Top Down Plan Generation: From Theory to Practice

Pit Fender [1], Guido Moerkotte [2]

University of Mannheim
Mannheim, Germany
[1]pfender@informatik.uni-mannheim.de
[2]moerkotte@informatik.uni-mannheim.de

***Abstract*— Finding the optimal execution order of join operations is a crucial task of today's cost-based query optimizers. There are two approaches to identify the best plan: bottom-up and top-down join enumeration. But only the top-down approach allows for branch-and-bound pruning, which can improve compile time by several orders of magnitude while still preserving optimality. For both optimization strategies, efficient enumeration algorithms have been published. However, there are two severe limitations for the top-down approach: The published algorithms can handle only (1) simple (binary) join predicates and (2) inner joins. Since real queries may contain complex join predicates involving more than two relations, and outer joins as well as other non-inner joins, efficient top-down join enumeration cannot be used in practice yet. We develop a novel top-down join enumeration algorithm that overcomes these two limitations. Furthermore, we show that our new algorithm is competitive when compared with the state of the art in bottom-up processing even without playing out its advantage by making use of its branch-and-bound pruning capabilities.**

I. Introduction

For a DBMS that provides support for a declarative query language like SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows it to be translated into many equivalent evaluation plans. The process of choosing a suitable plan from all alternatives is known as query optimization. The basis of this choice are a cost model and statistics over the data. Essential for the costs of a plan is the execution order of join operations in its operator tree, since the runtime of plans with different join orders can vary by several orders of magnitude. An exhaustive search for an optimal solution over all possible operator trees is computationally infeasible. To decrease complexity, the search space must be restricted. For the optimization problem discussed in this document, a well-accepted heuristic is applied: We consider all possible bushy join trees [1], but exclude cross products from the search, presuming that all considered queries span a connected query graph [1].

There are two principle approaches to find an optimal join order: bottom-up join enumeration via dynamic programming and top-down join enumeration through memoization. Both approaches face the same challenge: to efficiently find for a given set of relations all partitions into two subsets, such that both induce connected subgraphs and there exists an edge connecting the two subgraphs.

Currently, the following algorithms have been proposed in the literature: Moerkotte and Neumann [2] presented an efficient dynamic programming-based algorithm (DPccp). Fender and Moerkotte [3], [4], [5] proposed two equally competitive top-down join enumeration strategies (TDMinCutBranch and TDMinCutConservative), which are almost as efficient as DPccp. However, all three algorithms, DPccp, TDMinCutBranch and TDMinCutConservative are not ready yet to be used in real world scenarios because there exist two severe deficiencies in all of them. First, as has been argued in several places, hypergraphs must be handled by any plan generator [6], [7], [8]. Second, plan generators have to deal with outer joins and antijoins [9], [7]. These operators are, in general, not freely reorderable, i.e., there might exist different orderings, which produce different results. Because it has been shown that the non-inner join reordering problem can be reduced to hypergraphs, it remains the top goal of any plan generator to deal with hypergraphs [6], [10], [7]. Thus, Moerkotte and Neumann [10] extended DPccp to DPHyp to handle hypergraphs. Since DPHyp is a bottom-up join enumeration algorithm, it cannot benefit from branch-and-bound pruning. On the other hand, branch-and-bound pruning can significantly speed up plan generation [11], [4]. Therefore, we introduce a novel algorithm called TDMcCHyp (short for TDMinCutConservativeHyp) that is the first available plan generation algorithm which works top-down and is able to deal with hypergraphs. We show that it is almost as efficient as DPHyp even without utilizing its brauch-and-bound pruning capabilities.

This paper is organized as follows. Sec. II recalls some preliminaries. Sec. III shows a naive approach called TDBasicHyp for handling hyperedges. Sec. IV presents our novel partitioning strategy for hypergraphs. Sec. V contains the experimental evaluation, and Sec. VI concludes the paper.

II. Preliminaries

A. Hypergraphs

Let us begin with the definition of hypergraphs.

Definition 1: 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 $\prec$.

A hyperedge (u, v) is *simple* if $|u| = |v| = 1$. A hypergraph is *simple* if all its hyperedges are simple. We call all non-simple hyperedges complex hyperedges and all non-simple hypergraphs complex hypergraphs.

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: Let $H = (V, E)$ be a hypergraph and $V' \subseteq V$ a subset of nodes. The *node-induced subgraph* $H|_{V'}$ of H is defined as $H|_{V'} = (V', E')$ 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 (sub) graphs*, we define connectedness:

Definition 3: 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 $H|_{V'}$ and $H|_{V''}$ 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 $H|_{V'}$ is connected, then we call V' a *connected subgraph* or *csg* for short. The number of connected subgraphs is important for memoization: it directly corresponds to the number of entries in the memotable.

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).

B. Connected Subgraph and its Complement Pairs

Our focus is to determine an optimal join order for a given query. The execution order of join operations is specified by an operator tree of the physical algebra. For our purposes, we want to abstract from that representation and give the notion of a *join tree*. A join tree is a binary tree where the leaf nodes specify the relations referenced in a query, and the inner nodes specify the two-way join operations. The edges of the join tree represent sets of joined relations. Two input sets of relations that qualify for a join so that no cross products need to be considered are called a *connected subgraph and its complement pair* or ccp for short [2].

Definition 4: Let $H = (V, E)$ be a connected hypergraph, (S_1, S_2) is a *connected subgraph and its complement pair* (or *ccp* for short) if the following holds:

- S_1 with $S_1 \subset V$ induces a connected graph $H_{|S_1}$,
- S_2 with $S_2 \subset V$ induces a connected graph $H_{|S_2}$,
- $S_1 \cap S_2 = \emptyset$, and
- $\exists (u, v) \in E | u \subseteq S_1 \wedge v \subseteq S_2$.

The set of all possible ccps is denoted by P_{ccp}. We introduce the notion of *ccp for a set* to specify all those pairs of input sets that result in the same output set, if joined.

Definition 5: Let $H = (V, E)$ be a connected hypergraph graph and S a set with $S \subseteq V$ that induces a connected subgraph $H_{|S}$. For $S_1, S_2 \subset V$, (S_1, S_2) is called a *ccp for* S if (S_1, S_2) is a ccp and $S_1 \cup S_2 = S$ holds.

By $P_{ccp}(S)$, we denote the set of all ccps for S. Let $\mathcal{P}_{con}(V) = \{S \subseteq V \mid H_{|S} \text{ is connected } \wedge |S| > 1\}$ be the set of all connected subsets of V with more than one element, then $P_{ccp} = \cup_{S \in \mathcal{P}_{con}(V)} P_{ccp}(S)$ holds.

If (S_1, S_2) is a ccp, then (S_2, S_1) is one as well, and we consider them as symmetric pairs. We are interested in the set P_{ccp}^{sym} of all ccps, where symmetric pairs are accounted for only once. The choice which one of two symmetric pairs should be member of P_{ccp}^{sym} is arbitrary. Our approach is as follows. For some set of relations S_i, denote by $max_{\text{index}}(S_i)$ the highest index of all relations contained in S_i. Then, $(S_1, S_2) \in P_{ccp}^{sym}$ if $max_{\text{index}}(S_1) \leq max_{\text{index}}(S_2)$ holds; otherwise $(S_2, S_1) \in P_{ccp}^{sym}$. Analogously, we denote the set of all ccps for a set S containing either (S_1, S_2) or (S_2, S_1) by $P_{ccp}^{sym}(S)$. In order to be correct, any dynamic programming or memoization algorithm has to consider all elements of P_{ccp}. Further, only these have to be considered. Thus, the minimal number of cost function calls for bottom-up or top-down processing is exactly $|P_{ccp}|$. The lower bounds for join enumeration given by Ono and Lohman [1] for certain graph shapes correspond to $|P_{ccp}^{sym}|$.

C. Neighborhood

The main idea to generate ccps is to incrementally expand connected subgraphs by considering new nodes in the *neighborhood* of a subgraph.

We start with the definition of a *simple neighborhood* that relies only on simple edges and returns one set of vertices.

Definition 6: Let $H = (V, E)$ be a connected hypergraph and C be a subset of V. Then, the *simple neighborhood* of $C \subseteq V$ is defined as:

$$\mathcal{N}_{simple}(C) = \{x \mid x \in v \wedge (u, v) \in E \wedge u \subset C \wedge v \subset (V \setminus C) \wedge |u| = 1 \wedge |v| = 1\}.$$

We now give the definition of *neighborhood* for all edges, including hyperedges. We further restrict the neighborhood by some set of X of forbidden nodes. As we will see below, this is necessary in order to avoid the generation of ccps more than once.

