T U M

I N S T I T U T F Ü R I N F O R M A T I K

On the Performance and Pruning Power of Different Join Enumeration Strategies

Viktor Leis

TUM-I1106
März 11

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-03-I1106-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2011

Druck: Institut für Informatik der
Technischen Universität München

On the Performance and Pruning Power of Different Join Enumeration Strategies

Viktor Leis

Technische Universität München

leis@in.tum.de

***Abstract*— To find the optimal join order two different generative join enumeration strategies have been proposed. The most commonly used one is dynamic programming which proceeds bottom-up. The alternative is top down enumeration with memoization. For both strategies algorithms exist that enumerate only solutions without cartesian products, which is a commonly used heuristics. With top-down enumeration it is possible to further improve optimization time by pruning the search space while still obtaining the optimal solution.**

In this paper we compare the performance of the different join enumeration strategies and possible pruning strategies. We propose improvements to the pruning algorithms from the literature and empirically evaluate the effect of pruning using different synthetic queries and cost functions in order to understand when significant speedups can be achieved. We find that for many queries a speedup by a factor of 2 to 10 can be expected.

I. Introduction

The problem of finding the optimal join order has been studied extensively. The classical enumeration strategy is dynamic programming [1]. The dynamic programming proceeds bottom-up, building larger logical expressions by combining previously computed intermediate results. Another strategy is to compute the result top-down with memoization: The problem is broken into smaller sub-problems, which are then solved recursively. Intermediate results are stored ("memoized") because they might be reused later.

Because the time complexity of dynamic programming for finding the optimal join order with cartesian products is $O(3^n)$, a common heuristics is to ignore join orders that need cartesian products. This can be achieved using a generate and test strategy, i.e. to check if a solution would result in a cartesian product after creating it. Unfortunately, for large queries most of the optimization time is spent doing this check, as most possible solutions include cartesian products. This motivated the development of a more efficient algorithm called DPccp which was developed by Moerkotte and Neumann [2]. The algorithm uses a graph theoretic approach and operates on the query graph with the relations of the query as vertices and all join conditions as edges between the relations. DPccp is based on the observation that any connected subgraph of the query graph can be produced without using cartesian products. For a given query graph DPccp enumerates all connected subgraphs and its "complements". A complement of a subgraph is a connected subgraph that is disjoint but connected to that subgraph. This strategy ensures that these pairs can be joined without cartesian products. DPccp is optimal in that it enumerates each connected subgraph and complement pair (ccp) exactly once. The order of the enumerated pairs is suitable for dynamic programming, i.e. smaller subproblems are enumerated before bigger problems that use the smaller ones as building blocks. Table I shows all enumerated ccps for a cycle query with 4 relations.

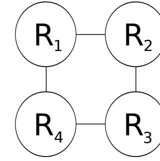


Fig. 1. Cycle join graph of a query with 4 relations

csg	complement	result
$\{R_3\}$	$\{R_4\}$	$\{R_3, R_4\}$
$\{R_2\}$	$\{R_3\}$	$\{R_2, R_3\}$
$\{R_2\}$	$\{R_3, R_4\}$	$\{R_2, R_3, R_4\}$
$\{R_2, R_3\}$	$\{R_4\}$	$\{R_2, R_3, R_4\}$
$\{R_1\}$	$\{R_4\}$	$\{R_1, R_4\}$
$\{R_1\}$	$\{R_3, R_4\}$	$\{R_1, R_3, R_4\}$
$\{R_1\}$	$\{R_2\}$	$\{R_1, R_2\}$
$\{R_1\}$	$\{R_2, R_3\}$	$\{R_1, R_2, R_3\}$
$\{R_1\}$	$\{R_2, R_3, R_4\}$	$\{R_1, R_2, R_3, R_4\}$
$\{R_1, R_2\}$	$\{R_4\}$	$\{R_1, R_2, R_4\}$
$\{R_1, R_2\}$	$\{R_3\}$	$\{R_1, R_2, R_3\}$
$\{R_1, R_2\}$	$\{R_3, R_4\}$	$\{R_1, R_2, R_3, R_4\}$
$\{R_1, R_4\}$	$\{R_3\}$	$\{R_1, R_3, R_4\}$
$\{R_1, R_4\}$	$\{R_2\}$	$\{R_1, R_2, R_4\}$
$\{R_1, R_4\}$	$\{R_2, R_3\}$	$\{R_1, R_2, R_4, R_4\}$
$\{R_1, R_2, R_4\}$	$\{R_3\}$	$\{R_1, R_2, R_3, R_4\}$
$\{R_1, R_2, R_3\}$	$\{R_4\}$	$\{R_1, R_2, R_3, R_4\}$
$\{R_1, R_3, R_4\}$	$\{R_2\}$	$\{R_1, R_2, R_3, R_4\}$

TABLE I

Enumerated connected subgraph (csg) and complement pairs for the cycle query graph from Figure 1

A top-down enumeration algorithm that avoids cartesian products was developed by DeHaan and Tompa [3]. The algorithm is based on minimal cuts. A cut is a set of edges that when deleted partitions the graph into two or more connected components. A cut is minimal if it divides the graph into exactly two connected components. Thus, all minimal cuts of a graph induce connected subgraph pairs which can be produced without using cartesian products. Each of the subgraphs are solved be applying the same procedure recursively. Figure 2 shows all resulting partitions for a cycle graph. A more efficient algorithm implementing this idea called MinCut-

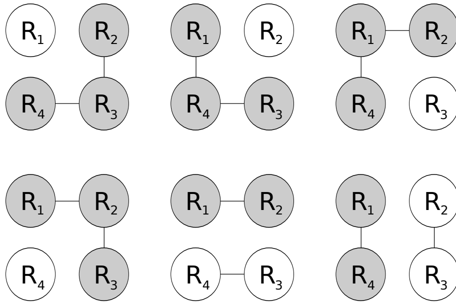


Fig. 2. The cycle graph from Figure 1 is partitioned into a gray and a white subgraph, such that both of them stay connected

BRANCH was developed by Fender and Moerkotte [4].

Usually dynamic programming performs marginally better than memoization, but the top-down approach has an advantage: more information is available, which can be used to prune the search space and to avoid exhaustively searching all of it while still guaranteeing to find the optimal solution.

