2012 IEEE 28th International Conference on Data Engineering

Effective and Robust Pruning for Top-Down Join Enumeration Algorithms

Pit Fender #1, Guido Moerkotte #2, Thomas Neumann #3, Viktor Leis #4

[1,2]*Database Research Group, Mannheim University*
68131 Mannheim, Germany
[1]fender@informatik.uni-mannheim.de
[2]moerkotte@informatik.uni-mannheim.de
[3,4]*Database Research Group, Technical University of Munich*
85748 Garching, Germany
[3]neumann@in.tum.de
[4]leis@in.tum.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. For both optimization strategies efficient algorithms have been published. However, only the top-down approach allows for branch-and-bound pruning. Two pruning techniques can be found in the literature. We add six new ones. Combined, they improve performance roughly by an average factor of $2-5$. Even more important, our techniques improve the worst case by two orders of magnitude.**

Additionally, we introduce a new, very efficient, and easy to implement top-down join enumeration algorithm. This algorithm, together with our improved pruning techniques, yields a performance which is by an average factor of $6-9$ higher than the performance of the original top-down enumeration algorithm with the original pruning methods.

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].

When designing a query optimizer, there are two approaches suitable 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). DeHaan and Tompa [3] proposed an efficient top-down algorithm (TDMinCutLazy). Recently, Fender and Moerkotte [4] proposed an alternative top-down join enumeration strategy (TDMinCutBranch), which is almost as efficient as DPccp.

Only top-down algorithms allow for branch-and-bound pruning strategies. DeHaan and Tompa categorized branch-and-bound based pruning strategies into *predicted cost bounding* PCB and *accumulated cost bounding* ACB [3]. These two can be combined into an effective pruning mechanism (*APCB*) [3].

Let us explicitly state two obvious observations. (1) Different top-down join enumeration algorithms lead to different enumeration orders. (2) Different enumeration orders (can) lead to a different pruning behavior. This leads to the question of the robustness of a pruning strategy, where we call a pruning strategy robust if it behaves equally well in different top-down enumeration strategies.

The main goal of this paper is to improve APCB (a) in terms of effectiveness, (b) in terms of robustness and (c) most importantly avoid the worst-case behavior otherwise observed. We achieve this by integrating six new techniques into APCB. Applying this improved version of APCB will accelerate plan generation by an average factor of $2-5$ and decrease the worst-case behavior by a factor of $10-98$.

And as we will see, it is more robust than APCB because its pruning efficiency is less dependent on the enumeration strategy used. To have a valid foundation for this statement, we need more than the two top-down enumeration algorithms described in the literature. Thus, we developed a new one (TDMinCutConservative), presented here. It is easier to implement than TDMinCutLazy and TDMinCutBranch, and it outperforms the existing algorithms. The combination of our effective new pruning with the efficiency

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.27

IEEE computer society

of TDMINCUTCONSERVATIVE outperforms the combination TDMINCUTLAZY with APCB by an average factor of $6-9$.

This paper is organized as follows. Sec. II recalls some preliminaries. Sec. III presents the novel partitioning strategy. Sec. IV explains the branch-and-bound pruning techniques ACB and PCB as well as our new technical advances. Sec. V contains the experimental evaluation, and Sec. VI concludes the paper.

II. FUNDAMENTALS

Before we start with our discussions, we give some fundamentals. In the first subsection, we explain important notions and then continue with an introduction to top-down join enumeration. We present a generic memoization algorithm for join optimization that can be instantiated with different enumeration strategies for csg-cmp pairs, which we also call partitioning strategies or algorithms.

A. Important Notions

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 2.1: Let $G = (V, E)$ be a connected query graph, (S_1, S_2) is a *connected subgraph and its complement pair* (or *ccp*) if the following holds:

- S_1 with $S_1 \subset V$ induces a connected graph $G_{|S_1}$,
- S_2 with $S_2 \subset V$ induces a connected graph $G_{|S_2}$,
- $S_1 \cap S_2 = \emptyset$, and
- $\exists (v_1, v_2) \in E \mid v_1 \in S_1 \wedge v_2 \in S_2$.

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

Definition 2.2: Let $G = (V, E)$ be a connected query graph and S a set with $S \subseteq V$ that induces a connected subgraph $G_{|S}$. 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 $\mathcal{P}(V)_{con} = \{S \subseteq V \mid G_{|S} \text{ is connected } \wedge |S| > 1\}$ be the set of all connected subsets of V with more than one element, then $P_{ccp} = \{P \in P_{ccp}(S) \mid S \in \mathcal{P}(V)_{con}\}$ 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^{sym}_{ccp} of all ccps, where symmetric pairs are accounted for only once, e.g., $(S_1, S_2) \in P^{sym}_{ccp}$ if $max_{index}(S_1) \leq max_{index}(S_2)$ holds, or $(S_2, S_1) \in P^{sym}_{ccp}$ otherwise. We give no constraints for choosing which one of two symmetric pairs should be member of P^{sym}_{ccp}, but leave this as a degree of freedom. The lower bounds for join enumeration given by Ono and Lohman [1] for certain graph shapes correspond to $|P^{sym}_{ccp}|$. Our conservative partitioning algorithm relies on the notion of the neighborhood of a set of nodes. We give its definition:

Definition 2.3: Let $G = (V, E)$ be an undirected graph. Then the *neighborhood of a set* $S \subseteq V$ is defined as:

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

B. Generic Top-Down Join Enumeration

As an introduction to top-down join enumeration, we give a generic memoization algorithm that we call TDPLANGEN. We present its pseudocode in Figure 1. Like dynamic programming, TDPLANGEN 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. At every recursion step of TDPGSUB, all possible join trees of two optimal subjoin trees that together would comprise the relations of S are built through BUILDTREE (line 3) that we explain in Appendix A, and the cheapest join tree is kept. 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 Appendix B.

The recursive descent stops when either $|S| = 1$ or TDPGSUB has already been called for that $G_{|S}$. 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 $G_{|S}$ as input, and the optimal join tree of $G_{|S}$ has not been found yet.

```
TDPLANGEN(G)
    ▷ Input: connected G=(V,E), V = ⋃_{1≤i≤|V|} {R_i}
    ▷ Output: an optimal join tree for G
1   for i ← 1 to |V|
2       do BestTree[{R_i}] ← R_i
3   return TDPGSUB(V)
TDPGSUB(G_|S)
    ▷ Input: connected sub graph G_|S
    ▷ Output: an optimal join tree for G_|S
1   if BestTree[S] = NULL
2       then for all (S_1, S_2) ∈ P^sym_ccp(S)
3            do BUILDTREE(G_|S, TDPGSUB(G_|S_1),
                          TDPGSUB(G_|S_2), ∞)
4   return BestTree[S]
```

Fig. 1. Pseudocode for TDPLANGEN

III. Graph-Based Join Enumeration

DeHaan and Tompa [3] were the first that presented a partitioning algorithm named MinCutLazy for top-down join enumeration with a complexity in $O(|V|^2)$ per emitted ccp [4]. Recently, with MinCutBranch Fender and Moerkotte [4] proposed another partitioning algorithm with a complexity in $O(1)$ per emitted ccp for chain, star, cycle and clique queries. In fact, MinCutBranch embedded in a memoization-based top-down join enumeration algorithm like TDPlanGen is nearly as efficient as Moerkotte's and Neumann's [2] state of the art in bottom-up join enumeration DPccp.

Therefore, MinCutBranch would be the partitioning algorithm of choice for branch-and-bound pruning. But instead of introducing MinCutBranch here, we chose to present a novel partitioning algorithm, which we call MinCutConservative. It:

- has a slightly better runtime performance, and
- is easier to implement.

A. Conservative Graph Partitioning

This section presents our novel partitioning algorithm named conservative partitioning, which we denote by MinCutConservative. It is an improvement of the advanced generate-and-test approach presented in [5]. The algorithm emits all ccps for a connected vertex set S for which $S \subseteq V$ holds and where V is the vertex set of the query graph $G = (V, E)$. We denote the instantiated top-down memoization variant by TDMinCutConservative. Its pseudo-code is given in Figure 2.