Definition 7: Let $H = (V, E)$ be a connected hypergraph, S a set of nodes ($S \subseteq V$) such that $H_{|S}$ is connected, and $X \subseteq V$ a set of excluded nodes. Then, the *neighborhood* of $C \subset S$ excluding X is defined as:

$$\mathcal{N}(S, C, X) = \{v \mid (u, v) \in E \wedge u \subseteq C \wedge v \subseteq S \setminus C \wedge v \cap X = \emptyset\}.$$

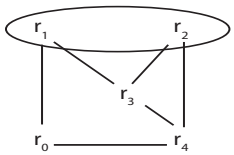

Fig. 1. Hypergraph $H(V,E) = (\{r_0,...,r4\}, \{ (\{r_0\},\{r_1,r_2\}), (\{r_0\},\{r_4\}), (\{r_1\},\{r_3\}), (\{r_2\},\{r_3\}), (\{r_2\},\{r_4\}), (\{r_3\},\{r_4\}) \})$

For our approach we need the notion of *minimal neighborhood*, where all subsumed nodesets $v \in \mathcal{N}(S,C,X)$ are eliminated. Again, this is necessary to avoid duplicate generation of ccps.

Definition 8: Let $S \subseteq V$, $C \subseteq V$, $X \subseteq V$ be sets of nodes with $C \subset S$ and $H_{|S}$ connected. Then, we define the *minimal neighborhood* of $C \subset S$ excluding X as:

$$\mathcal{N} \downarrow (S,C,X) = \{v \mid v \in N(S,C,X) \wedge \nexists u \in N(S,C,X) : u \subset v\}.$$

Thus, the minimal neighborhood only retains those hypernodes which are not proper supersets of others.

Let us exemplify Def. 6, 7 and 8 by the hypergraph of Fig. 1: $\mathcal{N}_{simple}(\{r_0\}) = \{r_4\}$, $\mathcal{N}_{simple}(\{r_0,r_4\}) = \{r_2,r_3\}$, $\mathcal{N}(\{r_0,...,r4\},\{r_0,r_4\},\emptyset) = \{\{r_1,r_2\},\{r_2\},\{r_3\}\}$, $\mathcal{N} \downarrow (\{r_0,...,r4\},\{r_0,r_4\},\emptyset) = \{\{r_2\},\{r_3\}\}$ and $\mathcal{N} \downarrow (\{r_0,...,r4\},\{r_0\},\emptyset) = \{\{r_1,r_2\},\{r_4\}\}$

III. Basic Memoization

As an introduction to top-down join enumeration, we give a basic memoization variant called TDBasicHyp, which we derive by utilizing a generic top-down algorithm, which invokes a naive partitioning algorithm. In the first subsection, we present our generic top-down algorithm. Then, we explain the naive partitioning strategy. Finally, we discuss the test for connectedness of hypergraphs.

A. Generic Top-Down Join Enumeration

Our generic top-down join enumeration algorithm TDPlanGenHyp is based on memoization. We present its pseudocode in Figure 2. Like dynamic programming, TDPlanGenHyp initializes the building blocks for atomic relations first (line 2). Then, in line 3 the subroutine TDPGSub is called, which traverses recursively through the search space. At the root invocation, the vertex set S corresponds to the vertex set V of the query graph. During every recursive call of TDPGSub, all possible join trees of two optimal subjoin trees that together would comprise the relations of S are built through a call to BuildTree (line 3). Among all those trees for a given set of relations S the cheapest join tree is kept by BuildTree which is explained later. We enumerate the optimal subjoin trees by iterating over the elements (S_1,S_2) of $P^{sym}_{ccp}(S)$ in line 2. This way, we derive the two optimal subjoin trees, each comprising exactly the relations in S_1 or S_2, respectively, by recursive calls to TDPGSub. The generation of $P^{sym}_{ccp}(S)$ is the task of a partitioning algorithm. Depending on the choice of the partitioning strategy, the overall performance of TDPlanGen can vary by orders of magnitude. For a naive partitioning strategy, we refer to Section III-B.

The recursive descent stops when either $|S| = 1$ or TDPGSub has already been called for that $H_{|S}$. In both cases, the optimal join tree is already known. To prevent TDPGSub from computing an optimal tree twice, $BestTree[S]$ is checked in line 1. $BestTree[S]$ yields a reference to an entry in an associative data structure called *memotable*. The data structure "memoizes" the optimal join tree generated for a set S. If $BestTree[S]$ equals NULL, this invocation of TDPGSub will be the first with $H_{|S}$ as input, and the optimal join tree of $H_{|S}$ has not been found yet.

```
TDPlanGenHyp(H)
    ▷ Input: H = (V,E) connected, V = ⋃_{1≤i≤|V|} {R_i}
    ▷ Output: an optimal join tree for H
1   for i ← 1 to |V|
2       BestTree[{R_i}] ← R_i
3   return TDPGSub(H_|V)

TDPGSub(H_|S)
    ▷ Input: H_|S connected
    ▷ Output: an optimal join tree for H_|S
1   if BestTree[S] = NULL
2       for all (S1,S2) ∈ P^sym_ccp(S)
3       do BuildTree(H_|S, TDPGSub(H_|S1),
                           TDPGSub(H_|S2))
4   return BestTree[S]
```

Fig. 2. Pseudocode for TDPlanGenHyp

The pseudocode of BuildTree is given in Figure 3. It is used to compare the cost of the join trees that belong to the same $H_{|S}$. Since the symmetric pairs (S_1,S_2) and (S_2,S_1) (line 2 of TDPGSub) are enumerated only once, we have to build two join trees (line 1 and line 4) and then compare their costs. We use the method CreateTree, which takes two disjoint join trees as arguments and combines them to a new join tree. If different join implementations have to be considered, among all alternatives the cheapest join tree has to be built by CreateTree. If the created join tree (line 1) is cheaper than $BestTree[S]$, or no tree for S has been built yet, $BestTree[S]$ gets registered with the $CurrentTree$. For the second tree, we just exchange the arguments (line 4). Again, the costs of the new join tree are compared to the costs of $BestTree[S]$. Only if the new join tree has lower costs, $BestTree[S]$ gets registered with the new join tree. Note that because of line 3, $BestTree[S]$ in line 6 cannot be NULL. Estimating the costs of the two possible join trees at the same time rather than separately and comparing them is more efficient, e.g., for cost functions as given in [12], where $card(T_x) \leq card(T_y) \Rightarrow cost(T_x \bowtie T_y) \leq cost(T_y \bowtie T_x)$ holds, where $card$ is the number of tuples or pages and T_x, T_y are (intermediate) relations.

B. Naive Partitioning

As we have already seen, the generic top-down enumeration algorithm iterates over the elements of $P^{sym}_{ccp}(S)$. Now, we show how the ccps for S can be computed by a naive generate-and-test strategy. We call our algorithm Partition$_{naive}$ and

```
BUILDTREE(H_|S, Tree_1, Tree_2)
    ▷ Input: induced graph H_|S, two sub join trees
1   CurrentTree ← CREATETREE(Tree_1, Tree_2)
2   if BestTree[S] = NULL ||
         cost(BestTree[S]) > cost(CurrentTree)
3       BestTree[S] ← CurrentTree
4   CurrentTree ← CREATETREE(Tree_2, Tree_1)
5   if cost(BestTree[S]) > cost(CurrentTree)
6       BestTree[S] ← CurrentTree
```

Fig. 3. Pseudocode for BUILDTREE

```
PARTITION_naive(H_|S)
    ▷ Input: a connected (sub) graph H_|S
    ▷ Output: P^sym_ccp(S)
1   for all C ⊂ S ∧ C ≠ ∅
2       if max_index(C) ≤ max_index(S \ C) ∧
             ISCONNECTED(H_|C) ∧ ISCONNECTED(H_|S\C)
3           emit(C, S \ C)
```

Fig. 4. Pseudocode for naive partitioning

give its pseudocode in Figure 4. In line 1, all $2^{|S|}-2$ possible non-empty and proper subsets of S are enumerated. For rapid subset enumeration, the method described in [13] can be used. We demand that from every symmetric pair only one is emitted. There are many possible solutions, but we make sure that the relation with the highest index represented in the graph is always contained in the complement $S \setminus C$ in line 2. Three conditions have to be met so that a partition $(C, S \setminus C)$ is a ccp. We check the connectivity of $H_{|C}$ and $H_{|S \setminus C}$ in line 2. We give a test for connectedness in Section III-C. The third condition that C needs to be connected to $S \setminus C$ is ensured implicitly by the requirement that the (sub) graph handed over as input is connected.

C. Test for Connectedness

Whereas the test for connectedness for query graphs with simple edges only is cheap ($O(|V|)$) and straightforward to implement (see ISCONNECTEDIMP in [5]), the test for connectedness of complex hypergraphs is more expensive ($O(|V| * |E|)$) and more delicate. Therefore, we explain it here in length. We give the pseudocode with ISCONNECTED in a modularized fashion (Figures 5, 6 and 7). This enables us to reuse some parts later on.

