Efficient Dynamic Programming Algorithms for Ordering Expensive Joins and Selections

Wolfgang Scheufele* Guido Moerkotte

Universität Mannheim
Lehrstuhl für Praktische Informatik III
68131 Mannheim, Germany
e-mail: {ws | moer}@pi3.informatik.uni-mannheim.de

Abstract. The generally accepted optimization heuristics of pushing selections down does not yield optimal plans in the presence of expensive predicates. Therefore, several researchers have proposed algorithms to compute optimal processing trees for queries with expensive predicates. All these approaches are incorrect—with one exception [3]. Our contribution is as follows. We present a formally derived and correct dynamic programming algorithm to compute optimal bushy processing trees for queries with expensive predicates. This algorithm is then enhanced to be able to (1) handle several join algorithms including sort merge with a correct handling of interesting sort orders, to (2) perform predicate splitting, to (3) exploit structural information about the query graph to cut down the search space. Further, we present efficient implementations of the algorithms. More specifically we introduce unique solutions for efficiently computing the cost of the intermediate plans and for saving memory space by utilizing bitvector contraction. Our implementations impose no restrictions on the type of query graphs, the shape of processing trees or the class of cost functions. We establish the correctness of our algorithms and derive tight asymptotic bounds on the worst case time and space complexities. We also report on a series of benchmarks showing that queries of sizes which are likely to occur in practice can be optimized over the unconstrained search space in less than a second.

1 Introduction

Traditional work on algebraic query optimization has mainly focused on the problem of ordering joins in a query. Restrictions like selections and projections are generally treated by "push-down rules". According to these, selections and projections should be pushed down the query plan as far as possible. These heuristic rules worked quite well for traditional relational database systems where the evaluation of selection predicates is of neglectable cost and every selection reduces the cost of subsequent joins. As pointed out by Hellerstein, Stonebraker [5], this is no longer true for modern database systems like object-oriented DBMSs

* Research supported by the German Research Association (DFG) under contract Mo 507/6-1.

that allow users to implement arbitrary complex functions in a general-purpose programming language. In this paper we present a dynamic programming algorithm for computing optimal bushy processing trees with cross products for conjunctive queries with expensive join and selection predicates. The algorithm is then enhanced to (1) handle several join algorithms including sort merge with a correct handling of interesting sort orders, to (2) perform predicate splitting, to (3) exploit structural information about the query graph to cut down the search space. There are no restrictions on the shape of the processing trees, the structure of the query graph or the type of cost functions. We then focus on *efficient algorithms* with respect to both the asymptotic time complexity and the hidden constants in the implementation. Our dynamic programming algorithm and its enhancements were formally derived by means of recurrences and time and space complexities are analyzed carefully. We present details of an efficient implementation and sketch possible generalizations. More specifically we introduce unique solutions for efficiently computing the cost of the intermediate plans and for saving memory space by utilizing bitvector contraction. A more detailed description of our algorithm can be found in our technical report [10].

The rest of the paper is organized as follows. Section 2 summarizes related work and clarifies the contribution of our approach over existing approaches. Section 3 covers the background for the rest of the paper. In section 4 we present the dynamic programming algorithm for ordering expensive selections and joins. Section 5 discusses the problems to be solved in an efficient implementation of these solutions. One of its main contributions is to offer a possibility for a fast computation of cost functions. This is a major point, since most of the optimization time in a dynamic programming approach is often spent on computing costs. A second major point introduces techniques for space saving measures. In section 6 we discuss several possible generalizations of our algorithm accounting for interesting sort orders, the option to split conjunctive predicates, and the exploitation of structural information from the join graph. Section 7 shows the results of timing measurements and section 8 concludes the paper.

2 Related Work and Contribution

Only few approaches exist to the problem of ordering joins and selections with expensive predicates. In the LDL system [4] and later on in the Papyrus project [1] expensive selections are modelled as artificial relations which are then ordered by a traditional join ordering algorithm producing left-deep trees. This approach suffers from two disadvantages. First, the time complexity of the algorithm cannot compete with the complexity of approaches which do not model selections and joins alike and, second, left-deep trees do not admit plans where more than one cheap selection is "pushed down". Another approach is based upon the "predicate migration algorithm" [5, 6] which solves the simpler problem of interleaving expensive selections in an existing join tree. The authors of [5, 6] suggest to solve the general problem by enumerating all join orders while placing the expensive selections with the predicate migration algorithm—in combination

