Counter Strike: Generic Top-Down Join Enumeration for Hypergraphs

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

Guido Moerkotte
University of Mannheim
Mannheim, Germany
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 to 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.

1. 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. Essential for the execution costs of a plan is the order of join operations, since the runtime of plans with different join orders can vary by several orders of magnitude. The search space considered here consists of all bushy join trees without cross products [16].

In principle, there are two 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: DPCCP, an efficient dynamic programming-based algorithm [12], TDMINCUTLAZY [3], as well as TDMINCUTBRANCH and TDMINCUTCONSERVATIVE), two competitive top-down join enumeration strategies [4, 7, 5].

However, all four algorithms (DPCCP, TDMINCUTLAZY, TDMINCUTBRANCH, 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 [2, 17, 19]. Second, plan generators have to deal with outer joins and antijoins [8, 17]. In general, these operators are not freely reorderable: some orderings produce wrong results. The non-inner join reordering problem can be correctly reduced to hypergraphs [2, 13, 17]. Consequently, Moerkotte and Neumann [13] 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 [3, 7], while still guaranteeing plan optimality.

In this paper, we present a novel generic framework that can be used by any existing partitioning algorithm for top-down join enumeration to efficiently handle hypergraphs. The central idea is to smartly convert hypergraphs to simple graphs and introduce effective means to avoid inefficiencies. This way, any existing partitioning algorithm for simple graphs can be used. We show that TDMCBHYP, resulting from instantiating our framework with the partitioning algorithm MINCUTBRANCH, is more efficient than existing partitioning algorithms for hypergraphs and as efficient as DPHYP even without pruning. With pruning, TDMCBHYP outperforms DPHYP by a factor of $1.1 - 11.5$.

This paper is organized as follows. Sec. 2 recalls some preliminaries. Sec. 3 shows a naive approach called TDBASICHYP for handling hyperedges. Sec. 4 presents our generic framework. Sec. 5 contains the experimental evaluation, and Sec. 6 concludes the paper.

2. PRELIMINARIES

Before we give the formal definitions necessary for our algorithm, let us demonstrate by means of a very simple example why hypergraphs (apart from the case where join predicates span more than two relations) are necessary when reordering more than plain joins. Consider the query

```
select * from   (R0 left outer join R1 on R0.A = R1.B)
                full outer join R2 on R1.C = R2.D
```

In a first step, it is translated into an initial operator tree:

$$(R_0 ⟕_{R_0.A=R_1.B} R_1) ⟗_{R_1.C=R_2.D} R_2$$

For this query, no valid reordering is possible. To prevent reordering, conflicts need to be detected and represented. At the core of every conflict presentation is a set of relations, called TES, associated with each operator in the initial operator tree [13, 17, 11]. To

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/14... $ 10.00.

describe the calculation of TES is beyond the current paper, but the intuition behind it is rather simple: before an operator can be applied to join two subplans, all relations in the TES must be present in the two subplans. For our example, we have $\texttt{TES}(\Join_{R_0.A=R_1.B}) = \{R_0, R_1\}$ and $\texttt{TES}(⟖_{R_1.C=R_2.D}) = \{R_0, R_1, R_2\}$. For non-commutative operators, it is important to distinguish between the relations contained in the left and right branch of the initial operator tree [1]. Intersection of the TES with the set of relations contained in the left and right branch of the operator tree gives a pair (L-TES, R-TES) of sets of relations. For $⟖_{p_{1,2}}$, this pair is $(\{R_0, R_1\}, \{R_2\})$. As we will see, this is a complex hyperedge.

2.1 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 (v, w) of non-empty subsets of V ($v \subset V$ and $w \subset V$) with the additional condition that $v \cap w = \emptyset$.*

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 (v, w) is simple *if $|v| = |w| = 1$. 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.*

Take a look at the complex hypergraph in Fig. 4(a) with $V = \{R_0, R_1, R_2, R_3\}$. Here, we have two simple edges $(\{R_0\}, \{R_2\}), (\{R_1\}, \{R_2\}))$ and two complex hyperedges $(\{R_0, R_2\}, \{R_3\}), (\{R_1, R_2\}, \{R_3\})$. Fig. 4(b) depicts a simple hypergraph.

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 $E' = \{(v, w) | (v, w) \in E, v \subseteq V', w \subseteq V'\}$. The node ordering on V' is the restriction of the node ordering of V.*

Next, 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 $(v, w) \in E$ such that $v \subseteq V'$, $w \subseteq V''$, and both $H|_{V'}$ and $H|_{V''}$ are connected.*

The node-induced subgraph $H|_{\{R_0, R_2, R_3\}}$ gained from the hypergraph of Fig. 4(a) with $E = \{(\{R_0\}, \{R_2\}), (\{R_0, R_2\}, \{R_3\})$ is connected, whereas $H|_{\{R_0, R_1, R_3\}}$ with $E = \emptyset$ is not.

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, we call V' a *connected subgraph* or *csg* for short. The number of connected subgraphs is important: it directly corresponds to the number of entries in the memotable.

We assume that all hypergraphs used here are connected. If not, we introduce complex hyperedges with a selectivity of one that join two disconnected subgraphs at a time.

For our framework, the notion of an *articulation hyperedge* is essential. We give its definition.

[1] For commutative operators it does not harm.

DEFINITION 4. *Let $H = (V, E)$ be a connected hypergraph, then we call a hyperedge (v, w) an articulation hyperedge if removing (v, w) from E would disconnect the graph H.*

All edges of the hypergraph shown in Fig. 5(a) are articulation hyperedges. The graph given in Fig. 4(a) has no complex articulation hyperedges, but two simple ones: $(\{R_0\}, \{R_2\})$ and $(\{R_1\}, \{R_2\})$. The graph in Fig. 4(b) has none. We observe that an articulation hyperedge cannot be part of any cycle. Hence, we call a hypergraph whose complex hyperedges are all articulation hyperedges a complex cycle-free hypergraph. Example cycle-free graphs are shown in Fig. 4(c) and Fig. 16(a).

2.2 Connected Subgraph and Its Complement Pairs

Our focus is on determining 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* (*ccp*) [12].

DEFINITION 5. *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*
- *$\exists (v, w) \in E \mid v \subseteq S_1 \land w \subseteq S_2$.*

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 6. *Let $H = (V, E)$ be a connected hypergraph 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.*

Note that if (S_1, S_2) is a ccp for S, then (S_2, S_1) is one as well. We call them symmetric pairs. By $P_{ccp}(S)$, we denote the set of all ccps for S. $P_{ccp}(\{R_0, R_1, R_2, R_3\})$ for the hypergraph of Fig. 4(a) consists of 6 *ccps*: $\{(\{R_0\}, \{R_1, R_2, R_3\}), (\{R_0, R_1, R_2\}, \{R_3\}), (\{R_0, R_2, R_3\}, \{R_1\})\}$ (symmetric counter pairs left out).

2.3 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 7. *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:*

$$\begin{aligned}\mathcal{N}_s(C) \quad = \quad & \{x \mid x \in w \ \land \ (v, w) \in E \ \land \ v \subset C \ \land \\ & w \subset (V \setminus C) \ \land \ |v| = 1 \ \land \ |w| = 1\}.\end{aligned}$$

We now give the definition of *neighborhood* for all edges, including hyperedges.

DEFINITION 8. *Let $H = (V, E)$ be a connected hypergraph, S a set of nodes ($S \subseteq V$) such that $H_{|S}$ is connected. Then the* neighborhood *of $C \subset S$ is defined as:*

$$\mathcal{N}(S, C) \quad = \quad \{w \mid (v, w) \in E \wedge v \subseteq C \wedge w \subseteq S \setminus C\}.$$

For the hypergraph of Fig. 4(a) with $S = V = \{R_0, R_1, R_2, R_3\}$, $\mathcal{N}(S, \{R_0, R_2\}) = \{\{R_1\}, \{R_3\}\}$ and $\mathcal{N}_s(\{R_0, R_2\}) = \{R_1\}$ holds. Furthermore, $\mathcal{N}(S, \{R_3\}) = \{\{R_0, R_2\}, \{R_1, R_2\}\}$ and $\mathcal{N}_s(\{R_3\}) = \emptyset$ holds.

2.4 Path, Cycles and Compound Relations

The following two definitions are important for the description of the structure of a simple hypergraph.

DEFINITION 9. *Let $G = (V, E)$ be a simple hypergraph, then a* path $x \rightarrow^* y$ *with the* length l *between vertices x and y is defined as a sequence of vertices $\langle z_0, z_1, z_2, ..., z_l\rangle$ in V such that $x = z_0$ and $y = z_l$ and $(\{z_{i-1}\}, \{z_i\}) \in E$ for $i = 1, 2, ...l$.*

With the definition of a path, we can define cycles.

DEFINITION 10. *Let $H = (V, E)$ be a simple hypergraph. Then a* cycle *is a path $\langle z_0, z_1, z_2, ..., z_l\rangle$ with $\forall_{0 \leq i \leq l} z_i \in V$ where $z_0 = z_l$ holds.*

DEFINITION 11. *Let $H = (V, E)$ be a simple hypergraph. A node $a \in V$ is an* articulation vertex *iff there exist two vertices $x \in V$ and $y \in V$, such that every path $x \xrightarrow{*} y$ in V contains a.*

The articulation vertices of a connected simple hypergraph $H = (V, E)$ are important when determining the biconnected components of a simple hypergraph.