The basic idea of ISCONNECTED is to start with a set of single disjoint vertex sets $O_i \subset C$ with $|O_i| = 1$. Since each vertex set contains only one vertex at a time, each vertex set O_i itself must be connected. Then, these sets are merged by finding connecting hyperedges. Consider a hyperedge (u, v) and two disjoint vertex sets O_i, O_j. If $u \subseteq O_i \ \wedge \ v \subseteq O_j$ holds, then the two vertex sets O_i, O_j are connected. We have to iterate over the set of hyperedges several times until only a single vertex set is left. Otherwise, we fail to merge some vertex sets O_i, O_j. If there is only a single vertex set left at the end, the (sub) hypergraph must be connected.

For performance reasons, our approach works in two steps. The first step merges vertex sets O_i, O_j by considering simple edges only. We do so by calling GETSIMPLECOMPONENTS in line 1 of ISCONNECTED. GETSIMPLECOMPONENTS chooses an arbitrary vertex in line 4 to start with and assigns it to L', which then gets enlarged by adding adjacent singleton vertex sets within the loop in lines 5 to 8. Note that instead of iterating over all simple edges, we exploit the precomputed simple neighborhood $\mathcal{N}_{simple}$ (Def. 6), which is much more efficient (line 8). If the vertex set L' cannot be enlarged further (line 5), we chose another new vertex $i \in I$ (line 4) as the new L'. The newness is ensured in line 9. The loop starts over with the new L'. We return from the call to ISCONNECTED with a set O_{set} of merged vertex sets O_i, once we have considered all single vertex sets $i \in I$ that have not been previously merged.

In line 3 of ISCONNECTED, we implemented an early exit, that only checks if O_{set} contains only one vertex set O_i. If so, it is obvious that all vertices in C must be connected. Otherwise, we have to consider the complex hyperedges in order to determine if all $O_j \in O_{set}$ could be merged to one O_i. This is done by calling MERGECOMPONENTS in line 7 of ISCONNECTED as the second step of our approach. MERGECOMPONENTS maintains two sets of vertex sets: O_{set} and O'_{set}. O'_{set} was initialized with an arbitrary element $T \in O_{set}$ (lines 4, 6 of ISCONNECTED) which was deleted from O_{set} (line 5 of ISCONNECTED) before the call. Now, MERGECOMPONENTS considers one vertex set $O_i \in O_{set}$ at a time (line 1) and tries to merge it with some vertex sets O_j (line 7) already added to O'_{set}. Therefore, we need to consider hyperedges that can connect O_i with O_j. Instead of iterating over the list of complex hyperedges, we make use of the neighbourhood $\mathcal{N}$ (Def. 7) in line 6. If two candidates O_i, O_j can be merged, we have to make sure that the now combined vertex set T cannot be further enlarged with other elements of O'_{set}. This is necessary because for the combined set T, other hyperedges might qualify where one of their two hypernodes can contain nodes from O_i and O_j at the same time. Therefore, the loop of line 4 is put in place. When T cannot be enlarged any further, we add it to O'_{set} in line 11. Note that all vertex sets O_j used for the enlargement of T have been deleted from O'_{set} before (line 9). Once all O_i of O_{set} have been considered, we know that no element O_j of O'_{set} can be merged with any other $O_k \in O'_{set}$.

When the call is returned, we check how many elements are contained in O'_{set}. We know that C can only be connected if O'_{set} has just one element. Since we define an empty set to be connected as well, we check for $|O'_{set}| \leq 1$ and return the result of this test (line 8).

IV. GRAPH-BASED JOIN ENUMERATION

There exist two equally efficient partitioning algorithms suitable for top-down join enumeration: MINCUTBRANCH [3], [5] and MINCUTCONSERVATIVE [4]. Both algorithms require that the query graph needs to be a simple hypergraph, i.e., contains no complex hyperedges. However, there are two important reasons for a partitioning algorithm to be able to handle complex hyperedges:

```
isConnected(H_|C)
    ▷ Input: a node induced subgraph H_|C
    ▷ Output: TRUE if H_|C is connected, FALSE otherwise
1   O_set ← getSimpleComponents(C, C, ∅)
2   if |O_set| ≤ 1
3       return TRUE
4   T ← O_i ∈ O_set
5   O_set ← O_set \ {T}
6   O'_set ← {T}
7   O'_set ← mergeComponents(C, ∅, O_set, O'_set)
8   return |O'_set| ≤ 1
```

Fig. 5. Pseudocode for isConnected

```
getSimpleComponents(S, I, X)
    ▷ Input: I ∪ X = S, X ∩ I = ∅
    ▷ Output: O_set a set<connected sets>
    ∀_{O_i,O_j ∈ O_set} O_i, O_j ⊆ I ∧ O_i ∩ O_j = ∅
1   O_set ← ∅
2   while I ≠ ∅
3       L ← ∅
4       L' ← i ∈ I
5       while L ≠ L'
6           D ← L' \ L
7           L ← L'
8           L' ← L' ∪ ((N_simple(D) ∩ S) \ X)
9       I ← I \ L'
10      O_set ← O_set ∪ {L'}
11  return O_set
```

Fig. 6. Pseudocode for getSimpleComponents

1) Non-inner joins are not freely reorderable. As a consequence, certain join reorderings result in different non-equivalent plans that return different query results when executed. However, it is well-known that valid operator orderings can be encoded by transforming simple edges into hyperedges [6], [10], [7].

```
mergeComponents(S, X, O_set, O'_set)
    ▷ Input: S ∩ X = ∅
             O_set, O'_set sets<connected sets>
    ▷ Output: O'_set a set<connected sets>
    ∀_{O_i,O_j ∈ O'_set} O_i, O_j ⊆ (S \ X) ∧ O_i ∩ O_j = ∅
1   for all O_i ∈ O_set
2       T ← O_i
3       t ← 0
4       while t ≠ |T|
5           t ← |T|
6           N_set ← N(S, T, X)
7           for all O_j ∈ O'_set
8               if ∃v ∈ N_set : v ⊆ O_j
9                   O'_set ← O'_set \ {O_j}
10                  T ← T ∪ O_j
11      O'_set ← O'_set ∪ {T}
12  return O'_set
```

Fig. 7. Pseudocode for mergeComponents

2) Real world queries may contain complex predicates, i.e., predicates referencing more than two relations. Those predicates can only be expressed with hyperedges [8].

Hence, both top-down partitioning algorithms cannot be used in real-world scenarios. With MinCutConservativeHyp, we present a partitioning algorithm that can efficiently handle complex hyperedges. The name was chosen because we adopted and extended a principle idea of MinCutConservative. As already said, MinCutConservativeHyp is a partitioning algorithm. Thus, it can be used to instantiate the generic top-down join enumeration algorithm (Section III-A). We denote the instantiated top-down memoization variant by TDMcCHyp (short for TDMinCutConservativeHyp). The pseudocode for MinCutConservativeHyp is given in Figure 8.

A. *Overview of* MinCutConservativeHyp

Before we present the algorithm in detail, we explain its basic idea. The goal of a partitioning algorithm is to compute the set $P^{sym}_{ccp}(S)$ for a connected vertex set S. As we will show in Section V-D, the generate-and-test approach of TDBasicHyp is not practical at all, because the majority of generated partitions is rejected by either one of the tests for connectedness in line 2 of Partition$_{naive}$. In certain scenarios (e.g., chain queries), this adds an exponential overhead for each emitted ccp. Clearly, this has to be avoided.

Let C be a set of vertices. For the time being, assume that C is initialized with an arbitrary single vertex $t \in S$. The general idea of MinCutConservativeHyp is to recursively enlarge a connected set C of vertices by adding members of its neighborhood $\mathcal{N}$. If at some point during the enlargement of C its complement $S \setminus C$ in S is connected as well, the algorithm has found a ccp for S. We ensure that at (almost) every instance of the algorithm's execution C is *c*onnected. Sometimes there are exceptions, and their handling is described below.

Besides, the connectedness of C's complement $S \setminus C$ MinCutConservativeHyp has to meet some more constraints before emitting a ccp: (1) Only one of two symmetric ccps is emitted, (2) the emission of duplicates has to be avoided, and (3) all ccps for S have to be computed as long as they comply with constraint (1).

Constraint (1) is ensured because the start vertex t – arbitrarily chosen as the initial neighborhood – is always contained in C and, therefore, can never be part of its complement. For the second constraint, the algorithm uses a filter set X of neighbors to e*x*clude from processing. After every recursive self-invocation of the algorithm, the adjacent hypernode v that was used to enlarge C is added to X. The presence of cycles in the hypergraph that involve complex hyperedges requires some additional precautions. Therefore, we introduce X_{map}, which is a mapping between vertices and vertex sets, and helps us to prevent duplicates. Later, we will see in detail how this ties in. For constraint (3), it is sufficient to ensure that all possible connected subsets of S that contain the start vertex t are considered when enlarging C.