In this paper we compare the performance and pruning power of the top-down and bottom-up join enumeration strategies for queries with a single select-project-join block. We only consider inner joins, as the optimal top-down enumeration algorithms do not support outer joins. In order to simplify our experiments, we also ignore the problem of interesting orders, even though they can be taken advantage of by the bottom-up and the top-down algorithms.

The rest of the paper is structured as follows: Based on the work of DeHaan and Tompa we introduce in Section II different pruning strategies and present an improved pruning algorithm. Section III describes the experimental setup for comparing the algorithms. Section IV evaluates the different join enumeration strategies and pruning algorithms. We conclude with section V by summarizing the results and discussing if pruning is beneficial in practice.

II. PRUNING THE SEARCH SPACE

Besides enumerating only connected subgraph pairs, the runtime of a join enumeration algorithm can be reduced by avoiding to enumerate some of those pairs, i.e. by pruning the search space. In this section we introduce several pruning methods. After discussing total-cost bounding for bottom-up enumeration, we introduce the powerful predicted-cost and accumulated-cost bounding algorithms for top-down enumeration. Finally we show some conditions that make it possible to directly compare possible solutions and thereby dismiss potential solutions early.

A. Total-Cost Bounding

When join alternatives are enumerated bottom-up, the fact that the cost of each subproblem must be less than the total cost can be used to speed up optimization time. Whenever the costs of all possible solutions for a subproblem are higher than the total cost bound, this subproblem cannot be part of the optimal solution. Dynamic programming with total cost bounding can therefore dismiss a possible solution if its left or right subproblem is too expensive. The necessary upper bound can be obtained by running a fast heuristics beforehand.

This simple optimization has been shown to be beneficial for plans with cartesian products [5], because there are many intermediate results with cartesian products that are very expensive and can therefore be pruned. It should be noted though that total-cost bounding is not really pruning the search space, as all ccps are enumerated.

Algorithm 1 Predicted-cost bounding

```
TDPCB(G = (V, E))
1  if memo[V] = NIL
2     then for (G_l, G_r) in PARTITION(G)
3             do if C(memo[V]) ≥ JOINLB(G_l, G_r)
4                   then p_l ← TDPCB(G_l)
5                        p_r ← TDPCB(G_r)
6                        p ← MAKEPLAN(p_l, p_r)
7                        if C(p) < C(bestPlan)
8                           then memo[V] ← p
9  return memo[V]

JOINLB(G_l = (V_l, E_l), G_r = (V_r, E_r))
1  jc ← JOINCOST(G_l, G_r)
2  return jc + lowerBound[V_l] + lowerBound[V_r]
```

B. Predicted-Cost Bounding

Top-down enumeration allows for actual pruning strategies because the algorithm has the freedom to decide whether to recurse into the subproblem. This means that in contrast to total cost bounding which must enumerate all ccps, an asymptotic speedup can be achieved.

Predicted-cost bounding is shown in Algorithm 1. We assume that the *memo* table is initialized with plans for all base relations, C is the cost function which computes the cost of a plan, and that $C(\text{NIL}) = \infty$. The only difference of Algorithm 1 from a basic top-down enumeration algorithm is the insertion of line 3, which compares the upper and lower bound for an expression. If the lower bound is higher than the upper bound, then it is not necessary to recurse into the left and right subproblems. The cheapest previously computed solution provides the upper bound for other expressions that produce the same results. The lower bound is a conservative prediction of the cost of an expression. Obviously, whether such a bound can be obtained and its quality depends on the cost function. Lower bounds for an expression might for instance be obtained by adding the cost of reading all base relations in that expression.

C. Accumulated-Cost Bounding

Accumulated-cost bounding, shown in Algorithm 2, maintains a budget which is passed down during recursion. Each join operator decreases the budget. The new budget is used as the budget for solving the left subproblem. After the solution was found, the budget is decreased again by the cost of the

Algorithm 2 Accumulated-cost bounding

```
TDACB(G = (V, E), b)
 1  if memo[V] = NIL and b ≥ lowerBound[V]
 2    then lowerBound[V] ← b
 3         b′ ← b
 4         for (G_l, G_r) in PARTITION(G)
 5         do b_l ← b′ − JOINCOST(G_l, G_r)
 6            p_l ← TDACB(G_l, b_l)
 7            if p_l ≠ NIL
 8              then b_r ← b_l − C(p_l)
 9                   p_r ← TDACB(G_r, b_r)
10                   if p_r ≠ NIL
11                     then p ← MAKEPLAN(p_l, p_r)
12                          memo[V] ← p
13                          b′ ← C(p)
14  if b < C(memo[V])
15    then return NIL
16    else return memo[V]
```

Algorithm 3 Improved accumulated-cost and predicted-cost bounding

```
TDAPCBSORT(G = (V, E), b)
 1  if memo[V] = NIL and b ≥ lowerBound[V]
 2    then if visited[V] = NIL
 3           then visited[V] = T
 4                b′ ← b
 5           else b′ ← max(b, lowerBound[V] · 2)
 6         (partitions, newLB) ← SORTEDPARTITION(G, b′)
 7         for (G_l, G_r) in partitions
 8         do if b′ ≥ JOINLB(G_l, G_r)
 9              then b_l ← b′ − JOINCOST(G_l, G_r)
10                   p_l ← TDAPCBSORT(G_l, b_l)
11                   if p_l ≠ NIL
12                     then b_r ← b_l − C(p_l)
13                          p_r ← TDAPCBSORT(G_r, b_r)
14                          if p_r ≠ NIL
15                            then p ← MAKEPLAN(p_l, p_r)
16                                 memo[V] ← p
17                                 b′ ← C(p)
18                                 newLB ← C(p)
19            newLB ← min(newLB, JOINLB(G_l, G_r))
20         lowerBound[V] ← newLB
21  if b < C(memo[V])
22    then return NIL
23    else return memo[V]

JOINLB(G_l = (V_l, E_l), G_r = (V_r, E_r))
 1  jc ← JOINCOST(G_l, G_r)
 2  return jc + lowerBound[V_l] + lowerBound[V_r]

SORTEDPARTITION(G = (V, E), b)
 1  newLB ← ∞
 2  partitions ← ∅
 3  for (G_l, G_r) in PARTITION(G)
 4  do if b ≥ JOINLB(G_l, G_r)
 5       then partitions ← partitions ∪ {(G_l, G_r)}
 6       else newLB ← min(newLB, JOINLB(G_l, G_r))
 7  return (SORT(partitions) by JOINLB, newLB)
```

left subproblem and used as the budget for solving the right subproblem.

