

# Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products

Guido Moerkotte  
University of Mannheim  
68131 Mannheim  
Germany

moerkotte@informatik.uni-mannheim.de

Thomas Neumann  
Max Planck Institute for Informatics  
66123 Saarbrücken  
Germany

neumann@mpi-inf.mpg.de

## ABSTRACT

Two approaches to derive dynamic programming algorithms for constructing join trees are described in the literature. We show analytically and experimentally that these two variants exhibit vastly diverging runtime behaviors for different query graphs. More specifically, each variant is superior to the other for one kind of query graph (chain or clique), but fails for the other. Moreover, neither of them handles star queries well. This motivates us to derive an algorithm that is superior to the two existing algorithms because it adapts to the search space implied by the query graph.

## 1. INTRODUCTION

For the overall performance of a database management system, the cost-based query optimizer is an essential piece of software. One important and complex problem any cost-based query optimizer has to solve is that of finding the optimal join order. In their seminal paper, Selinger et al. not only introduced cost-based query optimization but also proposed a dynamic programming algorithm to find the optimal join order for a given conjunctive query [7]. More precisely, they proposed to generate plans in the order of increasing size. Although they restricted the search space to left-deep trees, the general idea of their algorithm can be used to derive an algorithm to explore the space of bushy trees. Its pseudocode is shown in Fig. 1. The algorithm still forms the core of state of the art commercial query optimizers like the one of DB2 [2] and still is the foundation for further research on join ordering, e.g. in the context of distributed database management systems [3].

Given the widespread and prominent use of dynamic programming algorithms for finding a good join order, it came to us as a surprise that only two publications analyze the complexity of these algorithms. The first publication is the seminal paper by Ono and Lohman [5]. In order to under-

stand their results, it is important to note that the algorithm (see Fig. 1) follows a generate-and-test approach. Ono and Lohman analyse the number of times the tests succeed (see Fig. 1). Obviously, this number is highly dependent on the query graph. Therefore, Ono and Lohman consider chain, star, and clique queries. As pointed out by Vance, this only gives a lower bound for all dynamic programming algorithms [9]. The real complexity is the number of times the code within the inner loop (the test of the first `if`-statement in the innermost loop) is executed. Moreover, Vance gives an analytical result for clique queries for an unoptimized version of the size-based variant described above [9].

Vance and Maier proposed an algorithm which generates subsets extremely fast [9, 10]. They use this routine to generate optimal bushy join trees containing cross products. The core idea is to generate partial plans in a different order than the algorithm mentioned above. If cross products are to be considered, their algorithm can hardly be improved. However, as generating cross products vastly increases the search space [5], it is a very interesting exercise to modify their algorithm such that it excludes cross products. Modifying it such that it also follows the generate-and-test paradigm results in the algorithm given in Fig. 2. For this algorithm, no complexity bounds are known.

Our first contribution is that we analytically analyze the time complexity of both algorithms for chain, cycle, star, and clique queries (Sec. 2). This allows us to analytically compare their performance on these query graph instances. Let us call the first algorithm `DPsize` and the second `DPsub`. Then the comparison of these algorithms reveals (see also Sec. 2):

1. For chain and cycle queries, `DPsize` is highly superior to `DPsub`.
2. For star and clique queries, `DPsub` is highly superior to `DPsize`.
3. Both algorithms are far worse than the lower bound given by Ono and Lohman.

These findings immediately spawn the following question: Is it possible to derive a dynamic programming algorithm whose complexity meets the lower bound derived by Ono

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

```

DPsize
Input: a connected query graph with relations  $R = \{R_0, \dots, R_{n-1}\}$ 
Output: an optimal bushy join tree without cross products

for all  $R_i \in R$  {
  BestPlan( $\{R_i\}$ ) =  $R_i$ ;
}
for all  $1 < s \leq n$  ascending // size of plan
for all  $1 \leq s_1 < s$  { // size of left subplan
   $s_2 = s - s_1$ ; // size of right subplan
  for all  $S_1 \subset R : |S_1| = s_1$ 
     $S_2 \subset R : |S_2| = s_2$  {
      ++InnerCounter;
      if ( $\emptyset \neq S_1 \cap S_2$ ) continue;
      if not ( $S_1$  connected to  $S_2$ ) continue;
      ++CsgCmpPairCounter;
       $p_1 = \text{BestPlan}(S_1)$ ;
       $p_2 = \text{BestPlan}(S_2)$ ;
      CurrPlan = CreateJoinTree( $p_1, p_2$ );
      if ( $\text{cost}(\text{BestPlan}(S_1 \cup S_2)) > \text{cost}(\text{CurrPlan})$ ) {
        BestPlan( $S_1 \cup S_2$ ) = CurrPlan;
      }
    }
  }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan( $\{R_0, \dots, R_{n-1}\}$ );

```

**Figure 1: Algorithm DPsize**

and Lohman? If so, this variant would clearly outperform the other variants. The positive answer to this question is presented in Section 3.

