_3). \tag{3}$$

First, note that l-asscom and r-asscom are symmetric properties, i.e.,

$$\begin{aligned} \text{l-asscom}(\circ^a, \circ^b) &\leftrightarrow \text{l-asscom}(\circ^b, \circ^a), \\ \text{r-asscom}(\circ^a, \circ^b) &\leftrightarrow \text{r-asscom}(\circ^b, \circ^a). \end{aligned}$$

The following reasoning

$$\begin{aligned} (e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 &\equiv (e_2 \circ^a_{12} e_1) \circ^b_{23} e_3 && \text{if comm}(\circ^a_{12}) \\ &\equiv (e_2 \circ^b_{23} e_3) \circ^a_{12} e_1 && \text{if l-asscom}(\circ^a_{12}, \circ^b_{23}) \\ &\equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3) && \text{if comm}(\circ^a_{12}) \\ &\equiv (e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 && \text{if assoc}(\circ^a_{12}, \circ^b_{23}) \end{aligned}$$

implies that

$$\begin{aligned} \text{comm}(\circ^a_{12}), \text{assoc}(\circ^a_{12}, \circ^b_{23}) &\rightarrow \text{l-asscom}(\circ^a_{12}, \circ^b_{23}), \\ \text{comm}(\circ^a_{12}), \text{l-asscom}(\circ^a_{12}, \circ^b_{23}) &\rightarrow \text{assoc}(\circ^a_{12}, \circ^b_{23}). \end{aligned}$$

Thus, the l-asscom property is implied by associativity and commutativity, which explains its name. Quite similarly, the implications

$$\begin{aligned} \text{comm}(\circ^b_{23}), \text{assoc}(\circ^a_{12}, \circ^b_{23}) &\rightarrow \text{r-asscom}(\circ^a_{12}, \circ^b_{23}), \\ \text{comm}(\circ^b_{23}), \text{r-asscom}(\circ^a_{12}, \circ^b_{23}) &\rightarrow \text{assoc}(\circ^a_{12}, \circ^b_{23}) \end{aligned}$$

can be deduced.

Table 3 summarizes the l-/r-asscom properties. Again, entries with a footnote require that the predicates reject nulls. We assume that calls to assoc and l/r-asscom take care of this.

$\circ$	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	⟕	⟗	$\bar{\bowtie}$
$\times$	+/+	+/+	+/-	+/-	+/-	-/-	+/-
$\bowtie$	+/+	+/+	+/-	+/-	+/-	-/-	+/-
$\ltimes$	+/-	+/-	+/-	+/-	+/-	-/-	+/-
$\triangleright$	+/-	+/-	+/-	+/-	+/-	-/-	+/-
⟕	+/-	+/-	+/-	+/-	+/-	$+^1$/-	+/-
⟗	-/-	-/-	-/-	-/-	$+^2$/-	$+^3$/$+^4$	-/-
$\bar{\bowtie}$	+/-	+/-	+/-	+/-	+/-	-/-	+/-

[1] if p_{12} rejects nulls on $\mathcal{A}(e_1)$ (Eqv. 2)
[2] if p_{13} rejects nulls on $\mathcal{A}(e_3)$ (Eqv. 2)
[3] if p_{12} and p_{13} rejects nulls on $\mathcal{A}(e_1)$ (Eqv. 2)
[4] if p_{13} and p_{23} reject nulls on $\mathcal{A}(e_3)$ (Eqv. 3)

Table 3: The l-/r-asscom$(\circ^a, \circ^b)$ property

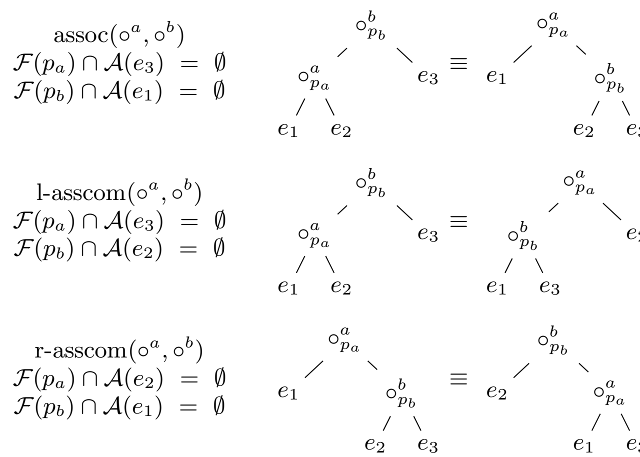

Figure 1: Transformation rules for assoc, l-asscom, and r-asscom

If an entry in one of the Tables 1 to 3 is marked with $-$ or its condition in the footnote is violated, we say that there is a *conflict* regarding this property. A conflict means that the application of the corresponding transformation rule results in an invalid plan.

3.2 Definition of the Core Search Space

Typically, for a given input query, the query optimizer constructs an initial operator tree. In a transformation-based plan generator, all valid plans are then generated by exhaustively applying transformations to the initial plan.

If commutativity, associativity, l-asscom, and/or r-asscom hold, this gives rise to the according transformations. Except for commutativity, these are shown in Fig. 1. All equivalences can be applied from left to right and from right to left. We define the *core search space* for a given initial plan to be the set of plans generated by exhaustively applying these four transformations to the initial plan.

Fig. 2 shows a larger operator tree. Let us consider several possibilities for the predicates of the top-most operator $\circ^b$. If $p_b \equiv R_0.a + R_1.a + R_2.a + R_3.a = R_4.a * R_5.a$, then no reordering is possible, since all tables are referenced. If $p_b \equiv R_2.a + R_3.a = R_4.a * R_5.a$, then applying associativity is possible from a syntactic point of view, since $\mathcal{F}(p_b) \cap \mathcal{T}(e_1)$ becomes in our example $\{R_2, R_3, R_4, R_5\} \cap \{R_0, R_1\} = \emptyset$. In fact, although the predicate is complex, it references only tables below $\circ^2$ and $\circ^3$, whose subtrees correspond to e_2 and e_3 in Fig. 1. Accordingly, we would write $\circ^b_{12}$. Clearly,

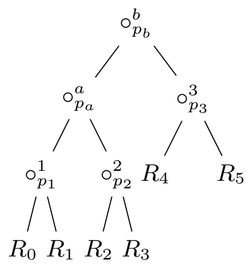

Figure 2: Example operator tree

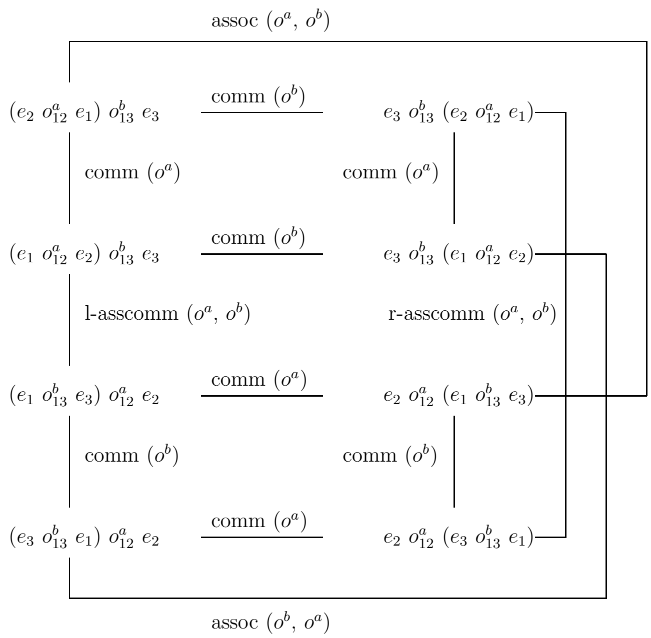

Figure 3: Core search space example

a binary predicate, e.g., $p_b \equiv R_0.a = R_5.a$, generates the largest search space and, thus, the highest opportunity for generating invalid plans and missing valid plans. This is the reason why we will restrict ourselves to binary predicates in Sec. 7.

Taking a look at the syntactic constraints shown in Fig. 1, we see that for non-degenerate predicates (see Def. 2) the following observation holds:

OBSERVATION 1. *The syntactic constraints for non-degenerate predicates imply that (1) either associativity or l-asscom can be applied for left nesting but not both, and (2) either associativity or r-asscom can be applied for right-nesting but not both.*

Thus, non-degenerate predicates simplify the handling of conflicts, since we have to take care of either associativity or l/r-asscom and never both at the same time.