Conservative partitioning is invoked by Partition$_{MinCut\ Conservative}$, which in turn immediately calls its recursive component MinCutConservative. The basic idea of branch partitioning is to successively enhance a **c**onnected set C by members of its neighborhood $\mathcal{N}(C)$ at every recursive iteration. The process starts with a single vertex $t \in S$ through a redefinition of $\mathcal{N}(\emptyset) = \{t\}$. This way, we ensure that at every instance of the algorithm's execution C is connected. Since S and $C \subset S$ are connected, for every possible complement $S \setminus C$ there must exist a join edge (v_1, v_2), where $v_1 \in C$ and $v_2 \in (S \setminus C)$ holds. If at some point of enlarging C its complement $S \setminus C$ in S is connected as well, the algorithm has found a ccp for S.

B. Correctness Constraints

Besides, the connectivity of the C's complement conservative partitioning has to meet some more constraints before emitting a ccp: (1) Symmetric ccps are emitted once, (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 is always contained in C and, therefore, can never be part of its complement $S \setminus C$. For the second constraint, the algorithm uses a filter set X of neighbors to exclude from processing. And for constraint (3), it is sufficient to ensure that all possible connected subsets of S that have a connected complement $S \setminus C$ are considered when enlarging C.

C. The Algorithm in Detail

There are certain scenarios, e.g., when star queries are considered, where constructing every possible connected subset C of S produces an exponential overhead because most of the 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. Before we explain this technique, 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 all the members of C's neighborhood $\mathcal{N}(C)$. Then every O_i where $1 \leq i \leq k$ 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 MinCutConservative the complement $S \setminus C$ is connected as well, it does not always suffice to enlarge C by only one of its neighbors but by a larger subset $\cup_{O_j, i \neq j}$ of its direct and indirect neighborhood. For the computation of all subsets O_i with $1 \leq i \leq k$, MinCutConservative invokes getConnectedParts in line 7, which calculates an output set O containing all subsets O_i. If the complement $S \setminus C$ is connected, the returned O contains only one O_i with $O_i = S \setminus C$. Appendix C explains getConnectedParts in detail.

Once $O = \{O_1, O_2, ..., O_k\}$ is returned, MinCutConservative invokes itself for k different times with a corresponding new $C' = S \setminus O_i$, while $1 \leq i \leq k$ holds. Note that if the old $S \setminus C$ is connected, there is only one branch of recursions with a new $C' = S \setminus O_1 = C \cup \{v\}$. All the child invocations will emit newly computed cpp accordingly in line 4. The recursive descent stops once the last neighbor $v \in \mathcal{N}(C)$ was added to C so that for the successive child invocation the condition of line 1 holds.

As mentioned for meeting constraint (2), conservative partitioning makes use of the filter set X. Therefore, the current v is added to X' (line 10) after MinCutConservative has returned from the k recursive calls of line 9. This ensures that in all other recursive descents invoked with the remaining $v \in \mathcal{N}(C)$ yet to be processed, the current v cannot be chosen as a neighbor again and is excluded from further consideration (line 6).

IV. Branch and Bound Pruning

As Sec. II-B outlined, the processing order of top-down join enumeration is demand-driven. This means that a join tree for

```
PARTITION_MinCutConservative(S)
    ▷ Input: a connected set S, arbitrary vertex t ∈ S
    ▷ Output: all ccps for S
1   MINCUTCONSERVATIVE(S, ∅, ∅)

MINCUTCONSERVATIVE(S, C, X)
    ▷ Input: connected set S, C ∩ X = ∅
    ▷ Output: ccps for S
 1  if C = S
 2     then return
 3  if C ≠ ∅
 4     then emit (C, S \ C)
 5  X' ← X
 6  for v ∈ (N(C) \ X)
 7        do O = GETCONNECTEDPARTS(S, C ∪ {v}, {v})
 8           for all O_i ∈ O
 9                 do MINCUTCONSERVATIVE(S, S \ O_i, X')
10           X' ← X' ∪ {v}
    ▷ N(∅) = {arbitrary element of t ∈ S}
```

Fig. 2. Pseudocode for MINCUTCONSERVATIVE

a subset of relations S is built only upon request through a recursive call of the top-down enumeration algorithm. Therefore, it then iterates over all possible partitions (ccps) (S_1, S_2) of S with $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$. Because of the demand-driven nature, top-down join enumeration can leverage the benefits of branch-and-bound pruning. The beauty of pruning is that it is risk-free: It can speed up processing by an order of magnitude by reducing the search space while at the same time it preserves optimality.

We now explain two different bounding strategies, *accumulated-cost* and *predicted-cost* bounding [3], and describe how they can be combined. Then, we describe our technical advances.

A. Accumulated-Cost Bounding

As implemented in Volcano [6], Cascades [7], and Columbia [8], this bounding technique passes a cost budget to the top-down enumeration procedure. We give the pseudo code for accumulated-cost bounding integrated in our generic top-down join enumeration algorithm in Figure 3 with TDPG$_{ACB}$ after [9]. During the recursive descent, each instance of TDPG$_{ACB}$ 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, the handed-over budget was not sufficient. Line 2 iterates over the ccps for S. The first step in the loop is to determine the cost for combining the two subjoin trees. These costs can be subtracted from the budget b (line 4) and handed over to the child invocation for S_1 (line 5). If no tree is returned, it becomes obvious that no join tree for the partition (S_1, S_2) can be constructed. Hence, there is no need to request a join tree for S_2. Otherwise, if a subjoin tree for S_1 has been found, the budget for S_2 is adjusted by decreasing b' with the cost for constructing the tree of S_1 (line 7). Line 7 invokes the recursive descent for S_2 with a tighter bound than for S_1. Upon return of the call, a join tree for S can only be registered with $BestTree[S]$ if (1) a tree for S_2 was returned, (2) the combined tree is cheaper than the budget allows for and (3) it is cheaper than all other trees produced so far. Line 9 takes care of (1), and BUILDTREE (Appendix A) handles (2) and (3). Every time a new and cheaper join tree for S has been registered with $BestTree[S]$, the budget b' for all other ccps for S can be adjusted (line 4) instead of resetting it to the b that was handed over. If after the enumeration of all ccps for S (line 2) no tree has been found that is cheaper than b, the lower bound $lB[S]$ is set to b. If a join for S is requested another time with a budget b'', the call can be returned immediately if the tree was already built and registered with $BestTree[S]$ or b'' is smaller than $lB[S]$ (line 1). Note that if the lower bound for S is not set, $lB[S]$ returns 0. The initial budget for the top-level call to TDPG$_{ACB}$ is set to ∞.

```
TDPG_ACB(G_|S, b)
    ▷ Input: connected sub graph G_|S, cost budget b
    ▷ Output: an optimal join tree for G_|S
 1  if BestTree[S] = NULL and lB[S] ≤ b
 2     then for all (S_1, S_2) ∈ P^sym_ccp(S)
 3          do c_⋈ ← cost of operator ⋈
 4             b' ← MIN(b, cost(BestTree[S])) − c_⋈
 5             lT ← TDPG_ACB(G_|S_1, b')
 6             if lT ≠ NULL
 7                then b' ← b' − cost(lT)
 8                   rT ← TDPG_ACB(G_|S_2, b')
 9                   if rT ≠ NULL
10                      then
                           BUILDTREE(G_|S, lT, rT, b)
11          if BestTree[S] = NULL
12             then lB[S] = b
13  return BestTree[S]
```

Fig. 3. Pseudocode for TDPG$_{ACB}$

B. Predicted-Cost Bounding

As has been shown, accumulated-cost bounding prunes the search space by passing budget information top-down. Predicted-cost bounding as the main contribution of Columbia [8] follows the opposite approach by estimating the costs of the subjoin trees that lie below in the recursive search tree. So the main idea is to find a lower bound in terms of join costs for a given ccp (S_1, S_2) 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, the cheapest join tree for S clearly cannot consist of a join between S_1 and S_2. Hence, the effort of subcalls with S_1 and S_2 can be spared.

In Figure 4, the pseudo code after [3] for top-down join enumeration enhanced with predicted-cost bounding is given with TDPG$_{PCB}$. It is identical to TDPGSUB except for line 3, where the lower bound estimate LBE is compared to the costs of $BestTree[S]$, which serves as an upper bound. If

there is no join tree for S known yet, $cost(BestTree[S])$ will return ∞.