DEFINITION 12. *Let $H = (V, E)$ be a connected simple hypergraph. A* biconnected component *is a connected subgraph $H_i^{BCC} = (V_i, E_i)$ of H with $H_i = \{v \mid (v = u \vee v = w) \wedge (v, w) \in E_i\}$, where the set of edges $E_i \subseteq E$ is maximal such that any two distinct edges $(u, w) \in E_i$ and $(x, y) \in E_i$ lie on a cycle $\langle v_0, v_1, v_2, ..., v_l\rangle$, where $u = v_0 \wedge u = v_l \wedge w = v_1 \wedge x = v_{j-1} \wedge y = v_j \wedge 0 < j < l$ and $\forall_{0 \leq i < j < l} v_i, v_j \in V \wedge v_i \neq v_j$ holds. If for an edge $(u, w) \in E_i$ no such cycle exists, the nodes $u, w \in V_i$ induce a biconnected component $H_i^{BCC} = (\{u, w\}, \{(u, w)\})$.*

The simple graph of Fig. 4(b) has just one biconnected component. Hence, there exists no articulation vertex and $H^{BCC} = (V, E)$ holds. Fig. 4(c) consists of 3 biconnected components: $H_1^{BCC} = H|_{\{R_0, R_2\}}$, $H_2^{BCC} = H|_{\{R_1, R_2\}}$, and $H_3^{BCC} = H|_{\{R_2, R_3\}}$. Thereby, R_2 is the only articulation vertex. Next, we define *compound relations*:

DEFINITION 13. *Let $H = (V, E)$ be a hypergraph. A* compound relation *u represents a group of nodes $V' = \{v_0, v_1, ...v_n\}$, where $v_i \in V$ and $u \notin V$ holds.*

3. BASIC MEMOIZATION

Although the basic variant of memoization has been discussed extensively elsewhere [3], we repeat it here since it requires some modifications in order to deal with hypergraphs [6]. It consists of three parts. The first part contains the top-level invocation together with the main recursion (Fig. 1). Its input consists of a hypergraph G and a set of (join) operators O. Both are derived from some input SQL query (see [11] for details). Like dynamic programming, TDPLANGENHYP first initializes the building blocks for single relations and adds them to the lookup table *BestTree*. It then calls the recursive routine TDPGSUB for the whole set V of nodes. TDPGSUB checks for the presence of an already derived best plan for any input set of nodes S. If such a plan does not exist, TDPGSUB iterates over all ccps (S_1, S_2) of S. If an operator is applicable (see below), the subroutine BUILDTREE generates the according plans and adds them to the lookup table (Fig. 2). BUILDTREE also considers interesting orders the usual way [15, 14]. The applicability test (Line 4) includes L-TES $\subseteq S_1 \wedge$ R-TES $\subseteq S_2$ and ensures correctness of the generated plan [11].

Whereas Line 2 declaratively specifies the set of ccps to be considered, any real implementation must provide a procedure to generate them explicitly. This is the third and exchangeable part of TDPLANGENHYP. One possibility is the naive partioning algorithm (Fig. 3). In its Line 1, all $2^{|S|} - 2$ possible non-empty and proper subsets of S are enumerated (see [20] for the efficient enumeration). 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 (for a connection test see [6]). The third condition that C needs to be connected to $S \setminus C$ is implied by the requirement that the (sub)graph handed over as input is connected. The frequent failure of this test is the main source of inefficiency of the basic partitioning algorithm.

```
TDPLANGENHYP(H, O)
   ▷ Input: connected H = (V, E), O set of operators
   ▷ Output: an optimal join tree for H
1  for i ← 1 to |V|
2      BestTree[{R_i}] ← R_i
3  return TDPGSUB(V)

TDPGSUB(S, O)
   ▷ Input: H_|S connected
   ▷ Output: an optimal join tree for H_|S
1  if BestTree[S] = NULL
2      for all (S_1, S_2) ∈ P_ccp(S)
3          for ∘ ∈ O
4              if APPLICABLE(∘,S_1,S_2)
5                  BUILDTREE(∘, TDPGSUB(S_1), TDPGSUB(S_2))
6  return BestTree[S]
```

Figure 1: Pseudocode for TDPLANGENHYP

4. GRAPH-BASED JOIN ENUMERATION

This section describes our generic framework that enables any existing partitioning algorithm for top-down join enumeration to deal with hypergraphs. In particular, we use it to enhance MINCUTBRANCH, which results in a novel partitioning algorithm that we call MINCUTBRANCHHYP. We call the instantiated top-down join enumeration variant TDMCBHYP. In [6], we constructed MINCUTCONSERVATIVEHYP as a derivative of MINCUTCONSERVATIVE that is part of TDMCCHYP. But the techniques we used there cannot be applied to other partitioning algorithms. This section presents an approach which is generic, more efficient in terms of performance and, thus, superior.

4.1 High-Level Overview

```
BUILDTREE(∘, S, Tree_1, Tree_2)
   ▷ Input: induced graph H_|S, two optimal partial plans
1  CurrentTree ← CREATETREE(Tree_1, Tree_2)
2  if BestTree[S] = NULL ||
       cost(BestTree[S]) > cost(CurrentTree)
3      BestTree[S] ← CurrentTree
4  if ∘ is commutative
5      CurrentTree ← CREATETREE(Tree_2, Tree_1)
6      if cost(BestTree[S]) > cost(CurrentTree)
7          BestTree[S] ← CurrentTree
```

Figure 2: Pseudocode for BUILDTREE

$\text{PARTITION}_{naive}(H_{|S})$

▷ **Input:** a connected (sub) graph $H_{|S}$

▷ **Output:** $P_{ccp}(S)$

1 **for all** $C \subset S \land C \neq \emptyset$

2 **if** $\text{isConnected}(H_{|C}) \land \text{isConnected}(H_{|S \setminus C})$

3 $P_{ccp} \leftarrow P_{ccp} \cup \{(C, S \setminus C)\}$

Figure 3: Pseudocode for naive partitioning

To explain our main ideas, let us make four important observations. These will highlight the problems we face and indicate solutions.

First, assume that we have two simple connected graphs $H_v = (v, E_v)$ and $H_w = (w, E_w)$ with $|v| > 1 \lor |w| > 1$. Now we want to connect both graphs. Introducing a complex hyperedge (v, w) covering the whole vertex sets on both sides is *more restrictive* than introducing a simple edge $(\{x\}, \{y\})$ where $x \in v \land y \in w$ holds. Here, we use the term *more restrictive* in the sense that partitioning the resulting set $v \cup w$ into all possible *ccp*s (Sec. 2.2) leaves us with much fewer choices if a complex hyperedge ($|P^{ccp}(v \cup w)| = 1$) is introduced instead of a simple edge. Consider the disconnected graph $H = (V = v \cup w, E = \{(\{R_0\}, \{R_1\})\})$ with $v = \{R_0, R_1\} \land w = \{R_2\}$. On the one hand, if we connect v and w by the complex hyperedge (v, w), then $P^{ccp}(v \cup w)$ has two *ccp*s: $(\{R_0, R_1\}, \{R_2\})$ and $(\{R_2\}, \{R_0, R_1\})$. One the other hand, if we choose $(\{R_0\}, \{R_2\})$, this gives rise to two additional *ccp*s: $(\{R_0, R_2\}, \{R_1\})$ and $(\{R_1\}, \{R_0, R_2\})$. Hence, the latter case is less restrictive.

For the **second** observation, we take a look at the naive partitioning strategy (Fig. 3). Line 1 of PARTITION_{naive} enumerates $2^{|v \cup w|} - 2$ subsets of $S = v \cup w$. Adding the complex hyperedge to H, only $C = v$ or $C = w$ make it past Line 2. This is clearly inefficient, since all other generated subsets of $S = v \cup w$ are rejected. Assume that we substitute the complex hyperedge (v, w) by a simple edge $(\{x\}, \{y\})$. Then, the graph becomes simple again. As a consequence, we can reuse a highly efficient graph-aware partitioning algorithm for simple graphs (e.g. MINCUTBRANCH). However, we have to be careful, since complex hyperedges are more restrictive and, thus, by converting hyperedges to simple edges, invalid *ccp*s might be generated. Therefore, we need to check the *ccp*s resulting from simple graphs for connectivity within the original hypergraph. We call partitions that are not valid *ccp*s of the original complex hypergraph *false ccp*s. In the example used in the first observation, the false *ccp*s are $(\{R_0, R_2\}, \{R_1\})$ and $(\{R_1\}, \{R_0, R_2\})$.

Third, if we represent a complex hyperedge (v, w) by a simple edge, there are $|v| * |w|$ possibilities to do so. For the graph presented in Fig. 4(a), the call to $\text{PARTITION}_{naive}(\{R_0, R_1, R_2, R_3\})$ generates 14 subsets assigned to C, but only $C = \{R_0\}$, $\{R_0, R_1, R_2\}$ and $\{R_0, R_2, R_3\}$ survive the test in Line 2. Thus, there exist only six valid *ccp*s as listed in Sec. 2.2. For the hypergraph given in Fig. 4(b), a graph-aware partitoning algorithm generates 12 partitions and therefore 6 false *ccp*s. Since the two hyperedges of Fig. 4(a) overlap, the mapping of Fig. 4(c) is one of the 4 possible combinations. Here, not a single false *ccp* is generated. We conclude that in certain cases, there are good (restrictive) and bad (less restrictive) mappings.

In Sec. 4.3, we present COMPUTEADJACENCYINFO and show how it exploits the first three observations.