If at some point the budget becomes negative, failure is communicated by returning NIL up the call stack signaling that the solution is invalid and another solution must be found. If failure occurs i.e. the budget is lower than the cheapest cost for an expression, then this budget is stored as a lower bound for that expression. The next time the same expression is requested, failure can be signaled immediately if the new budget is lower than the stored previous lower bound. Accumulated-cost bounding can be combined with predicted-cost bounding by initializing the lower bounds for all expressions with predicted costs and only processing a possible solution if the current budget is higher than the lower bound (see line 8 of Algorithm 3).

DeHaan and Tompa observed that accumulated-cost bounding can lead to the same logical expression being processed multiple times. This happens when the budget for the expression is too low to find a result and the same expression is processed with a rising budget multiple times. Since without pruning DPCCP and MINCUTBRANCH with memoization optimize each expression exactly once, accumulated-cost bounding can reduce performance. Specifically, optimizing large queries with star graphs can be much slower with accumulated-cost bounding than without pruning. This pathological behavior also occurs when accumulated-cost bounding is combined with predicted-cost bounding. Therefore this form of pruning in its simple form can not be recommended.

D. Improvements

We developed a new algorithm called TDAPCBSORT shown in Algorithm 3 which is based on a combination of accumulated-cost and predicted-cost bounding. The algorithm solves the problematic behavior of accumulated-cost bounding which was discussed in the previous section and includes further modifications that improve pruning effectiveness.

a) Rising budget: An expression is reprocessed when the current budget is higher than the budget of the last unsuccessful attempt. Therefore, one possibility to reduce the number of times an expression is reprocessed is to artificially increase the available budget if the expression is revisited. Instead of using the budget that was passed from above we use at least the doubled budget which was used the last time (line 5). This causes the budget to grow exponentially and reduces the number of times an expression is reprocessed dramatically. Less plans are pruned, because sometimes the result for an expression will be calculated unnecessarily. Nevertheless,

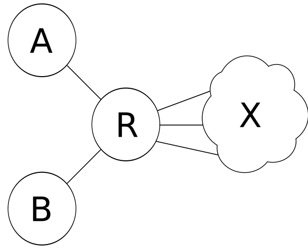


Fig. 3. In some cases the plans $\{A\} \bowtie \{B, R\} \cup X$ and $\{B\} \bowtie \{A, R\} \cup X$ can be compared without computing their costs

besides solving the pathological case this change proved to be beneficial in nearly all cases tested.

b) Sorting partitions: Since the cost of cheapest plan obtained so far is used as the budget for other join alternatives, the efficiency of pruning depends on the order in which the join alternatives are processed. A good choice in the beginning may avoid work later. It is therefore beneficial to sort all join alternatives by an estimate of their cost. The PLANBOUND function produces such an estimate, as it includes the cost of the top most join and lower bounds for the left and right sub-plans. Thus the algorithm processes the possible solutions in sorted order instead of the arbitrary order produced by MINCUTBRANCH.

c) Improved Lower Bounds: The $lowerBound$ structure initially contains a conservative estimate for the cost of each expression or 0 if no such estimate can be computed. Because lower bounds are used to prune the search space and to sort candidate solutions, it is very beneficial to increase them as high as possible. Each time an expression is processed, the lower bound can be improved. Either a solution was found for the available budget, then the lower bound is set to the cost of the cheapest solution (line 18). If no solution is available for the current budget, the lower bound is increased to the cheapest lower bound of all candidate solutions (line 19). Line 20 stores this improved bound in the $lowerBound$ structure.

E. Dominated Plans

Besides pruning based on upper and lower bounds top-down enumeration allows for another form of pruning. In some cases it is possible to show using certain necessary conditions that a solution has a greater or equal cost than another solution without computing their costs. Suppose we have a graph shown in Figure 3 with relations A and B which are both only connected to a relation R which may be connected to some arbitrary subgraph X. This pruning technique is only efficient for stars, as for most other query graphs this situation occurs very infrequently.

When computing the solution for the expression that includes all these relations ($\{A, B, R\} \cup X$), the top-down enumeration algorithm will besides others enumerate $\{A\} \bowtie (\{B\} \cup X)$ and $\{B\} \bowtie (\{A\} \cup X)$. The costs of these two expressions can be decomposed:

- $C(\{A\} \bowtie \{B, R\} \cup X) = \text{JOINCOST}(\{A\}, \{B, R\} \cup X) + C(\{A\}) + C(\{B, R\} \cup X)$
- $C(\{B\} \bowtie \{A, R\} \cup X) = \text{JOINCOST}(\{B\}, \{A, R\} \cup X) + C(\{B\}) + C(\{A, R\} \cup X)$

The cost of $\{B\} \bowtie (\{A\} \cup X)$ is greater or equal than the cost of $\{B\} \bowtie \{A, R\} \cup X$ and therefore can be dismissed if the following conditions hold:

$$\text{JOINCOST}(\{A\}, \{B, R\} \cup X) \leq \text{JOINCOST}(\{B\}, \{A, R\} \cup X) \quad (1)$$

$$C(\{A\}) \leq C(\{B\}) \quad (2)$$

$$C(\{B, R\} \cup X) \leq C(\{A, R\} \cup X) \quad (3)$$

The Equation 1 can be checked directly by evaluating the join cost function and comparing the two results. Equation 2 is trivially true, since the join cost of a single relation is 0. To show Equation 3, we observe that the graphs of $\{A, R\} \cup X$ and $\{B, R\} \cup X$ have the same structure. Only the cardinality and selectivity for A or B differ. If the join selectivities are independent, the cost function is monotonic, and the cost function depends on cardinalities only, then a higher cardinality and selectivity implies higher (or equal) cost. In fact, the selectivity may be less as long as the result cardinality of joining A to some expression is always higher than joining B to the same expression. Therefore, the third condition is necessarily true if:

$$|B| \leq |A| \wedge |B| \cdot s_{R,B} \leq |A| \cdot s_{R,A} \quad (4)$$

Equation 4 can be precomputed once for all pairs of relations before enumerating join orders. For each relation this results is a set of relations that potentially dominate it. While enumerating join orders only Equation 1 needs to be checked.

