Dynamic Programming Strikes Back

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

Thomas Neumann
Max-Planck Institute for Informatics
Saarbrücken, Germany
neumann@mpi-inf.mpg.de

ABSTRACT

Two highly efficient algorithms are known for optimally ordering joins while avoiding cross products: DPccp, which is based on dynamic programming, and Top-Down Partition Search, based on memoization. Both have two severe limitations: They handle only (1) simple (binary) join predicates and (2) inner joins. However, real queries may contain complex join predicates, involving more than two relations, and outer joins as well as other non-inner joins.

Taking the most efficient known join-ordering algorithm, DPccp, as a starting point, we first develop a new algorithm, DPhyp, which is capable to handle complex join predicates efficiently. We do so by modeling the query graph as a (variant of a) hypergraph and then reason about its connected subgraphs. Then, we present a technique to exploit this capability to efficiently handle the widest class of non-inner joins dealt with so far. Our experimental results show that this reformulation of non-inner joins as complex predicates can improve optimization time by orders of magnitude, compared to known algorithms dealing with complex join predicates and non-inner joins. Once again, this gives dynamic programming a distinct advantage over current memoization techniques.

Categories and Subject Descriptors

H.2 [**Systems**]: Query processing

General Terms

Algorithms, Theory

1. INTRODUCTION

For the overall performance of a database management system, the cost-based query optimizer is an essential piece of software. One important and complex problem any cost-based query optimizer has to solve is to find the optimal join order. In their seminal paper, Selinger et al. not only introduced cost-based query optimization but also proposed a dynamic programming algorithm to find the optimal join order for a given conjunctive query [21]. More precisely, they proposed to generate plans in the order of increasing size. Although they restricted the search space to left-deep trees, the general idea of their algorithm can be extended to the algorithm DPsize, which explores the space of bushy trees (see Fig. 1). The algorithm still forms the core of state-of-the-art commercial query optimizers like the one of DB2 [12].

Recently, we gave a thorough complexity analysis of DPsize [17]. We proved that DPsize has a runtime complexity which is much worse than the lower bound. This is mainly due to the tests (marked by '*' in Fig. 1), which fail far more often than they succeed. Furthermore, we proposed the algorithm DPccp, which exactly meets the lower bound. Experiments showed that DPccp is highly superior to DPsize. The core of their algorithm generates connected subgraphs in a bottom-up fashion.

The main competitor for dynamic programming is memoization, which generates plans in a top-down fashion. All known approaches needed tests similar to those shown for DPsize. Thus, with the advent of DPccp, dynamic programming became superior to memoization when it comes to generating optimal bushy join trees, which do not contain cross products. Challenged by this finding, DeHaan and Tompa successfully devised a top-down algorithm that is capable of generating connected subgraphs by exploiting minimal cuts [7]. With this algorithm, called Top-Down Partition Search, memoization can be almost as efficient as dynamic programming.

However, both algorithms, DPccp and Top-Down Partition Search, are not ready yet to be used in practice: there exist two severe deficiencies in both of them. First, as has been argued in several places, hypergraphs must be handled by any plan generator [1, 19, 23]. Second, plan generators have to deal with outer joins and antijoins [11, 19]. These operators are, in general, not freely reorderable. That is, there might exist different orderings, which produce different results. This is not true for the regular, inner join: any ordering gives the same result. Restricting the ordering to valid orderings for outer joins, that is those which produce the same result as the original query, has been the subject of the seminal work by Galindo-Legaria and Rosenthal [10, 11, 20]. They also propose a dynamic programming algorithm that takes into account the intricacy of outer joins. Their algorithm has been extended by Bhargava et al. to deal with hyperedges [1]. A more practical approach has been proposed by Rao et al. [19]. They also include the

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

```
DPsize (R = {R_0, ..., R_{n-1}})
for ∀ R_i ∈ R    dpTable[{R_i}] = R_i
for ∀ 1 < s ≤ n ascending // size of plan
    for ∀ 1 ≤ s_1 < s // size of left subplan
        for ∀ S_1 ⊂ R : |S_1| = s_1, S_2 ⊂ R : |S_2| = s − s_1
            if S_1 ∩ S_2 ≠ ∅ continue    (*)
            if ¬(S_1 connected to S_2) continue    (*)
            p =dpTable[S_1] ⋈ dpTable[S_2]
            if cost(p)<cost(dpTable[S_1 ∪ S_2])
                dpTable[S_1 ∪ S_2] = p
return dpTable[{R_0, ..., R_{n-1}}]
```

Figure 1: Algorithm DPsize

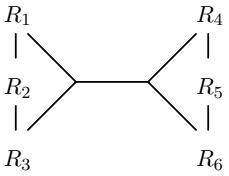

Figure 2: Sample hypergraph

antijoin. All these approaches use DPsize as their starting point. Thus, they suffer from a much higher than necessary runtime complexity.

In this paper, we introduce DPhyp, which can efficiently deal with hypergraphs (Sec. 2 and 3). Experiments will show that it is highly superior to existing approaches (Sec. 4). In a second step, we deal with left and full outer joins, antijoins, nestjoins, and their dependent counterparts (Section 5). It will be shown that non-inner joins can be dealt with by introducing new hyperedges. Thus, no extension to DPhyp except for calculating the new hyperedges is necessary to deal with a complete set of non-inner and dependent joins. This approach is highly superior to existing approaches even if no initial hyperedges are present, i.e. the query exhibits only simple predicates but non-inner joins.

2. HYPERGRAPHS

2.1 Definitions

Let us start with the definition of hypergraphs.

DEFINITION 1 (HYPERGRAPH). *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. *We assume that the nodes in* V *are totally ordered via an (arbitrary) relation* $\prec$. *The ordering on nodes is important for our algorithm.*

A hyperedge (u, v) *is* simple *if* $|u| = |v| = 1$. *A hypergraph is* simple *if all its hyperedges are simple.*

Note that a simple hypergraph is the same as an ordinary undirected graph. In our context, the nodes of hypergraphs are relations and the edges are abstractions of join predicates. Consider, for example, a join predicate of the form $R_1.a + R_2.b + R_3.c = R_4.d + R_5.e + R_6.f$. This predicate will result in a hyperedge $(\{R_1, R_2, R_3\}, \{R_4, R_5, R_6\})$. Fig. 2 contains an example of a hypergraph. The set V of nodes is $V = \{R_1, \ldots, R_6\}$. Concerning the node ordering, we assume that $R_i \prec R_j \iff i < j$. There are the simple edges $(\{R_1\}, \{R_2\})$, $(\{R_2\}, \{R_3\})$, $(\{R_4\}, \{R_5\})$, and $(\{R_5\}, \{R_6\})$. The hyperedge from above is the only true hyperedge in the hypergraph.

Note that is possible to rewrite the above complex join predicate. For example, it is equivalent to $R_1.a + R_2.b = R_4.d + R_5.e + R_6.f - R_3.c$. This leads to a hyperedge $(\{R_1, R_2\}, \{R_3, R_4, R_5, R_6\})$. If the query optimizer is capable of performing this kind of algebraic transformations, all derived hyperedges are added to the hypergraph, at least conceptually. We will come back to this issue in Section 6.

To decompose a join ordering problem represented as a hypergraph into smaller problems, we need the notion of subgraph. More specifically, we only deal with node-induced subgraphs.

DEFINITION 2 (SUBGRAPH). *Let* $H = (V, E)$ *be a hypergraph and* $V' \subseteq V$ *a subset of nodes. The* node induced subgraph $G|_{V'}$ *of* G *is defined as* $G|_{V'} = (V', E')$ *with* $E' = \{(u, v) | (u, v) \in E, u \subseteq V', v \subseteq V'\}$. *The node ordering on* V' *is the restriction of the node ordering of* V.

As we are interested in connected subgraphs, we give

DEFINITION 3 (CONNECTED). *Let* $H = (V, E)$ *be a hypergraph.* H *is* connected *if* $|V| = 1$ *or if there exists a partitioning* V', V'' *of* V *and a hyperedge* $(u, v) \in E$ *such that* $u \subseteq V'$, $v \subseteq V''$, *and both* $G|_{V'}$ *and* $G|_{V''}$ *are connected.*

If $H = (V, E)$ is a hypergraph and $V' \subseteq V$ is a subset of the nodes such that the node-induced subgraph $G|_{V'}$ is connected, then we call V' a *connected subgraph* or *csg* for short. The number of connected subgraphs is important for dynamic programming: it directly corresponds to the number of entries in the dynamic programming table. If a node set $V'' \subseteq (V \setminus V')$ induces a connected subgraph $G|_{V''}$, we call V'' a *connected complement* of V' or *cmp* for short.

Within this paper, we will assume that all hypergraphs are connected. This way, we can make sure that no cross products are needed. However, when dealing with hypergraphs, this condition can easily be assured by adding according hyperedges: for every pair of connected components, we can add a hyperedge whose hypernodes contain exactly the relations of the connected components. By considering these hyperedges as $\bowtie$ operators with selectivity 1, we get an equivalent connected hypergraph (i.e., one that describes the same query).

2.2 Csg-cmp-pair

With these notations, we can move closer to the heart of dynamic programming by defining a csg-cmp-pair.

DEFINITION 4 (CSG-CMP-PAIR). *Let* $H = (V, E)$ *be a hypergraph and* S_1, S_2 *two subsets of* V *such that* $S_1 \subseteq V$ *and* $S_2 \subseteq (V \setminus S_1)$ *are a connected subgraph and a connected complement. If there further exists a hyperedge* $(u, v) \in E$ *such that* $u \subseteq S_1$ *and* $v \subseteq S_2$, *we call* (S_1, S_2) *a* csg-cmp-pair.

Note that if (S_1, S_2) is a csg-cmp-pair, then (S_2, S_1) is one as well. We will restrict the enumeration of csg-cmp-pairs to those (S_1, S_2) which satisfy the condition that $\min(S_1) \prec \min(S_2)$, where $\min(S) = s$ such that $s \in S$ and $\forall s' \in S : s \neq s' \implies s \prec s'$. Since this restriction will hold for all csg-cmp-pairs enumerated by our procedure, we are sure that no duplicate csg-cmp-pairs are calculated. As a consequence, we have to take some care in order to ensure that our dynamic programming procedure is complete: if the binary operator we apply is commutative, the procedure to build a plan for $S_1 \cup S_2$ from plans for S_1 and S_2 has to take commutativity into account. However, this is not really a challenge.

Obviously, in order to be correct, any dynamic programming algorithm has to consider all csg-cmp-pairs [17]. Further, only these have to be considered. Thus, the minimal number of cost function calls of any dynamic programming algorithm is exactly the number of csg-cmp-pairs for a given hypergraph. Note that the number of connected subgraphs is far smaller than the number of csg-cmp-pairs. The problem now is to enumerate the csg-cmp-pairs efficiently and in an order acceptable for dynamic programming. The latter can be expressed more specifically. Before enumerating a csg-cmp-pair (S_1, S_2), all csg-cmp-pairs (S_1', S_2') with $S_1' \subseteq S_1$ and $S_2' \subseteq S_2$ have to be enumerated.