Fig. 3 shows an example of the core seach space for the expression $(e_1 \circ^a_{12} e_2) \circ^b_{13} e_3$. We observe that any expression in the core search space can be reached by a sequence of at most two applications of commutativity, at most one application of associativity, l-asscom, or r-asscom, finally followed by at most two applications of commutativity. The total number of applications of commutativity can be restricted to 2. More specifically, one application of commutativity to each operator in the plan suffices.

DPSUBE

$\triangleright$ **Input:** a set of relations $R = \{R_0, \ldots, R_{n-1}\}$
a set of operators O with associated conflict descriptors
$\triangleright$ **Output:** an optimal bushy operator tree

```
1  for all R_i ∈ R
2      BestPlan({R_i}) ← R_i
3  for 1 ≤ i < 2^n − 1 ascending
4      S ← {R_j ∈ R | (⌊i/2^j⌋ mod 2) = 1}
5      if |S| = 1
6          continue
7      for all S_1 ⊂ S, S_1 ≠ ∅
8          S_2 ← S \ S_1
9          for all ∘ ∈ O
10             if APPLICABLE(∘, S_1, S_2)
11                 build and handle the plans
                     BestPlan(S_1) ∘ BestPlan(S_2)
12                 if comm(∘)
13                     build and handle the plans
                         BestPlan(S_2) ∘ BestPlan(S_1)
14 return BestPlan(R)
```

Figure 4: Pseudocode for DPSUBE

4. GOAL OF THE PAPER

This section discusses how the complete core search space can be explored by a plan generator.

Therefore, we extend the simple dynamic programming algorithm DPSUB [14] to one called DPSUBE. The resulting pseudo-code is shown in Fig. 4. As input, DPSUBE takes the set of n relations $R = \{R_0, \ldots, R_{n-1}\}$ and the set of operators O containing $n-1$ operators which DPSUBE has to apply in order to build a plan. First, it constructs a plan for single relations (Line 2). Then, it enumerates all subsets S of relations by decoding an integer, which is interpreted as a bitvector encoding a subset of the set of relations. For each set of relations S, DPSUBE then enumerates all subsets S_1 of S (Line 7) and their complements S_2 (Line 8). Both of them must be non-empty. For each pair (S_1, S_2), all operators $\circ$ in O are then tested for applicability via a call to APPLICABLE (Line 10). If the operator is applicable, the best plans P_1 for S_1 and P_2 for S_2 are recalled from the dynamic programming table (DP-table for short) via *BestPlan* and combined into the plan $P_1 \circ P_2$ for S (Line 11). The costs of this plan are then calculated, and if this plan is cheaper than the existing one, it is added to the DP-table. Since this piece of code is straightforward, we do not detail on it. Note that only if an operator is applicable, DPSUBE also considers commutativity. Thus, the plan $P_2 \circ P_1$ is built and handled if comm($\circ$) (Line 12) holds. The goal of the paper is to provide different implementations for APPLICABLE.

5. CONFLICT DETECTION

5.1 Outline

In order to open our approach for new algebraic operators, we use a table-driven approach. We use four tables which contain the properties of the algebraic operators. These contain the information of Tables 1, 2 and 3. (The latter includes two tables.) Extending our approach only requires to extend these tables!

We develop our final approach in three steps. In each step, we introduce one of our conflict detectors CD-A, CD-B, and

CD-C. For these conflict detectors, we present a complete bundle consisting of three components:

1. a representation for conflicts,
2. a conflict detection (CD) algorithm, which detects the conflicts in the initial operator tree and produces a conflict representation for each operator contained in it, and
3. the implementation of APPLICABLE, which uses the conflict representation for an operator and then determines whether the operator can be applied in a given context.

Each of the subsequently discussed bundles is correct, but only the last one is complete.

The main idea in the following (the same as in [15, 20, 21]) is to extend the producer/consumer constraints modeled through `SES` (`NEL`) by adding more tables to it in order to restrict the explored search space to valid plans only. This is possible, since `SES` is used to express the syntactic constraints: all referenced attributes/tables must be present before an expression can be evaluated. Therefore, if we add more tables, the explored search space becomes smaller.

Let us now define `SES`. First of all, `SES` contains the tables referenced by a predicate. If some operator like the groupjoin $⋈\!\!\Gamma$ [16] introduces new attributes, they are treated as if they belonged to a new table. This new table is present in the set of accessible tables after the groupjoin has been applied. Let R be a table and $\circ_p$ any of our binary operators except a groupjoin. We give the pseudo code for the `SES` calculation:

$\textsc{Calc}_{\texttt{SES}}(\circ_p)$

$\triangleright$ **Input:** binary operator $\circ \in$ `LOP` carrying predicate p
if $\circ_p \in \{\bowtie, ⟗, ⟖, \triangleright, \ltimes\}$ $\triangleright$ and later: cross product $\times$
  return $\bigcup_{R \in \mathcal{F}_{\mathrm{T}}(p)} \{R\} \cap \mathcal{T}(\circ_p)$
elseif $\circ_{p;a_1:e_1,\ldots,a_n:e_n} \in \{⋈\!\!\Gamma\}$
  return $\bigcup_{R \in \mathcal{F}_{\mathrm{T}}(p) \cup \mathcal{F}_{\mathrm{T}}(e_i)} \{R\} \cap \mathcal{T}(\circ_p)$
else $\triangleright$ cross product $\times$
  return $\emptyset$

In case of non-degenerate predicates and in the absence of dependent operators (cf. Sec. 6.3) and table functions, $\textsc{Calc}_{\texttt{SES}}(\circ_p) = \mathcal{F}_{\mathrm{T}}(p)$. Note that $\textsc{Calc}_{\texttt{SES}}$ handles cross products, which we will not need until Sec. 6.2.

All conflict representations have a component called *total eligibility set* (`TES` for short) which contains a set of tables. We always initialize `TES` with `SES` as calculated above. Further, we assume that our conflict representation has two accessors `L-TES` and `R-TES` returning

$$\begin{aligned} \texttt{L-TES}(\circ) &:= \texttt{TES}(\circ) \cap \mathcal{T}(\mathrm{left}(\circ)) \text{ and} \\ \texttt{R-TES}(\circ) &:= \texttt{TES}(\circ) \cap \mathcal{T}(\mathrm{right}(\circ)). \end{aligned}$$

This distinction is necessary since we want to consider commutativity explicitly, and in those cases where commutativity does not hold, we want to prevent operators which occurred on the left-hand side of an operator from moving to its right-hand side or vice versa.

All our implementations of APPLICABLE conjunctively include the test $\texttt{L-TES} \subseteq S_1 \wedge \texttt{R-TES} \subseteq S_2$.

5.2 Approach CD-A

Let us first consider a simple operator tree with only two operators. Take a look at the upper half of Fig. 5. There, the application of associativity and l-asscom to some plan is illustrated. In case that associativity does not hold, we add $\mathcal{T}(e_1)$ to $\texttt{TES}(\circ^b)$. This prevents the plan on the right-hand side of the arrow marked with assoc. It does not, however, prevent the plan on the right-hand side of the arrow marked with l-asscom. Similarly, adding $\mathcal{T}(e_2)$ to $\texttt{TES}(\circ^b)$ does prevent the plan resulting from l-asscom but not the plan resulting from applying associativity. The lower part of Fig. 5 shows the actions needed if an operator is nested in the right argument. Again, we can precisely prevent the invalid plans.