Besides cardinalities, realistic join cost functions depend on other variables like the tuple size or the form of the join predicate. In this case Equation 4 is not sufficient. It can be extended by other dimensions like tuple size if the cost function is monotonic in it. If it is not possible to augment Equation 4 this form of pruning must be regarded as a heuristics.

III. EXPERIMENTAL SETUP

A. Queries

In contrast to algorithms that search the space exhaustively, the effectiveness of pruning depends on the selectivities and cardinalities of the query. Therefore, in order to evaluate pruning strategies, ”realistic” queries would be desirable. Unfortunately, the literature contains no empirical study of real-world queries. We must therefore resort to artificial but hopefully plausible queries.

As join graphs we use: chains, cycles, stars, grids, cliques, random acylic graphs (trees), and random cyclic graphs. For assigning the cardinalities and selectivities we follow the approach of [6] which is based on [7]. There are two classes of queries: random join queries and foreign-key queries. Random join queries get their cardinalities and domain sizes for attributes from the distributions in Figure 4. The selectivities are computed by choosing two random attributes and using $\frac{1}{\max(dom(A_1), dom(A_2))}$ as the selectivity. Neumann observed

relation size	prob.
10-100	15%
100-1,000	30%
1,000-10,000	25%
10,000-100,000	20%

domain size	prob.
2-10	5%
10-100	50%
100-500	35%
500-1,000	15%

Fig. 4. Relation and domain sizes for random join queries as proposed by Steinbrunn et al.

	Star			Fk-Star		
i	$\lvert R_i\rvert$	$s_{1,i}$	$\lvert R_i\rvert \cdot s_{1,i}$	$\lvert R_i\rvert$	$s_{1,i}$	$\lvert R_i\rvert \cdot s_{1,i}$
1	74209			74209000		
2	72719	6.44e-03	468.20	72719000	1.35e-08	0.98
3	140	4.96e-03	0.69	140000	7.14e-06	1.00
4	779	1.34e-03	1.04	779000	1.27e-06	0.99
5	6382	6.22e-03	39.70	6382000	1.57e-07	1.00

TABLE II

COMPARISON OF A RANDOM AND A FOREIGN KEY QUERY WITH A STAR JOIN GRAPH AND ROOT R_1

that this often leads to intermediate cardinalities less than 1 which are successively increased to become huge again. As this does not seem to be realistic he proposes foreign-key join queries.

Foreign-key join queries are based on key/foreign joins, which are very common in practice. For relation and domain sizes the same distributions are used as before, but the sizes are multiplied by 1000 to get larger data sets. With 10% probability the selectivity of a join edge is computed as described above. With 90% probability the selectivity is computed such that the cardinality of the result is equal to the cardinality of the relation with the foreign key. Additionally a Zipf-distributed part is removed. Table II shows an example for the two ways of assigning selectivities.

B. Cost Functions

To investigate the effect of the cost function on pruning effectiveness, we compare three different cost functions. The first function C_{out} computes the sum of all intermediate cardinalities. It is commonly used in the literature and has the plausible goal of minimizing intermediate cardinalities. The second cost function C_{sm} estimates the asymptotic cost of sorting both input relations to perform a sort-merge join: $C_{sm}(X, Y) = |X| \log(|X|) + |Y| \log(|Y|) + C(X) + C(Y)$. The most complex cost function C_{gh} estimates the IO cost of a grace hash join [8]. It distinguishes between seeks and sequential disk accesses and must be configured with actual disk parameters.

Of these, C_{gh} is computationally the most expensive function. For example, finding the solution for a star query with 18 relations using the DPCCP algorithm involves 1,114,112 invocations of the cost function. For the C_{gh} cost function the total optimization time is 519ms, for C_{out} which involves no computation the optimization time is 320ms. This corresponds to around 500 clock cycles per invocation of C_{gh}.

For predicted-cost bounding conservative lower bounds are necessary. For all cost function the lower bound is 0 if the expression consists only of a single relation. For C_{out} the lower bound of an expression is the cardinality of that expression. For C_{sm} the lower bound is $\sum_{R \in X} |R| \log |R|$ where X is the set of all relations in the expression. For C_{gh} we use the cost of sequentially reading each input relation from disk.

C. General Setup

We implemented all algorithms using the same infrastructure for representing plans, accessing the hash table which contains the plans, and computing the cardinalities. Because DPCCP and MINCUTBRANCH use set operations very heavily, we used machine words to represent sets. This allows set operations to be performed very efficiently in constant time using bit operations. This decision limits the maximum query size to the number of bits in a machine word. All programs were compiled using GCC 4.4.5 running Linux in 64 bit mode. The measurements were done on an AMD Athlon II X4 640 Processor with 3.0 GHz and 2GB DDR3 RAM.

For each query type we generated 1000 random queries. As the baseline algorithm for all comparisons we use DPCCP, which is the fastest known dynamic programming algorithm. The runtimes of the algorithms are scaled by the runtimes of DPCCP, i.e. a value of 0.1 means that the algorithm is 10 times faster than DPCCP. Additionally we report the number of cost function evaluations scaled by the respective value of DPCCP, as we found it to be a good machine independent measure of pruning effectiveness. All top-down algorithms use MINCUTBRANCH to enumerate join orders. We compare the following algorithms:

- DPCCP: bottom-up enumeration without pruning
- DPTCB: bottom-up enumeration with total-cost bounding
- TDMCB: top-down enumeration without pruning
- TDPCB: top-down enumeration with predicted-cost bounding (Algorithm 1)
- TDAPCBSORT: top-down enumeration with improved predicted-cost and accumulated-cost bounding (Algorithm 3)

IV. EVALUATION

A. No pruning

Table III shows the scaled execution time of TDMCB in comparison with DPCCP when the C_{out} cost function is used. For all query graphs DPCCP is faster, although the difference is bigger for graphs with cycles. For C_{gh} the difference is smaller, ranging from 1.03 to 1.22, since relative to C_{out} more time is spent computing the cost function as opposed to enumerating join orders. These results show the head start of DPCCP that must be overcome by TDMCB.