2.3 Neighborhood

The main idea to generate csg-cmp-pairs is to incrementally expand connected subgraphs by considering new nodes in the *neighborhood* of a subgraph. Informally, the neighborhood $N(S)$ under an exclusion set X consists of all nodes reachable from S that are not in X. We derive an exact definition below.

When choosing subsets of the neighborhood for inclusion, we have to treat a hypernode as a single instance: either all of its nodes are inside an enumerated subset or none of them. Since we want to use the fast subset enumeration procedure introduced by Vance and Maier [24], we must have a single bit representing a hypernode and also single bits for relations occurring in simple edges. Since these may overlap, we are constrained to choose one unique representative of every hypernode occurring in a hyperedge. We choose the node that is minimal with respect to $\prec$. Accordingly, we define:

$$\min(S) = \{s | s \in S, \forall s' \in S \; s \neq s' \implies s \prec s'\}$$

Note that if S is empty, then $\min(S)$ is also empty. Otherwise, it contains a single element. Hence, if S is a singleton set, then $\min(S)$ equals the only element contained in S. For our hypergraph in Fig. 2 and with $S = \{R_4, R_5, R_6\}$, we have $\min(S) = \{R_4\}$.

Let S be a current set, which we want to expand by adding further relations. Consider a hyperedge (u, v) with $u \subseteq S$. Then, we will add $\min(v)$ to the neighborhood of S. However, we have to make sure that the missing elements of v, i.e. $v \setminus \min(v)$, are also contained in any set emitted. We thus define

$$\overline{\min}(S) = S \setminus \min(S)$$

For our hypergraph in Fig. 2 and with $S = \{R_4, R_5, R_6\}$, we have $\overline{\min}(S) = \{R_5, R_6\}$.

We define the set of non-subsumed hyperedges as the minimal subset $E\downarrow$ of E such that for all $(u, v) \in E$ there exists a hyperedge $(u', v') \in E\downarrow$ with $u' \subseteq u$ and $v' \subseteq v$. Additionally, we make sure that none of the nodes of a hypernode are contained in a set X, which is to be excluded from neighborhood considerations. We thus define a set containing the *interesting hypernodes* for given sets S and X. We do so in two steps. First, we collect the potentially interesting hypernodes into a set $E\downarrow'(S, X)$ and then minimize this set to eliminate subsumed hypernodes. This step then results in $E\downarrow(S, X)$, with which the algorithm will work.

$$E\downarrow'(S, X) = \{v | (u, v) \in E, u \subseteq S, v \cap S = \emptyset, v \cap X = \emptyset\}$$

Define $E\downarrow(S, X)$ to be the minimal set of hypernodes such that for all $v \in E\downarrow'(S, X)$ there exists a hypernode v' in $E\downarrow(S, X)$ such that $v' \subseteq v$. Note that apart from the connectedness, we test exactly the conditions given in Def. 4. For our hypergraph in Fig. 2 and with $X = S = \{R_1, R_2, R_3\}$, we have $E\downarrow(S, X) = \{\{R_4, R_5, R_6\}\}$.

We are now ready to define the neighborhood of a hypernode S, given a set of excluded nodes X.

$$\mathcal{N}(S, X) = \bigcup_{v \in E\downarrow(S,X)} \min(v) \tag{1}$$

For our hypergraph in Fig. 2 and with $X = S = \{R_1, R_2, R_3\}$, we have $\mathcal{N}(S, X) = \{R_4\}$. Assuming a bit vector representation of sets, the neighborhood can be efficiently calculated bottom-up.

3. THE ALGORITHM

Before starting with the algorithm description we give a high-level overview of the general principles used in the algorithm:

1. The algorithm constructs ccps by enumerating connected subgraphs from an increasing part of the query graph;
2. both the primary connected subgraphs and its connected complement are created by recursive graph traversals;
3. during traversal, some nodes are *forbidden* to avoid creating duplicates. More precisely, when a function performs a recursive call it forbids all nodes it will investigate itself;
4. connected subgraphs are increased by following edges to neighboring nodes. For this purpose hyperedges are interpreted as $n : 1$ edges, leading from n of one side to one (specific) canonical node of the other side (cmp. Eq. 1).

Summarizing the above, the algorithm traverses the graph in a fixed order and recursively produces larger connected subgraphs. The main challenge relative to [17] is the traversal of hyperedges: First, the "starting" side of the edge can require multiple nodes, which complicates neighborhood computation. In particular the neighborhood can no longer be computed as a simple bottom-up union of local neighborhoods. Second, the "ending" side of the edge can lead to multiple nodes at once, which disrupts the recursive growth of components. The algorithm therefore picks a *canonical end node* (the 1 in the $n : 1$ of item 4 above, see also Eq. 1), starts recursive growth and uses the DP table to check if a valid constellation has been reached (this exploits the fact

that DP strategies enumerate subsets before supersets). We now discuss the details of the algorithm.

We give the implementation of our join ordering algorithm for hypergraphs by means of pseudocode for member functions of a class `DPhyp`. This allows us to minimize the number of parameters by assuming that the query hypergraph ($G = (V, E)$) and the dynamic programming table (`dpTable`) are class members.

The whole algorithm is distributed over five subroutines. The top-level routine `Solve` initializes the dynamic programming table with access plans for single relations and then calls `EmitCsg` and `EnumerateCsgRec` for each set containing exactly one relation. The member function `EnumerateCsgRec` is responsible for enumerating connected subgraphs. It does so by calculating the neighborhood and iterating over each of its subset. For each such subset S_1, it calls `EmitCsg`. This member function is responsible for finding suitable complements. It does so by calling `EnumerateCmpRec`, which recursively enumerates the complements S_2 for the connected subgraph S_1 found before. The pair (S_1, S_2) is a csg-cmp-pair. For every such pair, `EmitCsgCmp` is called. Its main responsibility is to consider a plan built up from the plans for S_1 and S_2. The following subsections discuss these five member functions in detail. We illustrate them with the example hypergraph shown in Fig. 2. The corresponding traversal steps are shown in Fig. 3, we will illustrate them during the algorithm description.

3.1 Solve

The pseudocode for `Solve` looks as follows:

```
Solve()
for each v ∈ V // initialize dpTable
    dpTable[{v}] = plan for v
for each v ∈ V descending according to ≺
    EmitCsg({v}) // process singleton sets
    EnumerateCsgRec({v}, B_v) // expand singleton sets
return dpTable[V]
```

In the first loop, it initializes the dynamic programming table with plans for single relations. In the second loop, it calls for every node in the query graph, in decreasing order (according to $\prec$) the two subroutines `EmitCsg` and `EnumerateCsgRec`. The algorithm calls `EmitCsg({v})` for single nodes $v \in V$ to generate all csg-cmp-pairs $(\{v\}, S_2)$ via calls to `EnumerateCsgCmp` and `EmitCsgCmp`, where $v \prec \min(S_2)$ holds. This condition implies that every csg-cmp-pair is generated only once, and no symmetric pairs are generated. In Fig. 3, this corresponds to single vertex graphs, e.g. step 1 and 2. The calls to `EnumerateCsgRec` extend the initial set $\{v\}$ to larger sets S_1, for which then connected subsets of its complement S_2 are found such that (S_1, S_2) results in a csg-cmp-pair. In Fig. 3, this is shown in step 2, for example, where `EnumerateCsgRec` starts with R_5 and expands it to $\{R_5, R_6\}$ in step 4 (step 3 being the construction of the complement). To avoid duplicates during enumerations, all nodes that are ordered before v according to $\prec$ are prohibited during the recursive expansion [17]. Formally, we define this set as $B_v = \{w | w \prec v\} \cup \{v\}$.

3.2 EnumerateCsgRec

The general purpose of `EnumerateCsgRec` is to extend a given set S_1, which induces a connected subgraph of G to a larger set with the same property. It does so by considering each non-empty, proper subset of the neighborhood of S_1. For each of these subsets N, it checks whether $S_1 \cup N$ is a connected component. This is done by a lookup into the `dpTable`. If this test succeeds, a new connected component has been found and is further processed by a call `EmitCsg`$(S_1 \cup N)$. Then, in a second step, for all these subsets N of the neighborhood, we call `EnumerateCsgRec` such that $S_1 \cup N$ can be further extended recursively. The reason why we first call `EmitCsg` and then `EnumerateCsgRec` is that in order to have an enumeration sequence valid for dynamic programming, smaller sets must be generated first. Summarizing, the code looks as follows:

```
EnumerateCsgRec(S1, X)
for each N ⊆ N(S1, X): N ≠ ∅
    if dpTable[S1 ∪ N]≠ ∅
        EmitCsg(S1 ∪ N)
for each N ⊆ N(S1, X): N ≠ ∅
    EnumerateCsgRec(S1 ∪ N, X ∪ N(S1, X))
```

Take a look at step 12. This call was generated by `Solve` on $S_1 = \{R_2\}$. The neighborhood consists only of $\{R_3\}$, since R_1 is in X (R_4, R_5, R_6 are not in X either, but not reachable). `EnumerateCsgRec` first calls *EmitCsg*, which will create the joinable complement (step 13). It then tests $\{R_2, R_3\}$ for connectedness. The according `dpTable` entry was generated in step 13. Hence, this test succeeds, and $\{R_2, R_3\}$ is further processed by a recursive call to `EnumerateCsgRec` (step 14). Now the expansion stops, since the neighborhood of $\{R_2, R_3\}$ is empty, because $R_1 \in X$.

3.3 EmitCsg

`EmitCsg` takes as an argument a non-empty, proper subset S_1 of V, which induces a connected subgraph. It is then responsible to generate the seeds for all S_2 such that (S_1, S_2) becomes a csg-cmp-pair. Not surprisingly, the seeds are taken from the neighborhood of S_1. All nodes that have ordered before the smallest element in S_1 (captured by the set $\mathcal{B}_{min(S_1)}$) are removed from the neighborhood to avoid duplicate enumerations [17]. Since the neighborhood also contains $\min(v)$ for hyperedges (u, v) with $|v| > 1$, it is not guaranteed that S_1 is connected to v. To avoid the generation of false csg-cmp-pairs, `EmitCsg` checks for connectedness. However, each single neighbor might be extended to a valid complement S_2 of S_1. Hence, no such test is necessary before calling `EnumerateCmpRec`, which performs this extension. The pseudocode looks as follows:

```
EmitCsg(S1)
X = S1 ∪ B_min(S1)
N = N(S1, X)
for each v ∈ N descending according to ≺
    S2 = {v}
    if ∃(u, v) ∈ E : u ⊆ S1 ∧ v ⊆ S2
        EmitCsgCmp(S1, S2)
    EnumerateCmpRec(S1, S2, X)
```

Take a look at step 20. The current set S_1 is $S_1 = \{R_1, R_2, R_3\}$, and the neighborhood is $\mathcal{N} = \{R_4\}$. As there is no hyperedge connecting these two sets, there is no call to `EmitCsgCmp`. However, the set $\{R_4\}$ can be extended to a valid complement, namely $\{R_4, R_5, R_6\}$. Properly extending the seeds of complements is the task of the call to `EnumerateCmpRec` in step 21.