Next, we were interested in the practical implication of our theoretical analysis. Hence, we implemented all three algorithms in a plan generator and ran experiments. Some typical results are presented in Section 4. Section 5 concludes the paper.

## 2. ALGORITHMS AND ANALYSIS

In this section, we present two dynamic programming algorithms to generate optimal bushy trees without cross products. The first two subsections discuss the pseudocode of these algorithms. We start by discussing the common infrastructure used by all our algorithms (including our new one). Section 2.1 sketches the most prominent size-chained variant. Section 2.2 presents two variants based on fast subset generation [9, 10]. Both subsections give analytical results on the time complexity of the algorithm they present. The analytical results independent of any algorithm are presented in Section 2.3.2 after introducing some definitions in Subsection 2.3.1. Section 2.4 applies the formulas presented so far and draws important conclusions.

### 2.1 Size-Driven Enumeration

In general, dynamic programming generates solutions for a larger problem in a bottom-up fashion by combining solutions for smaller problems [1]. Taking this description literally, we can construct optimal plans of size  $n$  by joining plans  $P_1$  and  $P_2$  of size  $k$  and  $n - k$ . We just have to take care that (1) the sets of relations contained in  $P_1$  and  $P_2$

do not overlap, and (2) there is a join predicate connecting a relation  $P_1$  with a relation in  $P_2$ . After this remark, we are prepared to understand the pseudocode for algorithm DPsize (see Fig. 1). A table BestPlan associates with each set of relations the best plan found so far. The algorithm starts by initializing this table with plans of size one, i.e. single relations. After that, it constructs plans of increasing size (loop over  $s$ ). Thereby, the first size considered is two, since plans of size one have already been constructed. Every plan joining  $n$  relations can be constructed by joining a plan containing  $s_1$  relations with a plan containing  $s_2$  relations. Thereby,  $s_i > 0$  and  $s_1 + s_2 = n$  must hold. Thus, the pseudocode loops over  $s_1$  and sets  $s_2$  accordingly. Since for every possible size there exist many plans, two more loops are necessary in order to loop over the plans of sizes  $s_1$  and  $s_2$ . Then, conditions (1) and (2) from above are tested. Only if their outcome is positive, we consider joining the plans  $p_1$  and  $p_2$ . The result is a plan CurrPlan. Let  $S$  be the relations contained in CurrPlan. If BestPlan does not contain a plan for the relations in  $S$  or the one it contains is more expensive than CurrPlan, we register CurrPlan with BestPlan.

The algorithm DPsize can be made more efficient in case of  $s_1 = s_2$ . The algorithm as stated cycles through all plans  $p_1$  joining  $s_1$  relations. For each such plan, all plans  $p_2$  of size  $s_2$  are tested. Assume that plans of equal size are represented as a linked list. If  $s_1 = s_2$ , then it is possible to iterate through the list for retrieving all plans  $p_1$ . For  $p_2$  we consider the plans succeeding  $p_1$  in the list. Thus, the complexity can be decreased from  $s_1 * s_2$  to  $s_1 * s_2 / 2$ . The following formulas are valid only for the variant of DPsize where this optimization has been incorporated (see [4] for details).

We now come to the first important contribution of our paper. If the counter InnerCounter is initialized with zero at the beginning of the algorithm DPsize, then we are able to derive analytically its value after DPsize terminates. Since this value of the inner counter depends on the query graph, we have to distinguish several cases. For chain, cycle, star, and clique queries, we denote by  $I_{\text{DPsize}}^{\text{chain}}$ ,  $I_{\text{DPsize}}^{\text{cycle}}$ ,  $I_{\text{DPsize}}^{\text{star}}$ , and  $I_{\text{DPsize}}^{\text{clique}}$  the value of InnerCounter after termination of algorithm DPsize.