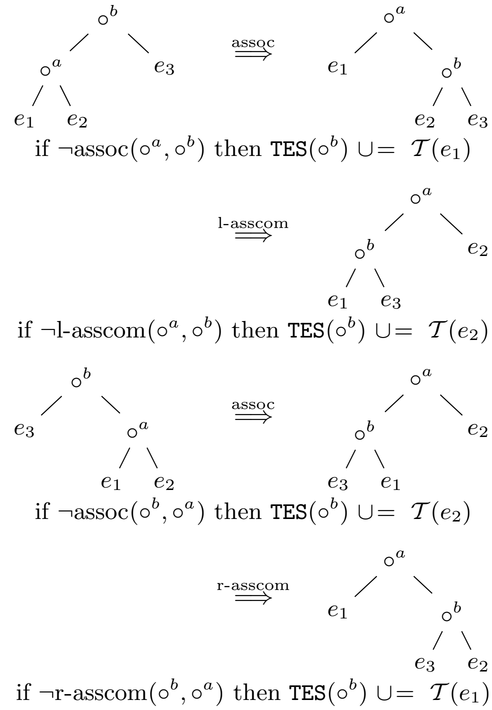

Figure 5: Calculating `TES` for simple operator trees

There is only one more problem we have to solve. It occurs if a conflicting operator $\circ_a$ is not a direct child of $\circ_b$, but instead a descendant situated deeper in the operator tree. This is possible since in general, the e_i are trees themselves. Some reordering could possibly move a conflicting operator $\circ_a$ up to the top of an argument subtree.

Thus, we have to calculate the total eligibility sets bottom-up by applying CD-A to every operator $\circ^b$ in the operator tree. The pseudo code of CD-A is:

CD-A($\circ^b$)

$\triangleright$ **Input:** operator $\circ^b$
$\texttt{TES}(\circ^b) \leftarrow \textsc{Calc}_{SES}(\circ^b)$
for $\forall\ \circ^a \in \texttt{STO}(\mathrm{left}(\circ^b))$
  if $\neg\mathrm{assoc}(\circ^a, \circ^b)$
    $\texttt{TES}(\circ^b) \leftarrow \texttt{TES}(\circ^b) \cup \mathcal{T}(\mathrm{left}(\circ^a))$
  if $\neg\text{l-asscom}(\circ^a, \circ^b)$
    $\texttt{TES}(\circ^b) \leftarrow \texttt{TES}(\circ^b) \cup \mathcal{T}(\mathrm{right}(\circ^a))$
for $\forall\ \circ^a \in \texttt{STO}(\mathrm{right}(\circ^b))$
  if $\neg\mathrm{assoc}(\circ^b, \circ^a)$
    $\texttt{TES}(\circ^b) \leftarrow \texttt{TES}(\circ^b) \cup \mathcal{T}(\mathrm{right}(\circ^a))$
  if $\neg\text{r-asscom}(\circ^b, \circ^a)$
    $\texttt{TES}(\circ^b) \leftarrow \texttt{TES}(\circ^b) \cup \mathcal{T}(\mathrm{left}(\circ^a))$

If we do not have degenerate predicates and cross products among the operators in the initial operator tree, we can safely use TES instead of $\mathcal{T}$.

The conflict representation comprises the TES for every operator. The pseudo code for APPLICABLE is:

APPLICABLE$_A(\circ, S_1, S_2)$

$\triangleright$ **Input:** binary operator $\circ$, set of tables S_1, S_2

return L-TES$(\circ) \subseteq S_1 \wedge$ R-TES$(\circ) \subseteq S_2$

Let us now see why APPLICABLE$_A$ is correct. We have to show that it prevents the generation of bad plans. Take the $\neg$assoc case with nesting on the left. Let the original operator tree contain $(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3$. Define the set of tables $R_2 := \mathcal{F}_{\mathrm{T}}(\circ^b_{23}) \cap \mathcal{T}(\mathrm{left}(\circ^b_{23}))$ and $R_3 := \mathcal{F}_{\mathrm{T}}(\circ^b_{23}) \cap \mathcal{T}(\mathrm{right}(\circ^b_{23}))$. Then $\mathtt{SES}(\circ^b_{23}) = R_2 \cup R_3$. Further, since $\neg\mathrm{assoc}(\circ^a_{12}, \circ^b_{23})$, we have

$$\mathtt{TES}(\circ^b_{23}) \supseteq \mathtt{SES}(\circ^b_{23}) \cup \mathcal{T}(e_1).$$

Note that we used $\supseteq$ and not equality, since due to other conflicts, $\mathtt{TES}(\circ^b)$ could be larger. Next, we observe that

$$\begin{aligned}
\mathtt{L\text{-}TES}(\circ^b_{23}) &\supseteq (\mathtt{SES}(\circ^b_{23}) \cup \mathcal{T}(e_1)) \cap \mathcal{T}(\mathrm{left}(\circ^b_{23}))\\
&\supseteq (\mathtt{SES}(\circ^b_{23}) \cap \mathcal{T}(\mathrm{left}(\circ^b_{23}))) \cup\\
&\quad\ (\mathcal{T}(e_1) \cap \mathcal{T}(\mathrm{left}(\circ^b_{23})))\\
&\supseteq ((R_2 \cup R_3) \cap \mathcal{T}(\mathrm{left}(\circ^b_{23}))) \cup (\mathcal{T}(e_1))\\
&\supseteq R_2 \cup \mathcal{T}(e_1)
\end{aligned}$$

and

$$\begin{aligned}
\mathtt{R\text{-}TES}(\circ^b_{23}) &\supseteq (\mathtt{SES}(\circ^b_{23}) \cup \mathcal{T}(e_1)) \cap \mathcal{T}(\mathrm{right}(\circ^b_{23}))\\
&\supseteq \mathtt{SES}(\circ^b_{23}) \cap \mathcal{T}(\mathrm{right}(\circ^b_{23}))\\
&\supseteq R_3.
\end{aligned}$$

Let S_1, S_2 be a pair of two arbitrary subsets of tables generated by DPSUBE. Then, the call APPLICABLE$(\circ^b, S_1, S_2)$ checks

$$\begin{aligned}
\mathtt{L\text{-}TES}(\circ^b_{23}) &\subseteq S_1 \text{ and}\\
\mathtt{R\text{-}TES}(\circ^b_{23}) &\subseteq S_2,
\end{aligned}$$

and fails if $S_1 \not\supseteq \mathcal{T}(e_1)$. Thus, neither $e_2 \circ^b_{23} e_3$ nor $e_3 \circ^b_{23} e_2$ will be generated and, hence, $e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3)$ will not be generated either. Similarly, if $\neg$l-asscom$(\circ^a, \circ^b)$, L-TES$(\circ^b)$ will contain $\mathcal{T}(e_2)$, and the test prevents the generation of $e_1 \circ^b e_3$. The remaining two cases can be checked analogously.

From this discussion, it follows that DPSUBE generates only valid plans. However, it does not generate all valid plans. It is thus incomplete, as we can see from the example shown in Fig. 6. Since $\neg\mathrm{assoc}(\ltimes_{0,1}, ⟗_{2,3})$, $\mathtt{TES}(⟗_{2,3})$ contains R_0 (line 4 of CD-A$(⟗_{2,3})$) . Thus, neither of the valid plans Plan 1 nor Plan 2 nor any of those derived from applying join commutativity to them will be generated.

5.3 Approach CD-B

In order to avoid this problem, we introduce the more flexible mechanism of *conflict rules*. A *conflict rule* (CR) is simply a pair of table sets denoted by $T_1 \rightarrow T_2$. With every operator node $\circ$ in the operator tree, we associate a set of conflict rules. Thus, our conflict representation now associates a TES and a set of conflict rules with every operator.

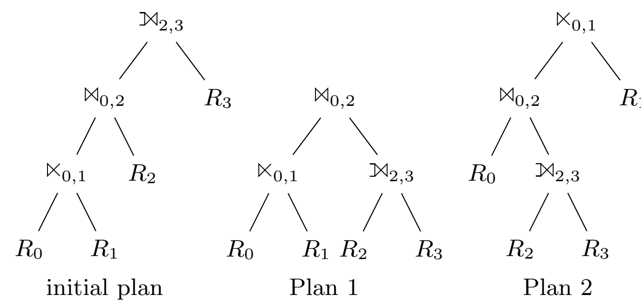

Figure 6: Example for incompleteness of CD-A

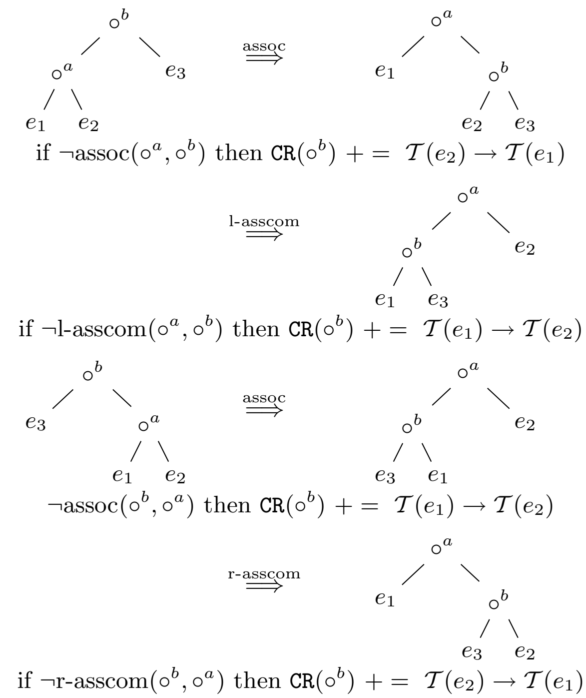

Figure 7: Calculating conflict rules for simple operator trees

Before we introduce their construction, let us illustrate their role in APPLICABLE$(\circ, S_1, S_2)$. A conflict rule $T_1 \rightarrow T_2$ is *obeyed* for S_1 and S_2 if with $S = S_1 \cup S_2$ the following condition holds:

$$T_1 \cap S \neq \emptyset \implies T_2 \subseteq S.$$

Thus, if T_1 contains a single table from S, S must contain all tables in T_2. Keeping this in mind, it is easy to see that the invalid plans are indeed prevented by the rules shown in Fig. 7 if they are obeyed. As we will see, the TES is restricted to SES in CD-B. Thus, the conflict rules allow for more flexibility: whereas the TES containment test is unconditioned, conflict rules represent a conditioned containment test.

The pseudo code for the new conflict detection is given in Fig. 8 with CD-B. As before, we apply CD-B bottom-up to every operator $\circ^b$ in the tree.

With the conflict rules, we need a new test for applicability. Now, the test given in Fig. 9 with APPLICABLE$_{B/C}(\circ, S_1, S_2)$ checks for two conditions:

1. L-TES $\subseteq S_1 \wedge$ R-TES $\subseteq S_2$ must hold (line 1), and
2. all rules in the rule set of $\circ$ must be obeyed (Lines 2-6).

CD-B($\circ^b$)

$\triangleright$ **Input:** operator $\circ^b$

$\mathtt{TES}(\circ^b) \leftarrow \text{CALC}_{SES}(\circ^b)$
for $\forall \circ^a \in \mathtt{STO}(\text{left}(\circ^b))$
  if $\neg\text{assoc}(\circ^a, \circ^b)$
    $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{right}(\circ^a)) \to \mathcal{T}(\text{left}(\circ^a))$
  if $\neg\text{l-asscom}(\circ^a, \circ^b)$
    $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{left}(\circ^a)) \to \mathcal{T}(\text{right}(\circ^a))$
for $\forall \circ^a \in \mathtt{STO}(\text{right}(\circ^b))$
  if $\neg\text{assoc}(\circ^b, \circ^a)$
    $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{left}(\circ^a)) \to \mathcal{T}(\text{right}(\circ^a))$
  if $\neg\text{r-asscom}(\circ^b, \circ^a)$
    $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{right}(\circ^a)) \to \mathcal{T}(\text{left}(\circ^a))$

Figure 8: Pseudocode for CD-B

Note that now all plans in Fig. 6 can be generated.

APPLICABLE$_{B/C}(\circ, S_1, S_2)$

$\triangleright$ **Input:** binary operator $\circ$, set of tables S_1, S_2

1 **if** $\mathtt{L\text{-}TES}(\circ) \subseteq S_1 \wedge \mathtt{R\text{-}TES}(\circ) \subseteq S_2$
2   **for all** $(T_1 \to T_2) \in \mathtt{CR}(\circ)$
3   **if** $T_1 \cap (S_1 \cup S_2) \neq \emptyset$
4   **if** $T_2 \not\subseteq (S_1 \cup S_2)$
5   **return** FALSE
6   **return** TRUE
7 **else**
8   **return** FALSE

Figure 9: Pseudocode for APPLICABLE$_{B/C}$

Again, this implementation of APPLICABLE is correct but not complete, as the example in Fig. 10 shows. Since assoc($\Join_{0,1}, \ltimes_{1,3}$), assoc($\Join_{1,2}, \ltimes_{1,3}$) and l-asscom($\Join_{1,2}, \ltimes_{1,3}$), the only conflict occurs due to $\neg$r-asscom($\Join_{0,1}, \ltimes_{1,3}$). Thus,

$$\mathcal{T}(\{R_3\}) \to \mathcal{T}(\{R_1, R_2\}) \ \in\ \mathtt{CR}(\Join_{0,1}).$$

The latter rule prevents the plan on the right-hand side of Fig. 10. Note that this is overly careful, since $R_2 \notin \mathcal{F}_{\mathrm{T}}(\ltimes_{1,3})$. In fact, r-asscom would never be applied in this example, since $\Join_{0,1}$ accesses table R_1, and applying r-asscom would thus destroy the producer/consumer relationship ($\mathcal{F}_{\mathrm{T}}(\Join_{0,1}) \cap \{R_1, R_2\} \neq \emptyset$) already checked by $\mathtt{SES}(\Join_{0,1})$.

5.4 Approach CD-C

The approach CD-C differs from CD-B only by the calculation of the conflict rules. The conflict representation and the procedure APPLICABLE remain the same. The idea is to learn from the above example and include only those tables under operator $\circ^a$ which occur in the predicate. However, we have to be careful to include special cases for degenerate predicates and cross products. The pseudo code is given with CD-C in Fig. 11. Let us revisit the example of Section 5.3. Since the only conflict occurs due to $\neg$r-asscom($\Join_{0,1}, \ltimes_{1,3}$), the rule set $\mathtt{CR}(\Join_{0,1})$ contains (Line 21 of CD-C) $\mathcal{T}(\{R_3\}) \to \mathcal{T}(\{R_1\}) \in \mathtt{CR}(\Join_{0,1})$. As a consequence, the plan on the right of Fig. 10 will not be prevented anymore.

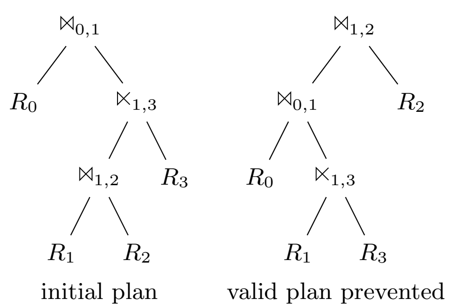

Figure 10: Example for incompleteness of CD-B

CD-C($\circ^b$)

$\triangleright$ **Input:** operator $\circ^b$

1 $\mathtt{TES}(\circ^b) \leftarrow \text{CALC}_{SES}(\circ^b)$
2 **for** $\forall \circ^a \in \mathtt{STO}(\text{left}(\circ^b))$
3   **if** $\neg\text{assoc}(\circ^a, \circ^b)$
4   **if** $\mathcal{T}(\text{left}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a) \neq \emptyset$
5       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{right}(\circ^a)) \to \mathcal{T}(\text{left}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a)$
6   **else**
7       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{right}(\circ^a)) \to \mathcal{T}(\text{left}(\circ^a))$
8   **if** $\neg\text{l-asscom}(\circ^a, \circ^b)$
9   **if** $\mathcal{T}(\text{right}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a) \neq \emptyset$
10       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{left}(\circ^a)) \to \mathcal{T}(\text{right}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a)$
11   **else**
12       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{left}(\circ^a)) \to \mathcal{T}(\text{right}(\circ^a))$
13 **for** $\forall \circ^a \in \mathtt{STO}(\text{right}(\circ^b))$
14   **if** $\neg\text{assoc}(\circ^b, \circ^a)$
15   **if** $\mathcal{T}(\text{right}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a) \neq \emptyset$
16       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{left}(\circ^a)) \to \mathcal{T}(\text{right}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a)$
17   **else**
18       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{left}(\circ^a)) \to \mathcal{T}(\text{right}(\circ^a))$
19   **if** $\neg\text{r-asscom}(\circ^b, \circ^a)$
20   **if** $\mathcal{T}(\text{left}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a) \neq \emptyset$
21       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{right}(\circ^a)) \to \mathcal{T}(\text{left}(\circ^a)) \cap \mathcal{F}_{\mathrm{T}}(\circ^a)$
22   **else**
23       $\mathtt{CR}(\circ^b) \mathrel{+}= \mathcal{T}(\text{right}(\circ^a)) \to \mathcal{T}(\text{left}(\circ^a))$