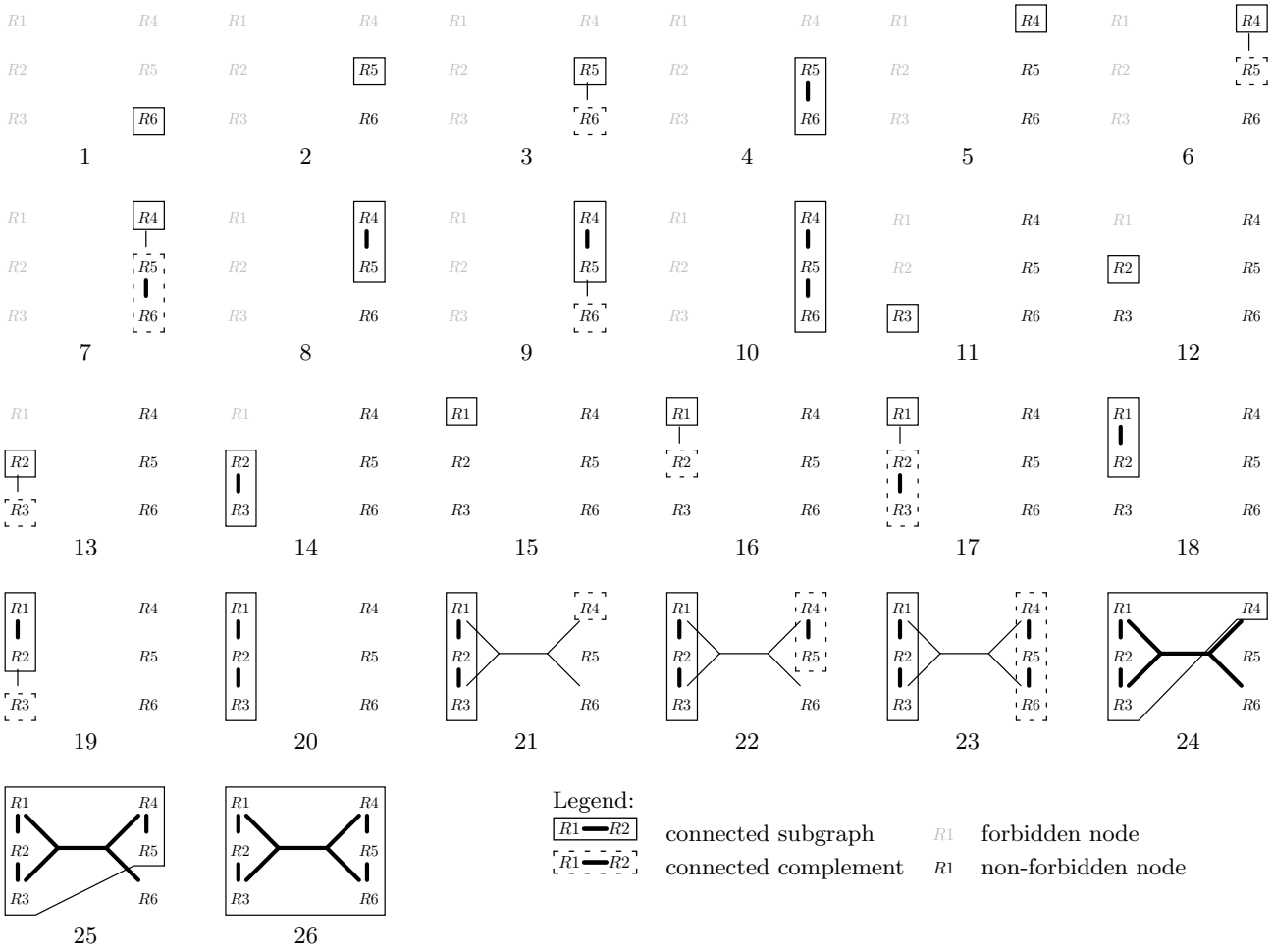

Figure 3: Trace of algorithm for Figure 2

3.4 EnumerateCmpRec

`EnumerateCsgRec` has three parameters. The first parameter S_1 is only used to pass it to `EmitCsgCmp`. The second parameter is a set S_2 which is connected and must be extended until a valid csg-cmp-pair is reached. Therefore, it considers the neighborhood of S_2. For every non-empty, proper subset N of the neighborhood, it checks whether $S_2 \cup N$ induces a connected subset and is connected to S_1. If so, we have a valid csg-cmp-pair (S_1, S_2) and can start plan construction (done in `EmitCsgCmp`). Irrespective of the outcome of the test, we recursively try to extend S_2 such that this test becomes successful. Overall, the `EnumerateCmpRec` behaves very much like `EnumerateCsgRec`. Its pseudocode looks as follows:

```
EnumerateCmpRec(S1, S2, X)
for each N ⊆ N(S2, X): N ≠ ∅
    if dpTable[S2 ∪ N]≠ ∅ ∧
            ∃(u, v) ∈ E : u ⊆ S1 ∧ v ⊆ S2 ∪ N
        EmitCsgCmp(S1, S2 ∪ N)
X = X ∪ N(S2, X)
for each N ⊆ N(S2, X): N ≠ ∅
    EnumerateCmpRec(S1, S2 ∪ N, X)
```

Take a look at step 21 again. The parameters are $S_1 = \{R_1, R_2, R_3\}$ and $S_2 = \{R_4\}$. The neighborhood consists of the single relation R_5. The set $\{R_4, R_5\}$ induces a connected subgraph. It was inserted into `dpTable` in step 6. However, there is no hyperedge connecting it to S_1. Hence, there is no call to `EmitCsgCmp`. Next is the recursive call in step 22 with S_2 changed to $\{R_4, R_5\}$. Its neighborhood is $\{R_6\}$. The set $\{R_4, R_5, R_6\}$ induces a connected subgraph. The according test via a lookup into `dpTable` succeeds, since the according entry was generated in step 7. The second part of the test also succeeds, as our only true hyperedge connects this set with S_1. Hence, the call to `EmitCsgCmp` in step 23 takes place and generates the plans containing all relations.

3.5 EmitCsgCmp

The task of `EmitCsgCmp(`S_1`,`S_2`)` is to join the optimal plans for S_1 and S_2, which must form a csg-cmp-pair. For this purpose, we must be able to calculate the proper join predicate and costs of the resulting joins. This requires that join predicates, selectivities, and cardinalities are attached to the hypergraph. Since we hide the cost calculations in an abstract function `cost`, we only have to explicitly assemble the join predicate. For a given hypergraph $G = (V, E)$ and

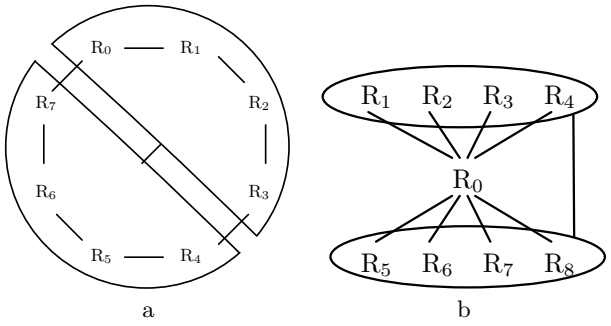

Figure 4: Cycle and Star with initial hyperedge ($n = 8$)

a hyperedge $(u, v) \in E$, we denote by $\mathcal{P}(u, v)$ the predicate represented by the hyperedge (u, v).

The pseudocode of `EmitCsgCmp` should look very familar:

```
EmitCsgCmp(S1, S2)
plan1 = dpTable[S1]
plan2 = dpTable[S2]
S = S1 ∪ S2
p = ⋀_{(u1,u2)∈E, ui⊆Si} P(u1, u2)
newplan = plan1 ⋈_p plan2
if dpTable[S]= ∅ ∨ cost(newplan) < cost(dpTable[S])
    dpTable[S] = newplan
newplan = plan2 ⋈_p plan1 // for commutative ops only
if cost(newplan) < dpTable[S]
    dpTable[S] = newplan
```

First, the optimal plans for S_1 and S_2 are recovered from the dynamic programming table. Then, we remember in S the total set of relations present in the plan to be constructed. The join predicate p is assembled by taking the conjunction of the predicates of those hyperedges that connect S_1 to S_2. Then, the plans are constructed and, if they are cheaper than existing plans, stored in `dpTable`.

The calculation of the predicate p seems to be expensive, since all edges have to be tested. However, we can attach the set of predicates

$$p_S = \{\mathcal{P}(u, v) | (u, v) \in E, u \subseteq S\}$$

to any plan class $S \subseteq V$. If we represent the p_S by a bit vector, then for a csg-cmp-pair we can easily calculate $p_{S_1} \cap p_{S_2}$ and just consider the result.

3.6 Memory Requirements

All dynamic programming variants `DPsize`, `DPsub`, `DPccp`, and `DPhyp` memoize the best plan for each subset of relations that induces a connected subgraph of the query graph. Since this is the major factor in memory consumption, the memory requirements of all algorithms are about the same. It is only about the same, because the number of bytes necessary for each such subset may differ sligthly. For example, `DPsub` needs an additional pointer to link plans of equal size.

4. EVALUATION

Unfortunately, there are no experiments on join ordering for hypergraphs reported in the literature. Thus, we had to invent our own experiments. The general design principle of our hypergraphs used in the experiments is that we start with a simple graph and add one big hyperedge to it. Then, we successively split the hyperedge into two smaller ones until we reach simple edges. As starting points, we use those graphs that have proven useful for the join ordering of simple graphs. In the literature, we often find the use of chain, cycle, star, and clique queries [17]. The behavior of join ordering algorithms on chains and cycles does not differ much: the impact of one additional edge is minor. Hence, we decided to use cycles as one starting point. Star queries have also been proven to be very useful to illustrate different performance behaviors of join ordering algorithms. Moreover, star queries are common in data warehousing and thus deserve special attention. Hence, we also used star queries as a starting point. The last potential candidate are clique queries. However, adding hyperedges to a clique query does not make much sense, as every subset of relations already induces a connected subgraph. Thus, we limited our experiments to hypergraphs derived from cycle and star queries.