For chain queries, we then have:  $I_{\text{DPsize}}^{\text{chain}}(n) =$

$$\begin{cases} 1/48(5n^4 + 6n^3 - 14n^2 - 12n) & n \text{ even} \\ 1/48(5n^4 + 6n^3 - 14n^2 - 6n + 11) & n \text{ odd} \end{cases}$$

For cycle queries, we have:  $I_{\text{DPsize}}^{\text{cycle}}(n) =$

$$\begin{cases} \frac{1}{4}(n^4 - n^3 - n^2) & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n) & n \text{ odd} \end{cases}$$

For star queries, we have:  $I_{\text{DPsize}}^{\text{star}}(n) =$

$$\begin{cases} 2^{2n-4} - 1/4 \binom{2(n-1)}{n-1} + q(n) & n \text{ even} \\ 2^{2n-4} - 1/4 \binom{2(n-1)}{n-1} + 1/4 \binom{n-1}{(n-1)/2} + q(n) & n \text{ odd} \end{cases}$$

with  $q(n) = n2^{n-1} - 5 * 2^{n-3} + 1/2(n^2 - 5n + 4)$ . For clique queries, we have:  $I_{\text{DPsize}}^{\text{clique}}(n) =$

$$\begin{cases} 2^{2n-2} - 5 * 2^{n-2} + 1/4 \binom{2n}{n} - 1/4 \binom{n}{n/2} + 1 & n \text{ even} \\ 2^{2n-2} - 5 * 2^{n-2} + 1/4 \binom{2n}{n} + 1 & n \text{ odd} \end{cases}$$

```

DPsub
Input: a connected query graph with relations  $R = \{R_0, \dots, R_{n-1}\}$ 
Output: an optimal bushy join tree
for all  $R_i \in R$  {
    BestPlan( $\{R_i\}$ ) =  $R_i$ ;
}
for  $1 \leq i < 2^n - 1$  ascending {
     $S = \{R_j \in R \mid (i/2^j) \bmod 2 = 1\}$ 
    if not (connected  $S$ ) continue; // *
    for all  $S_1 \subset S, S_1 \neq \emptyset$  do {
        ++InnerCounter;
         $S_2 = S \setminus S_1$ ;
        if ( $S_2 = \emptyset$ ) continue;
        if not (connected  $S_1$ ) continue;
        if not (connected  $S_2$ ) continue;
        if not ( $S_1$  connected to  $S_2$ ) continue;
        ++CsgCmpPairCounter;
         $p_1 = \text{BestPlan}(S_1)$ ;
         $p_2 = \text{BestPlan}(S_2)$ ;
        CurrPlan = CreateJoinTree( $p_1, p_2$ );
        if ( $\text{cost}(\text{BestPlan}(S)) > \text{cost}(\text{CurrPlan})$ ) {
            BestPlan( $S$ ) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan( $\{R_0, \dots, R_{n-1}\}$ );

```

**Figure 2: Algorithm DPsub**

Note that  $\binom{2^n}{n}$  is in the order of  $\Theta(4^n/\sqrt{n})$ .

Proofs of the above formulas as well as implementation details for the algorithm DPsize can be found in [4].

## 2.2 Subset-Driven Enumeration

Fig. 2 presents the pseudocode for the algorithm DPsub. The algorithm first initializes the table BestPlan with all possible plans containing a single relation. Then, the main loop starts. It iterates over all possible non-empty subsets of  $\{R_0, \dots, R_{n-1}\}$  and constructs the best possible plan for each of them. The enumeration makes use of a bitvector representation of sets: The integer  $i$  induces the current subset  $S$  with its binary representation. Taken as bitvectors, the integers in the range from 1 to  $2^n - 1$  exactly represent the set of all non-empty subsets of  $\{R_0, \dots, R_{n-1}\}$ , including the set itself. Further, by starting with 1 and incrementing by 1, the enumeration order is valid for dynamic programming: for every subset, all its subsets are generated before the subset itself.