The lower bound estimation procedure LBE depends upon the cost model used in the optimizer. For our experiments, we use the I/O cost model for ad hoc join costs as proposed in [10].

```
TDPG_PCB(G_|S)
    ▷ Input: connected sub graph G_|S
    ▷ Output: an optimal join tree for G_|S
1   if BestTree[S] = NULL
2     then for all (S1, S2) ∈ P^sym_ccp(S)
3            do if LBE(S1, S2) ≤ cost(BestTree[S])
4              then BUILDTREE(G_|S, TDPG_PCB(G_|S1),
                              TDPG_PCB(G_|S2), ∞)
5   return BestTree[S]
```

Fig. 4. Pseudocode for TDPG_{PCB}

C. Combining the Methods

A combination of both bounding methods is very beneficial, as [3] points out. This can be done by inserting the call for the LBE after line 2 of TDPG_{ACB}. The inserted code has the following form:

```
3.1   do if LBE(S1, S2) ≤ MIN(b, cost(BestTree[S]))
```

Again, if there is no join tree registered with $(BestTree[S])$, $cost$ will return ∞. In this case, b is chosen as the comparison value, in all other cases $cost(BestTree[S])$ will be the smaller value of the two. We denote the combined method by TDPG_{APCB}.

D. Technical Advances

This section describes our improvements to accumulated-predicted cost bounding. We name the new algorithm TDPG_{APCBI} because it is based on TDPG_{APCB}. The pseudo code is given in Figure 5.

First of all — and not indicated in the pseudo code — we propose an *advancement of the* LBE *method* to [3]. Instead of basing its computation solely on an estimation, we include information which we already know. Therefore, we increase the return value of LBE by the costs of the optimal join trees for S_1 or S_2 or at least $lB[S_1]$ or $lB[S_2]$ respectively, if this information is already available. However, we can add $lB[S_1]$ to the estimate only if the costs for $BestTree[S_1]$ are not known yet, that is if $BestTree[S_1] = \text{NULL}$ holds. The same is true for S_2.

Second, we make use of a *join heuristic* to decrease the initial budget. For our implementation we have used GOO [11] which is in $O(|V|^3)$. But instead of using only the cost of the whole join tree produced by the heuristic as a total bound, we also include the cost of its produced subtrees. Therefore, we introduce with $uB[S]$ another bound that is populated by the cost of the join and subjoin trees produced by the heuristic. If now an upper bound for S exists and the passed-in budget b is greater, we decrease b and set it to $uB[S]$ (line 11).

Third, we *improve the lower bounds*. Instead of setting lB to the current b when no join tree was found for b (compare to line 12 of TDPG_{ACB}), we compare b to the minimum of all LBE results and take the maximum of the two. This comes in handy when the predicted cost bounding component rejects all ccps for S (line 14 of TDPG_{PACBI}). Therefore, nlB is introduced in line 12 and set to ∞. Its minimum is computed over all ccps for S in line 15, notably before predicted cost bounding rejects a ccp (line 14). In line 35, we set its value if no join tree within the budget b was found. Lines 31, 32 and 33 extend this idea further by also considering cases where the corresponding LBE results are smaller than b, but join tree construction still fails because b is not high enough. Note that in Line 31 the value of nlB may still be decreased even when BUILDTREE constructs a join tree that is cheaper than b. But this is irrelevant, since $lB[S]$ is only set if no join tree could be constructed (line 34).

Fourth, a *rising budget* is proposed as the solution for the worst case behavior of accumulated cost bounding observed in [3]. Accumulated cost bounding is efficient by preventing top-down join enumeration from building expensive subtrees that cannot be part of an optimal solution. But in several cases, it might increase optimization time significantly compared to top-down join enumeration without it. This occurs when a join tree for S is requested several times, and each time the budget b that is passed in is slightly higher than before. If the slightly increased budgets are still too low to produce the cheapest join tree, the results are unnecessary computations of $P^{sym}_{ccp}(S)$ and the corresponding subtree requests that might even have the same cascading negative effect. As a solution to this problem, we count the number of times a request to TDPG_{PACBI} with the same S has been made in line 9 and store it in $attempts[S]$. If now a join tree for S is requested the second time and it has not already been constructed the first time, the budget is increased and set to $lB[S] * 2^{attempts[S]}$ if b is not higher already (line 8). If we have an upper bound for S, we make an exception and set b to that upper bound $uB[S]$ right away (line 7).

Fifth, TDPG_{PACBI} *tightens the budget* which is passed in to the call for requesting the left subjoin tree comprising the relations of S_1 more intelligently. Therefore, we include information about the costs of the right join tree comprising the relations of S_2 or at least its lower bound $lB[S_2]$. This is done through lines 19 to 21 and 23. Note that if the lower bound for S is not set, $lB[S]$ returns 0.

Sixth, we change the order in which the partitioning algorithm selects its next neighbor (Section II-A), e.g., line 6 of MINCUTCONSERVATIVE (Section III). In our implementation, the next neighbor is selected by the least significant bit of the bitset that stores the rest of the neighborhood to be processed. Therefore, we propose a re-numbering of the nodes in the query graph. The preferred processing order of neighbors is taken from the join tree produced by our join heuristic. We renumber the vertices by a breadth-first traversal of the join tree. As our experiments have shown, the effect is that the join tree and its subtrees produced by the heuristics are mostly planned first before other join trees during the top-down join enumeration.

```
TDPG_PACBI(G_|S, b)
    ▷ Input: connected sub graph G_|S, cost budget b
    ▷ Output: an optimal join tree for G_|S
 1  if BestTree[S] ≠ NULL and cost(BestTree[S]) ≤ b
 2     then return BestTree[S]
 3  if lB[S] ≤ b
 4     then return NULL
 5  if attempts[S] > 0
 6     then if b < uB[S]
 7             then b ← uB[S]
 8             else b ← MAX(b, lB[S] * 2^attempts[S])
 9  attempts[S] ← attempts[S] + 1
10  if uB[S] < b
11     then b ← uB[S]
12  nlB ← ∞
13  for all (S1, S2) ∈ P^sym_ccp(S)
14  do if LBE(S1, S2) > MIN(b, cost(BestTree[S]))
15       then nlB ← MIN(nlB, LBE(S1, S2))
16            continue
17     c⋈ ← minimal costs of operator ⋈
18     b' ← MIN(b, cost(BestTree[S]))
19     if BestTree[S2] ≠ NULL
20       then cr ← cost(BestTree[S2])
21       else cr ← lB[S2]
22     b' ← b' − c⋈
23     lT ← TDPG_PACBI(G_|S1, b' − cr)
24     if lT ≠ NULL
25       then c_lT ← cost(lT)
26            b' ← b' − c_lT
27            rT ← TDPG_PACBI(G_|S2, b')
28            if rT ≠ NULL
29              then BUILDTREE(G_|S, lT, rT, b)
30                   c_rT ← cost(rT)
31                   nlB ← MIN(nlB, c_lT + c_rT + c⋈)
32              else nlB ← MIN(nlB, c_lT + lB[S2] + c⋈)
33       else nlB ← MIN(nlB, lB[S1] + lB[S2] + c⋈)
34  if BestTree[S] = NULL
35     then lB[S] ← MAX(b, nlB)
36  return BestTree[S]
```

Fig. 5. Pseudocode for TDPG_{ACBI}

V. Evaluation

This section summarizes our experimental findings. We start by describing 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. [10]. They have the advantage of being very precise. The lower bound estimation method LBE (Section IV-B) bases its estimate on the intermediate relations that are the input for the next join.

B. Workload

To generate our workload, we have implemented a generic query graph generator. In a first step, it generates chain, star, cycle, and clique queries as well as random acyclic and cyclic graphs. For the latter, edges are randomly added by selecting two relation's indices using uniformly distributed random numbers. To generate cardinalities and selectivities, we follow the approach of [12], which is based on a proposal by Steinbrunn et. al. [13]. According to the kind of join, we can distinguish between foreign-key key joins and other joins, which we call random joins. Based on this, we generate foreign-key key join queries and random join queries. For random join queries, cardinalities and domain sizes are generated according to the scheme in Fig. 6. If — as proposed in [13] — selectivities are computed by choosing two random attributes and using $\frac{1}{max(dom(A_1),dom(A_2))}$, this often leads to intermediate cardinalities less than 1 which then are successively increased to become huge again. As this does not seem to be realistic, we propose and use foreign-key join queries. For relation and domain sizes, the same scheme as before is used. Then, with a probability of 10%, the selectivity of a join edge is computed as described above for random queries, and with a probability of 90% the selectivity is computed such that the cardinality of the result is equal to the cardinality of the relation with the foreign key.