Fig. 4a shows a starting cycle-based query. It contains eight relations $R_0, \ldots, R_7$. The simple edges are $(\{R_i\}, \{R_{i+1}\})$ for $0 \leq i \leq 7$ (with $R_{7+1} = R_0$). We then added the hyperedge $(\{R_0, \ldots, R_3\}, \{R_4, \ldots, R_7\})$. Each of its hypernodes consists of exactly half of the relations. From this graph (call it G_0), we derive hypergraphs $G_1, \ldots, G_3$ by successively splitting the hyperedge. This is done by splitting each hypernode into two hypernodes comprising half of the relations. That is, apart from the simple edges, G_1 has the hyperedges $(\{R_0, R_1\}, \{R_6, R_7\})$ and $(\{R_2, R_3\}, \{R_4, R_5\})$. To derive G_2, we split the first hyperedge into $(\{R_0\}, \{R_6\})$ and $(\{R_1\}, \{R_7\})$. G_3 additionally splits the second hyperedge into $(\{R_2\}, \{R_4\})$ and $(\{R_3\}, \{R_5\})$.

For star queries, we apply the same procedure. Fig. 4b shows an initial hypergraph derived from a star. It consists of nine relations $R_0, \ldots, R_8$ and simple edges $(\{R_0\}, \{R_i\})$ for $1 \leq i \leq 8$. The hyperedge is $(\{R_1, \ldots, R_4\}, \{R_5, \ldots, R_8\})$. More hypergraphs are generated by successively splitting this hyperedge as described above.

4.1 The Competitors

We ran `DPhyp` against `DPsize` and `DPsub`. For regular graphs, these algorithms are explained in detail in [17]. Since `DPsize` is the most frequently used dynamic programming algorithm, we give its pseudocode in Fig. 1. In order to deal with hypergraphs, the pseudocode does not have to be changed: only the second test marked by (*) has to be implemented in such a way that it is capable to deal with hyperedges instead of only regular edges. Whereas `DPsize` enumerates plans by increasing size, `DPsub` generates subsets. Assume the best plan for a set of relations S is to be found. Then `DPsub` generates all subsets $S_1 \subset S$ and joins the best plans for S_1 and $S_2 = S \setminus S_1$. Before doing so, there are tests checking that (S_1, S_2) is a csg-cmp-pair. Again, the pseudocode of `DPsub` does not have to be changed, but the test checking that S_1 and S_2 are connected has to be implemented in such a way that it can deal with hyperedges.

4.2 Cycle-Based Hypergraphs

For very small queries with 3 or fewer relations, there is no observable difference in the execution time of the different algorithms. Cycle queries with four relations exhibit a small difference. However, since each hypernode in the initial hyperedge consists of only two relations, there is only

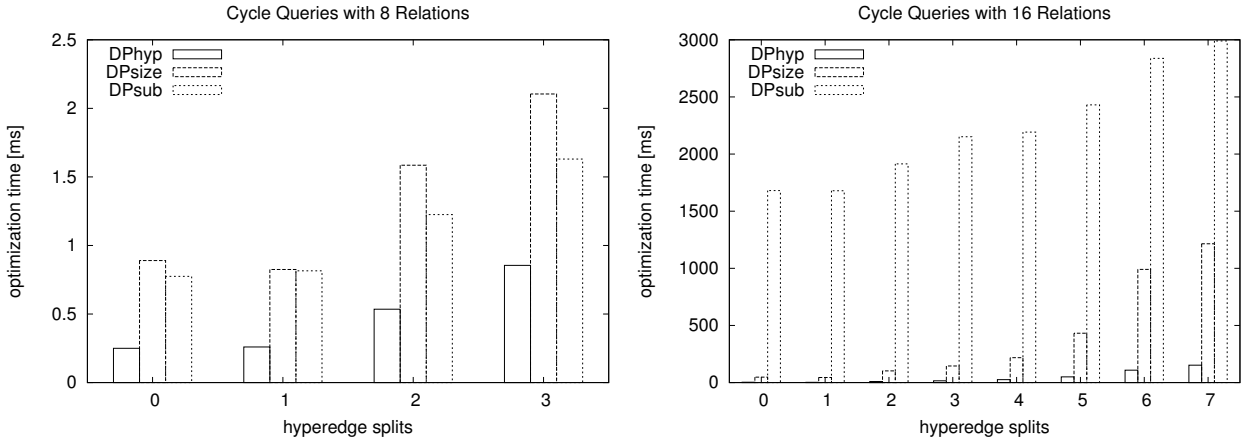

Figure 5: Results for cycle-based hypergraphs

one more derived hypergraph. Hence, we do not plot the runtimes but give them in tabular form. The runtime is given in milliseconds. The experiments were carried out on a PC with a 3.2 GHz Pentium D CPU. In the following table we show the result for cycles with 4 relations.

splits	DPhyp	DPsize	DPsub
0	0.02	0.035	0.035
1	0.025	0.025	0.025

Only small differences in runtime are observable here. This changes if we go to cycles with 8 and 16 relations. The cputimes in milliseconds are given in the graphs in Fig. 5. The first graph contains the results for cycles with 8 relations, the second one those for cycles with 16 relations. As we can see, in all cases DPhyp is superior to any of the other algorithms. Further, DPsize is superior to DPsub for large queries.

4.3 Star-Based Hypergraphs

Let us start by giving the results for star queries with four satellite relations in tabular form. The table is organized the same way as before.

splits	DPhyp	DPsize	DPsub
0	0.03	0.085	0.065
1	0.055	0.09	0.08

We already observe small runtime differences. For example, DPsize, which is used in commercial systems, is slower than DPhyp by a factor of almost two. Further, DPsub is slightly superior to DPsize, but less efficient than DPhyp. For larger star queries with eight and 16 satellite relations (see Fig. 6), these differences become rather huge. We observe that DPhyp is highly superior to DPsize and DPsub. Further, DPsub is superior to DPsize.

4.4 Queries with Regular Graphs

For completeness we also study the performance for regular graphs (i.e., simple hypergraphs without hyperedges), as these are more common in practice and DPhyp might have large constants than other approaches. But the results are similar to the hypergraph results (see Fig. 7), DPhyp is highly superior to DPsize and DPsub (note the logarithmic scale).

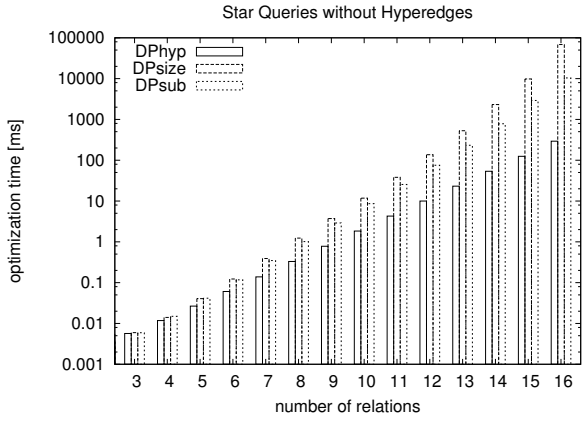

Figure 7: Results for star-based regular graphs

This is also true for other graph structures, where DPhyp performs exactly like DPccp on regular graphs.

5. NON-REORDERABLE OPERATORS

This section is organized as follows. We start with enumerating the set of binary operators which we handle. Then, we discuss their reorderability properties. Sec. 5.3 provides an overview of existing approaches. Problems occurring for non-commutative operators are discussed in Sec. 5.4. Hereafter, we introduce SESs and TESs, which capture possible conflicts among operators. Finally, we discuss issues concerning dependent joins and show how TESs can be used to generate the query hypergraph. An evaluation concludes this section.

5.1 Considered Binary Operators

Let us define the set of binary operators which we allow for in our plans. Besides the fully reorderable join ($\bowtie$), we also consider the following operators with limited reorderability capabilities: full outer join ($\fullouterjoin$), left outer join ($\leftouterjoin$), left antijoin ($\triangleright$), left semijoin ($\ltimes$), and left nestjoin ($\mathbin{\Gamma}$). Except for the nestjoin, these are standard operators. The nestjoin (also called binary grouping or MD-join) has been proposed

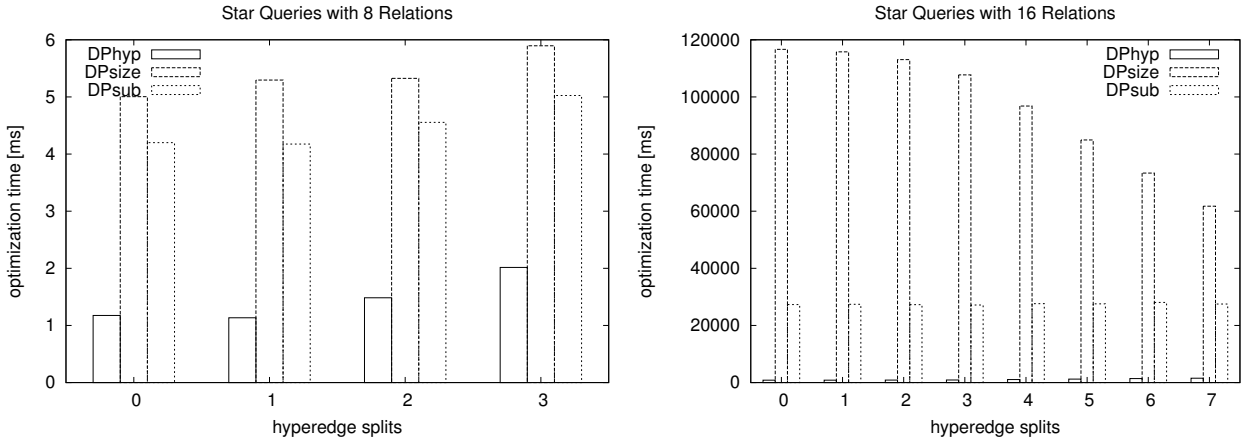

Figure 6: Results for star-based hypergraphs

to unnest nested queries in the object-oriented [6, 22], the relational [3], and the XML context [14]. It is further used to speed up data warehouse queries [5]. Since the various definitions of the nestjoin differ slightly, we use the most general one, from which all others can be derived by specialization. Let R and S be two relations, p a join predicate between them, a_i attribute names and e_i expressions in one free variable. Then we use the following definition of the nestjoin:

$$R \overset{g}{\bowtie}_{p;[a_1:e_1,\ldots,a_n:e_n]} S = \{r \circ s(r) | r \in R\}$$

with $s(r) = [a_1 : e_1(g(r)), \ldots, a_n : e_n(g(r))]$ and $g(r) = \{s | s \in S, p(r,s)\}$. Verbally, the operation can be described as follows. For every tuple $r \in R$, we collect all those tuples from S which successfully join with it. This gives $g(r)$. Then, the expressions e_i are evaluated with their free variable bound to $g(r)$. Often, e_i will consist of a single aggregate function call. Implementational issues for the nestjoin have been discussed in [5, 15].