There are certain scenarios, e.g., when simple hypergraphs like star queries are considered, where constructing every possible connected subset C of S produces an exponential overhead. This is because most of the produced complements $S \setminus C$ are not connected and the partitions $(C, S \setminus C)$ computed this way are not valid ccps. Therefore, the algorithm follows a *conservative* approach by enhancing C in such a way that the complement must be connected as well.

To explain this approach, we have to make some observations. From the recursive process of enlarging C, we know that the number of members in C must increase by at least one in every iteration. Furthermore, if a partition $(C, S \setminus C)$ is not a ccp for S, then $S \setminus C$ consists of $k \geq 2$ connected subsets $O_1, O_2, ..., O_k \subset (S \setminus C)$ that are disjoint and not connected to each other. Hence, those subsets $O_1, O_2, ...O_k$ can only be adjacent to C. Let $v_1, v_2, ..., v_l$ be hypernodes that are all the members of C's neighborhood $\mathcal{N}(S, C, \emptyset)$. Then every O_i with $1 \leq i \leq k$ must contain at least one such v_y where $1 \leq y \leq l$ and $k \leq l$ holds. The first ccp after recursively enlarging C by members of $S \setminus C$ would be generated when all subsets O_i with $1 \leq i \leq k$ but one are joined to C. Hence, in order to ensure that at every recursive iteration of MINCUTCONSERVATIVEHYP the complement $S \setminus C$ is connected as well, it does not always suffice to enlarge C by only one of its adjacent hypernodes, but by a larger subset $\cup_{O_j}$ of its direct and indirect neighborhood [2]. Section IV-D will explain how the subsets $O_1, O_2, ...O_k$ are computed with GETCONNECTEDCOMPONENTS.

Let C be a connected set of vertices and (u, v) a complex hyperedge with $|v| > 1$. Then if C is enlarged by v to $C' := C \cup v$, the new C' is not necessarily connected. For an example, we refer to Fig. 1. If C is set to $\{r_0\}$ and $v = \{r_1, r_2\}$, the new $C' = \{r_0, r_1, r_2\}$ is not connected anymore. Hence, we have to check every time whether C' is connected. DPHYP deals with this problem by exploiting its bottom-up processing nature (see ENUMERATECMPREC in [10]): C' is connected if and only if there exists an entry for C' in the dynamic programming table ($BestTree[C']$). We cannot rely on such a trick because TDMINCUTCONSERVATIVEHYP works top-down. Therefore, we introduce a set of vertex sets C_{set}. C_{set} keeps track of all connected subsets of C that are not connected to each other. To check if C is connected, we only have to ensure that C_{set} contains just one vertex set (which would be equal to C). Since the majority of the hypernodes of all hyperedges will be connected, maintaing C_{set} will be cheap. In the worst case, it is in $O(|C_{set}|^2)$, which is still cheaper than a call to ISCONNECTED ($O(|C|^2)$). MAINTAINCSET takes care of this task and is explained in Sec. IV-E.

B. The Algorithm in Detail

Now, we take a closer look at the pseudocode of MINCUTCONSERVATIVEHYP in Fig. 8. $\text{PARTITION}_{MinCutConservativeHyp}$ invokes MINCUTCONSERVATIVEHYP with $C = X = F = \emptyset$. C_{set} as well as X_{map} are also empty. With the exception of F, the intention behind every parameter of MINCUTCONSERVATIVEHYP has already been discussed. F is used as an optimization technique in form of a filter set to speed up duplicate avoidance and is discussed in Sec. IV-C.

Remember that S is the set for which we want to create ccps $(C, S \setminus C)$. MINCUTCONSERVATIVEHYP recursively enlarges C until the stop criterium with $|C| = |S| - 1$ (line 3) is met. C is enlarged by hypernodes that are members of its minimal neighborhood (Def. 8, line 8). To avoid duplicate ccps, vertices that have been added to C in previous invocations are excluded here and in further child invocations. In the root invocation, C is set to be empty, which means that its minimal neighborhood would be empty as well. To deal with the initial empty C, we redefine the minimal neighboorhood of the empty set to be the singleton set containing solely an arbitrary start vertex $t \in S$.

The loop in lines 8 to 20 iterates over all adjacent hypernodes v, where $v \in \mathcal{N} \downarrow (S, C, X) \wedge C \cup v \neq S$ holds. The condition $C \cup v \neq S$ ensures that the new C for the next child invocation is not equal to S, which would not give rise to any more ccps.

As has been discussed, we need to take special care of how we enlarge C such that the complement $S \setminus (C \cup v)$ is connected as well. Therefore, we have to determine if $S \setminus (C \cup v)$ consists of subsets $O_1, O_2, ...O_k$ that are disjoint and not interconnected to each other. Thus, we calculate $O := O_1, O_2, ...O_k$ by a call to GETCONNECTEDCOMPONENTS (explained in Sec. IV-D). Following from the discussion in Sec. IV-A, the next ccp is only emitted when all but one subset O_i with $1 \leq i \leq k$ are merged with the current C. As a consequence, we compute the next C simply by assigning O_i's complement in S to C' (line 11). All different O_i are then considered by processing the loop consisting of lines 10 to 15.

Referring to Sec. IV-A, a merge of an adjacent hypernode with C might not result in a connected C' anymore. The same can occur if even all but one of the connected subsets $O_1, O_2, ...O_k$ are merged with C. Therefore, MAINTAINCSET (explained in Section IV-E) manages the set of connected subsets of C by adding and merging $C' \setminus C$ with the elements of C_{set}. In line 15, MINCUTCONSERVATIVEHYP invokes itself. Before the next child invocation emits a partition $(C, S \setminus C)$, the condition of line 1 is checked. Here, we ensure that C is connected by checking the cardinality of C_{set}. Furthermore, we ensure that $S \setminus C$ is not empty.

The next section explains our duplicate avoidance technique.

C. Avoiding Duplicates

To prevent the emission of duplicate ccps, MINCUTCONSERVATIVEHYP distinguishes between two cases: the simple case and the complex case.

The first case occurs if solely a single vertex r is added to C, i.e., $|C' \setminus C| = 1$. In this case, we can apply the duplicate avoidance technique of MINCUTCONSERVATIVE [4], which we now briefly review. The new node r is added to the excluded set X' (line 17). As a consequence, r cannot be chosen again from the neighborhood (line 8) in any further child invocation. Additionally, we have to check that X' is disjoint with C' (line 12). To see this, imagine a vertex set v

```
PARTITION_{MinCutConservativeHyp}(S)
    ▷ Input: a connected set S
    ▷ Output: all ccps for S
1   for all r ∈ S
2       X_map[{r}] ← ∅
3   MINCUTCONSERVATIVEHYP(S, ∅, ∅, ∅, X_map, ∅)

MINCUTCONSERVATIVEHYP(S, C, C_set, X, X_map, F)
    ▷ Input: connected set S, C ∩ X = ∅, F ⊆ C
             C_set a set<connected sets>
             X_map a map<relation, set of relations>
    ▷ Output: ccps for S
 1  if |C_set| = 1 ∧ C ≠ S
 2      emit (C, S \ C)
 3  if |C| = |S| − 1
 4      return
 5  X' ← X
 6  X'_map ← X_map
 7  F' ← F
    ▷ N ↓ (S, ∅, ∅) = {arbitrary element of t ∈ S}
 8  for all v ∈ N ↓ (S, C, X) : C ∪ v ≠ S
 9      O ← GETCONNECTEDCOMPONENTS(S, C ∪ v)
10      for all O_i ∈ O
11          C' ← S \ O_i
12          if X' ∩ C' ≠ ∅ ∨ ((C' \ C) ∩ F' ≠ ∅ ∧
                        ¬CHECKXMAP(C', F', X'_map))
13              continue
14          C'_set ← MAINTAINCSET(S, C, C', C_set)
15          MINCUTCONSERVATIVEHYP(
                        S, C', C'_set, X', X'_map, F')
16      if |v| = 1
17          X' ← X' ∪ v
18      else
19          F' ← F' ∪ v
20          X'_map ← MAINTAINXMAP(v, X'_map)
```

Fig. 8. Pseudocode for MINCUTCONSERVATIVEHYP

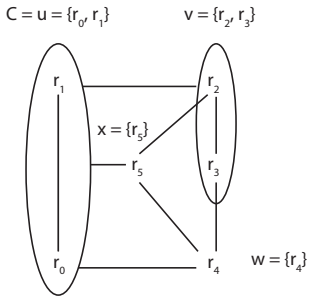

Fig. 9. Sample Hypergraph

with $|v| = 1$ was merged with X, and in a later iteration of the loop in line 8, another vertex set u with $|u| = 1$ is chosen to be merged with C'. Assume that the call to GETCONNECTEDCOMPONENTS yields $O = \{O_1, O_2, ...O_k\}$. Then, u must be part of some O_i $(1 \leq i \leq k)$. Now, if the previously chosen v is also part of O_i at some recursive call, then only ccps are emitted that have been emitted before. The check $X' \cap C' \neq \emptyset$ prevents this.