As DeHaan and Tompa pointed out, pruning techniques are quite unsuccessful with star queries. This makes them perfect for analyzing the overhead of a branch-and-bound pruning strategy. To use them for this purpose, we decrease the chances for pruning down to zero by setting the join selectivity of an edge to the reciprocal of the dimension's cardinality.

Relation Size	Prob.
10-100	15%
100-1000	30%
1000-10000	25%
10000-100000	20%

Domain Size	Prob.
2-10	5%
10-100	50%
100-500	35%
500-1000	15%

Fig. 6. Relation and domain sizes for random join queries, as proposed by Steinbrunn et al.

C. Organizational Overview

In our empirical analysis, we compare the accumulated-predicted cost bounding algorithm named APCB, which was given by DeHaan and Tompa (Section IV-C) [3], to our novel branch-and-bound pruning algorithm called APCBI. We instantiate both algorithms with three different partitioning strategies to calculate the ccps: MinCutLazy [3], MinCutBranch [4] and our novel conservative partitioning algorithm MinCutConservative (Section III). Throughout this section, we use the abbreviated names, as shown in Table

V-C. By TDPG$_{APCBI_Opt}$ we indicate the APCBI pruning strategy with pre-calculated optimal (tight) upper bounds. We derive those by extracting the costs of the optimal subjoin trees of each plan class after a run of DPccp and make them available with a lookup to $uB[S]$ (line 11 of TDPG$_{APCBI}$). Since APCBI_Opt is of no practical use, it is only shown here to indicate the theoretical lower bound of ACB. Therefore, we do not include the pre-computation time of the optimal upper bound's runtime results for TDPG$_{APCBI_Opt}$.

In order to put all top-down join enumeration algorithms into perspective, we include the results of Moerkotte and Neumann's DPccp [2] as the state of the art in bottom-up join enumeration via dynamic programming. We present our results in terms of the quotient of the algorithm's execution time and the execution time of DPccp. We refer to this quotient as the *normed time*. Table II shows the average, minimum, and maximum normed time over the whole workload for a particular graph type. Since the normed time for DPccp is always 1, we rather give its elapsed time in seconds.

Table III shows the details of the pruning behavior. With max_s or avg_s, we refer to the maximum or average of the following quotient. We divide the number of times a join tree was requested and successfully returned by the number of built join trees by DPccp. Note that the divisor corresponds to the number of plan classes, which equals the number of connected subgraphs. As a consequence, the subscript $_s$ indicates that we are talking about the normed number of plan classes for which a plan was *successfully* built. Analogously, max_f (avg_f) denotes the maximum (average) of the quotient where we divide the number of times a join tree was requested but not built by the number of join trees built by DPccp. The latter quotient gives us the ratio of *failed* builds compared to the total number of DPccp's optimal join tree builds.

In order not to overload the evaluation charts, we decided to include only a subset of the possible combinations between algorithms and pruning strategies we investigated. We present TDMcL to show the performance difference to DPccp. This deficiency can be overcome by pruning (TDMcL$_{APCB}$). We also present TDMcB$_{APCB}$ and TDMcB$_{APCBI}$ to allow for a comparison of the two pruning strategies. Finally, the results for TDMcC$_{APCBI}$ are presented because it dominates all the other combinations. The workload consists of more than 20000 query graphs. 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 11.0. We used the Intel C++ compiler with the compiler option O3.

D. Experiments

This section summarizes our experimental findings. We start with an evaluation of acyclic query graphs, present the results for cyclic queries later and investigate our different pruning advancements at the end.

1) Acyclic Query Graphs: The minimum, maximum, and average normed runtimes over the whole acyclic workload for chain, star, and random acyclic queries are given in Table II. We see, for example, for random acyclic queries that using APCBI instead of APCB improves the runtime of all three enumerators by a factor of $3.1 - 5.1$. We also see that for APCB the maximum normed runtimes sometimes deviate substantially from the average. APCBI does not show this behavior. Hence, APCBI exhibits a much better worst case behavior than APCB. Comparing the performance of TDMcL$_{APCB}$ and TDMcC$_{APCBI}$ for random acyclic graphs, we can determine an improvement factor of about 9.

TABLE I
ABBREVIATED NAMES OF DIFFERENT PARTITIONING ALGORITHMS AND PRUNING STRATEGIES.

Abbreviated Name	Partitioning Strategy	Pruning Type
TDMcL	MinCutLazy	none
TDMcL$_{PCB}$	MinCutLazy	TDPG$_{PCB}$
TDMcL$_{APCB}$	MinCutLazy	TDPG$_{APCB}$
TDMcL$_{APCBI}$	MinCutLazy	TDPG$_{APCBI}$
TDMcL$_{APCBI_Opt}$	MinCutLazy	TDPG$_{APCBI_Opt}$
TDMcB	MinCutBranch	none
TDMcB$_{PCB}$	MinCutBranch	TDPG$_{PCB}$
TDMcB$_{APCB}$	MinCutBranch	TDPG$_{APCB}$
TDMcB$_{APCBI}$	MinCutBranch	TDPG$_{APCBI}$
TDMcB$_{APCBI_Opt}$	MinCutBranch	TDPG$_{APCBI_Opt}$
TDMcC	MinCutConservat.	none
TDMcC$_{PCB}$	MinCutConservat.	TDPG$_{PCB}$
TDMcC$_{APCB}$	MinCutConservat.	TDPG$_{APCB}$
TDMcC$_{APCBI}$	MinCutConservat.	TDPG$_{APCBI}$
TDMcC$_{APCBI_Opt}$	MinCutConservat.	TDPG$_{APCBI_Opt}$

Fig. 7 and Fig. 9 show how the runtimes increase with the number of relations in the random acyclic query and chain queries respectively. Note that the relative order of the algorithms is independent of the number of relations.

Fig. 8 shows the density plot of the normed runtime for random acyclic graphs. TDMcC$_{APCBI}$ clearly shows the highest improvement factor of the normed runtimes for a large part of the queries.

Let us now delve into the details. We see that, except for star queries (Fig. 10), pruning decreases the optimization time for acyclic graphs. However, there are certain cases where pruning performs worse, e.g., for all star queries independent of the pruning method used.

In some rare cases, worsening by pruning occurs in combination with APCB. Our novel pruning strategy, on the other hand, proves to be risk free, again with the exception of star queries. The values of max_f give us the explanation for APCB's worst case behavior (Table III). Through the *improved lower bounds* and our *rising budget*, APCBI can prevent this negative effect (described in Section IV-D) as the corresponding values of max_f verify. Note, that [3] reported higher performance gains through APCB. The reason that we cannot repeat their results is that we implemented more realistic cost functions which are more expensive to compute (line 3 of TDPG$_{ACB}$). In fact, because of the more sophisticated cost functions used here, PCB alone turned out to be more efficient than combined with with ACB in most scenarios.

APCB is much less robust than APCBI, as the deviation of the avg_s and avg_f values between the corresponding three different TDMcX$_{APCB}$ results compared to TDMcX$_{APCBI}$ proves.

TABLE II
MINIMUM, MAXIMUM AND AVERAGE OF THE NORMED RUNTIMES.