Additionally to the above operators, we consider their dependent variants. Here, the evaluation of one side depends on the other side. Consider, for example, the left dependent join (or d-join for short) [6]. Let R be a relation and S an algebraic expression whose evaluation depends on R because it references attributes from R. Then, we define the d-join between R and S as follows:

$$R \bowtie^{d}_{p} S = \{r \circ s | r \in R, s \in S(r), p(r,s)\}$$

The d-join is very useful for table-valued functions with free variables [16], unnesting relational queries [9], object-oriented query processing [6], and XML query processing [4, 13, 14, 18].

It is straightforward to define the following dependent operators: left dependent join ($\overset{d}{\bowtie}$, d-join for short), dependent left outer join ($\overset{d}{⟕}$), dependent left antijion ($\overset{d}{▷}$), dependent left semijoin ($\overset{d}{⋉}$), and dependent left nestjoin ($\overset{g,d}{\bowtie}$). Again, different names have been supplied for these operators. For example, the d-join is sometimes called *[cross] apply* [9, 13, 18], and the dependent left outer join *outer apply* [13, 18].

Let $\mathcal{LOP}$ be the set of operators consisting of $\bowtie$, $▷$, $⋉$, $\overset{g}{\bowtie}$, $\overset{d}{\bowtie}$, $\overset{d}{⟕}$, $\overset{d}{▷}$, $\overset{d}{⋉}$, and $\overset{g,d}{\bowtie}$.

5.2 Reorderability

We will start with a definition that is at the core of what will be allowed in terms of reorderability and what will not.

Definition 5 (linear). *Let $\circ$ be a binary operator on relations. If for all relations S and T the following two conditions hold, then $\circ$ is called* left linear*:*

1. $\emptyset \circ T = \emptyset$ *and*
2. $(S_1 \cup S_2) \circ T = (S_1 \circ T) \cup (S_2 \circ T)$ *for all relations S_1 and S_2.*

Similarly, $\circ$ is called right linear *if*

1. $S \circ \emptyset = \emptyset$ *and*
2. $S \circ (T_1 \cup T_2) = (S \circ T_1) \cup (S \circ T_2)$ *for all relations T_1 and T_2.*

Observation 1. *All operators in $\mathcal{LOP}$ are left-linear, and $\bowtie$ is left- and right-linear.*

The full outer join is neither left- nor right-linear.

This observation simplifies the proofs of equivalences. We only have to prove that the operators are reorderable on single tuple relations. Before giving the reorderability results for our operators, we need some notation. Let S and T be two relation-valued algebraic expressions. Then we use the convention that a predicate p_{ST} references attributes from relations in S and T and no other relation.

We can now state the following equivalences.

Theorem 1 (Reorderability). *Let $\rightarrow^1$ and $\rightarrow^2$ be operators in $\mathcal{LOP}$. Then*

$$(R \rightarrow^1_{p_{RS}} S) \rightarrow^2_{p_{RT}} T = (R \rightarrow^2_{p_{RT}} T) \rightarrow^1_{p_{RS}} S \quad (2)$$

$$(R \bowtie_{p_{RS}} S) \rightarrow^2_{p_{ST}} T = R \bowtie_{p_{RS}} (S \rightarrow^2_{p_{ST}} T) \quad (3)$$

$$(R \bowtie_{p_{RS}} S) \rightarrow^2_{p_{RT}} T = S \bowtie_{p_{RS}} (R \rightarrow^2_{p_{RT}} T) \quad (4)$$

Another way to write the first equivalence by using the right variant of $\rightarrow^1$ is

$$(S \leftarrow^1_{p_{RS}} R) \rightarrow^2_{p_{RT}} T = S \leftarrow^1_{p_{RS}} (R \rightarrow^2_{p_{RT}} T)$$

With only very few exceptions, all valid reorderings are captured by the equivalences in the above theorem. Most of

these exceptions occur if the given expression can be simplified. For example, let the predicate p_{ST} be strong with respect to S.[1] Then

$$(R \bowtie_{p_{RS}} S) \bowtie_{p_{ST}} T = S \bowtie_{p_{RS}} (R \bowtie_{p_{RT}} T)$$

[11]. For this reason, we assume that all proposed simplifications [2, 11] have been applied. This is a typical assumption [19]. Another important assumption we make is that all predicates are strong on all tables. Predicates that are not strong are only reorderable if attached to regular joins. Hence, they can be treated by splitting query blocks [19]. Since the plan generator is called for each query block, we do not have to handle them.

5.3 Existing Approaches

A query (hyper-) graph alone does not capture the semantics of a query in a correct way [11]. What is needed is an initial operator tree equivalent to the query [19]. As mentioned, the initial operator tree has to be simplified. Then, our equivalences can be applied to derive all equivalent plans. Typically, not all valid reorderings will be equivalent to the original tree. Thus, any plan generation algorithm must be modified such that it restricts its search to valid reorderings. Several proposals to do so exist. For join trees with joins, left outer joins and full outer joins with predicates referencing only two relations, Galindo-Legaria and Rosenthal provided a procedure that analyzes paths in the query graph to detect conflicting reorderings [11]. Then, they modify a dynamic programming algorithm to take care of these conflicts. This approach was extended to conflict analysis with paths in hypergraphs [1]. As pointed out by Rao et al. there is a more efficient and easier to implement approach to deal with this problem [19]. They propose to compute a set of relations for every predicate that must be present in the arguments before the predicate can be evaluated. This set is called *extended eligibility list*, or EEL for short. Assume our algorithm enters `EmitCsgCmp` with sets S_1 and S_2 and the join predicates have an EEL E. Then, $E \subseteq S_1 \cup S_2$ must be checked. We could finish the current section at this point if there were not two problems. First, only regular joins, left outer joins, and antijoins are covered in [19]. Specifically, no dependent join operator is handled by their approach. Second, applying the test as late as in `EmitCsgCmp` results in enumerating csg-cmp-pair candidates, which will eventually fail. We would like to minimize this generation of irrelevant candidates.

5.4 Non-Commutative Operators

Only the join and the full outer join are commutative; all other operators are not. This requires some additional care. Consider, for example, the expression

$$(R_1 \bowtie_{p_1} R_2) \bowtie_{p_3} (R_3 \bowtie_{p_2} R_4)$$

Both $(\{R_1, R_2\}, \{R_3, R_4\})$ and $(\{R_3, R_4\}, \{R_1, R_2\})$ are valid csg-cmp-pairs. In order to build a plan for the pair $(\{R_3, R_4\}, \{R_1, R_2\})$, we must reestablish the fact that $\{R_3, R_4\}$ occurs on the right-hand side of a left outer join and build the plan accordingly. The same applies to $(\{R_1, R_2\}, \{R_3, R_4\})$. As a result, we would construct the same plan twice. Fortunately, our algorithm generates only csg-cmp-pairs (S_1, S_2) such that $S_1 < S_2$ if $<$ denotes lexicographical ordering among the sets of relations and is based on $\prec$, our ordering of single relations. To avoid the problem of reestablishing which part of a hyperedge occurred on the left-hand and which on the right-hand side, we order relations from left to right in the operator tree. That is, if R and S are two leaves in the operator tree and R occurs left of S, then $R \prec S$. Additionally, we associate with each hyperedge the operator from which it was derived. This operator can then be recovered by `EmitCsgCmp` to correctly build the plan.

[1] A predicate p is strong w.r.t. S if the fact that all attributes from S are NULL implies that p evaluates to false [11, 20].

5.5 Computing SESs and TESs

A procedure for calculating EELs bottom-up for operator trees containing joins, left outer joins and left antijoins is given in [19]. This approach is riddled with different cases and their complex interplay renders any extension impossible. Thus, we take a radically different approach by handling conflicts directly. Assume we have an expression $E = (R \circ_{p_1} S) \circ_{p_2} T$. Then we ask whether it is valid to transform E into $E' = R \circ_{p_1} (S \circ_{p_2} T)$. Similarly, given an expression E', we ask whether it can be safely reordered to an expression E. If we can detect a conflict, i.e. the reordering is invalid, then we report this. The problem is that we have to report conflicts for operators not only where one is a child of the other, but also for pairs of operators where one is a descendant of the other. To see why this is necessary, assume $\circ_2$ is a descendant operator in the left subtree of $\circ_1$. Then, due to valid reorderings of the left subtree of $\circ_1$, the operator $\circ_2$ might become a child of $\circ_1$. Then the conflict will count. During this rotation, all tables found on the right branches on the path from $\circ_2$ to $\circ_1$ in the original operator tree will be found in the right argument of $\circ_2$ in the reordered operator tree, where $\circ_2$ is a child of $\circ_1$. Our procedure will record conflicts for all pairs of operators where one is the descendant of the other.

Let us now formalize this approach. As usual, $\mathcal{F}(e)$ denotes the set of attributes occurring freely in an expression e, and $\mathcal{A}(R)$ denotes the set of attributes a relation R provides. For a set of attributes A, we denote by $\mathcal{T}(A)$ the set of tables to which these attributes belong. We abbreviate $\mathcal{T}(\mathcal{F}(e))$ by $\mathcal{F}_{\mathcal{T}}(e)$. Let $\circ$ be an operator in the operator tree. We denote by left($\circ$) (right($\circ$)) its left (right) successor. $\mathcal{STO}(\circ)$ denotes the operators under $\circ$ in the operator tree and $\mathcal{T}(\circ)$ the set of tables occurring in the subtree rooted at $\circ$, i.e. its leaves. Let $\circ_2$ be an operator in $\mathcal{STO}(\text{left}(\circ_1))$. Then we define RightTables$(\circ_1, \circ_2)$ as the union of $\mathcal{T}(\text{right}(\circ_3))$ for all $\circ_3$ on the path from $\circ_2$ (inclusive) to $\circ_1$ (exclusive). If $\circ_2$ is commutative, we add $\mathcal{T}(\text{left}(\circ_2))$ to RightTables$(\circ_1, \circ_2)$. Analogously, we define LeftTables$(\circ_1, \circ_2)$ in case $\circ_2 \in \mathcal{STO}(\text{right}(\circ_1))$.

The *syntactic eligibility set* (SES) is used to express the syntactic constraints: all referenced attributes/relations must be present before an expression can be evaluated. First of all, it contains the tables referenced by a predicate. Further, as we are also dealing with table functions and dependent join operators as well as nestjoins, we need the following extensions. Let R be a relation, T a table-valued function call, $\circ_p$ any join except a nestjoin, and $nj \in \{ \ltimes, \triangleright \}$. Then, we

define:

$$\begin{aligned}
\mathcal{SES}(R) &= \{R\} \\
\mathcal{SES}(T) &= \{T\} \\
\mathcal{SES}(\circ_p) &= \bigcup_{R \in \mathcal{F}_\mathcal{T}(p)} \mathcal{SES}(R) \cap \mathcal{T}(\circ_p) \\
\mathcal{SES}(nl_{p,[a_1:e_1,\ldots,a_n:e_n]}) &= \bigcup_{R \in \mathcal{F}_\mathcal{T}(p) \cup \mathcal{F}_\mathcal{T}(e_i)} \mathcal{SES}(R) \cap \mathcal{T}(nl)
\end{aligned}$$

We illustrate these definitions in the next subsection, where we discuss the handling of dependent join operations.