B. Total-Cost Bounding

To find out if total-cost bounding is beneficial for plans without cartesian products, we implemented the Greedy Operator Ordering (GOO) heuristics [9] and used its result as the upper bound. As can be seen in Table IV, total-cost bounding improves the running time for most query graphs. Only the

graph/size	DPccp	TDmcb	cost fn. eval.
chain 60	10ms	1.04	35,990
cycle 60	29ms	1.03	104,430
tree 24	50ms	1.10	187,870
star 16	73ms	1.14	245,760
grid 20	138ms	1.23	462,582
cyclic 16	146ms	1.36	481,974
clique 12	58ms	1.28	261,625

TABLE III

ABSOLUTE PERFORMANCE OF DPCCP (WITH THE C_{out} COST FUNCTION, MEANS FOR RANDOM GRAPHS), SCALED TIME OF TDMCB IN COMPARISON WITH DPCCP, AND THE NUMBER OF TIMES THE COST FUNCTION IS EVALUATED FOR THE GIVEN GRAPH

	DPtcb			
	time		cost fn. eval.	
graph/size	GOO	optimal	GOO	optimal
chain 60	1.05	0.95	0.79	0.79
fk-chain 60	0.96	0.82	0.30	0.29
cycle 60	0.93	0.89	0.55	0.54
fk-cycle 60	0.85	0.81	0.22	0.17
tree 24	0.98	0.97	0.86	0.86
fk-tree 24	0.83	0.82	0.48	0.47
star 16	0.99	0.99	0.88	0.88
fk-star 16	0.93	0.93	0.81	0.81
grid 20	0.66	0.66	0.10	0.10
fk-grid 20	0.59	0.58	0.02	0.02
cyclic 16	0.67	0.66	0.09	0.09
fk-cyclic 16	0.60	0.58	0.02	0.02
clique 12	0.71	0.70	0.04	0.03
fk-clique 12	0.75	0.70	0.16	0.06

TABLE IV

AVERAGE SCALED EXECUTION TIME AND SCALED NUMBER OF COST FUNCTION EVALUATIONS OF DPTCB RELATIVE TO DPCCP USING THE C_{out} COST FUNCTION

	TDpcb					
	time			cost fn. eval.		
graph/size	C_{out}	C_{sm}	C_{gh}	C_{out}	C_{sm}	C_{gh}
chain 60	0.84	0.98	0.93	0.45	0.43	0.55
fk-chain 60	0.98	1.13	1.01	0.59	0.59	0.65
cycle 60	0.53	0.68	0.58	0.25	0.24	0.28
fk-cycle 60	0.60	0.75	0.60	0.32	0.32	0.33
tree 24	0.15	0.18	0.32	0.09	0.08	0.21
fk-tree 24	0.57	0.69	0.64	0.40	0.39	0.44
star 16	0.10	0.12	0.16	0.06	0.06	0.10
fk-star 16	0.93	1.06	0.94	0.71	0.70	0.71
grid 20	0.05	0.07	0.05	0.02	0.02	0.02
fk-grid 20	0.07	0.09	0.07	0.03	0.03	0.03
cyclic 16	0.12	0.15	0.11	0.05	0.05	0.05
fk-cyclic 16	0.18	0.21	0.16	0.08	0.08	0.08
clique 12	0.20	0.22	0.17	0.10	0.10	0.10
fk-clique 12	0.57	0.60	0.51	0.40	0.40	0.40

TABLE V

AVERAGE SCALED EXECUTION TIME AND SCALED NUMBER OF COST FUNCTON EVALUATIONS OF TDPCB RELATIVE TO DPCCP

results for C_{out} are shown, as the other cost functions perform very similar. Total-cost bounding is clearly an improvement, in particular for complex graphs with many cycles. For chains we see a slow down, because our implementation of GOO takes $O(n^3)$ time, just as DPccp for this query graph. Thus, for chains the execution time of the heuristics is larger than the speedup from pruning. This is not the case if the more expensive C_{gh} cost function is used.

For most query graphs, queries with random selectivities benefit less from total-cost bounding than queries with foreign-key selectivities. This is because in random queries the cardinalities of intermediate results differ strongly, so the total cost is typically dominated by a few joins, while most other joins have a cost which is orders of magnitude lower. This means that the total cost bound is not very tight for many subproblems.

There is a tradeoff between the pruning power which is determined by the quality of the bound and the speed of the heuristics. To find out what the ideal pruning power of total-cost bounding is, we used the optimal cost as the upper bound but didn't include the time of obtaining it in the measurement. Therefore, the runtimes shown for the optimal variant in Table IV cannot be obtained in practice, but show the limits of this pruning technique. The number of cost function evaluations for both total-cost bounding variants differ in most cases by a few percent only, indicating that the upper bound of GOO is good enough for its purpose. The difference in runtime is therefore mostly caused be the time spent running the heuristics.

C. Top-down Enumeration with Pruning

Table V shows that predicted-cost bounding is more effective than total-cost bounding. In contrast to total-cost bounding, with predicted-cost bounding foreign-key queries perform worse than random queries, because the variance of the costs of possible solutions for an expression is lower (in other words: many plans have a similar cost) and therefore it takes longer to dismiss possible solutions. Obviously, the effectiveness of predicted-cost bounding depends on the quality the lower bound estimates for the specific cost function in use. Nevertheless, even though C_{out}, C_{sm}, and C_{gh} are quite different functions and therefore have very different predicted lower bounds, their predicted-cost bounding results are surprisingly similar.

Tables VI, VII, and Figure 6 show the results for TDapcbSort. TDapcbSort on average clearly outperforms all other tested algorithms. As with predicted-cost bounding, pruning is less effective for foreign-key queries. The worst result was obtained for foreign-key stars, where in the worst case a slowdown of 43% over DPccp was observed. This happens when the particular query allows no pruning and the overhead (e.g. for sorting the partitions) of the pruning algorithm occurs anyway. To investigate worst case behaviour further we created queries with cardinalities and selectivities all set to 1. All plans of these queries have the same cost, thus no pruning was possible. As expected TDapcbSort was slower than DPccp for all query graphs, with a maximal slowdown of 2.