Now, take a look at Fig. 5(a). A call to $\text{PARTITION}_{naive}(\{R_0, R_1, R_2, R_3, R_4\})$ results in the generation of 30 subsets assigned to C where just $C = \{R_0, R_1, R_2, R_3\}$ makes it past Line 2. Invocating COMPUTEADJACENCYINFO (Sec. 4.3) produces the simple graph of Fig. 5(b). Taking that graph as an input for a call

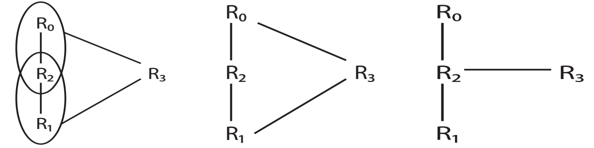

Figure 4: (a) overlapping hyperedges, (b) and (c) simple graphs

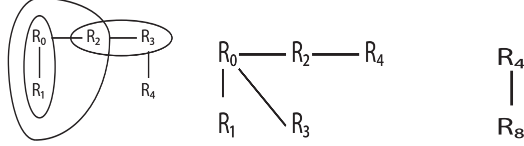

Figure 5: (a) hypergraph, (b) simp. graph and (c) final graph

to any graph-aware partitioning algorithm would return four pairs: $(\{R_0, R_1, R_2, R_3\}, \{R_4\})$, $(\{R_0, R_2, R_3, R_4\}, \{R_1\})$, $(\{R_0, R_1, R_3\}, \{R_2, R_4\})$ and $(\{R_0, R_1, R_2, R_4\}, \{R_3\})$ (symmetric counter pairs left out). But only the first partition is a valid *ccp*. Hence, the produced simple graph of Fig. 5(b) is less restrictive than the original one. In Fig. 5(a) we can see that R_1 cannot be separated from R_0, since otherwise, the connection to R_2 would be lost. Furthermore, R_2 cannot be separated from R_0, R_1, or the connection to R_3 would be lost. On top of that, R_2 and R_3 have to remain in the same subgraph, or the connection to R_4 breaks up. Concluding, it is only possible to separate R_4 from the rest, because all other combinations would end up in more than two connected subsets and therefore false *ccp*s.

From the example, we can draw our **fourth** observation: If a complex hyperedge (v, w) is essential for the connectedness of the hypergraph, i.e., it is an articulation hyperedge, then it is impossible to partition the graph by separating one or two of its complex hypernodes v or w. In other words: There exists no minimal cut involving an edge (s, t) with $s \subset v \land t \subset v$ within a hypernode v that is part of an articulation hyperedge (v, w).

In order to benefit from observation 4, we propose the concept of a compound relation (Def. 13). The basic idea is to group those vertices that compose a non-separable hypernode into a new artificial vertex. Particularly, we remove those vertices from the vertex (sub)set $S \subseteq V$ that have been grouped and introduce the compound relations as a new v by adding it to S (actually to some S', as we will see). In case that non-separable hypernodes are overlapping, we group all overlapping vertex sets together. Those steps are performed through COMPOSECOMPOUNDRELATIONS (Sec. 4.4). The result of this step is shown in Fig. 5(c), where R_8 is the compound relation representing R_0, R_1, R_2, R_3 of Fig. 5(a-b).

4.2 Embedding into the Framework

Fig. 6 gives PARTITION_X, which contains calls to all main functions of our framework. Line 1 calls COMPUTEADJACENCYINFO in order to map complex hyperedges temporarily to simple edges. Line 2 transforms certain hypernodes into compound relations in order to (1) regain some of the restrictiveness of the transformed hyperedges and to (2) speed up processing, since less vertices are involved. Line 3 determines whether the relatively expensive connection test assessing connectivity based on the original hypergraph is needed. The partitioning algorithm called in Line 4 only sees a simple graph with intermixed original and artificial vertex nodes (compound relations). Importantly, it does not needs any knowledge about the vertices representing compound relations. Finally, we loop through the emitted partitions of the simple graph (L. 5). We decode the emitted partitions (Sec. 4.4.3) by substituting the compound relations with the original vertices (L. 6, 7) and apply the connection test (L. 9) if needed (L. 8). Note that if a connection test is necessary (Sec. 4.5.1), the last step is very important

```
PARTITION_X(H_{|S})
    ▷ Input: H_{|S}
1   COMPUTEADJACENCYINFO(H_{|S})                       ▷ Sec. 4.3
2   S' ← COMPOSECOMPOUNDRELATIONS(H_{|S})              ▷ Sec. 4.4
3   con ← do we need connections tests                 ▷ Sec. 4.5.1
4   P^{sym}_{partitions} ← graph-aware partitioning algorithm (S')
5   for all (l', r') ∈ P^{sym}_{partitions}
6       l ← DECODE(S, l')                              ▷ Sec. 4.4.3
7       r ← DECODE(S, r')                              ▷ Sec. 4.4.3
8       if con = TRUE
9           if ISCONNECTED(H_{|l}) ∧ ISCONNECTED(H_{|r})
10              P_ccp ← P_ccp ∪ {(l, r)} ∪ {(r, l)}
11      else P_ccp ← P_ccp ∪ {(l, r)} ∪ {(r, l)}
```

Figure 6: Pseudocode for PARTITION$_X$

in order to filter out false *ccp*s. Missing to filter out false *ccp*s results in the generation of (sub)plans that rely on cross products and might be invalid [11]. Let $E_{comp} = \{(v, w) \mid v, w \in E \wedge (|v| > 1 \vee |w| > 1)\}$ be the set of complex hyperedges, then the complexity of the preprocessing step is in $O(|E_{comp}| * \frac{|V|^2}{2})$. The complexity of the enumeration algorithm in Line 4 remains unchanged. The complexity of the two additional connectivity tests is in $O(|V| + |\frac{|E_{comp}|^2}{2})$ per emitted *ccp* (false *ccp*s included). Note that in many cases the two tests can be avoided (see Sec. 4.5.1).

4.3 Generating the Adjacency Information

All graph-aware partitioning algorithms like MINCUTBRANCH [4], MINCUTAGAT [5], MINCUTLAZY [3] or MINCUTCONSERVATIVE [7] utilize the neighborhood information to extend connected sets. For our generic framework, we have to provide this information to the graph-aware partitioning algorithms. We therefore introduce the global variables shown in Fig. 7. Essentially, the algorithms have to rely only on the associative arrays N_s, N_h, which contain a representation of precomputed simple and hyper neighbourhoods. The other variables mainly exist for performance reasons and are explained below. The initial setup is done by COMPUTEADJACENCYINFO given in Fig. 9. First, we compute the simple neighborhood by iterating over all simple edges (L. 3 to 5). For Lines 6 and 7, we refer to Sec. 4.4.1, 4.4.2.

To understand Lines 10 to 37, recall the third observation. We solve the problem illustrated there by first pretending to substitute every complex hyperedge (v, w) with all possible combinations $(= |v| * |w|)$ of simple edges (L. 3 to 5). For every overlapping edge, we increase $card$ (L. 15). For every combination of indices i of $x_i \in v$ and j of $y_j \in w$, we compute an entry in the array $Ovlp$ (L. 1) by a call to COMPUTELOOKUPIDX (L. 14). The formula used in Line 1 of COMPUTELOOKUPIDX (Fig. 8) guarantees space efficiency ($\text{SIZEOF}(Ovlp) = \frac{|V|*(|V|-1)}{2}$). In Line 20 and 21 of COMPUTEADJACENCYINFO, we keep track of the simple edge and its array entry that is generated most frequently. After the generation of simple edges (they are not materialized yet), we check if we have found overlapping hyperedges (L. 22). If so, we materialize the simple edge that was generated most frequently (L. 29). At this point, we remove all other combinations of simple edges (L. 25 to 28) for the set of overlapping edges stored in $Ovlp[idx].E$ (L. 23). Lines 32 to 35 spot the next largest set of overlapping hyperedges, and the process is started again. Those complex hyperedges that do not overlap are substituted with one simple edge in Lines 36 and 37 through a call to STOREADJACENCYINFO (Fig. 10). We decided to store the substituted complex hyperedges not within the simple neighborhood N_s, but within N_h, where $_h$ stands for hyperneighborhood, although it is not an exact translation (Def. 8). Besides setting N_h, STOREADJACENCYINFO also updates $HEdgeLkp$, which keeps track of which (set of) complex hyperedges is mapped to a given simple edge $(\{x\}, \{y\})$.

```
1   declare N_s associative array of vertex sets
2   declare N_h associative array of vertex sets
3   declare HEdgeLkp array of hyper edge references
4   declare Label_map a map<vertex set → vertex set>
5   declare RevLabel_map a map<vertex set → vertex set>
6   declare Compound_map a map<vertex set → vertex set>
```

Figure 7: Global Variables

```
COMPUTELOOKUPIDX(x_i, y_j)
    ▷ Input: vertex labels i, j with x_i ∈ S ∧ y_j ∈ S
1   return MAX(i,j)*(MAX(i,j)-1)/2 + MIN(i, j)

MAINTAINLABELS(v)
    ▷ Input: vertex set v ∈ S
1   if KEYDOESNOTEXIST(Label_map, v)
2       z_k ← new vertex labeled k ← SIZE(Label_map)
3       Label_map[v] ← {z_k}
4       RevLabel_map[{z_k}] ← v
```