Algorithm	min	max	avg	min	max	avg	min	max	avg
	Chain			Star			Acyclic		
DPccp in sec	0.0001 s	0.0120 s	0.0031 s	0.0001 s	15.6530 s	1.8314 s	0.0002 s	4.4883 s	0.1540 s
TDMcL	0.7801 x	3.1256 x	1.4265 x	1.0000 x	2.2923 x	1.4007 x	0.7899 x	2.4272 x	1.3208 x
TDMcL$_{PCB}$	0.2600 x	2.0412 x	0.8451 x	1.0400 x	3.2256 x	1.4083 x	0.0126 x	2.0409 x	0.4297 x
TDMcL$_{APCB}$	0.2596 x	3.1248 x	0.9131 x	1.2500 x	3.7035 x	1.7643 x	0.0093 x	120.0473 x	1.0323 x
TDMcL$_{APCBI}$	0.1132 x	1.5846 x	0.5206 x	1.0667 x	2.5915 x	1.7290 x	0.0022 x	1.2266 x	0.2030 x
TDMcL$_{APCBI_Opt}$	0.0638 x	1.0769 x	0.3150 x	1.1667 x	2.4445 x	1.6322 x	0.0016 x	1.0528 x	0.1359 x
TDMcB	0.4899 x	1.5361 x	0.9167 x	0.6410 x	1.4000 x	1.0163 x	0.5000 x	1.6025 x	0.9623 x
TDMcB$_{PCB}$	0.1111 x	1.1176 x	0.4811 x	0.6000 x	1.7192 x	1.0442 x	0.0150 x	1.3987 x	0.3061 x
TDMcB$_{APCB}$	0.1347 x	1.4748 x	0.5150 x	0.9167 x	2.0253 x	1.3663 x	0.0043 x	72.7739 x	0.5935 x
TDMcB$_{APCBI}$	0.0724 x	0.7905 x	0.2839 x	0.9000 x	1.7778 x	1.3667 x	0.0016 x	0.7962 x	0.1212 x
TDMcB$_{APCBI_Opt}$	0.0379 x	0.5278 x	0.1797 x	0.9001 x	1.6667 x	1.3155 x	0.0009 x	0.5302 x	0.0853 x
TDMcC	0.5300 x	1.5650 x	0.9415 x	0.6290 x	1.5556 x	1.0152 x	0.6000 x	1.7240 x	0.9809 x
TDMcC$_{PCB}$	0.0833 x	0.8627 x	0.4044 x	0.5000 x	1.7192 x	1.0678 x	0.0058 x	1.1401 x	0.2339 x
TDMcC$_{APCB}$	0.0909 x	1.1333 x	0.4132 x	0.9167 x	1.9000 x	1.3811 x	0.0034 x	44.6346 x	0.3632 x
TDMcC$_{APCBI}$	0.0493 x	1.0286 x	0.2501 x	0.9667 x	1.7263 x	1.3875 x	0.0017 x	0.7640 x	0.1163 x
TDMcC$_{APCBI_Opt}$	0.0455 x	0.6500 x	0.1851 x	0.9167 x	1.8889 x	1.3539 x	0.0014 x	0.6897 x	0.0880 x
	Cycle			Clique			Cyclic		
DPccp in sec	0.0001 s	0.0280 s	0.0088 s	0.0001 s	22.1414 s	3.2147 s	0.0002 s	57.5836 s	5.4579 s
TDMcL	0.8182 x	2.1645 x	1.3303 x	1.3529 x	2.5210 x	1.7666 x	0.9500 x	3.0000 x	1.7445 x
TDMcL$_{PCB}$	0.1520 x	1.5388 x	0.5257 x	0.0294 x	1.3516 x	0.2989 x	0.0097 x	1.7442 x	0.1717 x
TDMcL$_{APCB}$	0.1480 x	1.5556 x	0.4739 x	0.0175 x	2.5935 x	0.2778 x	0.0086 x	9.0725 x	0.1546 x
TDMcL$_{APCBI}$	0.0879 x	1.4000 x	0.2937 x	0.0118 x	1.2174 x	0.1376 x	0.0044 x	0.8651 x	0.0655 x
TDMcL$_{APCBI_Opt}$	0.0500 x	0.8750 x	0.1722 x	0.0053 x	0.8400 x	0.0908 x	0.0031 x	0.6029 x	0.0491 x
TDMcB	0.5714 x	1.5556 x	0.8868 x	0.6666 x	1.3157 x	1.0008 x	0.4999 x	1.5228 x	1.0279 x
TDMcB$_{PCB}$	0.1167 x	0.9259 x	0.3260 x	0.0074 x	0.5652 x	0.1164 x	0.0036 x	0.9999 x	0.0677 x
TDMcB$_{APCB}$	0.0769 x	0.9259 x	0.2854 x	0.0063 x	1.0330 x	0.1134 x	0.0031 x	1.5716 x	0.0605 x
TDMcB$_{APCBI}$	0.0538 x	0.5429 x	0.1733 x	0.0043 x	0.6720 x	0.0619 x	0.0023 x	0.6319 x	0.0267 x
TDMcB$_{APCBI_Opt}$	0.0286 x	0.4255 x	0.1027 x	0.0032 x	0.2568 x	0.0294 x	0.0016 x	0.3469 x	0.0185 x
TDMcC	0.6000 x	1.6392 x	0.9073 x	0.7059 x	1.2857 x	0.9850 x	0.4999 x	1.5789 x	1.0179 x
TDMcC$_{PCB}$	0.1000 x	0.9176 x	0.2911 x	0.0072 x	0.5652 x	0.1068 x	0.0044 x	0.8721 x	0.0622 x
TDMcC$_{APCB}$	0.0833 x	0.8125 x	0.2713 x	0.0052 x	4.5459 x	0.2463 x	0.0045 x	4.7003 x	0.0977 x
TDMcC$_{APCBI}$	0.0435 x	0.6300 x	0.1572 x	0.0035 x	0.6870 x	0.0578 x	0.0024 x	0.4831 x	0.0250 x
TDMcC$_{APCBI_Opt}$	0.0192 x	0.4310 x	0.1039 x	0.0030 x	0.2838 x	0.0286 x	0.0016 x	0.3758 x	0.0182 x

We generated the selectivities for star queries in a way that disabled pruning. This is confirmed by $avg_s = 1$. Consequently, the normed runtime for star queries gives us an idea of the overhead of the pruning techniques. The overhead is largest for TDMcL$_{APCB}$ and TDMcL$_{APCBI}$, as indicated by the average normed runtimes of 1.7 compared to 1.4 (TDMcL).

The novel algorithm APCBI not only decreases the optimization time, but also requires less space for its memotable, as the avg_s values in Table III indicate. For random acyclic graphs, TDMcC$_{APCB}$ had to build 35% of DPccp built join trees, whereas TDMcC$_{APCBI}$ could drop this to a value of 14%.

2) Cyclic Query Graphs: The minimum, maximum, and average normed runtimes over the whole cyclic workload for cycle, clique, and random cyclic queries are given in Table II. We see, for example, for random cyclic queries that using APCBI instead of APCB improves the runtime of all three enumerators by an average factor of $2 - 4$. We also see that for APCB the maximum normed runtimes sometimes strongly deviate from the average. On the other hand, APCBI does not show such a worst case behavior. Our results indicate that APCB's runtime is up to a factor of 10 higher than APCBI.

As already outlined for acyclic graphs, APCBI compared to APCB proves to be much more robust. Again, we determine this from the deviation among the avg_s and avg_f values, which verify that the enumeration order has much less impact on the pruning behavior of APCBI than on the pruning behavior of APCB.

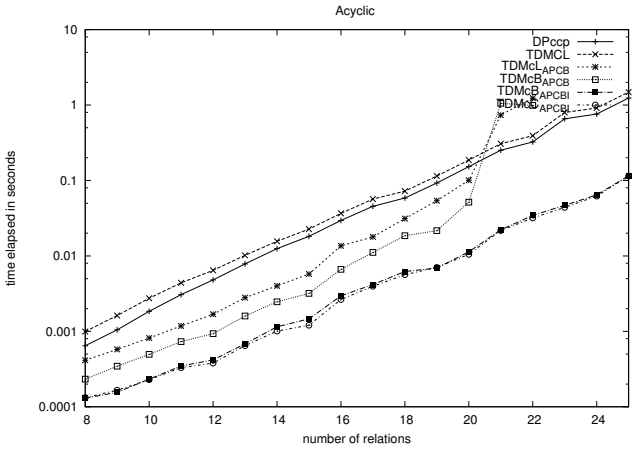

Fig. 7. Performance results for random acyclic queries that are neither chain nor star queries.

Cycle and clique queries belong to the same group of cyclic graphs, but in terms of the number of ccps, they are on two opposite sides of the spectrum. Cycles have the lowest number of edges that is possible for cyclic graphs, removing one edge would result in a chain query. Cliques have the maximal number of edges possible. Therefore, the potential for performance improvements through pruning is much higher for the latter, as the results in Table II show. The normed time for TDMcC$_{APCBI}$ decreases by a factor of 2.7 between the

TABLE III

AVERAGE AND MAXIMUM: NUMBER OF OPTIMAL JOIN TREES BUILT ($_s$) AND NUMBER OF FAILED JOIN TREE REQUESTS ($_f$) NORMED WITH THE NUMBER OF ALL POSSIBLE JOIN TREES.