This enumeration is very fast, since increment by one is a very fast operation. However, the relations contained in  $S$  may not induce a connected subgraph of the query graph. Therefore, we must test for connectedness. The goal of the next loop over all subsets of  $S$  is to find the best plan joining all the relations in  $S$ . Therefore,  $S_1$  ranges over all non-empty, strict subsets of  $S$ . This can be done very efficiently by applying the code snippet of Vance and Maier [9, 10]. Then, the subset of relations contained in  $S$  but not in  $S_1$  is assigned to  $S_2$ . Clearly,  $S_1$  and  $S_2$  are disjoint. Hence, only

connectedness tests have to be performed. Since we want to avoid cross products,  $S_1$  and  $S_2$  both must induce connected subgraphs of the query graph, and there must be a join predicate between a relation in  $S_1$  and one in  $S_2$ . If these conditions are fulfilled, we can construct a plan CurrPlan by joining the plans associated with  $S_1$  and  $S_2$ . If BestPlan does not contain a plan for the relations in  $S$  or the one it contains is more expensive than CurrPlan, we register CurrPlan with BestPlan.

For chain, cycle, star, and clique queries, we denote by  $I_{\text{DPsub}}^{\text{chain}}$ ,  $I_{\text{DPsub}}^{\text{cycle}}$ ,  $I_{\text{DPsub}}^{\text{star}}$ , and  $I_{\text{DPsub}}^{\text{clique}}$  the value of InnerCounter after termination of algorithm DPsub.

For chains, we have

$$I_{\text{DPsub}}^{\text{chain}}(n) = 2^{n+2} - n^n - 3n - 4 \quad (1)$$

For cycles, we have

$$I_{\text{DPsub}}^{\text{cycle}}(n) = n2^n + 2^n - 2n^2 - 2 \quad (2)$$

For stars, we have

$$I_{\text{DPsub}}^{\text{star}}(n) = 2 * 3^{n-1} - 2^n \quad (3)$$

For cliques, we have

$$I_{\text{DPsub}}^{\text{clique}}(n) = 3^n - 2^{n+1} + 1 \quad (4)$$

The number of failures for the additional check can easily be calculated as  $2^n - \#\text{csg}(n) - 1$ , where  $\#\text{csg}(n)$  denotes the number of non-empty connected subgraphs contained in the query graph.

## 2.3 Algorithm-Independent Results

### 2.3.1 Definition of #csg and #ccp

Consider a join ordering problem with  $n$  relations  $R_0, \dots, R_{n-1}$ . We assume the query graph to be connected. Any subset  $S$  of  $\{R_0, \dots, R_{n-1}\}$  induces a subgraph of the query graph. If the subgraph induced by  $S$  is connected, we call  $S$  a *connected subset* or simply *connected*. For a given query graph  $G$  in  $n$  relations, we denote by  $\#\text{csg}_G$  the number of non-empty connected subgraphs/subsets. For a given kind of query graph, every  $n$  uniquely determines a query graph. Since the kind of query graph will always be clear from the context, we write  $\#\text{csg}(n)$ .

Let  $S_1$  and  $S_2$  be two subsets of  $\{R_0, \dots, R_{n-1}\}$ . If there is a join predicate between a relation in  $S_1$  and another relation in  $S_2$ , we call  $S_1$  and  $S_2$  *connected*. Since we want to enumerate only bushy trees without cross products, we are only interested in connected sets  $S_1$  and  $S_2$  which are connected. Moreover, in order to form a valid join tree for relations in  $S := S_1 \cup S_2$ ,  $S_1$  and  $S_2$  may not overlap, i.e.  $S_1 \cap S_2 = \emptyset$ .

Summarizing, during plan generation we are interested in pairs  $(S_1, S_2)$  where

- $S_1$  is a non-empty subset of  $\{R_0, \dots, R_{n-1}\}$ , and

- $S_2$  is a non-empty subset of  $\{R_0, \dots, R_{n-1}\}$

such that

1.  $S_1$  is connected,
2.  $S_2$  is connected,
3.  $S_1 \cap S_2 = \emptyset$ ,
4. there exist nodes  $v_1 \in S_1$  and  $v_2 \in S_2$  such that there is an edge between  $v_1$  and  $v_2$  in the query graph.

These conditions imply that both  $S_1$  and  $S_2$  are strict subsets of  $\{R_0, \dots, R_{n-1}\}$ . Let us call a pair  $(S_1, S_2)$  fulfilling these conditions a *csg-cmp-pair*. Here, *csg* is the abbreviation of connected subgraph and *cmp* is the abbreviation of complement. The latter was chosen to emphasize the aspect of disjointness.