with a system R style dynamic programming algorithm endowed with pruning. The predicate migration approach has several severe shortcomings. It may degenerate to exhaustive enumeration, it assumes a linear cost model and it does not always yield optimal results [2]. Recently, Chaudhuri and Shim presented a dynamic programming algorithm for ordering joins and expensive selections [2]. Although they claim that their algorithm computes optimal plans for all cost functions, all query graphs, and even when the algorithm is generalized to bushy processing trees and expensive join predicates, the alleged correctness has not been proved at all. In fact, it is not difficult to find counterexamples disproving the correctness for even the simplest cost functions and processing trees. This bug was later discovered and the algorithm restricted to work on regular cost functions only [3]. Further, it does not generate plans that contain cross products. The algorithm is not able to consider different join implementations. Especially the sort merge join is out of the scope of the algorithm due to its restriction to regular cost functions. A further disadvantage is that the algorithm does not perform predicate splitting. The contribution of our algorithm and its enhancements are: (1) It works on arbitrary cost functions. (2) It generates plans with cross products. (3) It can handle different join algorithms. (4) It is capable of exploiting interesting sort orders. (5) It employs predicate splitting. (6) It uses structural information to restrict the search space. Our final contribution are tight time and space bounds.

3 Preliminaries

We consider simple *conjunctive queries* [12] involving only single table selections and binary joins (selection-join-queries). A query is represented by a set of relations $R_1, \ldots, R_n$ and a set of query predicates $p_1, \ldots, p_n$, where p_k is either a join predicate connecting two relations R_i and R_j or a selection predicate which refers to a single relation R_k (henceforth denoted by σ_k). All predicates are assumed to be either *basic predicates* or conjunctions of basic predicates (*conjunctive predicates*). Basic predicates are simple built-in predicates or predicates defined via user-defined functions which may be expensive to compute.

Let $R_1, \ldots, R_n$ be the relations involved in the query. Associated with each relation is its *cardinality* $n_i = |R_i|$. The predicates in the query induce a *join graph* $G = (\{R_1, \ldots, R_n\}, E)$, where E contains all pairs $\{R_i, R_j\}$ for which exists a predicate p_k relating R_i and R_j. For every join or selection predicate $p_k \in P$, we assume the existence of a *selectivity* f_k [12] and a cost factor c_k denoting the costs for a single evaluation of the predicate.

A *processing tree* for a select-join-query is a rooted binary tree with its internal nodes having either one or two sons. In the first case the node represents a selection operation and in the latter case it represents a binary join operation. The tree has exactly n leaves, which are the relations $R_1, \ldots, R_n$. Processing trees are classified according to their shape. The main distinction is between *left-deep trees* and *bushy trees*. In a left-deep tree the right subtree of an internal node does not contain joins. Otherwise it is called a *bushy tree*.

There are different implementations of the join operator each leading to different cost functions for the join and hence to different cost functions for the whole processing tree. We do not want to commit ourselves to a particular cost function, instead the reader may select his favorite cost function from a large class of *admissible cost functions* which are subject to the following two requirements. First, the cost function is *decomposable* and thus can be computed by means of recurrences. Second, the costs of a processing tree are *(strictly) monotonously increasing with respect to the costs of its subtrees.* This seems to be no major restriction for "reasonable" cost functions. It can be shown [9] that such cost functions guarantee that every optimal solution satisfies the "principle of optimality" which we state in the next section. In order to discuss some details of an efficient implementation we assume that the cost function can be written as a recurrence involving several auxiliary functions, an example is the *size function* (the number of tuples in the result of a subquery).

4 The Dynamic Programming Algorithm

Let us denote the set of relations occurring in a bushy plan P by $Rel(P)$ and the set of relations to which selections in P refer by $Sel(P)$. Let R denote a set of relations. We denote by $Sel(R)$ the set of all selections referring to some relation in R. Each subset $V \subseteq R$ defines an *induced subquery* which contains all the joins and selections that refer to relations in V only. A *subplan* P' of a plan P corresponds to a subtree of the expression tree associated with P. A *partition* of a set S is a pair of nonempty disjoint subsets of S whose union is exactly S. For a partition S_1, S_2 of S we write $S = S_1 \uplus S_2$. By a *k-set* we simply mean a set with exactly k elements.