Algorithm	avg_s	max_s	avg_f	max_f	avg_s	max_s	avg_f	max_f	avg_s	max_s	avg_f	max_f
	Chain				Star				Acyclic			
TDMcL$_{PCB}$	0.79	1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.45	1.00	0.00	0.00
TDMcL$_{APCB}$	0.64	1.00	0.21	3.76	1.00	1.00	0.00	0.00	0.23	0.90	1.98	256.89
TDMcL$_{APCBI}$	0.39	0.71	0.14	0.77	1.00	1.00	0.00	0.00	0.15	0.72	0.11	0.86
TDMcL$_{APCBI_Opt}$	0.31	0.70	0.14	0.51	1.00	1.00	0.00	0.00	0.12	0.55	0.09	0.64
TDMcB$_{PCB}$	0.74	1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.48	1.00	0.00	0.00
TDMcB$_{APCB}$	0.70	1.00	0.01	0.68	1.00	1.00	0.00	0.00	0.43	1.00	0.01	0.35
TDMcB$_{APCBI}$	0.39	0.73	0.15	0.77	1.00	1.00	0.00	0.00	0.15	0.67	0.12	0.80
TDMcB$_{APCBI_Opt}$	0.30	0.58	0.16	0.51	1.00	1.00	0.00	0.00	0.12	0.57	0.09	0.64
TDMcC$_{PCB}$	0.66	1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.38	1.00	0.00	0.00
TDMcC$_{APCB}$	0.64	1.00	0.01	0.49	1.00	1.00	0.00	0.00	0.35	1.00	0.01	0.39
TDMcC$_{APCBI}$	0.33	0.69	0.14	0.61	1.00	1.00	0.00	0.00	0.14	0.62	0.10	0.84
TDMcC$_{APCBI_Opt}$	0.30	0.70	0.15	0.49	1.00	1.00	0.00	0.00	0.12	0.56	0.09	0.63
	Cycle				Clique				Cyclic			
TDMcL$_{PCB}$	0.61	0.88	0.00	0.00	0.34	0.82	0.00	0.00	0.18	0.92	0.00	0.00
TDMcL$_{APCB}$	0.42	0.70	0.15	2.32	0.16	0.73	0.95	53.59	0.08	0.86	0.32	78.10
TDMcL$_{APCBI}$	0.27	0.56	0.07	0.33	0.09	0.55	0.02	0.11	0.03	0.54	0.02	0.40
TDMcL$_{APCBI_Opt}$	0.17	0.56	0.08	0.32	0.05	0.55	0.02	0.16	0.02	0.44	0.02	0.35
TDMcB$_{PCB}$	0.61	0.89	0.00	0.00	0.30	0.73	0.00	0.00	0.16	0.96	0.00	0.00
TDMcB$_{APCB}$	0.41	0.78	0.12	1.95	0.14	0.64	2.04	73.56	0.06	0.79	0.64	173.82
TDMcB$_{APCBI}$	0.27	0.56	0.09	0.38	0.09	0.64	0.02	0.17	0.03	0.53	0.02	0.46
TDMcB$_{APCBI_Opt}$	0.17	0.40	0.09	0.27	0.04	0.35	0.02	0.13	0.02	0.41	0.02	0.33
TDMcC$_{PCB}$	0.56	0.89	0.00	0.00	0.30	0.77	0.00	0.00	0.16	0.96	0.00	0.00
TDMcC$_{APCB}$	0.40	0.78	0.14	0.71	0.13	0.65	15.80	448.20	0.06	0.82	4.63	559.89
TDMcC$_{APCBI}$	0.24	0.56	0.09	0.31	0.09	0.64	0.02	0.14	0.03	0.45	0.02	0.41
TDMcC$_{APCBI_Opt}$	0.17	0.41	0.08	0.30	0.05	0.42	0.02	0.13	0.02	0.43	0.02	0.36

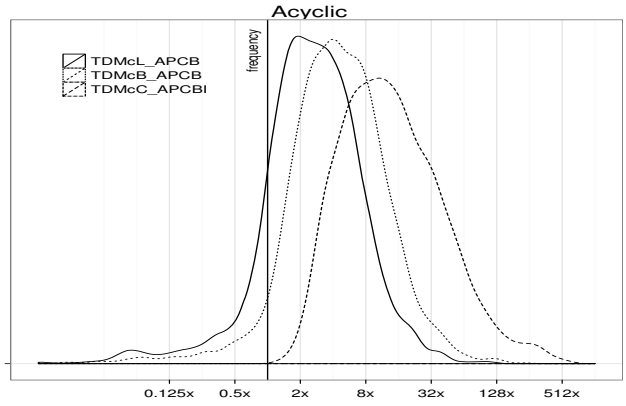

Fig. 8. Density plot of acyclic queries.

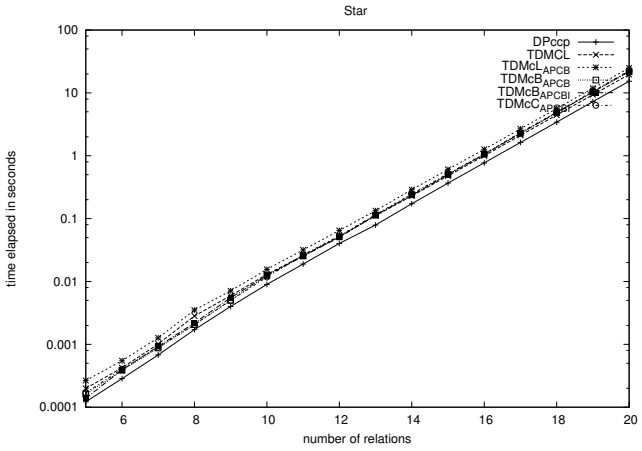

Fig. 10. Performance results for star queries.

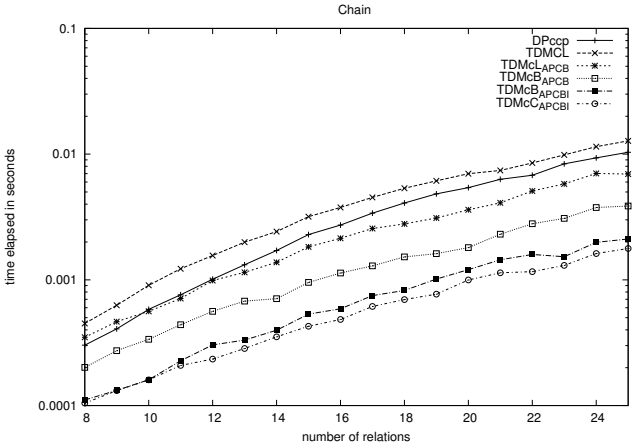

Fig. 9. Performance results for chain queries.

two graphs. This is also true for the space requirements of the memotable, as can be seen from the different avg_s values (Table III). Whereas TDMcC$_{APCBI}$ still requires 24% of DPccp's size of the memotable, the space requirement drops down to 9% when clique queries are optimized.

Comparing the performance of TDMcL$_{APCB}$ and TDMcC$_{APCBI}$ for random cyclic graphs, we see an improvement factor of more than 6. Fig. 13 supports our claim that TDMcC$_{APCBI}$ is the best performing join enumeration algorithm for cyclic graphs. The differences between the two different pruning algorithms vary on average between a factor of 2.3 for MinCutBranch and 3.9 for MinCutConservative. The speedup factor of TDMcC$_{APCBI}$, compared to DPccp, has an average of 40. This factor increases significantly with the number of relations in the query.

Fig. 14 shows the corresponding density plot. As can be

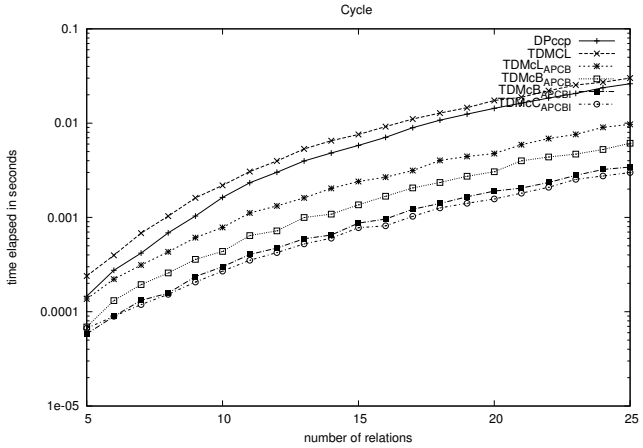

Fig. 11. Performance results for cycle queries.