In the following, we are interested in (1) the number of connected, non-empty subsets and (2) the number of csg-cmp-pairs. Obviously, these numbers depend on the query graph. For csg-cmp-pairs, it is also important to note that if  $(S_1, S_2)$  is a csg-cmp-pair, then  $(S_2, S_1)$  is one as well. We denote the total number of csg-cmp-pairs including symmetric pairs by  $\#ccp$ . Given this, we immediately understand the `CsgCmpPairCounter` in the algorithms. After termination it gives us  $\#ccp$ . Ono and Lohman counted the number of csg-cmp-pairs by excluding symmetric pairs. Hence, their formulas return  $\#ccp$  divided by two. It is important to note that  $\#ccp$  only depends on the query graph. That is, the value of `CsgCmpPairCounter` (and, hence, for `OnoLohmanCounter`) after termination is the same for `DPsize`, `DPsub`, and the new algorithm `DPccp`. It is very important to note that for any correct dynamic programming algorithm  $\#ccp$  provides a lower bound on the number of calls to `CreateJoinTree`.

### 2.3.2 Formulas for $\#csg$ and $\#ccp$

We analyse the join ordering problem for chain, cycle, and clique queries. For each kind of query graph, we calculate the number of connected subgraphs ( $\#csg$ ) and the number of csg-cmp-pairs ( $\#ccp$ ).

For a chain query in  $n$  relations, we have

$$\#csg(n) = \frac{n(n+1)}{2} \quad (5)$$

$$\#ccp(n) = \frac{(n+1)^3 - (n+1)^2 + 2(n+1)}{3} \quad (6)$$

This result is due to Ono and Lohman [5].

For a cycle query in  $n$  relations, we have

$$\#csg(n) = n^2 - n + 1 \quad (7)$$

$$\#ccp(n) = n^3 - 2n^2 + n \quad (8)$$

For star queries in  $n$  relations, we have

$$\#csg(n) = 2^{n-1} + n - 1 \quad (9)$$

$$\#ccp(n) = (n-1)2^{n-2} \quad (10)$$

This result is due to Ono and Lohman [5].

For clique queries in  $n$  relations, we have

$$\#csg(n) = 2^n - 1 \quad (11)$$

$$\#ccp(n) = 3^n - 2^{n+1} + 1 \quad (12)$$

This result is due to Ono and Lohman [5].

## 2.4 Sample Numbers

Fig. 3 contains tables with values produced by our formulas for input query graph sizes between 2 and 20. For different kinds of query graphs, it shows the number of csg-cmp-pairs ( $\#ccp$ ), and the values for the inner counter after termination of `DPsize` and `DPsub` applied to the different query graphs.

Looking at these numbers, we observe the following:

- For chain and cycle queries, the `DPsize` soon becomes much faster than `DPsub`.
- For star and clique queries, the `DPsub` soon becomes much faster than `DPsize`.
- Except for clique queries, the number of csg-cmp-pairs is orders of magnitude less than the value of *InnerCounter* for all DP-variants.

From the latter observation we can conclude that in almost all cases the tests performed by both algorithms in their innermost loop fail. Both algorithms are far away from the theoretical lower bound given by  $\#ccp$ . This conclusion motivates us to derive a new algorithm whose *InnerCounter* value is equal to the number of csg-cmp-pairs.

## 3. THE NEW ALGORITHM DPCCP

### 3.1 Problem Statement

The algorithm `DPsub` solves the join ordering problem for a given subset  $S$  of relations by considering all pairs of disjoint subproblems which were already solved. Since the enumeration of subsets is very fast, this is a very efficient strategy if the search space is dense, e.g. for clique queries. However, if the search space is sparse, e.g. for chain queries, the `DPsub` algorithm considers many subproblems which are not connected and, therefore, are not relevant for the solution, i.e. the tests in the innermost loop fail for the majority of cases. The main idea of our algorithm `DPccp` is that it only considers pairs of connected subproblems. More precisely, the algorithm considers exactly the csg-cmp-pairs of a graph. Note that this is also the lower bound for any dynamic programming algorithm [9].

Thus, our goal is to efficiently enumerate all csg-cmp-pairs  $(S_1, S_2)$ . Clearly, we want to enumerate every pair once and only once. Further, the enumeration must be performed in an order valid for dynamic programming. That is, whenever a pair  $(S_1, S_2)$  is generated, all non-empty subsets of  $S_1$  and  $S_2$  must have been generated before as a component of a pair. The last requirement is that the overhead for generating a single csg-cmp-pair must be constant or at most linear. This condition is necessary in order to beat `DPsize` and `DPsub`.