Figure 11: Pseudocode for CD-C

Correctness of CD-C. We show that APPLICABLE$_{B/C}$ for the $\neg$ assoc case with nesting on the left is correct. The remaining cases can be proven similarly. Let the original operator tree contain $(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3$. Since $\neg$assoc($\circ^a, \circ^b$) (line 3 of CD-C($\circ^b$)), one of the following (line 5 or line 7) holds:

$$\begin{aligned} \mathtt{CR}(\circ^b) \quad &\mathrel{+}= \quad \mathcal{T}(e_2) \to \mathcal{T}(e_1) \text{ or} \\ &\mathrel{+}= \quad \mathcal{T}(e_2) \to \mathcal{T}(e_1') \text{ with } e_1' \subset e_1 \wedge e_1' \neq \emptyset. \end{aligned}$$

The second case subsumes the first case. Thus, it suffices to show the second case. To construct $e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3)$ (right hand side of Eqv. 1), $(e_2 \circ^b_{23} e_3)$ must be constructed first. We show that APPLICABLE$_{B/C}(\circ^b, S_1, S_2)$ with either (A) $\mathcal{T}(e_2) \subseteq S_1 \wedge \mathcal{T}(e_3) \subseteq S_2$ or (B) $\mathcal{T}(e_3) \subseteq S_1 \wedge \mathcal{T}(e_2) \subseteq S_2$ returns FALSE. If the test in line 1 fails, FALSE is returned and we are done. Otherwise, $\mathtt{L\text{-}TES}(\circ) \subseteq S_1$ holds. Note that since we are trying to construct $(e_2 \circ^b_{23} e_3)$, $\mathcal{T}(e_1) \cap (S_1 \cup S_2) = \emptyset$ must hold. On the other hand, the conflict rule $T_1 \to T_2$ with $T_1 = \mathcal{T}(e_2)$ and $T_2 \supseteq \mathcal{T}(e_1')$ is contained in $\mathtt{CR}(\circ^b)$. Thus, for this rule $T_1 \to T_2$: $T_1 \cap (S_1 \cup S_2) \neq \emptyset$ and $T_2 \not\subseteq (S_1 \cup S_2)$ hold in both cases (A) and (B). This cleary shows that FALSE is returned. Hence, CD-C and, consequently, CD-B are correct.

5.5 Rule Simplification

It is well-known that larger TES have a positive impact on the runtime of plan generators like DPHYP [15] or TDMCCHYP [7] (see also Sec. 6). Further, reducing the number of rules slightly decreases plan generation time. Thus, applying laws like

$$\begin{aligned} R_1 \to R_2, R_1 \to R_3 &\equiv R_1 \to R_2 \cup R_3 \\ R_1 \to R_2, R_3 \to R_2 &\equiv R_1 \cup R_3 \to R_2 \end{aligned}$$

can be used to rearrange the rule set for efficient evaluation. However, we are much more interested in eliminating rules altogether by adding their right-hand side to the TES. For some operator $\circ$, consider a conflict rule $R_1 \to R_2$. If $R_1 \cap \mathtt{TES}(\circ) \neq \emptyset$, we can add R_2 to TES due to the existential quantifier on the left-hand side of a rule in the definition of *obey*. Further, if $R_2 \subseteq \mathtt{TES}(\circ)$, we can safely eliminate the rule. Applying these rearrangements is often possible, since both $\mathcal{T}(\text{left}(\circ^a)) \cap \mathcal{F}_\text{T}(\circ)$ and $\mathcal{T}(\text{right}(\circ^a)) \cap \mathcal{F}_\text{T}(\circ)$ will be non-empty.

6. MINOR ISSUES

6.1 Larger TES, Faster Plan Generation

Typically, a query graph is used to model the producer/consumer constraints defined in a given input query, and, thus, to represent the possible search space. For queries with complex predicates, i.e., predicates that reference more than two relations, the query graph is a hypergraph.

DEFINITION 3. *A* hypergraph *is a pair $H = (V, E)$ such that*

1. *V is a non-empty set of nodes, and*
2. *E is a set of hyperedges, where a* hyperedge *is an unordered pair (u, v) of non-empty subsets of V ($u \subset V$ and $v \subset V$) with the additional condition that $u \cap v = \emptyset$.*

We call any non-empty subset of V a hypernode.

Every plan generator based on dynamic programming or memoization constructs an optimal plan for a set of relations S by combining all suitable pairs of optimal subplans for sets of relations (S_1, S_2) where $S = S_1 \cup S_2 \wedge S_1 \neq \emptyset \wedge S_2 \neq \emptyset$ must hold (see Section 4). Furthermore, the producer/consumer constraints have to be met. Therefore, only certain sets S together with their combinations of S_1, S_2 are allowed. These combinations of S_1, S_2 are called csg-cmp pairs.

DEFINITION 4. *Let $H = (V, E)$ be a hypergraph and S_1, S_2 two non-empty subsets of V with $S_1 \cap S_2 = \emptyset$. Then, the pair (S_1, S_2) is called a* csg-cmp-pair *if the following conditions hold:*

1. *S_1 and S_2 induce a connected subgraph of H, and*
2. *there exists a hyperedge $(u, v) \in E$ such that $u \subseteq S_1$ and $v \subseteq S_2$.*

(See [15] for induced subgraphs and connectedness.)

Simple plan generators like DPSUB, DPSIZE [14], and MEMOIZATIONBASIC [6] generate various combinations for (S_1, S_2) and then possibly reject some (most) of them later on if they turn out to be invalid (APPLICABLE fails). This is not very efficient. In fact, the reason for DPHYP's [15] efficiency is that it only enumerates valid csg-cmp pairs. The above problem can be avoided if we use the TES (which are contained in all three conflict detectors and are (possibly) enlarged by rule simplification) to generate hyperedges instead of using them only within APPLICABLE. Hence, the hyperedges can directly cover most of the possible conflicts, if not all (see Sec. 7). The construction of the hyperedges proceeds as follows. For every operator $\circ$, we construct a hyperedge (l, r) such that $r = \mathtt{TES}(\circ) \cap \mathcal{T}(\text{right}(\circ)) = \mathtt{R\text{-}TES}(\circ)$ and $l = \mathtt{TES}(\circ) \setminus r = \mathtt{L\text{-}TES}(\circ)$. These hyperedges are then the input to DPHYP. Two things are important to observe. First, in case of non-empty rule sets, the applicable test must still be executed. Second, since $\mathtt{SES} \subseteq \mathtt{TES}$, no other hyperedges have to be constructed.

Let us now come to the question why larger TES result in higher efficiency. The efficiency of an advanced plan generator is directly correlated to the number of csg-cmp-pairs. Obviously, larger TES result in larger hypernodes in the hyperedges (l, r). Potentially, a hyperedge (l, r) gives rise to a csg-cmp-pair (l, r) if both l and r induce connected subgraphs. Further, every (S_1, S_2) with $S_1 \supseteq l$, $S_2 \supseteq r$, $S_1 \cap S_2 = \emptyset$ is a potential csg-cmp-pair. Thus, enlarging (l, r) decreases the number of csg-cmp-pairs.

6.2 Cross Products and Degenerate Predicates

Cross products and degenerate predicates are a little brittle. Consider the example $(R_1 \times R_2) \bowtie_{1,3} (R_3 \ltimes_{3,4} R_4)$. So far, nothing prevents DPSUBE to consider invalid plans like $R_1 \bowtie_{1,3} (R_3 \ltimes_{3,4} (R_2 \times R_4))$. Note that in order to prevent this plan, we would have to detect conflicts on the "other side" of the plan. Since cross products and degenerate predicates should be rare in real queries, it suffices to produce correct plans. We have no ambition to explore the complete search space. Thus, we just want to make sure that in these abnormal cases, the plan generator still produces a correct plan. This can be achieved by conjunctively adding the check

$$\mathcal{T}(\text{left}(\circ)) \cap S_1 \neq \emptyset \wedge \mathcal{T}(\text{right}(\circ)) \cap S_2 \neq \emptyset$$

to the test for APPLICABLE($\circ$, S_1, S_2). This results in a correct test, but about a third of the valid search space will not be explored if cross products are present in the initial operator tree. However, note that if the initial plan does not contain cross products and degenerate predicates, this test will always succeed such that in this case still the whole core search space is explored. Moreover, still a larger portion of the core search space is explored when comparing this approach to the one by Rao et al. [20, 21]. There, two separate runs of the plan generator for each of the arguments of a cross product are performed, which hinders any reordering of operators with cross products. Note that Moerkotte and Neumann's approach cannot handle cross products [14].