Figure 8: Additional Pseudocode

```
COMPUTEADJACENCYINFO(H_{|S})
    ▷ Input: vertex (sub)set S ⊆ V
1   declare Ovlp as an array of a struct (E, card, x, y)
2   declare HEdges_set ← ∅ as a set of hyper edges
3   for all ({x}, {y}) ∈ E
4       N_s[x] ← N_s[x] ∪ {y}
5       N_s[y] ← N_s[y] ∪ {x}
6   for all v : (v, w) ∈ E ∨ (w, v) ∈ E
7       MAINTAINLABELS(v)       ▷ adding artificial nodes
8   max ← 0
9   idx ← 0
10  for all (v, w) ∈ E | |v| > 1 ∨ |w| > 1
11      HEdges_set ← HEdges_set ∪ {(v, w)}
12      for all x_i ∈ v
13          for all y_j ∈ w
14              lkp ← COMPUTELOOKUPIDX(x_i, y_j)
15              Ovlp[lkp].card ← Ovlp[lkp].card + 1
16              Ovlp[lkp].E ← Ovlp[lkp].E ∪ {(v, w)}
17              Ovlp[lkp].x ← x_i
18              Ovlp[lkp].y ← y_j
19              if Ovlp[lkp].card > max
20                  max ← Ovlp[lkp].card
21                  idx ← lkp
22  while max > 1               ▷ for overlapping hyperedges
23      for all (v, w) ∈ Ovlp[idx].E
24          HEdges_set ← HEdges_set \ {(v, w)}
25          for all x_i ∈ v
26              for all y_j ∈ w
27                  lkp ← COMPUTELOOKUPIDX(x_i, y_j)
28                  Ovlp[lkp].card ← Ovlp[lkp].card − 1
29      STOREADJINFO(Ovlp[idx].x, Ovlp[idx].y, Ovlp[idx].E)
30      max ← 0
31      idx ← 0
32      for all i : 0 ≤ i < SIZEOF(Ovlp)    ▷ find next max
33          if Ovlp[i].card > max
34              max ← Ovlp[i].card
35              idx ← i
36  for all (v, w) ∈ HEdges_set   ▷ |v| = 1 ∧ |w| = 1 holds
37      STOREADJINFO(x ∈ v, y ∈ w, {(v, w)})
```

Figure 9: Pseudocode for COMPUTEADJACENCYINFO

4.4 Composing Compound Relations

This section discusses how the information of non-separable hypernodes is encoded into the simple graph to make it more restrictive by preventing false *ccp*s (fourth observation Sec. 4.1).

4.4.1 Merging Compound Relations

In the following, we focus on the details of finding non-separable hypernodes and merging them into compound relations. The pro-

STOREADJINFO(x_i, y_j, E)

▷ **Input:** vertex x_i, y_j with $x_i \in S \wedge y_j \in S$
1 $N_h[x_i] \leftarrow N_h[x_i] \cup \{y_j\}$
2 $N_h[y_j] \leftarrow N_h[y_j] \cup \{x_i\}$
3 $lkp \leftarrow$ COMPUTELOOKUPIDX(x_i, y_j)
4 $HEdgeLkp[lkp] \leftarrow HEdgeLkp[lkp] \cup$ REFERENCES(E)

Figure 10: Pseudocode for STOREADJACENCYINFO

cess is started by invoking COMPOSECOMPOUNDRELATIONS. In Lines 2 to 11, the variables for the recognition of biconnected components are initialized. Hereby, only S' will be used later on. Line 12 invokes the recognition of the non-separable hypernodes. Upon GETBCCINFO's completion, S' will hold only original vertices that are not part of any non-separable hypernode and $Compound_{map}$ will store for the rest of the nodes $v \in S \setminus S'$ the mapping to their corresponding compound relations. Note that a given v can be mapped to more than one compound relation. Through the information stored in $Compound_{map}$, we merge overlapping hypernodes to a new compound relation that represents the union of hypernodes (L. 13 to 31).

Therefore, we loop in Line 14 through S'', which contains all original vertices that are represented by at least one compound relation. We declare h in order to store the union of overlapping hypernodes and initialize it in Line 15. With Z and I, we keep track of the compound relations that represent the overlapping hypernodes. I maintains those we already have investigated and Z those we still have to consider. Z is initialised (L. 18) with the compound relations that represent v (which was arbitrary chosen from S'' in Line 16). Within the loop in Lines 20 to 27, we investigate all compound relations contained in Z by incrementally removing relations in Line 22 and possibly adding relations in Line 26. Line 24 applies a reverse lookup (through $RevLabel_{map}$) of the compound relation u that was chosen out of Z (L. 21). We add to h the result of the lookup, which are the vertices represented by u. For every vertex x (L. 25) contained in one of the hypernodes in question, we consult $Compound_{map}$ (L. 26) to enlarge Z with compound relations that correspond to x minus those already investigated (and kept in I). That way, all compound relations in question have to be added at one point to Z either in Line 18 or Line 26. By incrementally taking one element at a time out of Z and adding the vertices it encompasses to the new hypernode h, we ensure that h gets maximally enlarged. Line 27 removes all vertices contained in h from S'', and the process continues until the last h of overlapping non-separable hypernodes is found.

We call MAINTAINLABELS (L. 28) in order to ensure that h gets a representative in form of an artificial vertex assigned. Note that if h contains just one hypernode and is no merger of overlapping ones, there is already a compound relation assigned to h. This is because then MAINTAINLABELS was already invoked with the same argument in Line 7 of COMPUTEADJACENCYINFO. In Line 29, the new vertex set S' (as returned later on in Line 33) is enlarged with the compound relation that represents the new h. Lines 30 and 31 make the compound relation known to the vertices it represents. Finally, we call MANAGEADJACENCYINFO in order to set up the precomputed neighborhoods N_s and N_h.

The pseudocode for MANAGEADJACENCYINFO is given in Fig. 12. Within the first loop (L. 1 to 7), we enhance the precomputed neighbourhoods by adding the corresponding compound relation for each (L. 3, 6) adjacent vertex that is represented by one (L. 4, 7). The second loop iterates over the compound relations and adds an entry into N_s and N_h for each of them. The corresponding value is set to the union of the precomputed neighbourhoods of all the vertices that are represented by the compound relation in question.

COMPOSECOMPOUNDRELATIONS($H_{|S}$)

▷ **Input:** connected (sub)graph $H_{|S}$
▷ **Output:** S' a set of vertex sets
1 **declare** $Compound'_{map}$ a map<vertex set, vertex set>
2 **declare** stack of edges E_{stack}
3 **for** each vertex $x \in S$
4   $color[x] \leftarrow$ WHITE
5   $low[x] \leftarrow |S| + 1$
6   $\pi[x] \leftarrow$ NIL
7   $parent[x] \leftarrow$ NIL
8   $desc[x] \leftarrow \{x\}$
9 $S' \leftarrow S$
10 $count \leftarrow 0$
11 $t \leftarrow$arbitrary $x \in S$
12 GETBCCINFO(t)
13 $S'' \leftarrow S'' \setminus S'$ ▷ cont. only vertices represented by artificial v.
14 **while** $S'' \neq \emptyset$
15   $h \leftarrow \emptyset$ ▷ h stores the new hypernode
16   $v \leftarrow \{y\} : y \in S''$
17   $S'' \leftarrow S'' \setminus v$
18   $Z \leftarrow Compound_{map}[v]$
19   $I \leftarrow \emptyset$
20   **while** $Z \neq \emptyset$
21     $u \leftarrow \{z\} : z \in Z$
22     $Z \leftarrow Z \setminus u$
23     $I \leftarrow I \cup u$
24     $h \leftarrow h \cup RevLabel_{map}[u]$
25   **for all** $x \in h$
26       $Z \leftarrow Z \cup (Compound_{map}[\{x\}] \setminus I)$
27     $S'' \leftarrow S'' \setminus h$
28   MAINTAINLABELS(h) ▷ adding artificial node for h
29   $S' \leftarrow S' \cup Label_{map}[h]$
30   **for all** $x \in h$
31     $Compound'_{map}[\{x\}] \leftarrow Label_{map}[h]$
32 MANAGEADJACENCYINFO
33 **return** S'

Figure 11: Pseudocode for COMPOSECOMPOUNDRELATIONS

MANAGEADJACENCYINFO

1 **for all** $x \in S$
2   $neighs \leftarrow N_s[x]$
3   **for all** $y \in neighs$
4     $N_s[x] \leftarrow N_s[x] \cup Compound'_{map}[\{y\}]$
5   $hyperneighs \leftarrow N_h[x]$
6   **for all** $y \in hyperneighs$
7     $N_h[x] \leftarrow N_h[x] \cup Compound'_{map}[\{y\}]$
8 **for all** $x \in S' \setminus S$
9   $h \leftarrow RevLabel_{map}[\{x\}]$
10   **for all** $y \in h$
11     $N_s[x] \leftarrow N_s[x] \cup N_s[y]$
12     $N_h[x] \leftarrow N_h[x] \cup N_h[y]$

Figure 12: Pseudocode for MANAGEADJACENCYINFO

4.4.2 Discovering Non-Separable Hypernodes

As has been said, GETBCCINFO is responsible for discovering the non-separable hypernodes. This is done by determining the complex articulation hyperedges. During the transformation of a complex hypergraph into a simple hypergraph, the complex articulation hyperedges are mapped to simple hyperedges. Now, if the complex hyperedge's substitute is recognized as a biconnected component (Def. 12) in the simple graph, this indicates that the complex hyperedge must be an articulation hyperedge. Actually, it is possible that there are overlapping hyperedges mapped to the same simple edge, but we will take care of this case. Thus, in order to determine non-separable hypernodes, we have to discover the biconnected components of the simple graph. Due to lack of space we cannot detail on the recognition of biconnected components and refer to [1, 5].