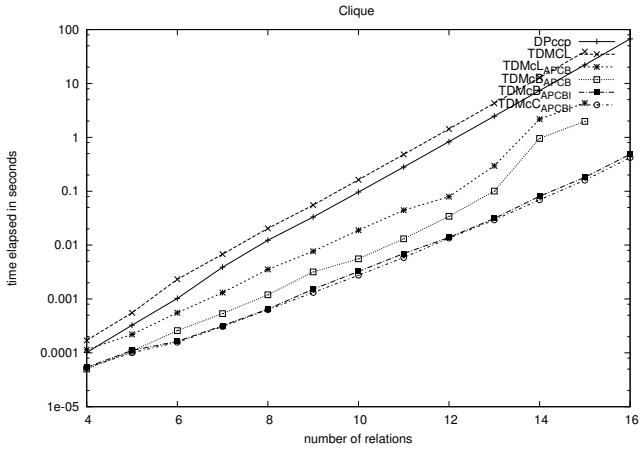

Fig. 12. Performance results for clique queries.

seen, TDMcC$_{APCBI}$ is much steeper and farther to the right than the other competitors. This means that for a much higher fraction of queries it achieves a far lower runtime.

3) Impact of Technical Advances: This section investigates how the six new pruning advancements (Section IV-D) improve the efficiency of APCB. Figure 15 presents the aggregated runtime results. We measured the runtime of every single pruning advancement on top of TDMcC$_{APCB}$. This

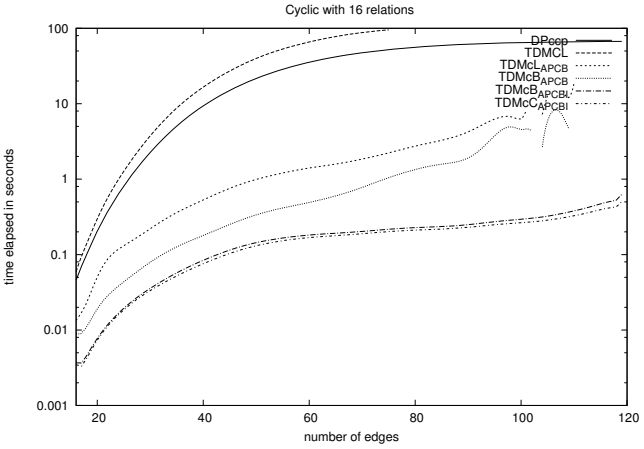

Fig. 13. Performance results for cyclic queries with 16 vertices.

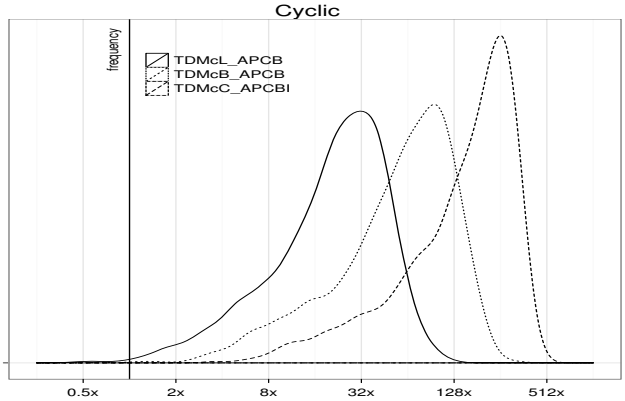

Fig. 14. Density plot of cyclic queries with 16 relations.

was done with the exception of *remapping the query graph* (Advancement no. 6) and the *join heuristic* (Advancement no. 2), which we measured as a whole, since remapping depends on a heuristic. As already mentioned, the join heuristic we implemented is GOO [11]. Looking at the results (Figure 15) for the acyclic queries, we can see that all advancements significantly decrease runtime, with the exception of GOO and remapping the graph, which itself appears to be counterproductive. That is why we also measured all advancements at once, except for the latter. This combination is displayed as the third last bar. When we compare it to APCBI, we can see that only in combination with the other advancements it further improves runtime. For acyclic graphs the rising budget is the most significant improvement.

These observations do not hold for random cyclic graphs. Here, all advancements achieve nearly the same performance gains. Note that these gains become more significant for workloads with a larger number of vertices, whereas the performance of APCB alone decreases. As already pointed out, the performance decreases because the evaluation of the cost functions is relatively expensive if plan classes are planned several times without producing a plan. The last bar shows APCBI with the upper bounds for all different connected plan classes set to the cost of the corresponding optimal plan. But compared to APCBI, it brings an improvement of 24% at the most. Hence, let us emphasize that there is not much potential for improving accumulated cost bounding strategies in the future, since this number is only of theoretical nature and can never be achieved in practice.

VI. CONCLUSION

Accumulated and predicted cost bounding are two long known pruning strategies. DeHaan and Tompa investigated how both bounding methods can be combined to yield the effective APCB pruning method. We improved this method by incorporating six new technical advances. The resulting pruning method APCBI has three major advantages. First, it prunes much more effectively than APCB. Second, it is more robust, and third, it avoids worst case behavior by several orders of magnitude.

To have a better basis for our robustness statements, we presented a new top-down enumeration method. It turns out

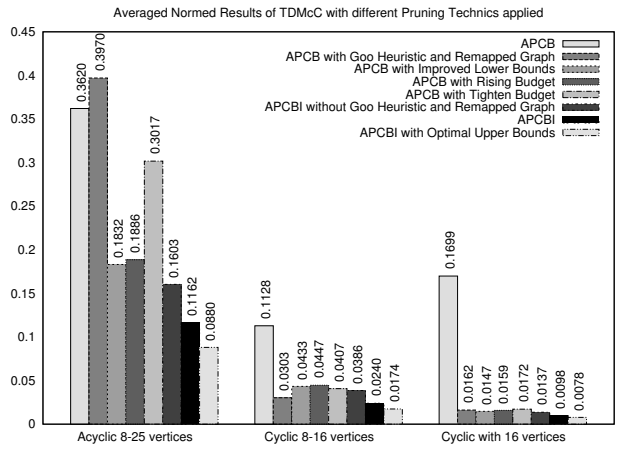

Fig. 15. Performance of different Pruning Advancements.

that this method is superior to the two enumerators published so far. Finally, we have shown that there is only a minor potential for future improvements of our techniques.

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, pp. 930–941.

[3] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration," in *SIGMOD*, 2007, pp. 1098–1109.

[4] P. Fender and G. Moerkotte, "A new, highly efficient, and easy to implement top-down join enumeration algorithm," in *ICDE*, 2011.

[5] P. Fender and G. Moerkotte, "Reassessing top-down join enumeration," to appear in *IEEE TKDE*, 2012.

[6] G. Graefe and W. J. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in *Proceedings of the Ninth International Conference on Data Engineering*. IEEE Computer Society, 1993.

[7] G. Graefe, "The cascades framework for query optimization," *IEEE Data Eng. Bull.*, vol. 18, no. 3, 1995.

[8] 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*, 2001.

[9] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration (extended version)," University of Waterloo, Tech. Rep., 2007.

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

[11] L. Fegaras, "A new heuristic for optimizing large queries," in *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, ser. DEXA, 1998.

[12] T. Neumann, "Query simplification: graceful degradation for join-order optimization," in *SIGMOD*, 2009.

[13] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *The VLDB Journal*, vol. 6, pp. 191–208, August 1997.

[14] B. Vance and D. Maier, "Rapid bushy join-order optimization with cartesian products," in *SIGMOD*, 1996, pp. 35–46.