There is a second issue concerning cross products. In some rare cases, they might be beneficially introduced, even if the initial plan does not demand them. In this case, we can proceed as proposed by Rao et al. [20, 21]. For each relation R, a *companion set* is calculated which contains all relations that are connected to R only by inner join predicates. Within a companion set, all join orders and introductions of cross products are valid.

6.3 Unnesting

Dependent joins (d-joins, ⧓) and the dependent variants of the other binary operators (⧔, ▶, ⟕, ⟗) play a central role in unnesting nested queries [2, 3, 8, 5]. Their incorporation into our approach is simple. In general, we just need to extend the matrices containing the assoc and l/r-asscom properties. In the special case of the dependent operators this is even simpler, since they have the same properties as their independent counterparts, except that they are not commutative. In any case, the conflict detectors can be used as is. However, the plan generator has to be adapted to dependent operators [14].

6.4 Pushing Grouping

Pushing group-by operators is a well-known technique to speed up data warehouse queries [24, 25]. To decide whether a group-by operator can be pushed and to do so is handled in Lines 11 and 13 of DPsube. Thus, these techniques are untouched by our conflict detector.

7. EVALUATION

In order to evaluate the different approaches, we implemented a transformation-based plan generator. It exhaustively applies the transformation rules defined in Sec. 3 until no new plan can be generated. Additionally, we implemented all known conflict detectors and used them within DPsube (see Sec. 4). Thereby, we modified DPsube such that it does not prune dominated plans but instead keeps all generated plans. This set of plans was then compared with the set of plans generated by the transformation-based plan generator. This way, we found (1) invalid plans and (2) valid plans not generated by DPsube equipped with some given conflict detector. Since NEL/EEL allows only for join, antijoin and left outerjoin but CD-X allow for more operators, we run experiments for two sets of operators ($\{\bowtie, \triangleright, ⟕\}$ and $\{\bowtie, \ltimes, \triangleright, ⟕, ⟗\}$).

For any given set of operators, we generated all possible initial plans for a given number of relations (varied between 3 and 7). For each initial plan, the different plan generators were called. The generation of all initial plans for n relations proceeds in three steps. In a first step, we unrank all integers from 1 to $\mathcal{C}(n-1)$, where $\mathcal{C}$ denotes the Catalan numbers ($\mathcal{C}(n)$ is the number of binary trees in n inner nodes), using the method proposed by Liebehenschel [13]. This gives us raw binary trees. In the second step, an operator from the operator set is attached to every inner node, making sure that every combination is generated exactly once. In the last step, we generate binary predicates by exploring all possibilities to reference one relation from the operator's left subtree and one from its right subtree. We did not generate complex predicates, since this simplifies the enumeration of the core search space (cf. Sec. 3.2).

Tables 4 and 5 show the results. The columns contain the number of relations (n), the number of distinct queries (initial operator trees), the number of plans the transformation-based plan generator generates for these queries, and for each conflict detector the number of invalid plans (I) and the number of plans not found (missing, M). The conflict detector EEL-F is a fixed version of the original NEL/EEL approach (see Sec. 8.1.1). Additionally, Table 5 contains for CD-C the number of rule sets which are empty after applying rule simplifications, and the number of non-empty rule sets.

From Table 4 we see that both the EEL/NEL approach and the SES/TES approach produce invalid plans. From Table 5 we see that CD-A and CD-B lose large fractions of the valid search space but CD-C does not. We also see that about 70% of all rule sets are empty if we apply rule simplification.

8. RELATED WORK

8.1 DP-based Plan Generation

If an input query involves binary operators other than $\bowtie$ and $\times$, not all transformations as discussed in Section 3.2 are valid. Thus, any plan generator must be modified such that it restricts its search to valid transformations only. Otherwise, without these restrictions the generated plan may not be equivalent to the input query and, therefore, the result might be wrong.

There exist several proposals to restrict the search space. First, the problem of outerjoin simplification and reordering has been studied extensively by Galindo-Legaria and Rosenthal [23, 9, 10]. They identified a subclass of join/outerjoin queries where the query graph unambiguously determines the semantics of a query. For this type of queries, they proposed a procedure that analyzes paths in the query graph to detect conflicting reorderings. They enhanced a conventional dynamic programming algorithm to deal with these conflicts. Although very useful, their approach is restricted to joins and outerjoins and the query graph must exhibit special properties. In order to handle complex predicates, Bhargava, Goel and Iyer [1] extended this approach and present a conflict detection which analyzes paths in hypergraphs. Again, the approach is limited to joins and outerjoins. Rao et al. presented a method that is not restricted to joins/outerjoins. They additionally consider antijoins [21, 20]. They propose to use the initial operator tree instead of the query graph in order to maintain the semantics of the input query. Their idea is to calculate a set of relations associated with every predicate (operator). This set of relations (called EEL, for *extended eligibility list*) must be available before the predicate can be evaluated. EELs are a superset of `NEL`s. Moerkotte and Neumann [15] adopted the idea of `EEL` and called it `TES`. Their approach considers all join operators in `LOP` and their dependent counterparts. Additionally, they reformulated non-inner joins as complex predicates by modeling their reordering conflicts in the form of hyperedges in order to make plan generation more efficient.

Both the approach of Rao et al. as well as Moerkotte and Neumann's approach are not correct. Both generate invalid plans. We will present examples demonstrating why they fail. Further, we present a fix for the algorithm of Rao et al.

8.1.1 *Outerjoin and Antijoin Reordering Using EELs*

First, we explain the approach of Rao et al. in short [20, 21]. Then, we give a counter-example that shows the incorrectness of their method. After that, we make an attempt to repair the proposed EEL computation algorithm.

Conflict Detection with EELs The main idea of [20, 21] is to compute an *extended eligibility list* (`EEL`), which is a shorthand representation of possible reordering conflicts. In [20], Rao et al. proposed an algorithm called Calc$_{\texttt{EEL}}$ to

n	#Queries	#Plans	EEL		EEL-F		TES		CD-A		CD-B		CD-C	
			I	M	I	M	I	M	I	M	I	M	I	M
3	26	88	0	0 %	0	1.14 %	0	0 %	0	0 %	0	0 %	0	0 %
4	344	4059	2	0 %	0	2.02 %	23	2.24 %	0	3.30 %	0	2.02 %	0	0 %
5	5834	301898	296	0 %	0	2.51 %	3964	6.47 %	0	8.54 %	0	5.38 %	0	0 %
6	117604	32175460	41108	0 %	0	2.70 %	605914	12.23 %	0	14.66 %	0	9.77 %	0	0 %
7	2708892	4598129499	6349126	0 %	0	2.71 %	99179293	19.05 %	0	21.06 %	0	15.04 %	0	0 %

Table 4: Small operator set: join, left outerjoin, antijoin

	#Queries	#Plans	CD-A		CD-B		CD-C		Rule Sets	
n			I	M	I	M	I	M	$\emptyset$	$\neg\emptyset$
3	62	203	0	0	0	0	0	0	107	17
4	1114	11148	0	473 (4.24 %)	0	246 (2.21 %)	0	0	2725	617
5	25056	934229	0	102019 (10.92 %)	0	55725 (5.96 %)	0	0	77484	22740
6	661811	108294798	0	20113801 (18.57 %)	0	11868102 (10.96 %)	0	0	2432717	876338
7	19846278	16448441514	0	4329578881 (26.32 %)	0	2793701760 (16.98 %)	0	0	83560096	35517572

Table 5: Large operator set: join, left/full outerjoin, semijoin, antijoin

compute the EEL for each predicate carried by a join operator $\circ \in \{\bowtie, ⟕, \triangleright\}$. The pseudo code of $\textsc{Calc}_{\texttt{EEL}}$ is shown in Fig. 12.