Consider an optimal plan P for an induced subquery involving the nonempty set of relations $Rel(P)$ and the set of selections $Sel(P)$. Obviously, P has either the form $P \equiv (P_1 \bowtie P_2)$ for subplans P_1 and P_2 of P, or the form $P \equiv \sigma_i(P')$ for a subplan P' of P and a selection $\sigma_i \in Sel(P)$. The important fact is now that the subplans P_1, P_2 are necessarily *optimal plans* for the relations $Rel(P_1), Rel(P_2)$ and the selections $Sel(P_1), Sel(P_2)$, where $Rel(P_1) \uplus Rel(P_2) = Rel(P)$, $Sel(P_1) = Sel(P) \cap Sel(R_1)$, $Sel(P_2) = Sel(P) \cap Sel(R_2)$. Similarly, P' is an optimal bushy plan for the relations $Rel(P')$ and the selections $Sel(P)$, where $Rel(P') = Rel(P)$, $Sel(P') = Sel(P) - \{\sigma_i\}$. Otherwise we could obtain a cheaper plan by replacing the suboptimal part by an optimal one which would be a contradiction to the assumed optimality of P (note that our cost function is decomposable and monotone). The property that optimal solutions of a problem can be decomposed into a number of "smaller", likewise optimal solutions of the same problem, is known as *Bellman's optimality principle.* This leads immediately to the following recurrence for computing an optimal bushy plan[1] for a set of relations R and a set of selections S.

[1] min() is the operation which yields a plan with minimal costs among the addressed set of plans. Convention: $\min_\emptyset(\dots) := \lambda$ where λ denotes some artificial plan with cost ∞.

$$\mathrm{opt}(R,S) = \begin{cases} \min(\min_{\emptyset \subset R' \subset R}(\mathrm{opt}(R', S \cap \mathrm{Sel}(R')) \bowtie & \text{if } \emptyset \subseteq S \subseteq R, \\ \qquad \mathrm{opt}(R \setminus R', S \cap \mathrm{Sel}(R \setminus R'))) & \\ \qquad \min_{\sigma_i \in S}(\sigma_i(\mathrm{opt}(R, S \setminus \{\sigma_i\})))) & \\ R_i & \text{if } R = \{R_i\}, \\ & \quad S = \emptyset \end{cases} \tag{1}$$

The join symbol $\bowtie$ denotes a join *with the conjunction of all join predicates* that relate relations in R' to relations in $R \setminus R'$. Considering the join graph, the conjuncts of the join predicate correspond to the predicates associated with the edges in the cut $(R', R \setminus R')$. If the cut is empty the join is actually a *cross product*.
In our first algorithm we will treat such joins and selections with conjunctive predicates as single operations with according accumulated costs. The option to split such predicates will be discussed in section 6.2 where we present a second algorithm.

Based on recurrence (1), there is an obvious recursive algorithm to solve our problem but this solution would be very inefficient since many subproblems are solved more than once. A much more efficient way to solve this recurrence is by means of a table and is known under the name of *dynamic programming* [8, 11]. Instead of solving subproblems recursively, we solve them one after the other in some appropriate order and store their solutions in a table. The overall time complexity then becomes (typically) a function of the number of distinct subproblems rather than of the larger number of recursive calls. Obviously, the subproblems have to be solved in the right order so that whenever the solution to a subproblem is needed it is already available in the table. A straightforward solution is the following. We enumerate all subsets of relations by increasing size, and for each subset R we then enumerate all subsets S of the set of selections occurring in R by increasing size. For each such pair (R, S) we evaluate the recurrence (1) and store the solution associated with (R, S).