The total eligibility set ($\mathcal{TES}$) – to be introduced next – captures the syntactic constraints *and* additional reorderability constraints. Assume the operator tree contains two operators $\circ_1$ and $\circ_2$, where $\circ_2 \in \mathcal{STO}(\circ_1)$. If they are not reorderable, i.e. there occurs a *conflict*, this is expressed by adding the $\mathcal{TES}$ of $\circ_2$ to the $\mathcal{TES}$ of $\circ_1$.

After initializing $\mathcal{TES}(\circ)$ with $\mathcal{SES}(\circ)$ for every operator $\circ$, the following procedure is called bottom-up for every operator to complete the calculation of $\mathcal{TES}(\circ)$.

CalcTES($\circ_{p_1}$) // operator $\circ_1$ and its predicate p_1
  for $\forall\ \circ_{p_2} \in \mathcal{STO}(\text{left}(\circ_{p_1}))$
    if `LeftConflict`$((\circ_{p_2}), \circ_{p_1})$ // add $\circ_{p_2} < \circ_{p_1}$
      $\mathcal{TES}(\circ_{p_1}) = \mathcal{TES}(\circ_{p_1}) \cup \mathcal{TES}(\circ_{p_2})$
  for $\forall\ \circ_{p_2} \in \mathcal{STO}(\text{right}(\circ_{p_1}))$
    if `RightConflict`$(\circ_{p_1}, (\circ_{p_2}))$ // add $\circ_{p_2} < \circ_{p_1}$
      $\mathcal{TES}(\circ_{p_1}) = \mathcal{TES}(\circ_{p_1}) \cup \mathcal{TES}(\circ_{p_2})$
  for $\forall\ \fallingdotseq_{p',[a_i:e_i]} \in \mathcal{STO}(\circ_{p_1})$
    if $\exists a_i : a_i \in \mathcal{F}(p_1)$ // add $\fallingdotseq_{p',[a_i:e_i]} < \circ_{p_1}$
      $\mathcal{TES}(\circ_{p_1}) = \mathcal{TES}(\circ_{p_1}) \cup \mathcal{TES}(\fallingdotseq_{p',[a_i:e_i]})$

where

$$\begin{aligned}
\texttt{LeftConflict}((\circ_{p_2}), \circ_{p_1})) &= \text{LC} \wedge \text{OC}(\circ_{p_2}, \circ_{p_1}) \\
\texttt{RightConflict}(\circ_{p_1}, (\circ_{p_2})) &= \text{RC} \wedge \text{OC}(\circ_{p_1}, \circ_{p_2})
\end{aligned}$$

and

$$\begin{aligned}
\text{LC}((\circ_{p_2}), \circ_{p_1}) &= \mathcal{F}_\mathcal{T}(p_1) \cap \text{RightTables}(\circ_{p_1}, \circ_{p_2}) \neq \emptyset \\
\text{RC}(\circ_{p_1}, (\circ_{p_2})) &= \mathcal{F}_\mathcal{T}(p_1) \cap \text{LeftTables}(\circ_{p_1}, \circ_{p_2}) \neq \emptyset \\
\text{OC}(\circ_1, \circ_2) &= (\circ_1 = \bowtie \wedge \circ_2 = \fallingdotseq) \vee (\circ_1 \neq \bowtie \wedge \\
&\quad \neg(\circ_1 = \circ_2 = \fallingdotseq) \\
&\quad \wedge \neg(\circ_1 = \fallingdotseq \wedge \circ_2 \in \{\bowtie, \fallingdotseq\}))
\end{aligned}$$

where each operator also stands for its dependent counterpart. We show of the derivations of these conditions here in the appendix.

5.6 Dependent Join Operators

When reordering dependent joins, some care is required, as can been seen in the following equivalences:

$$\begin{aligned}
R \bowtie_{p_{RS}} (S(R) \bowtie_{p_{ST}} T(R)) &= (R \bowtie_{p_{RS}} S(R)) \bowtie_{p_{ST}} T \\
R \bowtie_{p_{RS}} (S \bowtie_{p_{ST}} T(R)) &= (R \bowtie_{p_{RS}} S) \bowtie_{p_{ST}} T(R)
\end{aligned}$$

In the first equivalence, the join between S and T on the left-hand side must be turned into a dependent join on the right-hand side. In the second equivalence, the first dependent join between R and S becomes a regular join between R and S on the right-hand side and the regular join between S and T on the left-hand side becomes a dependent join on the right-hand side.

The general decision of whether to use a dependent or regular join (semijoin, antijoin, ...) can be made rather simple due to the numbering and enumeration properties of our algorithm discussed in Sec. 5.4. We attach only regular binary operators with hyperedges. When a hyperedge is used by `EmitCsgCmp` to generate a plan, we retrieve this operator. Then, `EmitCsgCmp` has to turn it into its dependent counterpart if and only if the following condition holds:

$$\mathcal{F}_\mathcal{T}(P_2) \cap S_1 \neq \emptyset$$

where P_2 is the best plan for S_2.

5.7 Faster TESs Handling Using Hypergraphs

Note the following: if the hypernodes in the hyperedges of the query graphs become larger, the search space decreases. We could use $\mathcal{TES}$ directly to test for conflicts in `EmitCsgCmp`, as described. However, taking the introductory statement into account, we use $\mathcal{TES}$ to construct the hypergraph, which then serves as the input to our algorithm. For every operator $\circ$, we construct a hyperedge (l, r) such that

$$r = \mathcal{TES}(\circ) \cap \mathcal{T}(\text{right}(\circ))$$

and

$$l = \mathcal{TES}(\circ) \setminus r$$

Again, it is more efficient, as the hyperedges directly cover all possible conflicts. Note that this significantly reduces the search space. Even for the relatively simple example of a star query of anti-joins, the explored search space is reduced from $O(n^2)$ to $O(n)$, the runtime from $O(n^3)$ to $O(n)$. The hypergraph formulation greatly speeds up the handling of non-inner joins.

5.8 Evaluation

We ran several experiments to evaluate the different algorithms under different settings. Due to lack of space, we selected two typical experiments.

In the first experiment, we wanted to answer the question how much we benefit from the search space reduction in Sec. 5.7. We compared a generate-and-test paradigm using TESs with deriving hypergraphs from TESs. We construct a left-deep operator tree for a star query with 16 relations, with an increasing number of antijoins. Thus, the search space size decreases over time, as the antijoins are more restrictive than inner joins. The results are shown in Fig. 8a. For both approaches the optimization time decreases as the search space shrinks, but the hypergraph performs much better. The reason is that a TES-test-based approach generates many plans which have to be discarded, while the hypergraph-based formulation can avoid generating them. This shows that hypergraphs can greatly reduce the optimization time when handling non-inner joins, even though the original query does not induce a hypergraph.

Antijoins are very restrictive. Hence, the relevant search space shrinks quite fast. Outer joins are more interesting, as they can be reordered relatively to each other, which increases the search space again. To study this effect and to get a better comparison with the other algorithms, we construct a cycle query with 16 relations similar to the star query above, and replaced inner joins with outer joins. Note that cycle queries are very favorable for `DPsize`. `DPsub` is so

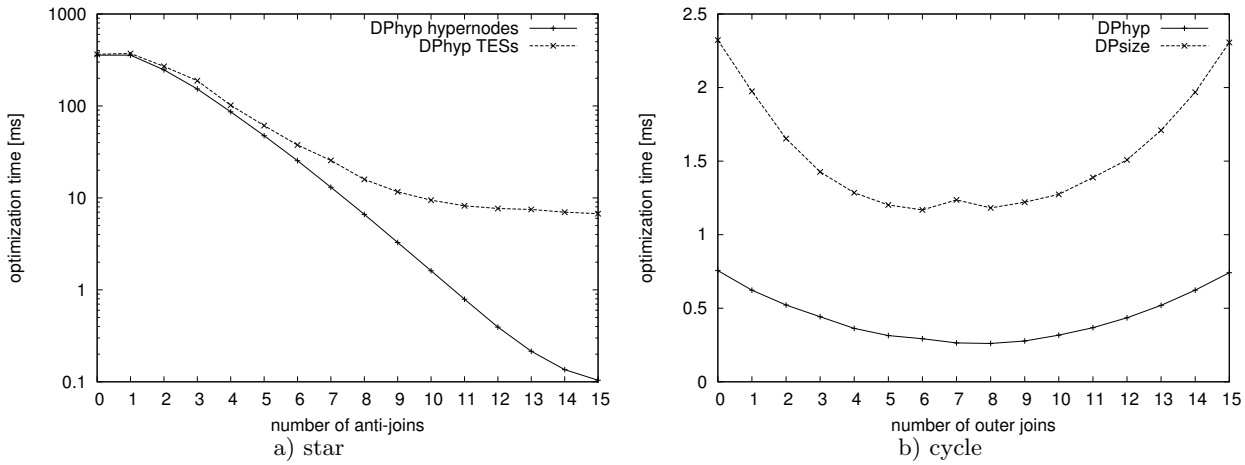

Figure 8: Star and Cycle Query with 16 relations

slow that we excluded it (> 1400 ms). The results are shown in Fig. 8b. The runtime decreases at first, as the outer joins cannot be reordered with inner joins. As the number of outer joins increases, the search space increases again as the outer joins are associative. Both algorithms benefit from the search space reduction, but `DPhyp` is clearly faster than `DPsize` in all cases. Apparently, `DPhyp` profits even more from the reduced search space than `DPsize`, as the ratio between slowest and fastest optimization is ≈ 2.88 for `DPhyp` and ≈ 1.96 for `DPsize`.

6. TRANSLATION OF JOIN PREDICATES

As mentioned in Section 2, hypergraph edges are formed by using the relations from both sides of the join condition as edge anchors. For example the join predicate

$$f_1(R_1.a, R_2.b, R_3.c) = f_2(R_4.d, R_5.e, R_6.f)$$

forms the hyperedge

$$(\{R_1, R_2, R_3\}, \{R_4, R_5, R_6\}).$$

But for some predicates this construction is not as straightforward. For example, the very similar join predicate $R_1.a + R_2.b + R_3.c = R_4.d + R_5.e + R_6.f$ could be translated into the same hyperedge, but also to other hyperedges like

$$(\{R_1, R_2\}, \{R_3, R_4, R_5, R_6\}),$$

as R_3 could be moved to the other side of the equation. Finally, some predicates do not have an inherent ordering, like $f(R_1.a, R_2.b, R_3.c) = true$. Note that while the previous cases could be subsumed under the last form of predicates, it is not desirable to do so, as it implies a nested loop evaluation.