The condition in Line 10 of GETBCCINFO indicates, if evaluated to TRUE, that a biconnected component was found. More pre-

```
GETBCCINFO(x)
    ▷ Input: vertex x ∈ S
1   color[x] ← GRAY
2   count ← count + 1
3   df[x] ← count
4   low[x] ← df[x]
5   for all y ∈ (N_s[x] ∪ N_hyp[x]) : y ∈ S
6       if color[y] = WHITE
7           PUSH(E_stack, ({x}, {y}))
8           π[y] ← x
9           GETBCCINFO(y)
10          if low[y] ≥ df[x]
11              desc ← ∅
12              repeat  ({e1}, {e2}) ← POP(E_stack)
13                  if e1 ≠ x
14                      desc ← desc ∪ desc[e1]
15                      parent[e1] ← x
16                  if e2 ≠ x
17                      desc ← desc ∪ desc[e2]
18                      parent[e2] ← x
19              until ({e1}, {e2}) = ({x}, {y})
20              if low[x] = low[y]   ▷ is t articulation vertex?
21                  parent[x] ← x
22              if low[x] ≠ low[y] ∧ y ∈ N_Hyp[x]∧
                        desc ∩ N_Simp[x] = ∅
23                  FINDINITIALCOMPOUNDS(x,y)
24              desc[x] ← desc[x] ∪ desc
25          low[x] ← MIN(low[x], low[y])
26      else if y ≠ π[x]
27          PUSH(E_stack, ({x}, {y}))
28          low[x] ← MIN(low[x], df[y])
29  color[x] ← BLACK
```

Figure 13: Pseudocode for GETBCCINFO

cisely: It means that x is either the start node t (assigned in L. 11 and handed over in L. 12 of Fig. 11), or an articulation vertex was found that is the only link to another biconnected component. In Line 11, we declare $desc$ to store the descendants of x, i.e., all vertices z where every possible path $z \rightarrow^* t$ would involve x. Those descendants are gathered in Lines 14 and 17 and finally stored with (possibly) other descendants of x (x can be the parent vertex for several biconnected components) in Line 24.

Lines 12 to 19 will pop all edges $(\{e_1\}, \{e_2\})$ belonging to this biconnected component from the stack of edges E_{stack}. Thereby, we update $desc$ and set the parent for every vertex (L. 15, 18) in the biconnected component. As has been mentioned, in case $x = t$ holds it is possible that x is not an articulation vertex but only a member of the current biconnected component. In order to differentiate between the two cases later on, we set x's parent to itself (L. 21) if x is not an articulation vertex (L. 20).

Line 22 checks for several conditions: (1) if x is an articulation vertex $low[x] \neq low[y]$ and not just t, (2) if $(\{x\}, \{y\})$ substitutes an hyperedge and (3) if in the original hypergraph x is not connected to any other node $z \in desc$ by a simple edge. Only if all three conditions are met, FINDINITIALCOMPOUNDS is invoked in Line 23.

The pseudocode of FINDINITIALCOMPOUNDS is given in Fig. 14. Entering FINDINITIALCOMPOUNDS, we know that there must exist at least one complex hyperedge (v, w) in the original graph with $x \in v \wedge y \in w$. With the help of the lookup index computed from the labels i, j (L. 1), we get the hyperedge references of the original hypergraph via the global array $HEdgeLkp$ (which was set up by STOREADJINFO). At this point, it is possible that more than one reference is returned. In this case, the referenced complex hyperedges must overlap. Although not necessary, but for reasons of simplicity, we demand that just one reference exists (L. 3) before FINDINITIALCOMPOUNDSSUB for the hypernode v and w is called.

```
FINDINITIALCOMPOUNDS(x_i, y_j)
    ▷ Input: vertex set v ∈ S
1   lkp ← COMPUTELOOKUPIDX(x_i, y_j)
2   if |HEdgeLkp[lkp]| = 1
3       (v, w) ← HEdgeLkp[lkp]
4       FINDINITIALCOMPOUNDSSUB(v)
5       FINDINITIALCOMPOUNDSSUB(w)

FINDINITIALCOMPOUNDSSUB(v)
1   if |v| > 1
2       for all x ∈ v
3           S' ← S' \ {x}
4           Compound_map[{x}] ← Compound_map[{x}]∪
                        Label_map[v]  ▷ artificial v. as rep
```

Figure 14: Pseudocode for FINDINITIALCOMPOUNDS

FINDINITIALCOMPOUNDSSUB ensures that the handed over hypernode v is really complex (L. 1). If so, for every vertex x we (1) remove x from the vertex set S' and (2) x as part of the hypernode v is mapped to its corresponding compound relation $Label_{map}[v]$.

4.4.3 Decoding Compound Relations

As Sec. 4.1 explains, we need to substitute the compound relations in every emitted partition of the partitioning algorithm. This is done by DECODE as given in Fig. 15. In Line 1, we initialize *decoded* with the original vertices that are not represented by any compound relation. After that, we loop over the compound relations (L. 2) contained in C' and substitute them with the group of vertices they represent (L. 3).

4.4.4 Compound Relations - An Example

Let us get back to our motivation example for compound relations of Sec. 4.1. From the graph shown in Fig. 5(a), we gained the simple graph of Fig. 5(b) by calling COMPUTEADJACENCYINFO with $RevLabel_{map} = \{.., (\{R_5\} \rightarrow \{R_0, R_1\}), (\{R_6\} \rightarrow \{R_0, R_1, R_2\}), (\{R_7\} \rightarrow \{R_2, R_3\})\}$. But as it turned out, this was not restrictive enough, since three of the four generated partitions were false *ccp*s (symmetric counter pairs ignored). By invoking GETBCCINFO, we gain the initial mapping from members of non-separable hypernodes to compound relations: $Compound_{map} = \{(\{R_0\} \rightarrow \{R_5, R_6\}), (\{R_1\} \rightarrow \{R_5, R_6\}), (\{R_2\} \rightarrow \{R_6, R_7\}), (\{R_3\} \rightarrow \{R_7\})\}$ with $S' = \{\{R_4\}\}$. Once we reach Line 28 in COMPOSECOMPOUNDRELATIONS, MAINTAINLABELS($\{R_0, R_1, R_2, R_3\}$) is called. The resulting simple graph is given in Fig. 5(c). Any graph-aware partitioning algorithm will produce only one partition: $(\{R_4\}, \{R_8\})$ (and its symmetric partition $(\{R_8\}, \{R_4\})$). And finally, a call to DECODE($\{R_0, R_1, R_2, R_3, R_4\}, \{R_8\}$) returns $\{R_0, R_1, R_2, R_3\}$. Since both $\{R_4\}$ and the decoded set $\{R_0, ..., R_3\}$ are connected, the partition is proved to be a *ccp* and is returned without generating false *ccp*s.

4.5 Further Optimization Techniques

Let us briefly recall that non-inner joins are not freely reorderable because certain join reorderings result in different non-equivalent plans that return different query results when executed. Furthermore, it is well known that valid operator orderings can be

```
DECODE(S, C')
    ▷ Input: vertex sets S, C'
1   decoded ← S ∩ C'
2   for all x ∈ C' \ S        ▷ only artificial nodes
3       decoded ← decoded ∪ RevLabel_map[x]
4   return decoded
```

Figure 15: Pseudocode for DECODE

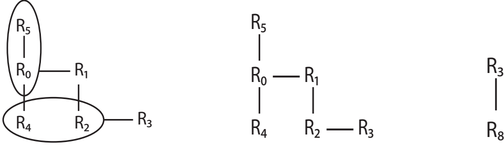

Figure 16: (a) hypergraph, (b) simp. graph and (c) final graph

encoded by transforming simple edges into hyperedges [2, 11, 13, 17]. Complex hypergraphs that are a result of those transformations can be mainly categorized as *complex cycle-free hypergraphs* (Sec. 2.1). We strongly believe that among all complex query graphs that can be found in real-world scenarios, the majority belongs to this category. The only common exception will be graphs that contain complex hyperedges originating from complex predicates.

4.5.1 Avoiding Connection Tests

For complex cycle-free hypergraphs, there are certain scenarios where we do not need a connection test. Since we have to run at least one connection test for every emitted join partition, saving the effort in doing so would increase efficiency significantly. In general, two conditions have to be met in order to be able to avoid the connection tests: (1) There are no vertices left in S' emitted by COMPOSECOMPOUNDRELATIONS which are also a member of any complex hypernode. This is an even weaker condition than that all complex hyperedges need to be articulation hyperedges. (2) All complex hypernodes need to be connected.

Even if a complex hypernode is not connected, we might still be able to enlarge it, i.e., by merging it with adjacent vertices. But we cannot risk to restrict the graph by enlarging the complex hypernode too much so that we would miss to emit valid *ccp*s. Thus, we have to determine under which circumstances it is safe to enlarge a hypernode. Note that enlarging a node has a positive side effect: The number of nodes in S' is decreased because more vertices are represented by the same compound relation. That in turn increases the graph-aware partitioning algorithms performance drastically, since fewer vertices are to be partitioned.

Before we determine how to enlarge a hypernode, let us take a look at Fig. 16(a). Here, R_3 is only connected to the rest of the graph through R_2, R_4. But the latter is not connected. The only way to connect R_2 with R_4 is through R_1 and R_0, R_5. Thus, there exists only one valid *ccp*: $(\{R_0, R_1, R_2, R_4, R_5\}, \{R_3\})$ (and its symmetric partition). In fact, if we do not include R_0, R_1, R_5, we have to partition the graph into at least three connected subgraphs.

Fig. 16(b) shows the transformed graph of Fig. 16(a) after applying COMPUTEADJACENCYINFO. We can can observe that R_1 and R_0 lie on every possible path $R_2 \rightarrow^* R_4$. We, can generalize our observation: If there exists a non-separable hypernode that is not connected, it can be enlarged with all vertices that lie on every possible path (in the mapped simple graph) between the connected subsets of the hypernode. Since the vertices that are candidates for the enlargement have to lie on every path, all vertices that qualify for the enlargement in the end are articulation vertices by definition.

4.5.2 Enlarging Disconnected Hypernodes

Non-connected hypernodes are enlarged by MAXIMIZECOMPOUNDRELATIONS, as given in Fig. 17. Since this method relies on the knowledge of the biconnected components of the graph, we invoke it after calling GETBCCINFO, but before merging the overlapping hypernodes in COMPOSECOMPOUNDRELATIONS.

First, we gather the set of compound relations (L. 1 of MAXIMIZECOMPOUNDRELATIONS), which can actually be done in FINDINITIALCOMPOUNDSSUB. Next, we check whether the corresponding hypernode h (L. 2) is connected (L. 3). The loop of Lines 7 to 16 is responsible to enlarge the hypernode h. Therefore, we use two sets Z and I, whereby Z holds the vertices of the initial h (L. 5) and I keeps track of the already investigated vertices of the intial h. Once all members of Z are investigated, the stop condition of the loop is met.