$\textsc{Calc}_{\texttt{EEL}}$ computes the EELs in a single bottom-up traversal (lines 4-20) of the initial operator tree. During the traversal, it maintains for each relation R an outerjoin set $outer_R$ and an antijoin set $anti_R$. Initially, both sets contain only the corresponding relation itself (lines 2, 3). Thereby, $outer_R$ stores all relations that are linked together through either inner or antijoin predicates (lines 13-16). And $anti_R$ keeps track of all relations $R \in \mathcal{T}(\text{left}(\circ)) \cap \texttt{NEL}(\circ)$ that are linked through $\circ \in \{⟕\}$ (lines 17-20). Essentially, this means that R has to be on the preserving side of a one-sided outerjoin predicate. As the name implies, $outer_R$ is used to compute the EEL for an outerjoin predicate (lines 6-8). Similarly, $anti_R$ is used to compute the EEL for an antijoin predicate (lines 9-12). The test of APPLICABLE is $\texttt{EEL} \subseteq S_1 \cup S_2$.

$\textsc{Calc}_{\texttt{EEL}}$

$\triangleright$ **Input:** $\mathcal{T}(\circ)$, $\texttt{NEL}(\circ)$ where $\circ \in \{\bowtie, ⟕, \triangleright\}$
$\triangleright$ **Output:** $\texttt{EEL}(\circ)$
1 **for each** $R \in \mathcal{T}(\text{topmost } \circ)$
2 $\quad outer_R \leftarrow \{R\}$
3 $\quad anti_R \leftarrow \{R\}$
4 **for each** operator $\circ$ during bottom-up traversal
5 $\quad \texttt{EEL}(\circ) \leftarrow \texttt{NEL}(\circ)$
6 $\quad$ **if** $\circ \in \{⟕\}$
7 $\quad\quad W \leftarrow \bigcup_{R \in \mathcal{T}(right(\circ)) \cap \texttt{NEL}(\circ)} outer_R$
8 $\quad\quad \texttt{EEL}(\circ) \leftarrow \texttt{EEL}(\circ) \cup W$
9 $\quad$ **elseif** $\circ \in \{\triangleright\}$
10 $\quad\quad V \leftarrow \bigcup_{R \in \mathcal{T}(left(\circ)) \cap \texttt{NEL}(\circ)} anti_R$
11 $\quad\quad U \leftarrow \{R | R \in \mathcal{T}(right(\circ)) \cap \texttt{NEL}(\circ)\}$
12 $\quad\quad \texttt{EEL}(\circ) \leftarrow \texttt{EEL}(\circ) \cup V \cup U$
13 $\quad$ **if** $\circ \in \{\bowtie, \triangleright\}$
14 $\quad\quad W \leftarrow \bigcup_{R \in \texttt{NEL}(\circ)} outer_R$
15 $\quad\quad$ **for each** $R \in W$
16 $\quad\quad\quad outer_R \leftarrow W$
17 $\quad$ **elseif** $\circ \in \{⟕\}$
18 $\quad\quad V \leftarrow \bigcup_{R \in \mathcal{T}(left(\circ)) \cap \texttt{NEL}(\circ)} anti_R$
19 $\quad\quad$ **for each** $R \in \mathcal{T}(right(\circ)) \cap \texttt{NEL}(\circ)$
20 $\quad\quad\quad anti_R \leftarrow anti_R \cup V$

Figure 12: Pseudocode for $\textsc{Calc}_{\texttt{EEL}}$

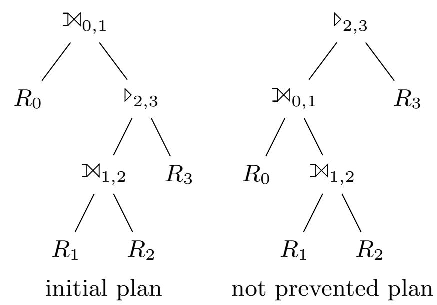

Figure 13: Example showing the incorrectness of $\textsc{Calc}_{\texttt{EEL}}$

The EEL computation is not correct. Fig. 13 shows an example where EELs as computed by $\textsc{Calc}_{\texttt{EEL}}$ do not prevent the generation of invalid plans. The initial plan is given on the left. The plan on the right of Fig. 13 can be derived by applying $\text{assoc}(⟕_{0,1}, \triangleright_{2,3})$. A look at Table 2 reveals that $\text{assoc}(⟕_{0,1}, \triangleright_{2,3})$ is not valid. Thus, the initial plan and the not prevented plan are not equivalent. We can verify this by using the relations in Table 6 as input for both plans. The result for the initial plan is given in Table 7. Clearly, this differs from the result of the invalid plan (Table 8).

Table 9 shows $anti_R$ and $outer_R$ during $\textsc{Calc}_{\texttt{EEL}}$ execution. Table 10 displays the results of $\textsc{Calc}_{\texttt{EEL}}$. According to $\texttt{EEL}(⟕_{0,1})$ and $\texttt{EEL}(\triangleright_{2,3})$, the antijoin $\triangleright_{2,3}$ can be applied on top of the outerjoin $⟕_{0,1}$, which is wrong. $\texttt{EEL}(⟕_{0,1})$ should contain $\{R_0, R_1, R_2, R_3\}$ in order to be correct because $\neg\text{assoc}(⟕_{0,1}, \triangleright_{2,3})$ holds.

R_0
A
1

R_1	
A	B
1	1

R_2	
B	C
1	1

R_3
C
1

Table 6: Example Relations.

$R_0 ⟕_{R_0.A=R_1.A}((R_1 ⟕_{R_1.B=R_2.B} R_2) \triangleright_{R_2.C=R_3.C})$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$	$R_3.C$
1	NULL	NULL	NULL	NULL	NULL

Table 7: Result for executing initial plan Fig. 13 using relations of Table 6 as input.

$(R_0 \mathbin{⟕}_{R_0.A=R_1.A}(R_1 \mathbin{⟕}_{R_1.B=R_2.B} R_2)) \triangleright_{R_2.C=R_3.C}$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$	$R_3.C$
$\emptyset$					

Table 8: Result for executing initial plan Fig. 13 using relations of Table 6 as input.

R	$outer_R$	$anti_R$
R_0	$\{R_0\}$	$\{R_0\}$
R_1	$\{R_1\}$	$\{R_0, R_1\}$
R_2	$\{R_2, R_3\}$	$\{R_1, R_2\}$
R_3	$\{R_2, R_3\}$	$\{R_3\}$

Table 9: $anti_R$ **and** $outer_R$ **sets after executing** CALCEEL**.**

Fixing the EEL computation CALCEEL can be fixed: we only have to eliminate the intersection with NEL($\circ$) in Lines 7 and 18 as in

7 $\quad W \leftarrow \bigcup_{R \in \mathcal{T}(right(\circ))} outer_R$

With this fix, CALCEEL prevents reordering conflicts, but is not complete any more. Hence, we traded in correctness for incompleteness, which still is a better choice. This can be verified by using $R_0 \mathbin{⟕}_{0,1}(R_1 \mathbin{⟕}_{1,2} R_2)$ as input plan. The modified CALCEEL procedure now calculates $\texttt{EEL}(\mathbin{⟕}_{0,1}) = \{R_0, R_1, R_2\}$, which prevents $(R_0 \mathbin{⟕}_{0,1} R_1) \mathbin{⟕}_{1,2} R_2$, although the latter is an equivalent and valid plan because assoc($\mathbin{⟕}_{0,1}$, $\mathbin{⟕}_{1,2}$) holds. Thus, $\texttt{EEL}(\mathbin{⟕}_{0,1})$ should contain $\{R_0, R_1\}$ only.

8.1.2 Join Reordering using TESs

Before a join operator can be applied, the plan generator needs to ensure that the producer/consumer constraints are fulfilled. The conventional test is to check if the SES($\circ$) of some operator $\circ$ is a subset of $\mathcal{T}(\circ)$. Moerkotte and Neumann extend this test to prevent reordering conflicts [15]. Therefore, they introduce the notion of the *total eligibility set* (TES for short). The TES is defined to be a set of relations that is attached to any binary operator $\circ$. Before $\circ$ can be applied (line 11 of DPSUBE), APPLICABLE ensures that all elements of TES($\circ$) are present in $S_1 \cup S_2$. Since TES is an extension of SES, SES $\subseteq$ TES holds.