The second case is much more complex. We proceed in several steps. In step 1, let us observe that it is easy to generate duplicate ccps. Consider the hypergraph in Fig. 9 with $S = \{r_0, ...r_5\}$. Let us forward to the point of the algorithms execution where $C = \{r_0, r_1\}$ holds. To continue we have to recursivly enlarge C with its neighbours. Hence we have to walk the cycle in the direction $C \rightarrow v \rightarrow w$ which generates C, $C \cup v$, $C \cup v \cup w$. The opposite direction gives us $C \cup w$, $C \cup w \cup v$. Thus, it is easy to generate duplicate connected components, and because $C \cup v \cup w$ would be generated twice, the corresponding ccp $(C \cup v \cup w, x)$ would be emitted twice.

In step 2, we convince ourselves that the duplicate avoidance technique of MINCUTCONSERVATIVE does not help here. Imagine we have to partition the hypergraph in Fig. 9 with $S = \{r_0, ...r_5\}$. Then, the minimal neighborhood $\mathcal{N} \downarrow (S, C, X)$ with $C = \{r_0, r_1\}$ contains $v = \{r_2, r3\}$. Once v is chosen in line 8, v is added to C. Assume we apply the technique used in MINCUTCONSERVATIVE. Then, we have to add all members of v to X'. Because of $r_3 \in v$, that would imply that we cannot generate $C \cup w \cup \{r_3\}$ with $w = \{r_4\}$ later on, which breaks our correctness constraint (3) (Section IV-A).

In step 3, we present our duplicate avoidance strategy. Therefore, we introduce X_{map} that represents a mapping between vertices and vertex sets. The mapping is managed by MAINTAINXMAP presented in Figure 10. Now, instead of adding all vertices of the hypernode v to X' (case 1), we alter X_{map}. We register for every member of v either (1) the whole hypernode $A = v$ (line 3 of MAINTAINXMAP), or (2) merge v with an already registered vertex set (line 5 of MAINTAINXMAP).

To check if we are allowed to add $\{r_3\}$ to C (Fig. 9), we have to consult our X_{map}. This is done by a call to CHECKXMAP (line 12 of MINCUTCONSERVATIVEHYP). Fig.11 shows the pseudocode. CHECKXMAP checks for all vertices $r \in v$ that are added to C if their mapping $X_{map}[r]$ is (1) either empty, or (2) does not overlap fully with the new C so that $X_{map}[r] \cap C \neq X_{map}[r]$ holds (line 2 of CHECKXMAP). Let us explain the intuition behind the latter by referring to our example scenario. When we enlarged $C = u$ with v, we set $X_{map}[r_2] = v$ and $X_{map}[r_3] = v$ after returning from the child invocation. In the next iteration of the loop (8 of MINCUTCONSERVATIVEHYP), we merge $C = u$ with w. If we now want to add r_3 to $C = u \cup w$, CHECKXMAP allows this because $X_{map}[r_3] = v$ and $v \setminus C \neq \emptyset$ holds. In general, any combination of vertices in v is allowed to be added to C, as long as not all elements of v are added to C. At that point where we would try to add the last member r of v to C as well, $X_{map}[r]$ returns v, and since $v \subset C$ holds, CHECKXMAP returns FALSE. Since CHECKXMAP ensures that not all members of $X_{map}[r]$ are included into C', we have to pay attention during the maintenance of X_{map} in the case the value of a given entry $X_{map}[r]$ is already assigned with. Before we enlarge $X_{map}[r]$ (line 5 of MAINTAINXMAP) we have to ensure that $X_{map}[r]$ not already contains A (second part of the condition in line 2). For those cases, we have to curtail $X_{map}[r]$ (line 3) instead of enlarging it. Thus we

```
MAINTAINXMAP(A, X_map)
   ▷ Input: set A of relations
            X_map a map<relation, set of relations>
   ▷ Output: modified X_map
1  for all r ∈ A
2      if X_map[r] = ∅ ∨ A ⊆ X_map[r]
3          X_map[r] ← A
4      else
5          X_map[r] ← X_map[r] ∪ A
6  return X_map
```

Fig. 10. Pseudocode for MAINTAINXMAP

```
CHECKXMAP(C, F, X_map)
   ▷ Input: set C, F
            X_map a map<relation, set of relations>
   ▷ Output: FALSE if duplicate ccps have to be avoided
1  for all r ∈ (C ∩ F)
2      if X_map[r] ≠ ∅ and X_map[r] \ C = ∅
3          return FALSE
4  return TRUE
```

Fig. 11. Pseudocode for CHECKXMAP

allow only for proper subsets of A (and not of the superset $X_{map}[r]$ of A) to be added to C in later recursions of MINCUTCONSERVATIVEHYP.

Iterating over all members of C in line 1 of CHECKXMAP is relatively expensive. Therefore, we introduce a vertex filter set F. In line 19 of MINCUTCONSERVATIVEHYP, we add all elements of a hypernode v to F. It is easy to see that we now only need to consider the vertices of C that are also element of F in line 1 of CHECKXMAP. Furthermore, we can even avoid the whole effort of evaluating CHECKXMAP if all the vertices in $C' \setminus C$ with which C is going to be enlarged are disjoint to F (line 12 of MINCUTCONSERVATIVEHYP). If we assume that all complex hyperedges of a hypergraph are there because of non-inner join conflict encodings, the hypergraph will not have any cycles involving a complex hyperedge. In this case, $(C' \setminus C) \cap F' = \emptyset$ always holds, and CHECKXMAP is never called.

D. GETCONNECTEDCOMPONENTS

The pseudocode for GETCONNECTEDCOMPONENTS is given in Figure 12. As has been said, GETCONNECTEDCOMPONENTS is designed to compute all disjoint, connected but not interconnected subsets $O_1, O_2, ...O_k$ of Cs complement in S. The main idea of GETCONNECTEDCOMPONENTS is similar to the one used in ISCONNECTED (Section III-C). In line 1, we call GETSIMPLECOMPONENTS (explained in Section III-C) to compute all the subsets of $S \setminus C$ that induce disjoint connected hypergraphs consisting of simple edges only.

The result of that call is stored into O_{set}. Now, we only need to check whether the simple connected components can be merged further by considering complex hyperedges. This task is delegated to MERGECOMPONENTS (Section III-C) in line 5 of GETCONNECTEDCOMPONENTS. For more details, we refer to Section III-C.

```
GETCONNECTEDCOMPONENTS(S, C)
   ▷ Input: C ⊂ S a connected set
   ▷ Output: O'_set a set<connected sets>
   ∀_{O_i,O_j ∈ O'_set} O_i, O_j ⊆ (S \ C) ∧ O_i ∩ O_j = ∅
1  O_set ← GETSIMPLECOMPONENTS(S, S \ C, C)
2  T ← O_i ∈ O_set
3  O_set ← O_set \ {T}
4  O'_set ← {T}
5  O'_set ← MERGECOMPONENTS(S, C, O_set, O'_set)
6  return O'_set
```

Fig. 12. Pseudocode for GETCONNECTEDSETS

```
MAINTAINCSET(S, C, C', C_set)
   ▷ Input: set S of relations, C' ⊆ S, C ⊆ C'
            C_set a set<connected sets>
   ▷ Output: modified C_set
1  I ← C' \ C
2  O_set ← GETSIMPLECOMPONENTS(S, I, S \ I)
3  C'_set ← MERGECOMPONENTS(S, S \ C', O_set, C_set)
4  return C_set
```

Fig. 13. Pseudocode for MAINTAINCSET

E. MAINTAINCSET

The purpose of MAINTAINCSET is to speed up the test for connectedness of C. The result of the test is needed in order to determine if a partition $(C, S \setminus C)$ is a valid ccp (line 1 of MINCUTCONSERVATIVEHYP). The pseudocode is shown in Figure 13. As the name implies, MAINTAINCSET manages C_{set}, which contains all disjoint, connected but not interconnected subsets of C. If C_{set} contains only one vertex set, this means that C consists only of one connected subset and must therefore be connected. Besides the C_{set} also S, C and C' are handed over. The latter two input parameters are used to determine all vertices that were added to C. We assign the result to the vertex set I (line 1).

We utilize GETSIMPLECOMPONENTS (line 2) to compute all the subsets of I that induce connected hypergraphs consisting only of simple edges. The computed subsets are stored in O_{set} and handed over to MERGECOMPONENTS (line 3), which merges them with the components of C_{set}. For every subset of O_{set}, there are two possibilities. Either it can be merged with one subset of C_{set} or it unions two or more subsets of C_{set} together by forming one unified vertex set. But one member of O_{set} will always be adjacent to at least one connected subset in C_{set}. There is only one exception that occurs when MAINTAINCSET is called from MINCUTCONSERVATIVEHYP's root invocation, then C_{set} will be empty and C'_{set} will be returned containing only $\{t\}$ (a vertex set with the start vertex t).