The idea is as follows: We take an element of Z and assign it to x. There are two possibilities either x is already part of h or it is an ancestor of an element of h. In the latter case, it must be an articulation vertex and/or the start vertex t (L. 11 of Fig. 11). If it is an articulation vertex, we can add it to h (L. 10), since all paths between $desc[x] \cap h$ and other members of the hypernode $h \setminus desc[x]$ must contain x. We choose the next x to be its parent (L. 12). Note that the descendants of x are either already processed or are part of different biconnected components. In the latter case, they will be processed later on if they intersect with Z, or they are of no interest. If they are not of interest, this is because they will not be part of every path connecting the different subsets of h.

Before we continue with the next x, we have to check in Line 9 if (1) x was not already processed, i.e., $x \notin Z$ or (2) the descendants of x cover the whole hypernode h. In the latter case, we do not need to go any further (following the parents), because we would process other biconnected components that are not of interest. We can discard the members of those components since they cannot be part of every path between the disconnected members of the non-connected hypernode h.

Since we might have interrupted the loop (L. 9) because $h \nsubseteq desc[x]$ holds, we still have to add x to h (L. 14). But there is the chance that $x = t$ holds where t is the start vertex. Now there are two possibilities: either x is also an articulation vertex or it is not (see Sec. 4.4.2). The differentiation between the two cases was encoded through Line 21 of Fig. 13. Therefore, we have to ensure that $parent[x] = \text{NIL}$ holds first, otherwise x might not be contained in every possible path between the disconnected parts of h.

Finally, the condition of 17 checks if h was enlarged. If so, we have to apply the changes by invoking MAXIMIZECOMPOUNDRELATIONSSUB. Now there are two possibilities for the new h: (1) either there is no compound relation assigned or (2) there is one assigned because the new h is also the endpoint of a different articulation hyperedge. In the Lines 1 to 5 and 6 to 10, we change the assignments of the $Compound_{map}$ and the vertex set S' according to both cases. In Line 11, we update the corresponding hyperedge.

Let us get back to our example of Fig. 16 with the disconnected hypernode $\{R_2, R_4\}$. Before invoking MAXIMIZECOMPOUNDRELATIONS, the following holds: $RevLabel_{map} = \{..,(\{R_6\} \rightarrow \{R_0, R_5\}), (\{R_7\} \rightarrow \{R_2, R_4\})\}$ and $Compound_{map} = \{(\{R_0\} \rightarrow \{R_6\}),(\{R_2\} \rightarrow \{R_7\}),(\{R_4\} \rightarrow \{R_7\}),(\{R_5\} \rightarrow \{R_6\})\}$. Once MAXIMIZECOMPOUNDRELATIONS returns, $RevLabel_{map} = \{..,(\{R_7\} \rightarrow \{R_0, R_1, R_2, R_4\})\}$ holds. Furthermore, the entry for R_0 in $Compound_{map}$ was changed to $\{R_6, R_7\}$ and $(\{R_1\} \rightarrow \{R_7\})$ was inserted. Note that the entry for R_5 remains the same. After merging the hypernodes, we gain the simple graph of Fig. 16(c) with $S' = \{R_3, R_8\}$. Note that now all two conditions for avoiding the connection test are met.

4.5.3 Additional Considerations

Due to the nature of top-down join enumeration, a partitioning algorithm is called many times, each time with a different subgraph $H_{|S}$. With PARTITION$_X$ (Sec. 4.1, Fig. 6), we gave an overview of our generic framework. It contains room for improvements. COMPUTEADJACENCYINFO only needs to be called once: if $S = V$ (i.e., the whole graph $H = (V, E)$) is handed over. In all other cas-

MAXIMIZECOMPOUNDRELATIONS(S, S')

$\triangleright$ **Input:** vertex sets S, S'
1 **for all** $v \in$ set of compounds relations
2 $\quad h \leftarrow RevLabel_{map}[v]$
3 $\quad$ **if** ISCONNECTED(h) $=$ TRUE
4 $\quad\quad$ **continue**
5 $\quad Z \leftarrow h$
6 $\quad I \leftarrow \emptyset$
7 $\quad$ **while** $Z \neq \emptyset$
8 $\quad\quad x \leftarrow y \in Z$
9 $\quad\quad$ **while** $h \nsubseteq desc[x] \wedge x \notin I$
10 $\quad\quad\quad h \leftarrow h \cup \{x\}$
11 $\quad\quad\quad I \leftarrow I \cup \{x\}$
12 $\quad\quad\quad x \leftarrow parent[x]$
13 $\quad\quad$ **if** $parent[x] =$ NIL
14 $\quad\quad\quad h \leftarrow h \cup \{x\}$
15 $\quad\quad Z \leftarrow Z \setminus \{x\}$
16 $\quad\quad I \leftarrow I \cup \{x\}$
17 $\quad$ **if** $h \neq RevLabel_{map}[v]$
18 $\quad\quad$ MAXIMIZECOMPOUNDRELATIONSSUB(v, h)

MAXIMIZECOMPOUNDRELATIONSSUB(v, h)

1 **if** KEYDOESNOTEXIST($Label_{map}, h$) $\neq$ TRUE
2 $\quad z_k \leftarrow$ new vertex labeled $k \leftarrow$ SIZE($Label_{map}$)
3 $\quad S' \leftarrow (S' \setminus v) \setminus h$
4 $\quad$ **for all** $x \in h$
5 $\quad\quad Compound_{map}[\{x\}] \leftarrow$
$\quad\quad\quad (Compound_{map}[\{x\}] \setminus v) \cup \{z_k\}$
6 **else** CHANGEKEY($Label_{map}, RevLabel_{map}[v] \rightarrow h$)
7 $\quad RevLabel_{map}[v] \leftarrow h$
8 $\quad S' \leftarrow S' \setminus h$
9 $\quad$ **for all** $x \in h \setminus RevLabel_{map}[v]$
10 $\quad\quad Compound_{map}[\{x\}] \leftarrow Compound_{map}[\{x\}] \cup v$
11 update alle hyperedges (v, w) where $v \subseteq h \vee w \subseteq h$ holds

Figure 17: Pseudocode for MAXIMIZECOMPOUNDRELATIONS

es, we can reuse the information stored in the global variables (Fig. 7) by passing them down. In order to being able to modify them, we make a copy, which is handed over to the child invocations of TDPGSUB. Making a copy pays off, since we can eliminate edges that are not fully contained in the vertex set S. Further, we cleanse the array of precomputed hyperneighbours N_h. These two steps increase the partitioning algorithm's efficiency as well as the efficiency of connection tests.

Additional to COMPUTEADJACENCYINFO, we need to call GETBCCINFO only once. We simply reuse its gathered information stored in $Compound_{map}$ during all other calls of PARTITION$_X$. For those invocations of PARTITION$_X$ where $S \neq V$ and V is the vertex set of the query graph, we need a modified version of COMPOSECOMPOUNDRELATIONS. It has to be one that does not call GETBCCINFO and MAXIMIZECOMPOUNDRELATIONS and that invokes an adapted version of MANAGEADJACENCYINFO.

In case there are no complex hyperedges in the vertex subset S, we can even skip the methods of our framework. Then we can call the graph-aware partitioning algorithm right away. In that case, the partitioning algorithm only needs to exploit the information stored in N_s.

Finally, we propose to store the information whether a vertex set C is connected or not into the memotable. Besides TRUE and FALSE we need UNKNOWN. Now every time ISCONNECTED($H_{|C}$) is called, we check whether an entry in the memotable for the given vertex set C exists. If not, we create one and invoke the connection test since its current value is UNKNOWN to set it to TRUE or FALSE. In all other cases, we just return its value, which saves us additional connection tests. Note that since our transformed hypergraphs are relatively restrictive simple graphs, there will be only a few entries in the memotable with the value FALSE.

5. EVALUATION

We compare the performance of the three top-down enumerators TDMCBHYP (derived by instantiating our framework using MINCUTBRANCH), TDMCCHYP [6], and the naive partitioning TDBASICHYP. Further, to assess the potential of the optimizations proposed (Sec. 4.5), we include the unimproved implementation (without the techniques of Sec. 4.4, 4.5) of TDMCBHYP called TDMCBHYP$_{\text{naive}}$. Last, we instantiated our framework (without the techniques of Sec. 4.4, 4.5) with the algorithm proposed by DeHaan and Tompa [3]. We call this variant TDMCCHYP$_{\text{naive}}$. In order to investigate the pruning benefits, we added pruning to the first two algorithms, yielding TDMCBHYP$_{\text{pruning}}$ and TDMCCHYP$_{\text{pruning}}$, where we used the improved accumulated-predicted cost bounding method [7]. Note that this pruning method still guarantees plan optimality. **Indeed, all algorithms guarantee plan optimality.**

All plan generators (no matter whether they work top-down or bottom-up) use a common infrastructure (memotable, cardinality estimation, cost functions, and the conflict detector CD-A [11]). Consequently, the different plan generators differ only in those parts responsible for enumerating *ccp*s and for pruning (if applied). For the cost estimation of joins, we decided to use the formulas developed by Haas et al. [10], since they are very precise.

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 Intel C++ compiler with option O3.

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*. For DPHYP we present the absolute execution time.

This section is organized as follows. The first two subsections evaluate the performance for random cyclic and acyclic graphs. Then, we dissect the impact of pruning. After that, we take a look at TPC-H and TPC-DS queries. Finally, we try to detect the possible overhead of the hypergraph handling mechanism compared to an algorithm specialized on simple graphs.