In general, the relations involved in a join predicate can be classified into three groups: Those that must appear on one side of the join, those that must appear on the other side of the join and those that can appear on any of the two. To simplify the already not overly intuitive discussion about hypergraph edges, our hypergraph definition in Section 2 could not express this degree of freedom caused by the relations in the third group. After explaining all the mechanisms, we can now generalize the hypergraphs to include this freedom.

DEFINITION 6 (GENERALIZED HYPERGRAPH). *A* generalized 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 a triple (u, v, w) of non-empty subsets of V ($u \subset V$ and $v \subset V$) with the additional condition that u, v and w are pairwise disjoint.*

We call any non-empty subset of V a hypernode. *A hyperedge (u, v, w) is* simple *if $|u| = |v| = 1 \wedge |w| = 0$. A generalized hypergraph is* simple *if all its hyperedges are simple.*

DEFINITION 7 (CONNECTED HYPERNODES). *Two hypernodes V_1, V_2 in a generalized hypergraph $H = (V, E)$ are connected if $\exists (u, v, w) \in E$ such that $u \subseteq V_1 \wedge v \subseteq V_2 \wedge w \subseteq (V_1 \cup V_2)$ or $u \subseteq V_2 \wedge v \subseteq V_1 \wedge w \subseteq (V_1 \cup V_2)$.*

Intuitively, the triple (u, v, w) connects all nodes in u with all nodes in v, where the nodes in w can appear on any side of the edge. All other definitions follow analogously. Note that while these generalized hypergraphs are difficult to visualize for humans, they are easy to use in practice and the previously described algorithms require no changes. In particular, following such a hyperedge (e.g., for neighborhood computation) is simple, as one side of the hyperedge is known: Given a hypernode V_1 and an edge (u, v, w) such that $v \subseteq V_1$, the neighbouring hypernode V_2 must be $v \cup (w \setminus V_1)$. This makes use of the fact that we create a join tree, i.e., that V_1 and V_2 must be disjoint.

Overall, the usage of generalized hypergraphs does not complicate the optimization algorithm. It is interesting to note, though, that the generalized hypergraph interacts with the non-reorderable operators. The initial edges (u, v, w) can be derived directly from the join predicates, where the w part implies degrees of freedom. When handling non-reorderable joins, the hyperedge computation from Section 5.7 places some relations explicitly on separate sides of a join. Thus, initially unordered relations from w can be moved to u or v due to reorderability constraints. As a consequence, the search space shrinks, as the resulting hyperedge is more restrictive. This illustrates why this relatively complex triple form is required: Using only pairs of hypernodes is too restrictive for some predicates, while considering hyperedges

as connecting an unordered set of nodes (as is sometimes done for hypergraphs) is wasteful for the search space. By combining them, we can both maintain expressiveness and preserve an efficient exploration of the search space.

7. RELATED WORK

We already discussed closely related approaches in Section 5.3. Hence, we give only a brief overview here. While there exist many algorithms for ordering inner joins (see [16] for an overview), there exist only very few to deal with other join operators and hypergraphs.

The basic idea of using csg-cmp-pairs for join enumeration for simple graphs was published in [17]. DeHaan and Tompa used the same idea to formulate a top down algorithm [7]. In both cases, neither hypergraphs nor operators other than inner joins have been considered.

Galindo-Legaria and Rosenthal extend `DPsize` to deal with full and left outer joins [11]. They extend `DPsize` by incorporating a conflict analysis, which analyzes paths in the query graph to detect conflicting join operators. However, the extension to hypergraphs was left to Bhargava et al. [1]. The main idea here is to analyze paths in a hypergraph to detect possible conflicts.

A much simpler ordering test using EELs has been proposed by Rao et al. [19]. It performs a bottom-up traversal of the initial operator tree and builds relation dependencies, handling left outer joins and antijoins. They extend `DPsize` with an EEL test much in the same way as our first (less efficient) alternative (see Sec. 5.8). A more thorough discussion of their approach and the differences to our approach can be found in Sec. 5.3 and 5.5.

8. CONCLUSION

We presented `DPhyp`, a join enumeration algorithm capable of handling hypergraphs and a much wider class of join operators than previous approaches. The extension to hypergraphs enables us to optimize queries with non-inner joins much more efficiently than before, even for queries with binary join predicates.

Although our algorithm is way the fastest competitor for join ordering for complex queries, there is still plenty of room for future research. First, the generation of csg-cmp-pairs still does some *generate-and-test*. It will be interesting to see whether connected subgraphs of hypergraphs can be generated without any tests. Further, compensation is a means to allow for more reordering if there is a conflict [1, 11, 19]. Our algorithm does not incorporate compensation. Thus, this is a natural next step to consider. Lately, a new approach for a top-down join enumeration algorithm has been proposed by DeHaan and Tompa [7]. It is only a linear factor apart from the optimal solution and thus highly superior to existing top-down join enumeration algorithms. It suffers from the same issues as `DPccp` did, namely, no hypergraph and no outer join support. It will be interesting to see how their algorithm can be extended to deal with these issues.

Acknowledgement. We thank Guy Lohman for pointing out the importance of hypergraphs, Simone Seeger for her help preparing the manuscript, and Vasilis Vassalos and the anonymous referees for their help to improve the readability of the paper.

9. REPEATABILITY ASSESSMENT RESULT

All the results in this paper were verified by the SIGMOD repeatability committee.

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] G. Bhargava, P. Goel, and B. Iyer. Simplification of outer joins. In *CASCOM*, 1995.

[3] S. Bitzer. Design and implementation of a query unnesting module in Natix. Master's thesis, U. of Mannheim, 2007.

[4] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. In *ICDE*, pages 705–716, 2005.

[5] D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-Join: An Operator for Complex OLAP. In *ICDE*, pages 524–533, 2001.

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

[7] D. DeHaan and F. Tompa. Optimal top-down join enumeration. In *SIGMOD*, pages 785–796, 2007.

[8] C. Galindo-Legaria. *Outerjoin Simplification and Reordering for Query Optimization*. PhD thesis, Harvard University, 1992.

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

[10] 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.

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

[12] P. Gassner, G. Lohman, and K. Schiefer. Query optimization in the IBM DB2 family. *IEEE Data Engineering Bulletin*, 16:4–18, Dec. 1993.

[13] E. Kogan, G. Schaller, M. Rys, H. Huu, and B. Krishnaswarmy. Optimizing runtime XML processing in relational databases. In *XSym*, pages 222–236, 2005.

[14] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *TODS*, 31(3):968–1013, 2006.

[15] N. May and G. Moerkotte. Main memory implementations for binary grouping. In *XSym*, pages 162–176, 2005.

[16] G. Moerkotte. Building query compilers. available at db.informatik.uni-mannheim.de/moerkotte.html.en, 2006.

[17] 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.

[18] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, and E. K. D. Churin. Xquery implementation in a relational database system. In *VLDB*, pages 1175–1186, 2005.

[19] 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. IBM Tech. Rep. RJ 10203.

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

[21] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[22] H. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, University of Twente, 1995.

[23] J. Ullman. *Database and Knowledge Base Systems*, volume Volume 2. Computer Science Press, 1989.

[24] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *SIGMOD*, pages 35–46, 1996.

APPENDIX

A. EQUIVALENCES AND CONFLICT RULES

In Section 5.5 we needed conflcit rules for all considered operators. The following two subsections provide an overview over all equivalences and conflict rules. The first subsection deals with operators nested on the left side, the second with operators nested on the right side. Note that this is redundant, but it is better for an overview of the situation. There is a parenthesized comment on every equivalence or conflict rules. Within these comments, you sometimes find p_{ST} (p_{RS}) strong. This has to be read as p_{ST} (p_{RS}) strong with respect to S. Further, the numbers 4.?? refer to equivalences in Galindo-Legarias thesis [8].

A.1 Equivalences and Conflict Rules for Left Nesting

Let R, S, and T be arbitrary algebraic expression in our operators. Then, assume that we have an expression E

$$(R \circ_{p_1} S) \circ_{p_2} T$$

and would like to reorder it to E', which looks as follows:

$$R \circ_{p_1} (S \circ_{p_2} T)$$

For our original expression E, we observe the following:

$$\begin{aligned} \mathcal{F}_{\mathcal{T}}(T) &\subseteq \mathcal{T}(R) \cup \mathcal{T}(S) \\ \mathcal{F}_{\mathcal{T}}(S) &\subseteq \mathcal{T}(R) \\ \mathcal{F}_{\mathcal{T}}(p_1) &\subseteq \mathcal{T}(R) \cup \mathcal{T}(S) \\ \mathcal{F}_{\mathcal{T}}(p_2) &\subseteq \mathcal{T}(R) \cup \mathcal{T}(S) \cup \mathcal{T}(R) \\ \mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(T) &= \emptyset \end{aligned}$$

These are syntactic constraints, that must be covered.

If

$$\mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(R) \neq \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(S) \neq \emptyset$$

then reodering E to E' is not possible. However, this is also covered by our syntactic constraints $\mathcal{SES}$.

If

$$\mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(R) = \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(S) = \emptyset$$

then p_2 can be a constant predicate like *true* or *false*. The first case can lead to e.g. cross products. In the second case, simplifications apply. If neither is the case, then the predicate might reference attributes outside the arguments. This kind of algebraic expression results from some nested queries [6]. Anyway, in this paper we do not consider the case that $\mathcal{F}_{\mathcal{T}}(p_2)$ has no intersection with any argument relations.

Hence, we have to consider only two cases here:

L1 $\mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(R) \neq \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(S) = \emptyset$

L2 $\mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(R) = \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(S) \neq \emptyset$

Case L1 allows for free reorderability, if no full outerjoin is present (see Theorem 1). We will consider Case L2 below.

For commutative operators ($\bowtie$, $\between$), Case L1 can be recast to Case L2 by normalzing the operator tree (prior to calculating $\mathcal{SES}$ and $\mathcal{TES}$) by demanding that

$$\mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(S) \neq \emptyset$$

for all commutative operators $\circ_{p_1}$, which occur on the left under some other operator. After this step, all possible conflicts are of Case L2.

Fig. 9 contains a complete listing of valid and invalid cases for reordering E to E'. It follows from these equivalences and conflict rules, that we can safely detect conflicts of reorderability for E, if we check

$$\begin{aligned} \text{L2} \wedge ((&\circ_{p_1} = \bowtie \wedge \circ_{p_2} = \between) \\ &\vee (\circ_{p_1} \neq \bowtie \\ &\quad \wedge (\ \neg(\circ_{p_1} = \between \wedge \circ_{p_2} = \between) \\ &\qquad \wedge \ \ \neg(\circ_{p_1} = \between \wedge \circ_{p_2} \in \{\bowtie, \between\}))))\end{aligned}$$

Then, we can reorder E to E' if and only if the above condition does not return *true*, i.e. returns *false*.