Moerkotte and Neumann propose an algorithm called CALC$_{TES}$. It calculates the TES for every $\circ \in$ LOP. Its pseudo code can be found in [15]. As it turns out, Moerkotte's and Neumann's approach is neither correct nor complete: it generates wrong plans and misses good plans.

Fig. 14 contains an example showing the incorrectness of the SES/TES approach: the plan on the right is not equivalent to the initial plan on the left. Applying assoc($\bowtie_{1,2}, \triangleright_{2,3}$) as a first step and assoc($\mathbin{⟕}_{0,1}, \triangleright_{2,3}$) thereafter transforms the initial plan into the plan on the right. To see that the plan on the right is invalid, consider the different results in Tables 11

$\circ$	NEL	EEL
$\mathbin{⟕}_{1,2}$	$\{R_1, R_2\}$	$\{R_1, R_2\}$
$\triangleright_{2,3}$	$\{R_2, R_3\}$	$\{R_1, R_2, R_3\}$
$\mathbin{⟕}_{0,1}$	$\{R_0, R_1\}$	$\{R_0, R_1\}$

Table 10: Computed NEL and EEL after executing CALCEEL**.**

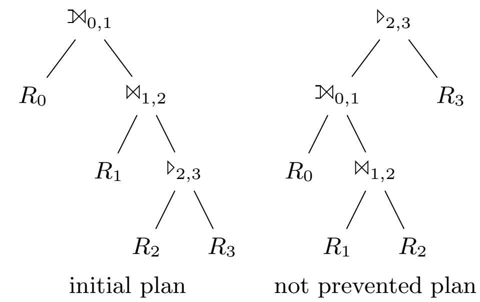

Figure 14: Example showing the incorrectness of CALC$_{TES}$

$R_0 \mathbin{⟕}_{R_0.A=R_1.A}(R_1 \bowtie_{R_1.B=R_2.B}(R_2 \triangleright_{R_2.C=R_3.C} R_3))$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$
1	NULL	NULL	NULL	NULL

Table 11: Result for executing initial plan Fig. 14 using relations of Table 6 as input.

and 12, which are based on the same input relations as before (Table 6).

Table 13 shows the results of applying CALC$_{TES}$ to the initial plan. (For details, see [15].) Due to the actual values of TES($\mathbin{⟕}_{0,1}$) and TES($\triangleright_{2,3}$), APPLICABLE allows that the antijoin $\triangleright_{2,3}$ moves on top of $\mathbin{⟕}_{0,1}$, which is invalid since $\neg$assoc($\mathbin{⟕}_{0,1}, \triangleright_{2,3}$) holds. In order to prevent the reordering, TES($\mathbin{⟕}_{0,1}$) should contain $\{R_0, R_1, R_2, R_3\}$.

8.2 Transformation-Based Plan Generation

Besides DP-based and memoization-based algorithms for plan generation, there exists plenty of work on transformation-based plan generators [11, 17]. Unfortunately, transformation-based plan generators are less efficient. First, they are memory consuming, since they cannot prune plans because they need them to generate more plans via transformations. Second, they generate an exponential number of duplicates, as pointed out by Pellenkoft, Galindo-Legaria, and Kersten [17, 18, 19]. They also propose a solution to avoid the generation of duplicates, but this solution only works for acyclic query graphs. Thus, until better transformation-based algorithms are found, DP-based or memoization-based plan generators are the approaches of choice. Note that the latter also need a correct conflict detector.

$(R_0 \mathbin{⟕}_{R_0.A=R_1.A}(R_1 \bowtie_{R_1.B=R_2.B} R_2)) \triangleright_{R_2.C=R_3.C} R_3$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$
$\emptyset$				

Table 12: Result for executing right plan Fig. 14 using relations of Table 6 as input.

$\circ$	SES	TES
$\triangleright_{2,3}$	$\{R_2, R_3\}$	$\{R_2, R_3\}$
$\bowtie_{1,2}$	$\{R_1, R_2\}$	$\{R_1, R_2\}$
$\mathbin{⟕}_{0,1}$	$\{R_0, R_1\}$	$\{R_0, R_1, R_2\}$

Table 13: Computed SES and TES after executing CALC$_{TES}$**.**

8.3 Beyond the Core Search Space

There exist several approaches to go beyond the core search space. The first one is based on generalized outerjoins [1, 4, 23]. To incorporate generalized outerjoins into our framework, we need to extend the property matrices. Another approach replaces all outerjoins by joins [12]. To make this work correctly, a complete semijoin reduction is performed upfront and virtual rows are introduced. Yet another interesting approach is proposed by Rao et al. [22]. They deliberately apply wrong reorderings and then compensate for it. For the compensation they rely on a new operator called best match that is relatively expensive. If no best-match operator is available in the runtime system, only compensation-free plans can be generated. However, whether a plan needs compensation or not is decided when the top-most operator is put in place. Hence, plans that need compensation are still built and have to be thrown away afterwards. Consequently, when compared to our approach far more (sub-) plans are generated but no cheaper plan is found. Moreover, our optimization techniques as described in Sec. 5.5 and Sec. 6.1 cannot be applied which renders modern plan generators like DPHyp [15] or TD-McCHyp [7] almost useless. Furthermore, we conjecture that [22] does not cover the whole core search space. For example, for the initial plan $(R_1 ⟕_{p_{12}} R_2) ⟕_{p_{23}} R_3$, the alternative $R_1 ⟕_{p_{12}} (R_2 ⟕_{p_{23}} R_3)$ cannot be produced. It is future research to see how compensation can be incorporated into our approach.

9. CONCLUSION

We showed that existing approaches to reorder a not necessarily strict superset of $\{⋈, ⟕, ▷\}$ are incorrect: they produce invalid plans. We then presented the first valid conflict detectors. The third one of them, CD-C, is not only correct but also complete. It thus explores the full core search space. Further, our approach is extensible. If new algebraic operators pop up, we only need to extend four matrices containing their properties. The code itself remains unchanged.

Acknowledgements. We thank Simone Seeger for her help preparing the manuscript. We are also grateful to the anonymous referees for their comments, which helped us to significantly improve the paper.

10. REFERENCES

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

[2] S. Cluet and G. Moerkotte. Nested queries in object bases. In *DBPL*, pages 226–242, 1993.

[3] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

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

[5] M. Elhemali, C. Galindo-Legaria, T. Grabs, and M. Joshi. Execution strategies for SQL subqueries. In *SIGMOD*, pages 993–1003, 2007.

[6] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *ICDE*, pages 864–875, 2011.

[7] P. Fender and G. Moerkotte. Top down plan generation: From theory to practice. In *ICDE*, pages 1105–1116, 2013.

[8] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, pages 571–581, 2001.

[9] C. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *ICDE*, pages 402–409, 1992.

[10] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *TODS*, 22(1):43–73, Marc 1997.

[11] G. Graefe and W. McKenna. Extensibility and search efficiency in the volcano optimizer generator. In *ICDE*, pages 209–218, 1993.

[12] G. Hill and A. Ross. Reducing outer joins. *VLDB Journal*, 18:599–610, 2009.

[13] J. Liebehenschel. Ranking and unranking of lexicographically ordered words: An average-case analysis. *J. of Automata, Languages, and Combinatorics*, 2:227–268, 1997.

[14] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy trees without cross products. In *VLDB*, pages 930–941, 2006.

[15] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, pages 539–552, 2008.

[16] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11):843–851, 2011.

[17] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. Complexity of transformation-based optimizers and duplicate-free generation of alternatives. Technical Report CS-R9639, CWI, 1996.

[18] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *VLDB*, pages 306–315, 1997.

[19] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. Duplicate-free generation of alternatives in transformation-based optimizers. In *DASFAA*, pages 117–124, 1997.

[20] J. Rao, B. Lindsay, G. Lohman, H.Pirahesh, and D. Simmen. Using EELs: A practical approach to outerjoin and antijoin reordering. Technical Report RJ 10203, IBM, 2000.

[21] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen. Using EELs: A practical approach to outerjoin and antijoin reordering. In *ICDE*, pages 595–606, 2001.

[22] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In *SIGMOD*, pages 671–682, 2004.

[23] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *SIGMOD*, pages 291–299, 1990.

[24] W. Yan and P.-A. Larson. Performing group-by before join. In *ICDE*, pages 89–100, 1994.

[25] W. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357, 1995.