5.1 Random Acyclic Query Graphs

There are two situations giving rise to complex hyperedges: (1) the TES indicating non-reorderability of non-inner joins and (2) complex predicates referencing more than two relations.

In order to distinguish these two cases, we first generated random binary operator trees where the operators can be any join. However, only simple predicates, i.e., those referencing exactly two relations, were generated. From the operator trees we then generated the resulting hypergraphs using the method described in [11] and sketched at the beginning of Sec. 2. We denote this case by acyclic/non-inner/simple.

To clearly separate the second case from the first, we have to use inner joins only. Thus, we generated random operator trees with inner joins only and used the method described in [13] to extend these with complex predicates to hypergraphs. We denote this case by acyclic/inner/complex.

The results for both cases are shown in Fig. 19 and on the left side of Table 1.

When comparing the performance results between acyclic/non-inner/simple and acyclic/inner/complex, each algorithm retains its unique trend. Further, we observe the following for acyclic query graphs. The performance of TDBASICHYP is unacceptable. The optimizations proposed in Sec. 4.5 result in an average improvement of about a factor of two, as can be seen by comparing TDMCBHYP and TDMCBHYP$_{\text{naive}}$. Further, TDMCBHYP performs best on average in all cases and has the lowest worst case normed

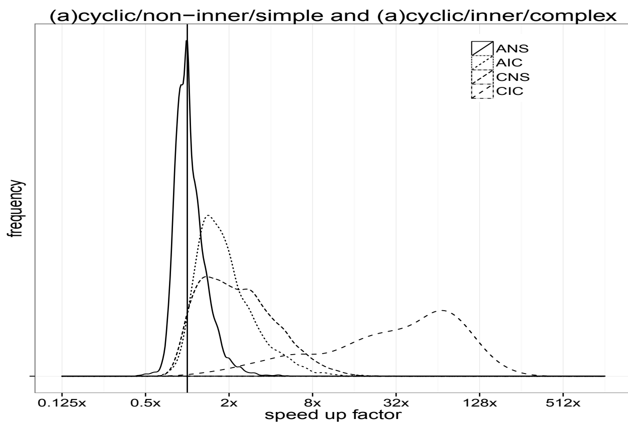

Figure 18: Density plots for $\text{TDMCBHYP}_{pruning}$

runtime. Comparing the performance of TDMCBHYP with that of the state of the art bottom-up plan generator DPHYP, we see that on average, their performance is about the same. Looking at the max/min normed runtimes, we see that TDMCBHYP is slower by a factor of at most 2.3 and sometimes faster by a factor of roughly $0.12^{-1} = 8.3$.

5.2 Random Cyclic Query Graphs

Let us look at the performance for random cyclic queries. Again, we distinguish between the case where complex hyperedges are only due to conflicts of non-inner joins and the case where they result from complex predicates. We denote these cases by cyclic/non-inner/simple and cyclic/inner/complex. The results are shown in Fig. 20 and on the right side of Table 1.

Here, for any fixed number of relations, the variations in runtime are enormous. More specifically, they heavily depend on the number of edges. This is obvious, as we imagine to add edges to a chain until we get a clique. Hence, we fixed the number of relations to some medium number (15) and varied the number of edges from 15 (cycle) to 40 in case of cyclic/no-inner/simple and up to 80 in case of cyclic/inner/complex.

Considering Fig. 20, we observe that (except for TDBASICHYP) the runtimes of all algorithms increase heavily with the number of edges. This is due to the increased search space size. TDBASICHYP only shows a slight increase in runtime if more edges are present. This can easily be explained by observing that adding edges leads to a higher connectivity within the graph and thus to more ccps. Thus, the validity test of TDBASICHYP fails less frequently, resulting in more calls to the cost function.

Compared to the acyclic case, we observe the larger distance in runtime between the algorithms with and without pruning. Taking a look at the avg column of Table 1, we see that $\text{TDMCBHYP}_{\text{pruning}}$ outperforms DPHYP by a factor of approximately $0.09^{-1} = 11$ (on average!). In the best case, the $\text{TDMCBHYP}_{\text{pruning}}$ outperforms DPHYP by a factor of $1/0.004 = 250$. Only for a low number of vertices, the runtimes are comparable with those of acyclic queries. Further note that the optimizations proposed in Sec. 4.5 again result in a profound runtime saving. As in the acyclic case, TDMCBHYP clearly dominates all other algorithms, in both variants (without and with pruning).

5.3 Dissecting Pruning Performance

Different queries together with different cardinalities and selectivities embed different inherent pruning potentials. We thus decided to illustrate this potential using density plots (see. Fig. 18). The x-axis gives the speed-up factor achieved by pruning. The y-axis shows its frequency, i.e., how often a certain speed-up factor was observed during our experiments with random cyclic and acyclic queries.

The results are shown for $\text{TDMCBHYP}_{pruning}$ for the acyclic/non-inner/simple (ANS), acyclic/inner/complex (AIC), cyclic/non-inner/simple (CNS), and cyclic/inner/complex (CIC) workloads. We observe that the pruning behavior for the different cases is rather different. First, in the worst case (ANS), we have a steep peak around 1. This means that the pruning potential is poor. It becomes larger in case of AIC, but still for cyclic queries in the cases CNS and CIC we observe a much higher optimization potential. In order to determine the pruning potential for realistic queries, we turn to the TPC-H and TPC-DS benchmarks.

5.4 TPC-H and TPC-DS

As our initial plans, we used the plans generated by a commercial DBMS. Thus, we could benefit from optimization techniques such as unnesting and subplan sharing. We used complex hyperdeges to prevent reordering conflicts. The runtimes reported here do not include the preparation time for computing the query graphs.

We considered all TPC-H queries except for those that did not contain any join (Q1 and Q6). Then, we summed up the runtimes for all queries for each of DPHYP, TDMCBHYP, and $\text{TDMCBHYP}_{\text{pruning}}$. For each algorithm, let us call this its H-total time. Then, the ratio of DPHYP's H-total time divided by $\text{TDMCBHYP}_{\text{pruning}}$'s H-total time is 1.7. The ratio of TDMCBHYP's H-total time divided by $\text{TDMCBHYP}_{\text{pruning}}$'s H-total time is 1.6. Without pruning, TDMCBHYP beats DPHYP by roughly five percent for the TPC-H workload. For those queries referencing more than 4 tables, detailed results are shown in Table 2. Hereby Q2, Q20 reference 5 tables, Q5, Q7, Q9, Q21 6 tables, and Q8 8 tables. The number of join edges is given in Table 2 accordingly.

Again, for the TPC-DS queries, we summed up the plan generation times for all queries in TPC-DS for each of the above algorithms. For a given algorithm, we call this its DS-total time. Then, the ratio of DPHYP's DS-total time divided by $\text{TDMCBHYP}_{\text{pruning}}$'s DS-total time is 2.28. The ratio of TDMCBHYP's DS-total time divided by $\text{TDMCBHYP}_{\text{pruning}}$'s DS-total time is 2.37.

5.5 Overhead Detection

In order to determine the possible overhead induced by hyperedges, we evaluated the algorithms on the standard cases of simple query graphs: chains, cycles, and cliques. We included TDMCB, an algorithm which is not capable of handling hypergraphs. We run these three different query graph classes for different numbers of relations (n). The results are shown in Table 3. Note that the runtimes of TDMCB and TDMCBHYP are almost identical, indicating that there is no measurable overhead. This may be due to the fact that in both cases ccp enumeration is fast, compared to cost function calculation.

6. CONCLUSION

We presented a generic framework which allows us to reuse any top-down enumerator for simple graphs to handle hypergraphs. We further demonstrated that one possible instantiation (TDMCBHYP) outperforms existing enumerators for hypergraphs and is comparable in performance to DPHYP even without pruning. It is also faster than existing top-down enumerators and has zero measurable overhead compared to TDMCB. With pruning, TDMCBHYP is currently unbeatable. Besides pruning, another advantage of top-down plan generation is that it is easily parallelizable. A feature we are likely to explore in the future.

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

Algorithm	min	max	avg	min	max	avg
	acyclic/non-inner/simple			cyclic/non-inner/simple		
DPHyp	0.0001 s	0.0588 s	0.0007 s	0.0001 s	1.2638 s	0.0076 s
TDBasicHyp	1.1232 ×	15396.4991 ×	963.2282 ×	0.8571 ×	15755.3138 ×	902.7267 ×
TDMcLHyp$_{naive}$	0.3333 ×	13.9975 ×	2.2518 ×	0.2763 ×	12.6669 ×	2.0992 ×
TDMcBHyp$_{naive}$	0.2708 ×	13.4980 ×	1.9281 ×	0.2105 ×	11.7512 ×	1.6415 ×
TDMcCHyp	0.1458 ×	3.6001 ×	1.1349 ×	0.1169 ×	3.4285 ×	1.0932 ×
TDMcBHyp	0.1148 ×	2.2992 ×	0.9385 ×	0.0921 ×	2.0660 ×	0.8806 ×
TDMcCHyp$_{Pruning}$	0.0641 ×	4.5006 ×	1.2055 ×	0.0494 ×	4.0007 ×	0.7149 ×
TDMcBHyp$_{Pruning}$	0.0647 ×	2.2495 ×	0.9166 ×	0.0306 ×	1.8777 ×	0.5057 ×
Algorithm	min	max	avg	min	max	avg
	acyclic/inner/complex			cyclic/inner/complex		
DPHyp	0.0001 s	0.4359 s	0.0096 s	0.0002 s	43.3824 s	2.1668 s
TDBasicHyp	2.1001 ×	36667.0371 ×	973.5695 ×	1.0243 ×	49977.4253 ×	19.8760 ×
TDMcLHyp$_{naive}$	0.6552 ×	5.3914 ×	1.4863 ×	1.0910 ×	8.0412 ×	1.6961 ×
TDMcBHyp$_{naive}$	0.5345 ×	4.8697 ×	1.2344 ×	0.7500 ×	2.5111 ×	1.1730 ×
TDMcCHyp	0.6207 ×	3.2274 ×	1.3606 ×	0.8000 ×	1.9246 ×	1.3573 ×
TDMcBHyp	0.4310 ×	1.7543 ×	0.9978 ×	0.6429 ×	1.5828 ×	1.0539 ×
TDMcCHyp$_{Pruning}$	0.0849 ×	4.0002 ×	0.9746 ×	0.0056 ×	1.9245 ×	0.1384 ×
TDMcBHyp$_{Pruning}$	0.0352 ×	1.7741 ×	0.5837 ×	0.0040 ×	1.2536 ×	0.0872 ×