As another improvement we modified TDapcbSort to dismiss possible solutions if Equations 1 and 4 from Section II E hold. The speedup was between 60% for foreign-key stars and 100% for random stars. For all other query graphs very

	TDapcbSort (time)																	
	C_{out}						C_{sm}						C_{gh}					
graph/size	mean	min.	5%	median	95%	max.	mean	min.	5%	median	95%	max	mean	min	5%	median	95%	max.
chain 60	0.52	0.42	0.44	0.48	0.69	0.96	0.67	0.58	0.59	0.64	0.84	1.09	0.71	0.42	0.46	0.64	1.16	1.49
fk-chain 60	0.57	0.42	0.45	0.53	0.80	1.08	0.73	0.58	0.60	0.70	0.99	1.35	0.66	0.37	0.42	0.60	1.07	1.36
cycle 60	0.34	0.32	0.32	0.34	0.39	0.53	0.50	0.48	0.48	0.49	0.54	0.63	0.40	0.34	0.35	0.38	0.53	0.73
fk-cycle 60	0.35	0.32	0.33	0.34	0.41	0.55	0.51	0.48	0.48	0.50	0.57	0.68	0.35	0.30	0.31	0.34	0.46	0.57
tree 24	0.03	0.00	0.01	0.03	0.09	0.25	0.05	0.00	0.01	0.04	0.13	0.36	0.12	0.00	0.01	0.08	0.35	0.84
fk-tree 24	0.21	0.00	0.02	0.17	0.51	1.65	0.28	0.01	0.03	0.24	0.68	1.67	0.30	0.00	0.03	0.26	0.68	1.58
star 16	0.04	0.01	0.01	0.04	0.08	0.18	0.05	0.01	0.02	0.05	0.10	0.23	0.07	0.01	0.02	0.06	0.14	0.34
fk-star 16	0.67	0.03	0.11	0.53	1.18	1.29	0.78	0.04	0.13	0.62	1.37	1.43	0.71	0.03	0.12	0.57	1.24	1.38
grid 20	0.01	0.01	0.01	0.01	0.02	0.04	0.01	0.01	0.01	0.01	0.03	0.05	0.01	0.01	0.01	0.01	0.03	0.08
fk-grid 20	0.02	0.01	0.01	0.01	0.03	0.11	0.02	0.01	0.01	0.02	0.04	0.13	0.01	0.01	0.01	0.01	0.03	0.11
cyclic 16	0.06	0.02	0.04	0.06	0.10	0.23	0.08	0.03	0.05	0.07	0.12	0.27	0.06	0.02	0.03	0.05	0.10	0.29
fk-cyclic 16	0.10	0.03	0.04	0.09	0.20	0.34	0.13	0.04	0.06	0.11	0.24	0.40	0.09	0.02	0.04	0.08	0.18	0.31
clique 12	0.13	0.10	0.11	0.13	0.16	0.20	0.16	0.13	0.14	0.15	0.19	0.23	0.10	0.07	0.08	0.10	0.13	0.17
fk-clique 12	0.29	0.20	0.25	0.29	0.34	0.40	0.33	0.23	0.28	0.33	0.39	0.44	0.27	0.17	0.22	0.27	0.32	0.38

TABLE VII

Scaled execution time of TDapcbSort

	TDapcbSort					
	time			cost fn. eval.		
graph/size	C_{out}	C_{sm}	C_{gh}	C_{out}	C_{sm}	C_{gh}
chain 60	0.52	0.67	0.71	0.127	0.113	0.335
fk-chain 60	0.57	0.73	0.66	0.177	0.180	0.306
cycle 60	0.34	0.50	0.40	0.047	0.044	0.095
fk-cycle 60	0.35	0.51	0.35	0.056	0.055	0.079
tree 24	0.03	0.05	0.12	0.014	0.013	0.079
fk-tree 24	0.21	0.28	0.30	0.149	0.147	0.200
star 16	0.04	0.05	0.07	0.024	0.024	0.043
fk-star 16	0.67	0.78	0.71	0.566	0.561	0.566
grid 20	0.01	0.01	0.01	0.002	0.002	0.003
fk-grid 20	0.02	0.02	0.01	0.003	0.003	0.003
cyclic 16	0.06	0.08	0.06	0.013	0.012	0.016
fk-cyclic 16	0.10	0.13	0.09	0.033	0.033	0.033
clique 12	0.13	0.16	0.10	0.038	0.037	0.038
fk-clique 12	0.29	0.33	0.27	0.198	0.197	0.201

TABLE VI

Average scaled execution time and scaled number of cost functon evaluations of TDapcbSort relative to DPccp

few subproblems have the necessary graph structure so that a small slowdown because of the added overhead was observed.

D. Discussion

We have seen that predicted-cost bounding alone achieves a significant speedup over dynamic programming and is faster than total-cost bounding. When predicted-cost bounding is combined with accumulated-cost bounding using the TDapcbSort algorithm, query optimization time can be reduced even more. In a separate experiment we evaluated TDapcbSort with predicted cost of 0 which yielded very variable results which differed very strongly depending on the cost function and the query graph. Predicted-cost bounds are therefore necessary and the interaction between predicted-cost and accumulated-cost bounding is very beneficial. Figure 5 shows the growth of the average absolute runtime for cycle, star, and grid queries using different algorithms and the C_{out} cost function. For all query graphs pruning becomes more effective as the size of the query grows.

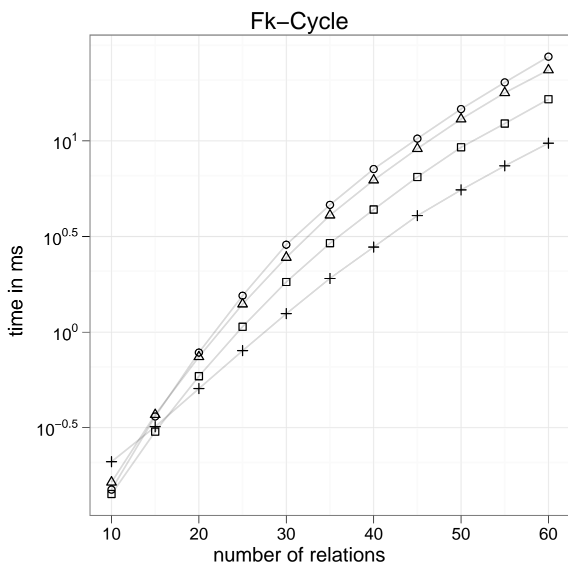
The effect of the cost function on pruning was fairly small, as long as the function allows one to determine some reasonable good predicted lower bound for an expression. Pruning effectiveness strongly depends on the query graph. Query graphs with many cycles like grids benefit from pruning very much, while for simple acyclic graphs like chains pruning is less effective. Pruning effectiveness additionally depends on the selectivities of the query. If the different join alternatives produce large differences in the cardinalities of intermediate results, then pruning occurs early and is very effective. If the opposite is the case, then the different join alternatives have very similar costs which means that most of the search space must be searched. This leads to the interesting observation that pruning is least effective where it matters least, because many different plans have almost the same cost anyway.