A.2 Equivalences and Conflict Rules for Right Nesting

Let R, S, and T be arbitrary algebraic expression in our operators. Then, assume that we have an expression E

$$R \circ_{p_1} (S \circ_{p_2} T)$$

and would like to reorder it to E', which is defined as follows:

$$(R \circ_{p_1} S) \circ_{p_2} T$$

For our original expression E, we observe the following:

$$\begin{aligned} \mathcal{F}_{\mathcal{T}}(S) &\subseteq \mathcal{T}(R) \\ \mathcal{F}_{\mathcal{T}}(T) &\subseteq \mathcal{T}(R) \cup \mathcal{T}(S) \\ \mathcal{F}_{\mathcal{T}}(p_1) &\subseteq \mathcal{T}(R) \cup \mathcal{T}(S) \cup \mathcal{T}(R) \\ \mathcal{F}_{\mathcal{T}}(p_2) &\subseteq \mathcal{T}(S) \cup \mathcal{T}(R) \\ \mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(R) &= \emptyset \end{aligned}$$

These are syntactic constraints, that must be covered. If

$$\mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(S) \neq \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(T) \neq \emptyset$$

then reodering E to E' is not possible. However, this is also covered by our syntactic constraints $\mathcal{SES}$. Again, we do not consider the case where

$$\mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(R) = \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_2) \cap \mathcal{T}(S) = \emptyset$$

here.

Hence, we have to consider only two cases here:

R1 $\mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(S) = \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(T) \neq \emptyset$

R2 $\mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(S) \neq \emptyset \ \wedge \ \mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(T) = \emptyset$

$(R \bowtie_{p_{RS}} S) \bowtie_{p_{ST}} T$	$=$	$R \bowtie_{p_{RS}} (S \bowtie_{p_{ST}} T)$	(join associativity), 4.44
$(R \ltimes_{p_{RS}} S) \bowtie_{p_{ST}} T$	$\neq$	$R \ltimes_{p_{RS}} (S \bowtie_{p_{ST}} T)$	(lhs not possible)
$(R \rhd_{p_{RS}} S) \bowtie_{p_{ST}} T$	$\neq$	$R \rhd_{p_{RS}} (S \bowtie_{p_{ST}} T)$	(lhs not possible)
$(R \,\bar{\ltimes}_{p_{RS}} S) \bowtie_{p_{ST}} T$	$\neq$	$R \,\bar{\ltimes}_{p_{RS}} (S \bowtie_{p_{ST}} T)$	(lhs not possible)
$(R \bowtie_{p_{RS}} S) \bowtie_{p_{ST}} T$	$\neq$	$R \bowtie_{p_{RS}} (S \bowtie_{p_{ST}} T)$	(false, lhs simplifiable if p_{ST} strong, 4.48)
$(R \mathbin{⟗}_{p_{RS}} S) \bowtie_{p_{ST}} T$	$\neq$	$R \mathbin{⟗}_{p_{RS}} (S \bowtie_{p_{ST}} T)$	(false, lhs simplifiable, GOJ 4.54)
$(R \bowtie_{p_{RS}} S) \ltimes_{p_{ST}} T$	$=$	$R \bowtie_{p_{RS}} (S \ltimes_{p_{ST}} T)$	(linearity, 4.44)
$(R \ltimes_{p_{RS}} S) \ltimes_{p_{ST}} T$	$\neq$	$R \ltimes_{p_{RS}} (S \ltimes_{p_{ST}} T)$	(lhs not possible)
$(R \rhd_{p_{RS}} S) \ltimes_{p_{ST}} T$	$\neq$	$R \rhd_{p_{RS}} (S \ltimes_{p_{ST}} T)$	(lhs not possible)
$(R \,\bar{\ltimes}_{p_{RS}} S) \ltimes_{p_{ST}} T$	$\neq$	$R \,\bar{\ltimes}_{p_{RS}} (S \ltimes_{p_{ST}} T)$	(lhs not possible)
$(R \bowtie_{p_{RS}} S) \ltimes_{p_{ST}} T$	$\neq$	$R \bowtie_{p_{RS}} (S \ltimes_{p_{ST}} T)$	(false, lhs simplifiable if p_{ST} strong, 4.48)
$(R \mathbin{⟗}_{p_{RS}} S) \ltimes_{p_{ST}} T$	$\neq$	$R \mathbin{⟗}_{p_{RS}} (S \ltimes_{p_{ST}} T)$	(false)
$(R \bowtie_{p_{RS}} S) \rhd_{p_{ST}} T$	$=$	$R \bowtie_{p_{RS}} (S \rhd_{p_{ST}} T)$	(linearity)
$(R \ltimes_{p_{RS}} S) \rhd_{p_{ST}} T$	$\neq$	$R \ltimes_{p_{RS}} (S \rhd_{p_{ST}} T)$	(lhs not possible)
$(R \rhd_{p_{RS}} S) \rhd_{p_{ST}} T$	$\neq$	$R \rhd_{p_{RS}} (S \rhd_{p_{ST}} T)$	(lhs not possible)
$(R \,\bar{\ltimes}_{p_{RS}} S) \rhd_{p_{ST}} T$	$\neq$	$R \,\bar{\ltimes}_{p_{RS}} (S \rhd_{p_{ST}} T)$	(lhs not possible)
$(R \bowtie_{p_{RS}} S) \rhd_{p_{ST}} T$	$\neq$	$R \bowtie_{p_{RS}} (S \rhd_{p_{ST}} T)$	(false)
$(R \mathbin{⟗}_{p_{RS}} S) \rhd_{p_{ST}} T$	$\neq$	$R \mathbin{⟗}_{p_{RS}} (S \rhd_{p_{ST}} T)$	(false)
$(R \bowtie_{p_{RS}} S) \,\bar{\ltimes}_{p_{ST}} T$	$=$	$R \bowtie_{p_{RS}} (S \,\bar{\ltimes}_{p_{ST}} T)$	(linearity)
$(R \ltimes_{p_{RS}} S) \,\bar{\ltimes}_{p_{ST}} T$	$\neq$	$R \ltimes_{p_{RS}} (S \,\bar{\ltimes}_{p_{ST}} T)$	(lhs not possible)
$(R \rhd_{p_{RS}} S) \,\bar{\ltimes}_{p_{ST}} T$	$\neq$	$R \rhd_{p_{RS}} (S \,\bar{\ltimes}_{p_{ST}} T)$	(lhs not possible)
$(R \,\bar{\ltimes}_{p_{RS}} S) \,\bar{\ltimes}_{p_{ST}} T$	$\neq$	$R \,\bar{\ltimes}_{p_{RS}} (S \,\bar{\ltimes}_{p_{ST}} T)$	(lhs not possible)
$(R \bowtie_{p_{RS}} S) \,\bar{\ltimes}_{p_{ST}} T$	$\neq$	$R \bowtie_{p_{RS}} (S \,\bar{\ltimes}_{p_{ST}} T)$	(false)
$(R \mathbin{⟗}_{p_{RS}} S) \,\bar{\ltimes}_{p_{ST}} T$	$\neq$	$R \mathbin{⟗}_{p_{RS}} (S \,\bar{\ltimes}_{p_{ST}} T)$	(false)
$(R \bowtie_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$=$	$R \bowtie_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(linearity, 4.45)
$(R \ltimes_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \ltimes_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(lhs not possible)
$(R \rhd_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \rhd_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(lhs not possible)
$(R \,\bar{\ltimes}_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \,\bar{\ltimes}_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(lhs not possible)
$(R \bowtie_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$=$	$R \bowtie_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(extra, if p_{ST} strong, 4.46)
$(R \mathbin{⟗}_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$=$	$R \mathbin{⟗}_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(if p_{ST} strong, 4.51)
$(R \bowtie_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \bowtie_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(false, GOJ 4.54)
$(R \ltimes_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \ltimes_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(lhs not possible)
$(R \rhd_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \rhd_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(lhs not possible)
$(R \,\bar{\ltimes}_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \,\bar{\ltimes}_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(lhs not possible)
$(R \bowtie_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$\neq$	$R \bowtie_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(false)
$(R \mathbin{⟗}_{p_{RS}} S) \mathbin{⟗}_{p_{ST}} T$	$=$	$R \mathbin{⟗}_{p_{RS}} (S \mathbin{⟗}_{p_{ST}} T)$	(if p_{ST} and p_{RS} strong, 4.50)

Figure 9: Equivalences and conflict rules for left nesting

Case R1 allows for free reorderability, if no full outerjoin is present (see Theorem 1). We will consider Case R2 below.

For commutative operators ($\bowtie$, $\mathbin{⟗}$), Case R1 can be recast to Case R2 by normalzing the operator tree (prior to calculating $\mathcal{SES}$ and $\mathcal{TES}$) by demanding that

$$\mathcal{F}_{\mathcal{T}}(p_1) \cap \mathcal{T}(S) \neq \emptyset$$

for all commutative operators $\circ_{p_2}$, which occur on the right under some other operator. After this step, all possible conflicts are of Case R2.

For space reasons we ommit the table for the right nesting, it is symmetric to Fig 9. It follows from these equivalences and conflict rules, that we can safely detect conflicts of reorderability for E, if we check

$$\begin{aligned}
&\text{R2} \wedge ((\circ_{p_1} = \bowtie \wedge \circ_{p_2} = \mathbin{⟗}) \\
&\quad \vee (\circ_{p_1} \neq \bowtie \\
&\qquad \wedge (\quad \neg(\circ_{p_1} = \mathbin{⟗} \wedge \circ_{p_2} = \mathbin{⟗}) \\
&\qquad\quad \wedge \quad \neg(\circ_{p_1} = \mathbin{⟗} \wedge \circ_{p_2} \in \{\mathbin{⟗}, \mathbin{⟗}\}))))
\end{aligned}$$

Then, we can reorder E to E' if and only if the above condition does not return *true*, i.e. returns *false*.

A.3 Summary

Note that the conditions at the end of the preceding subsections differ only in L2 and R2. Hence, we can factorize these conditions into a condition $\text{OC}(\circ_1, \circ_2)$ which is defined as

$$\begin{aligned}
\text{OC}(\circ_1, \circ_2) \quad = \quad & (\circ_1 = \bowtie \wedge \circ_2 = \mathbin{⟗}) \vee (\circ_1 \neq \bowtie \wedge \\
& \neg(\circ_1 = \circ_2 = \mathbin{⟗}) \\
& \wedge \neg(\circ_1 = \mathbin{⟗} \wedge \circ_2 \in \{\mathbin{⟗}, \mathbin{⟗}\}))
\end{aligned}$$