Table 1: Results for random queries

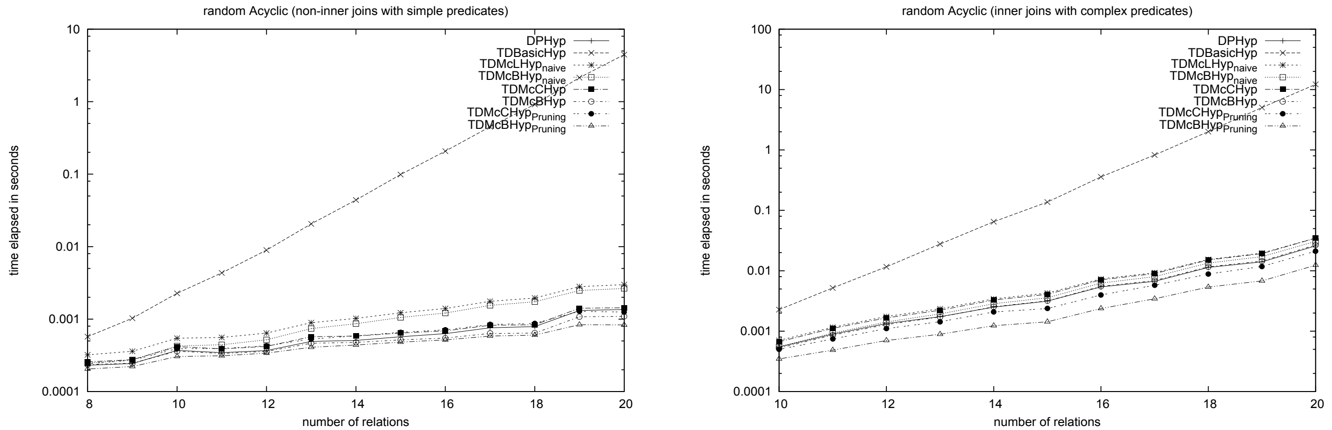

Figure 19: Left: acyclic/non-inner/simple. Right: acyclic/inner/complex

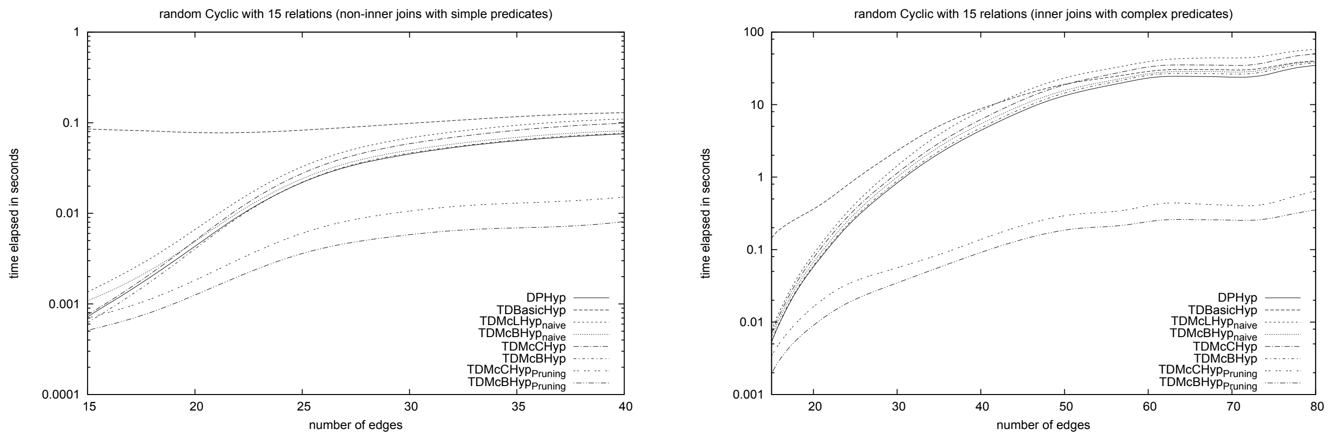

Figure 20: Left: cyclic/non-inner/simple. Right: cyclic/inner/complex

7. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

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

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

Algorithm	Q2 $\|E\|$=4	Q5 $\|E\|$=6	Q7 $\|E\|$=6	Q8 $\|E\|$=7	Q9 $\|E\|$=6	Q20 $\|E\|$=4	Q21 $\|E\|$=5
TDMCBHYP	0.76 ×	0.70 ×	0.76 ×	1.00 ×	1.09 ×	1.64 ×	1.05 ×
TDMCBHYP$_{Pruning}$	0.50 ×	0.29 ×	0.31 ×	0.75 ×	0.60 ×	0.66 ×	0.75 ×
Speed-Up-Factor through Pruning	1.53 ×	2.46 ×	2.46 ×	1.33 ×	1.82 ×	2.48 ×	1.41 ×

Table 2: Normed runtime and Speed-Up-Factors (against TDMCBHYP) for TPC-H Queries with more than 4 tables referenced

Algorithm	min	max	avg	min	max	avg	min	max	avg
	Chain $(8 \leq \|V\| \leq 22)$			Cycle $(8 \leq \|V\| \leq 20)$			Clique $(8 \leq \|V\| \leq 15)$		
DPHYP	0.00 s	0.01 s	0.00 s	0.00 s	0.03 s	0.01 s	0.02 s	65.95 s	12.01 s
TDMCB	0.56 ×	1.19 ×	0.93 ×	0.68 ×	1.14 ×	0.97 ×	0.98 ×	1.07 ×	1.02 ×
TDMCBHYP	0.56 ×	1.19 ×	0.94 ×	0.68 ×	1.14 ×	0.97 ×	0.98 ×	1.08 ×	1.02 ×
TDMCBHYP$_{Pruning}$	0.09 ×	0.76 ×	0.29 ×	0.07 ×	0.40 ×	0.20 ×	0.00 ×	0.10 ×	0.02 ×

Table 3: Chain, Cycle and Clique queries: minimum, maximum and average of the normed runtimes

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

[5] P. Fender and G. Moerkotte. Reassessing top-down join enumeration. *IEEE TKDE*, 24(10):1803–1818, 2012.

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

[7] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In *ICDE*, pages 414–425, 2012.

[8] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst.*, 22(1):43–73, 1997.

[9] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[10] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB Journal*, 6:241–256, 1997.

[11] G. Moerkotte, P. Fender, and M. Eich. On the correct and complete enumeration of the core search space. In *SIGMOD*, pages 493–504, 2013.

[12] 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*, pages 930–941, 2006.

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

[14] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, pages 960–971, 2004.

[15] T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *ICDE*, pages 461–472, 2004.

[16] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[17] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen. Using EELs: A practical approach to outerjoin and antijoin reordering. In *ICDE*, pages 585–594, 2001.

[18] L. D. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H. min Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *IDEAS*, pages 20–33, 2001.

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

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

APPENDIX

A. BRANCH AND BOUND PRUNING

As Sec. 3 has pointed out, TDPGSUB builds a join tree for a (sub)set of relations S upon request through a recursive call of the top-down join enumeration algorithm. Because the processing order for the generation of (sub)plan is demand-driven, two branch and bound pruning methods can be exploited: accumulated-cost bounding and predicted-cost bounding. A comprehensive description can be found in [3, 7].

The main idea of accumulated-cost bounding is to pass a cost budget to the top-down join enumeration procedure [9, 18]. During the recursive descent, each instance of the top-down procedure subtracts costs from the handed-over budget as soon as they become known. The descent is aborted once the budget drops below zero. Every call that returns with a join tree has produced an optimal join tree. If no join tree is returned, then the handed-over budget was not sufficient. The goal of accumulated-cost bounding is to tighten the budget as much as possible, because this way the search might be curtailed sooner and ultimately the compile time decreases. There are three ways of adjusting the budget: (1) The budget for the sub call with S_1 (Fig. 1 L. 5) is set to the current budget decreased by the cost of the join operator $\circ$, (2) the budget for the sub call with S_2 is set to the current budget decreased by the cost of the join operator $\circ$ plus the cost of the join tree of S_1 and (3) if a join tree is built for S_1 and S_2 the current cost budget is set to the join tree's costs (if it is cheaper than existing ones for the same S). Some additional care has to be taken in the presence of different properties.

Whereas accumulated-cost bounding prunes the search space by passing budget information top-down, predicted-cost bounding follows the opposite approach by estimating what are the costs of the subjoin trees that lie below in the recursive search tree [18]. So the main idea is to find a lower bound in terms of join costs for a given *ccp* $(S1, S2)$ before actually requesting the two corresponding optimal subjoin trees to be built through two different recursive descents. If now the estimate which is specific for that *ccp* is larger than the cost of a join tree already built for S, then the cheapest join tree for S clearly cannot consist of a join between $S1$ and $S2$. Hence, the effort of subcalls with $S1$ and $S2$ can be spared.

For our experiments we applied a combination of both methods together with further improvements as described in [7]. With all pruning techniques as applied in [7], plan optimality is preserved.