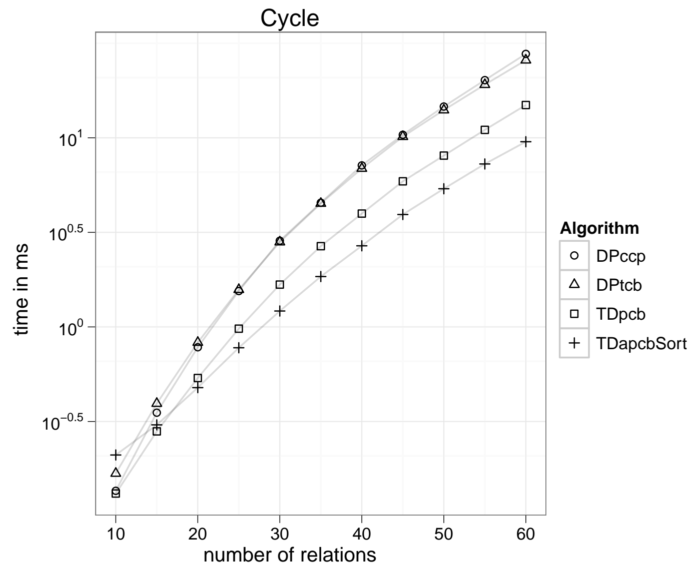
V. Conclusion

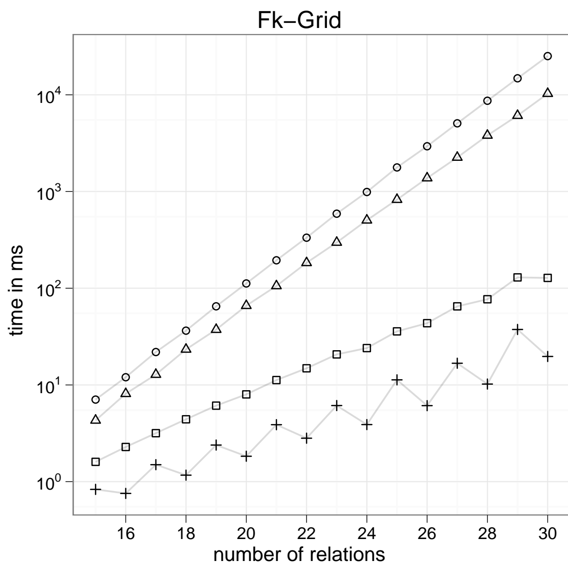
We have shown that top-down enumeration with MinCutBranch using pruning can achieve a speedup over dynamic programming by a factor of 2 to 10 for many query graphs and even up to 100 for grid queries if predicted-cost and accumulated-cost bounding is combined. These results were observed for different cost functions. An open question which we did not investigate is the effect of interesting orders on pruning effectiveness. Total-cost bounding for dynamic programming improves execution time by at most a factor of 2 and is therefore not competitive with top-down pruning.

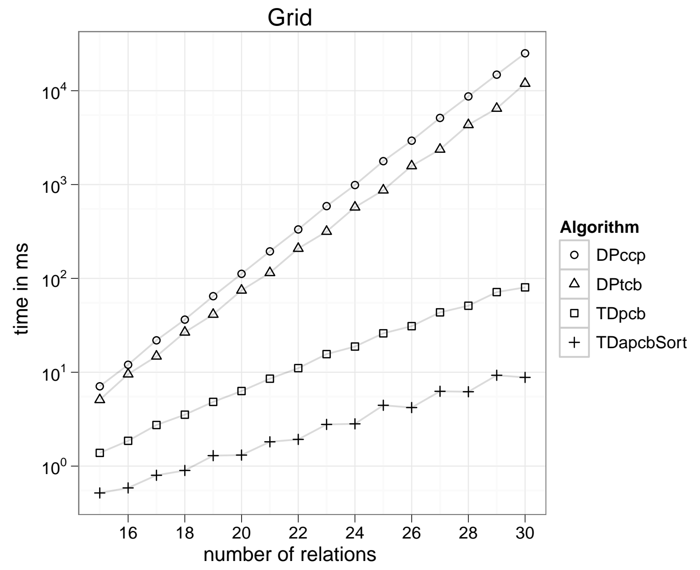
Pruning is particularly effective for complex query graphs with a high degree of cyclicity and for queries with a high variance of intermediate cardinalities. Unfortunately, most real world queries do not satisfy these properties, as they typically use foreign-key joins and have few cycles. Therefore, it must be considered an engineering tradeoff if the additional effort of implementing our non-trivial pruning algorithm on top of an already quite complex top-down enumeration algorithm is worthwhile.

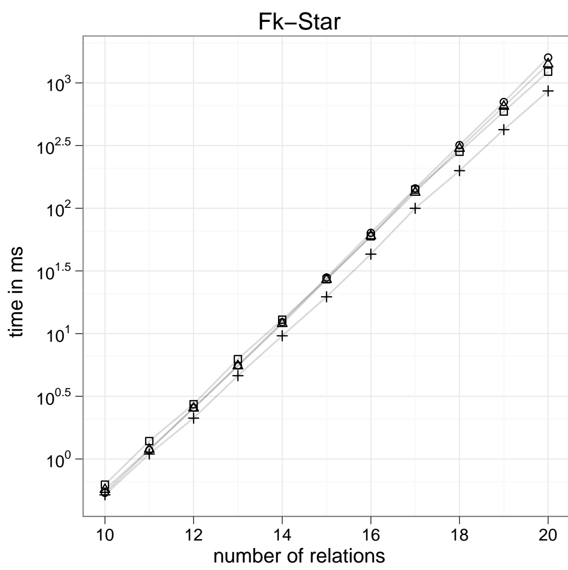
Additionally, it might be impossible or computationally too expensive to compute predicted lower bounds for complex real-world cost functions. But these lower bounds are nec-