APPENDIX

A. Building a Join Tree

Our generic top-down join enumeration algorithm relies on a method to construct the join trees and compare their costs. We give the pseudocode of BUILDTREE in Figure 16. It is used to compare the cost of the join trees that belong to the same $G_{|S}$. 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 5) and then compare their costs. If the method succeeds, the cheapest join tree gets registered with $BestTree[S]$ as long as its costs are within a certain budget b (line 2 and 6). An actual budget is only passed in within the context of branch-and-bound pruning, otherwise the budget will be assigned with ∞. For constructing a join tree, we call 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, under 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 there even has no tree for S been built yet, $BestTree[S]$ gets registered with the cT as long as the condition of line 2 holds. For building the second tree, we just exchange the arguments (line 5). 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 or there has been no join tree registered yet and cT is within the budget b, $BestTree[S]$ gets registered with the new join tree.

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 [10], where $card(S_x) \leq card(S_y) \Rightarrow cost(S_x \bowtie S_y) \leq cost(S_y \bowtie S_x)$ holds, with $card$ denoting the number of tuples or pages and S_x, S_y denoting disjoint sets of relations.

BUILDTREE$(G_{|S}, Tree_1, Tree_2, b)$
▷ **Input:** $G_{|S}$, two sub join trees, cost budget b
1 $cT \leftarrow$ CREATETREE$(Tree_1, Tree_2)$
2 **if** $cost(cT) \leq b$
3 **then if** $BestTree[S] =$ NULL or $cost(BestTree[S]) > cost(cT)$
4 **then** $BestTree[S] \leftarrow cT$
5 $cT \leftarrow$ CREATETREE$(Tree_2, Tree_1)$
6 **if** $cost(cT) \leq b$
7 **then if** $BestTree[S] =$ NULL or $cost(BestTree[S]) > cost(cT)$
8 **then** $BestTree[S] \leftarrow cT$

Fig. 16. Pseudocode for BUILDTREE

B. Naive Partitioning

As we have already seen, the generic top-down enumeration algorithm iterates over the elements of $P_{ccp}^{sym}(S)$. Now, we show how the csg-cmp pairs for S can be computed by a naive generate-and-test strategy. We call our algorithm PARTITION$_{naive}$ and give its pseudocode in Figure 17. In line 1, all $2^{|V|} - 2$ possible non-empty and proper subsets of V are enumerated. For rapid subset enumeration, the method described in [14] 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 $V \setminus S$ in line 2. As we will see, the partitioning strategies proposed by DeHaan and Tompa ensure this by design. Three conditions have to be met such that a partition $(S, V \setminus S)$ is a ccp. We check the connectivity of $G_{|S}$ and $G_{|V \setminus S}$ in line 2. The third condition that S needs to be

PARTITION$_{naive}(G)$

▷ **Input:** a connected graph $G = (V, E)$
▷ **Output:** $P^{sym}_{ccp}(V)$

1 **for all** $S \subset V \wedge S \neq \emptyset$
2 **do if** $max_{index}(S) \leq max_{index}(V \setminus S)$ && $G_{|S}$ is connected and $G_{|V \setminus S}$ is connected
3 **then** emit$(S, V \setminus S)$

Fig. 17. Pseudocode for naive partitioning

connected to $V \setminus S$ is ensured implicitly by the requirement that the graph we hand over as input is connected.

C. Exploring Connected Components

This section explains how GETCONNECTEDPARTS works. For performance reasons, the computation of the connected components is implemented as a twofold strategy. The first part of GETCONNECTEDPARTS implements an improved connection test, and the second part is only executed if the connection test fails. The pseudo code is given in Figure 18.

We start by explaining the first part. It is based on the following two observations. First of all: Knowing that conservative partitioning ensures that $C \cup \{v\}$ is connected, it should be sufficient for testing the connectivity of $S \setminus (C \cup \{v\})$ to check if the neighbors $\mathcal{N}(C \cup \{v\})$ are connected to each other within the complement $S \setminus (C \cup \{v\})$. In other words, we can ensure the connectivity of the complement by testing for a weaker condition that checks if all elements of $\mathcal{N}(C \cup \{v\})$ lie on a common path that does not contain any vertices of $C \cup \{v\}$. And second: If we know that the old complement $S \setminus C$ that still includes the current v was connected, it suffices to check if only the elements of $\mathcal{N}(\{v\}) \setminus C$ are connected within $S \setminus (C \cup \{v\})$.

Because MINCUTCONSERVATIVE ensures that $S \setminus C$ has to be connected, we implemented our test by checking only the latter, even weaker condition. On average, the latter test is cheaper to execute than a common connection test. In the best case, its complexity is in $O(1)$. The worst case is identical to the complexity of the common connection test, which is in $O(|S \setminus (C \cup \{v\})|)$.

Now we are ready to explain the first part (lines of 1 to 14) of GETCONNECTEDPARTS. In Line 1 of GETCONNECTEDPARTS, we compute those neighbors of T that are not in C and save them in the set N. Note that T is a one-element set containing the v previously added to C. If N contains only one element, $S \setminus C$ must be connected and the test returns the whole complement (line 3). As a starting point for the test, we choose an arbitrary vertex $n \in N$ (line 5) and assign it to L'. Throughout the test, the set U contains all the vertices of N that have not been reached yet by the generation of neighbors of $\{n\}$. In the loop of Line 7 to 11, the indirect neighborhood [2] of $\{n\}$ is computed by enlarging L' with a generation of neighbors of the previous L' in every iteration. Thereby, L holds the previous L' (line 9), and D holds all the new elements of L'. Note that except from the first iteration of the loop, $\mathcal{N}(L) \cap (S \setminus C) = D$ holds.

GETCONNECTEDPARTS(S, C, T)

▷ **Input:** $C \subset S$ a connected set, $T \subseteq C$
▷ **Output:** set O of connected disjoint sets O_i
$\forall_{O_i, O_j \in O}\ O_i \subseteq (S \setminus C) \wedge O_i \cap O_j = \emptyset$

1 $N \leftarrow \mathcal{N}(T) \setminus C$
2 **if** $|N| \leq 1$
3 **then return** $\{S \setminus C\}$
4 $L \leftarrow \emptyset$
5 $L' \leftarrow n \in N$
6 $U \leftarrow N \setminus L'$
7 **while** $L \neq L'$ and $U \neq \emptyset$
8 **do** $D \leftarrow (L' \setminus L)$
9 $L \leftarrow L'$
10 $L' \leftarrow L' \cup \mathcal{N}(D) \setminus C$
11 $U \leftarrow U \setminus L'$
12 **if** $U = \emptyset$
13 **then return** $\{S \setminus C\}$
14 **else** $O \cup \{L\}$
▷ now explore all other to C adjacent subsets
15 $U \leftarrow N \setminus L$
16 **while** $U \neq \emptyset$
17 **do** $L \leftarrow \emptyset$
18 $L' \leftarrow n \in U$
19 **while** $L \neq L'$
20 **do** $D \leftarrow (L' \setminus L)$
21 $L \leftarrow L'$
22 $L' \leftarrow L' \cup \mathcal{N}(D) \setminus C$
23 $U \leftarrow U \setminus L$
24 $O \cup \{L\}$
25 **return** O

Fig. 18. Pseudocode for GETCONNECTEDPARTS

Because $|D| < |L'|$ holds, we compute the next generation of neighbors from D instead of L' (line 10), which is clearly more efficient. If we could reach new elements of N, we must subtract them from U (line 11). Once U equals the empty set, the loop is interrupted (line 7), and it is obvious that all $n \in N$ are reachable within $S \setminus C$ so that $S \setminus C$ is connected. Hence, the whole complement is returned (line 13). If on the other hand L' cannot be enlarged further so that $L = L'$ holds, we have computed all indirect neighbors of $\{n\}$ (within $S \setminus C$) and meet the loop's stop condition. In that case, we could not reach every element of N so that $N \setminus L' \neq \emptyset$ must hold and we have computed our first O_i (line 14).

In the latter case, the second part of GETCONNECTEDPARTS is executed to compute the remaining O_j. In line 15, the set U is assigned with all neighbors of C that are not part of the already calculated O_1. U indicates if there might be any other O_j left to compute. Hence, U serves as an abort criteria for the loop of lines 16 to 24. The inner loop of lines 19 to 22 resembles the loop (7 to 11) of our first part. But this time we need to implement an early exit, as we have done by checking the old U (line 7). This was only necessary to discover as early as possible that all neighbors of $T = \{v\}$ are connected to each other. The sole purpose of this inner loop is to compute the remaining O_j.