F. An Example

We illustrate the execution of MINCUTCONSERVATIVEHYP by an example. Table I shows the execution steps for the cyclic query given in Figure 1. The first column is the table entry number that serves as reference. The second column

keeps track of the recursion level. The root invocation is indicated with a 0. Columns 3−6 show the input parameters of MINCUTCONSERVATIVEHYP. S is set to $\{r_0, r_1, r_2, r_3, r_4\}$ and remains unchanged. The last column displays the result of the minimal neighborhood (Def. 8) $\mathcal{N} \downarrow (S, C, X)$. The start vertex t is set to r_0. Although C is a vertex set, we display the order of insertion instead of using a set notation (Column 3). In this example, MINCUTCONSERVATIVE emits six ccps. However, one partition is generated but not emitted because C is not connected at that point (entry no. 11). We have omitted the calls to MAINTAINCSET because its results are implicitly given as a parameter of the next child invocation. MAINTAINXMAP is called only once (entry no. 13). It registers for every member of the hypernode $\{r_1, r_2\}$ the hypernode itself. At the same time, F is enlarged with $\{r_1, r_2\}$. We can see that our filter technique is quite effective since CHECKXMAP is called only once (entry no. 17). All the other times $(C' \setminus C) \cap F' \neq \emptyset$ does not hold. There is only one time where the result of GETCONNECTEDCOMPONENTS contains more than one vertex set (entry no. 23). At this instance, v is assigned with $\{r_3\}$ and C is assigned with $\{r_0, r_4\}$. Instead of setting C' to $C \cup v$ it is set to $S \setminus O_i = S \setminus \{r_1\} = \{r_0, r_2, r_3, r_4\}$. Otherwise, the complement $S \setminus C' = \{r_1, r_2\}$ would not be connected during the next child invocation.

V. EVALUATION

This section describes our empirical evaluation. We start by explaining our setup in Sections V-A and V-B. Then, we give an organizational overview (Section V-C). Finally, we present the results (Section V-D).

A. Implementation

For all plan generators, no matter whether they work top-down or bottom-up, a shared optimizer infrastructure was established. It contains the common functions to instantiate, fill, and look up the memotable, initialize and use plan classes, estimate cardinalities, calculate costs, and compare plans. Thus, the different plan generators differ only in those parts of the code responsible for enumerating ccps and to utilize pruning (if applied).

For the cost estimation of joins, we decided to use the formulas developed by Haas et al. [12]. They have the advantage of being very precise.

B. Workload

To generate our workload, we have implemented two kinds of query graph generators.

The first graph generator is based on a random operator tree generator that attaches all different inner and non-inner join operators. Here we make only one assumption. More than half of all attached join operators should be inner join operators. We then compute acyclic complex hypergraphs from these generated operator trees with the method proposed in [10]. To gain cyclic graphs, we determine all subgraphs of a given hypergraph that are connected by inner join edges only. Within those subgraphs, we randomly generate more inner join edges. We call the generated graphs non-inner join query graphs although the majority of edges are still inner join edges.

The second graph generator generates random acyclic and cyclic graphs, containing simple edges only. Thereby the edges are randomly added by selecting two relation's indices using uniformly distributed random numbers. After having generated a simple graph, the generator starts transforming simple edges to complex hyperedges at random. Therefore, it randomly chooses between 3 parameters: (1 and 2) the size of the hypernodes connected through the new hyperedge and (3) if the simple edge that is transformed is part of a cycle or not, i.e., is the only connection between two connected subgraphs or not. An edge is only transformed if the resulting complex hyperedge is not subsumed by any other edge. Essentially, this generator generates hypergraphs with complex hyperedges that model complex join predicates involving more than two relations. Therefore, we call the generated graphs complex predicate query graphs.

To generate cardinalities and selectivities, we follow the approach of Fender et. al. [4].

C. Organizational Overview

In our empirical analysis, we compare the runtime results of our new algorithm TDMCCHYP (short for TDMINCUTCONSERVATIVEHYP) with our naive approach called TDBASICHYP. In order to put the two top-down join enumeration algorithms into perspective, we include the results of Moerkotte and Neumann's DPHYP as the state-of-the art in bottom-up join enumeration via dynamic programming. Since top-down processing has branch-and-bound pruning capabilities, we also want to investigate its benefits. Therefore, we include a pruning variant of TDMCCHYP to which we refer as TDMCCHYP$_{pruning}$ in the following. For our pruning variant, we implemented pruning techniques as described in [4].

We present our results in terms of the quotient of the algorithm's execution time and the execution time of DPHYP. We refer to this quotient as the *normed time*. Table II and Table III show the average, minimum, and maximum normed time over the whole workload of non-inner join and complex predicate queries.

Since the normed time for DPHYP is always 1, we rather give its elapsed time in seconds. Figure 14 displays the runtime results for acyclic queries. We give the number of vertices on the abscissa and the execution time in log scale on the ordinate. We draw lines to connect the averaged execution times.

For randomly generated cyclic queries, the algorithms' performance results deviate significantly for the same number of vertices. Thus, we cannot show the results for different numbers of vertices at the same time. Figure 15 presents the results for 15 vertices only. At the abscissa, we choose to display the number of edges and again the execution time in log scale on the ordinate.

We include only those query graphs in our evaluation that all plan generators could process in less than 100 seconds. The workload consists of more than 50000 query graphs. We generated graphs up to 20 vertices of non-inner join queries

TABLE I

EXEMPLIFIED EXECUTION OF MINCUTCONSERVATIVEHYP FOR THE GRAPH OF FIG. 1 WITH $S = \{r_0, r_1, r_2, r_3, r_4\}$

Entry No.	Level	C	C_{set}	X	X_{map}	F	$\mathcal{N}\downarrow$
1	0	$\emptyset$	$\emptyset$	$\emptyset$	empty	$\emptyset$	$\{\{t = r_0\}\}$
2	1	r_0	$\{\{r_0\}\}$	$\emptyset$	empty	$\emptyset$	$\{\{r_1, r_2\}, \{r_4\}\}$
3	1	emitting $(\{r_0\}, \{r_1, r_2, r_3, r_4\})$					
4	1	$v = \{r_1, r_2\} \rightarrow$ GETCONNECTEDCOMPONENTS$(S, \{r_0, r_1, r_2\}) \rightarrow \{\{r_3, r_4\}\}$					
5	2	$r_0 \rightarrow r_1, r_2$	$\{\{r_0\}, \{r_1\}, \{r_2\}\}$	$\emptyset$	empty	$\emptyset$	$\{\{r_3\}, \{r_4\}\}$
6	2	$v = \{r_3\} \rightarrow$ GETCONNECTEDCOMPONENTS$(S, \{r_0, r_1, r_2, r_3\}) \rightarrow \{\{r_4\}\}$					
7	3	$r_0 \rightarrow r_1, r_2 \rightarrow r_3$	$\{\{r_0, r_1, r_2, r_3\}\}$	$\emptyset$	empty	$\emptyset$	$\{\{r_4\}\}$
8	3	emitting $(\{r_0, r_1, r_2, r_3\}, \{r_4\})$					
9	2	$v = \{r_4\} \rightarrow$ GETCONNECTEDCOMPONENTS$(S, \{r_0, r_1, r_2, r_4\}) \rightarrow \{\{r_3\}\}$					
10	3	$r_0 \rightarrow r_1, r_2 \rightarrow r_4$	$\{\{r_0, r_2, r_4\}, \{r_1\}\}$	$\{r_3\}$	empty	$\emptyset$	$\emptyset$
11	3	no emission of $(\{r_0, r_1, r_2, r_4\}, \{r_3\})$ since $\lvert C_{set}\rvert > 1$					
12	1	$v = \{r_4\} \rightarrow$ GETCONNECTEDCOMPONENTS$(S, \{r_0, r_4\}) \rightarrow \{\{r_1, r_2, r_3\}\}$					
13	1	call to MAINTAINXMAP$(\{r_1, r_2\}$, empty$) \rightarrow r_1 : \{r_1, r_2\}, r_2 : \{r_1, r_2\}$					
14	2	$r_0 \rightarrow r_4$	$\{\{r_0, r_4\}\}$	$\emptyset$	$r_1 : \{r_1, r_2\}, r_2 : \{r_1, r_2\}$	$\{r_1, r_2\}$	$\{\{r_2\}, \{r_3\}\}$
15	2	emitting $(\{r_0, r_4\}, \{r_1, r_2, r_3\})$					
16	2	$v = \{r_2\} \rightarrow$ GETCONNECTEDCOMPONENTS$(S, \{r_0, r_2, r_4\}) \rightarrow \{\{r_1, r_3\}\}$					
17	2	call to CHECKXMAP$(\{r_0, r_2, r_4\}, \{r_1, r_2\}, r_1 : \{r_1, r_2\}, r_2 : \{r_1, r_2\}) \rightarrow$ returns TRUE					
18	3	$r_0 \rightarrow r_4 \rightarrow r_2$	$\{\{r_0, r_2, r_4\}\}$	$\emptyset$	$r_1 : \{r_1, r_2\}, r_2 : \{r_1, r_2\}$	$\{r_1, r_2\}$	$\{\{r_3\}\}$
19	3	emitting $(\{r_0, r_2, r_4\}, \{r_1, r_3\})$					
20	3	$v = \{r_3\} \rightarrow$ GETCONNECTEDCOMPONENTS$(S, \{r_0, r_2, r_3, r_4\}) \rightarrow \{\{r_1\}\}$					
21	4	$r_0 \rightarrow r_4 \rightarrow r_2 \rightarrow r_3$	$\{\{r_0, r_2, r_3, r_4\}\}$	$\emptyset$	$r_1 : \{r_1, r_2\}, r_2 : \{r_1, r_2\}$	$\{r_1, r_2\}$	$\{\{r_1\}\}$
22	4	emitting $(\{r_0, r_2, r_3, r_4\}, \{r_1\})$					
23	2	$v = \{r_3\} \rightarrow$ GETCONNECTEDCOMPONENTS$(S, \{r_0, r_3, r_4\}) \rightarrow \{\{r_1\}, \{r_2\}\}$					
24	3	$r_0 \rightarrow r_4 \rightarrow r_1, r_3$	$\{\{r_0, r_1, r_3, r_4\}\}$	$\{r_2\}$	$r_1 : \{r_1, r_2\}, r_2 : \{r_1, r_2\}$	$\{r_1, r_2\}$	$\emptyset$
25	3	emitting $(\{r_0, r_1, r_3, r_4\}, \{r_2\})$					