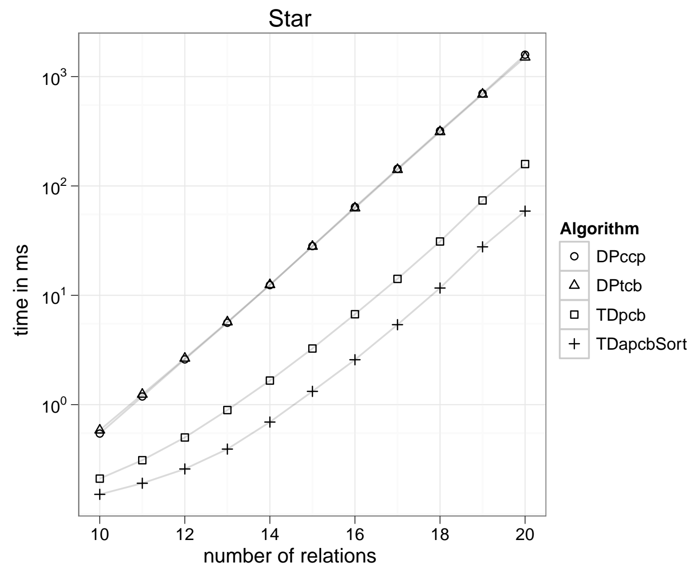


Fig. 5. Average optimization time

essary, because accumulated-cost bounding alone is not very effective. Another disadvantage of pruning is that optimization time varies quite strongly for the same query graph depending on the cardinalities and selectivities. Furthermore, in contrast to DPCCP outer joins are currently not supported by any optimal top-down enumeration algorithm.

References

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database management system,” in *SIGMOD Conference*, 1979, pp. 23–34.

[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 Conference*, 2007, pp. 785–796.

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

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

[6] T. Neumann, “Query simplification: Graceful degradation for join-order optimization,” in *SIGMOD Conference*, 2009, pp. 403–414.

[7] M. Steinbrunn, G. Moerkotte, and A. Kemper, “Heuristic and randomized optimization for the join ordering problem,” *VLDB J.*, vol. 6(3), pp. 191–208, 1997.

[8] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla, “Seeking the truth about ad hoc join costs,” *VLDB J.*, vol. 6(3), pp. 241–256, 1997.

[9] L. Fegaras, “A new heuristic for optimizing large queries,” in *DEXA*, 1998, pp. 726–735.

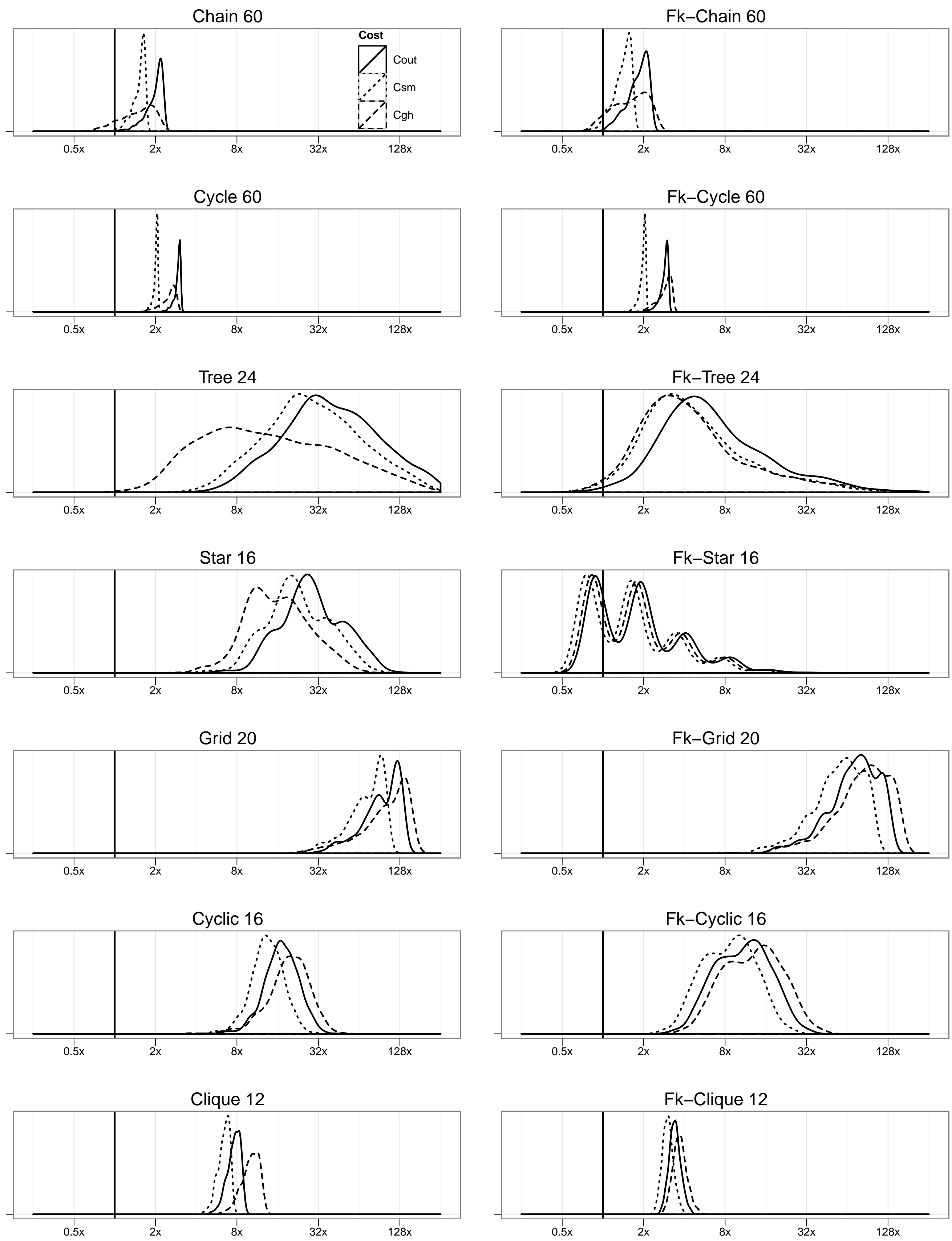


Fig. 6. Smoothed density estimate for the speedup factor of TDAPCBSORT relative to DPCCP