and graphs up to 22 vertices of complex predicate queries. Thereby among the cyclic queries the number of edges per number of vertices is evenly distributed. In fact, when we generated the cyclic queries we took care that the minimal number of edges was equal to the number of vertices and that the maximal number of edges was at least twice the number of vertices. Every graph had to have at least one complex hyperedge.

Our experiments were conducted on an Intel Pentium D with 3.4 GHz, 2 Mbyte second level cache and 3 Gbyte of RAM running openSUSE 12.1. We used the GNU C++ compiler with the compiler option O3.

D. Experiments

This section summarizes our experimental findings. We start with an evaluation of random acyclic queries, present the results for random cyclic queries later and investigate how much TDMCCHYP can benefit from applying branch-and-bound pruning techniques at the end.

1) Random Acyclic Query Graphs: When comparing the performance results between non-inner join queries and complex predicate queries as given in Figure 14 and Tables II, III, each algorithm retains its unique trend. Nevertheless, on average the algorithms were able to faster compile plans for non-inner join queries. As it turns out, the average size of a hypernode associated to an average hyperedge is distinctly higher for non-inner join queries. Hence the number of ccps for non-inner join queries has to be lower, which makes them easier to compile since less plans have to be considered.

With a maximal normed runtime of 28,500 (Table III) and an average normed runtime of 730 to 860, TDBASICHYP performs worst. With a rising number of vertices the difference to DPHYP and TDMCCHYP increases significantly. For 18 and more vertices, TDBASICHYP needs more than one second to run. If we look at the average compile-time of TDBASICHYP for queries with more than 20 vertices, it becomes indisputable that it could not be used in practice. On the other hand, the runtimes of DPHYP and TDMCCHYP are negligible. On average, DPHYP performs slightly better. But in some cases, it can also become slower than TDMCCHYP: up until a factor of $0.17^{-1} = 6$. Since the absolute runtimes of both algorithms are so low, the performance of DPHYP and TDMCCHYP is virtually the same.

2) Random Cyclic Query Graphs: Again, the compile-time results of non-inner join queries resemble each other when compared with the results for complex predicate queries. But still non-inner join queries take less time to compile.

Looking at Figure 15 one can observe that the normed runtime of TDBASICHYP highly depends on the number of edges considered. The normed runtime is similar to the runtime of acyclic queries if the number of edges is low compared to the number of vertices. However, with a rising number of edges, the normed runtime decreases. This is no surprise, since the number of ccps increases exponentially in the number of edges for a fixed number of vertices. With a higher number of ccps, fewer partitions considered by PARTITION$_{naive}$ are rejected by ISCONNECTED. This makes PARTITION$_{naive}$ more competitive. Furthermore, the test for connectedness becomes cheaper on average: Since more connecting edges might exist, interconnected subsets are faster to merge. In fact, at some point the runtime of TDBASICHYP converges with the runtime of DPHYP.

With an average factor of 1.04 to 1.19, the runtime of TDMCCHYP is slightly higher compared to the runtime of

TABLE II

MINIMUM, MAXIMUM AND AVERAGE OF THE NORMED RUNTIMES FOR NON-INNER JOIN GRAPHS.

Algorithm	min	max	avg	min	max	avg
	random Acyclic			random Cyclic		
DPHYP	0.0001 s	0.0624 s	0.0007 s	0.0001 s	1.4908 s	0.0079 s
TDBASICHYP	1.4615 ×	11820.6168 ×	858.4679 ×	1.0159 ×	12146.2795 ×	723.6721 ×
TDMCCHYP	0.1707 ×	3.0236 ×	1.0730 ×	0.1231 ×	2.8985 ×	1.0428 ×
TDMCCHYP$_{Pruning}$	0.0860 ×	3.4495 ×	1.0623 ×	0.0190 ×	2.9595 ×	0.6519 ×

TABLE III

MINIMUM, MAXIMUM AND AVERAGE OF THE NORMED RUNTIMES FOR COMPLEX PREDICATE GRAPHS.

Algorithm	min	max	avg	min	max	avg
	random Acyclic			random Cyclic		
DPHYP	0.0001 s	0.4759 s	0.0106 s	0.0003 s	85.4760 s	4.6123 s
TDBASICHYP	1.8182 ×	28488.7710 ×	730.0430 ×	0.9555 ×	43728.5736 ×	16.2903 ×
TDMCCHYP	0.6316 ×	2.7727 ×	1.2418 ×	0.7500 ×	2.3330 ×	1.1894 ×
TDMCCHYP$_{Pruning}$	0.0370 ×	2.5000 ×	0.8539 ×	0.0018 ×	1.6668 ×	0.0978 ×

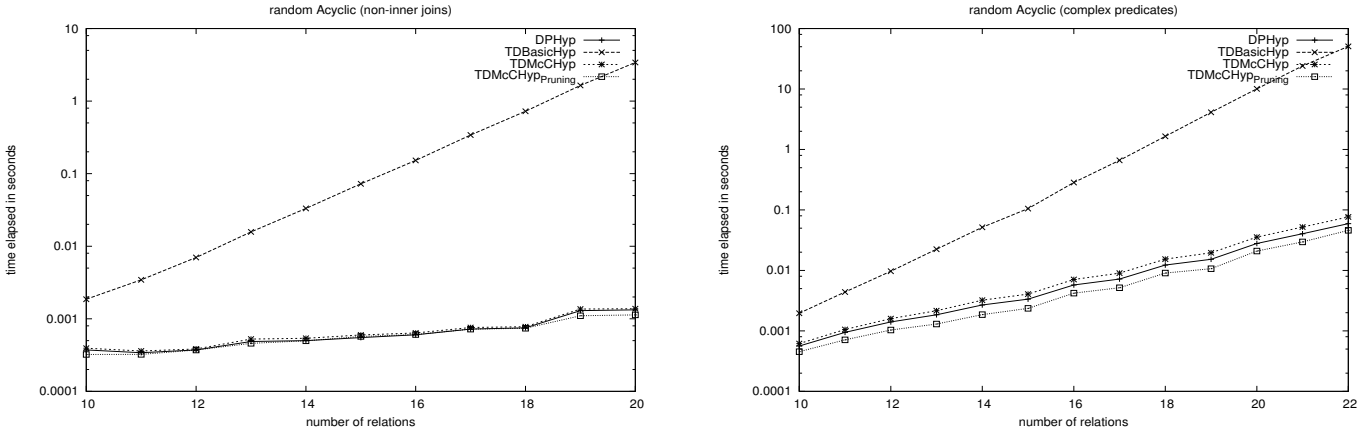

Fig. 14. Performance results of random acyclic graphs (left: non-inner join graphs, right: complex predicate graphs).

DPHYP. As the minimal normed runtime of 0.12 indicates, TDMCCHYP has a clear advantage over DPHYP in certain scenarios because TDMCCHYP enlarges C by the whole adjacent hypernode at once. DPHYP on the other hand incrementally enlarges a connected set by one member of the adjacent hypernode at a time. There can be as many invocations of DPHYP's sub-methods without any emission of ccps as there are proper connected subsets of the adjacent hypernode. However, the number of connected subsets is limited to the number of sets that contain the same representative vertex of the neighborhood (see [10] for more details). In the worst case this number corresponds to $2^{|n|-1}-2$, with $|n|$ being the size of the hypernode. Hence we can conclude that TDMCCHYP is more robust.

We can see from Table II and Table III that TDMCCHYP's normed runtime is on average slightly lower for non-inner join graphs than it is for complex predicate graphs. This is because non-inner join queries do not contain hyperedges that are part of a cycle (Sec. V-B). Hence as explained in Sec. IV-C CHECKXMAP is never evaluated when non-inner join queries are considered.

In conclusion, TDMCCHYP is clearly to be favoured over TDBASICHYP. Compared to DPHYP, it is slightly slower but its runtime behaviour is more reliable. Next, we will discuss how TDMCCHYP benefits from branch-and-bound pruning.

3) Analyzing the Pruning Performance: In order to study the pruning capabilities of TDMCCHYP, we enhanced it with the improved accumulated-predicted cost bounding method, as proposed in [4].

The overall pruning performance is given in Tables II and III. We can see that on average, TDMCCHYP$_{pruning}$ has a lower normed runtime than TDMCCHYP. The maximal normed runtimes in all different workloads are a little higher than TDMCCHYP but they are never twice as high.

If we compare TDMCCHYP$_{pruning}$ to DPHYP, we see that on average, TDMCCHYP$_{pruning}$ dominates DPHYP distinctly. The only exception to this are random acyclic non-inner join queries.

Let us dwell more into the details by considering the density plots of Figure 16. The plots show how the normed runtime results are distributed. Figure 16 depicts the density plots for random acyclic queries and for cyclic queries with 10 and 15 vertices. Additionally, we distinguished between non-inner join graphs (on the left of Figure 16) and complex predicate queries (on the right of Figure 16).

The reason why DPHYP outperforms TDMCCHYP$_{pruning}$ for random acyclic non-inner join queries lies in the size of the average hypernode of those hypergraphs. As discussed in Section V-D.2, the number of ccps is very low for the given number of vertices in this particular workload. Since there are not so many different plans, TDMCCHYP$_{pruning}$ has almost

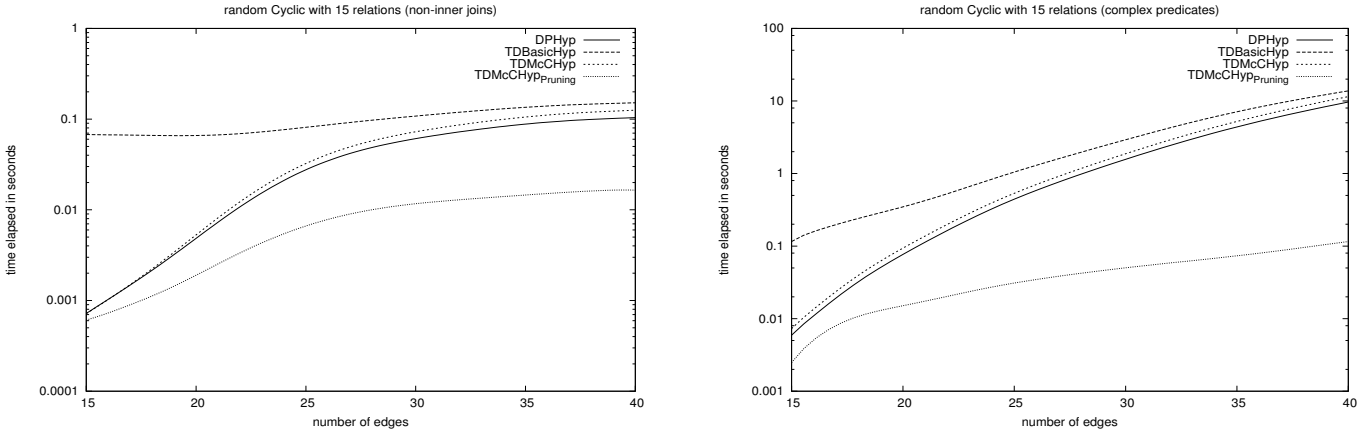

Fig. 15. Performance results of random cyclic graphs with 15 vertices (left: non-inner join graphs, right: complex predicate graphs).

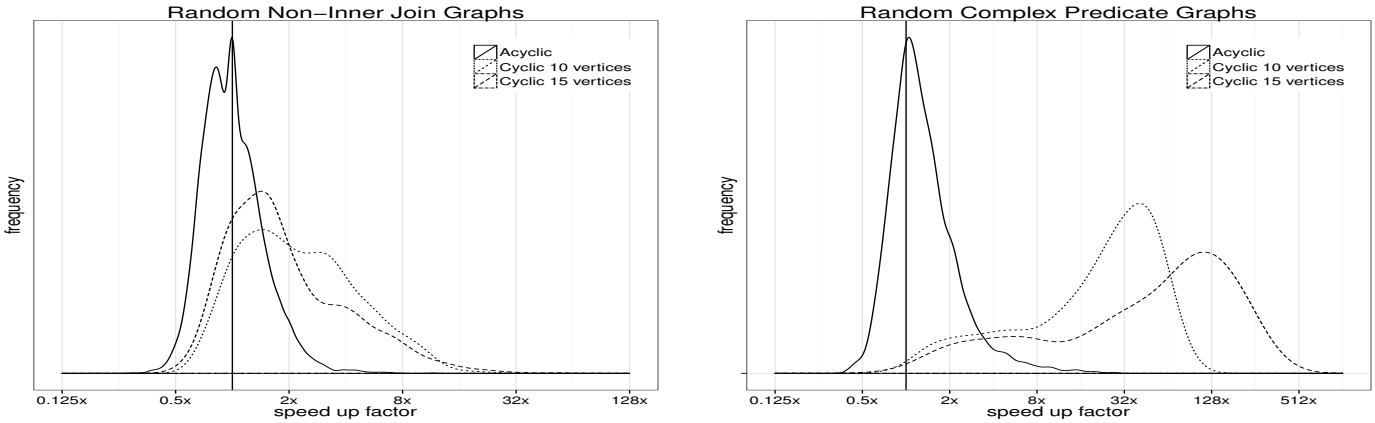

Fig. 16. Density plots of the normed runtime of TDMcCHyp$_{pruning}$ (left: non-inner join graphs, right: complex predicate graphs).

no possibilities to prune plans. We can see this when looking at the small performance gains between the averaged normed runtimes of TDMcCHyp$_{pruning}$ and TDMcCHyp. However, one needs to keep in mind that the absolute runtimes are very low. When we look at the random cyclic queries, we can see factors of up to 550 of performance gains. Those factors will rise when graphs with more numbers of vertices, a higher number of edges and with a higher simple edge to complex edge ratio are considered.

VI. Conclusion

With TDMcCHyp, we have presented the first top-down join enumeration algorithm that can handle query graphs that have the form of complex hypergraphs. Our empirical analysis has shown that TDMcCHyp performs almost as good as DPHyp, which is the state-of-the art in bottom-up join enumeration and TDMcCHyp's runtime behaviour is even more robust. Exploiting its capabilities to apply branch-and-bound pruning techniques TDMcCHyp$_{pruning}$ outperforms DPHyp almost in every workload scenario.

We belief that with TDMcCHyp top-down join enumeration is ready to be used in practice.

Acknowledgment. We thank Simone Seeger for her help preparing this manuscript and the referees for their thorough feedback.

References

[1] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization," in *VLDB*, 1990.
[2] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products," in *VLDB*, 2006.
[3] P. Fender and G. Moerkotte, "A new, highly efficient, and easy to implement top-down join enumeration algorithm," in *ICDE*, 2011.
[4] P. Fender, G. Moerkotte, T. Neumann, and V. Leis, "Effective and robust pruning for top-down join enumeration algorithms," in *ICDE*, 2012.
[5] P. Fender and G. Moerkotte, "Reassessing top-down join enumeration," *IEEE TKDE*, vol. 24, no. 10, pp. 1803–1818, 2012.
[6] G. Bhargava, P. Goel, and B. Iyer, "Hypergraph based reorderings of outer join queries with complex predicates," 1995.
[7] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen, "Using EELs: A practical approach to outerjoin and antijoin reordering," 2001.
[8] J. Ullman, *Database and Knowledge Base Systems*. Computer Science Press, 1989, vol. Volume 2.
[9] C. Galindo-Legaria and A. Rosenthal, "Outerjoin simplification and reordering for query optimization," in *TODS*, vol. 22, no. 1, 1997.
[10] G. Moerkotte and T. Neumann, "Dynamic programming strikes back," in *SIGMOD*, 2008.
[11] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration," in *SIGMOD*, 2007.
[12] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla, "Seeking the truth about ad hoc join costs," *VLDB Journal*, vol. 6, 1997.
[13] B. Vance and D. Maier, "Rapid bushy join-order optimization with cartesian products," in *SIGMOD*, 1996.