For the following algorithm we assume a given select-join-query involving n relations $\mathcal{R} = \{R_1, \ldots, R_n\}$ and $m \leq n$ selections $\mathcal{S} = \{\sigma_1, \ldots, \sigma_m\}$. In the following, we identify selections and relations to which they refer. Let P be the set of all join predicates $p_{i,j}$ relating two relations R_i and R_j. By R_S we denote the set $\{R_i \in \mathcal{R} \mid \exists\, \sigma_j \in \mathcal{S} : \sigma_j \text{ relates to } R_i\}$ which consists of all relations in $\mathcal{R}$ to which some selection in $\mathcal{S}$ relates. For all $U \subseteq \mathcal{R}$ and $V \subseteq U \cap R_S$, at the end of the algorithm $T[U, V]$ stores an optimal bushy plan for the subquery (U, V).

```
proc Optimal-Bushy-Tree(R, P)
1  for k = 1 to n do
2      for all k-subsets M_k of R do
3          for l = 0 to min(k, m) do
4              for all l-subsets P_l of M_k ∩ R_S do
5                  best_cost_so_far = ∞;
6                  for all subsets L of M_k with 0 < |L| < k do
7                      L' = M_k \ L, V = P_l ∩ L, V' = P_l ∩ L';
8                      p = ⋀{p_{i,j} | p_{i,j} ∈ P, R_i ∈ V, R_j ∈ V'};    // p=true might hold
9                      T = (T[L, V] ⋈_p T[L', V']);
10                     if Cost(T) < best_cost_so_far then
11                         best_cost_so_far = Cost(T);
12                         T[M_k, P_l] = T;
13                     fi;
14                 od;
15                 for all R ∈ P_l do
16                     T = σ_R(T[M_k, P_l \ {R}]);
17                     if Cost(T) < best_cost_so_far then
18                         best_cost_so_far = Cost(T);
19                         T[M_k, P_l] = T;
20                     fi;
21                 od;
22             od;
23         od;
24     od;
25 od;
26 return T[R, S];
```

Complexity of the algorithm: In [10] we show that the number of considered partial plans is $3^n(5/3)^m + (2m/3 - 2) \cdot 2^n(3/2)^m + 1$ which we can rewrite as $[3(5/3)^c]^n + (2cn/3 - 2)[2(3/2)^c]^n + 1$ if we replace m by the fraction of selections $c = m/n$. Assuming an asymptotic optimal implementation of the enumeration part of the algorithm (see section 5), the amount of work per considered plan is constant and the asymptotic time complexity of our algorithm is $O([3\,(5/3)^c]^n + n[2\,(3/2)^c]^n)$. Assuming that the relations are re-numbered such that all selections refer to relations in $\{R_1, \ldots, R_m\}$, the space complexity of the algorithm is $O(2^{n+m})$. Note that not every table entry $T[i, j], 0 \leq i < n, 0 \leq j < m$ represents a valid subproblem. In fact, the number of table entries used by the algorithm to store the solutions of subproblems is $2^n(3/2)^m - 1$. In section 5.3 we discuss techniques to save space.

5 An Efficient Implementation

5.1 Fast Enumeration of Subproblems

The frame of our dynamic programming algorithm is the systematic enumeration of subproblems consisting of three nested loops iterating over subsets of relations and predicates, respectively.

The first loop enumerates all nonempty subsets of the set of all relations in the query. It turns out that enumerating all subsets strictly by increasing size seems not to be the most efficient way. The whole point is that the order of enumeration only has to guarantee that for every enumerated set S, all subsets of S have already been enumerated. One of such orderings, which is probably the most suitable, is the following. We use the standard representation of n-sets, namely bitvectors of length n. A subset of a set S is then characterized by a bitvector which is component-wise smaller than the bitvector of S. This leads to the obvious ordering in which the bitvectors are arranged according to their value as binary numbers. This simple and very effective enumeration scheme (*binary counting method*) is successfully used in [13]. The major advantage is that we can pass over to the next subset by merely incrementing an integer, which is an extremely fast hardwired operation.
The next problem is to enumerate subsets S of a fixed subset M of a set Q. If Q has n elements then we can represent M by a bitvector m of length n. Since M is a subset of Q some bit positions of m may be zero. Vance and Maier propose in [13] a very efficient and elegant way to solve this problem. In fact, they show that the following loop enumerates all bitvectors S being a subset of M, where $M \subseteq Q$.

$S = 0;$
repeat
  $\ldots$
  $S = M \,\&\, (S - M);$
until $S = 0$

We assume two's-complement arithmetic. Bit-operations are denoted as in the language C. As an important special case, we mention that $M \,\&\, -M$ yields the bit with the *smallest* index in M. Similarily, one can count downward using the operation $S = M \,\&\, (S-1)$. Combining these operations, one can show that the operation $S = M \,\&\, ((S \mid (M \,\&\, (S-1))) - M)$ (with initial value $S = M \,\&\, -M$) iterates through each single bit in the bitvector M in order of increasing indices.

5.2 Efficient Computation of the Cost Function

Now, we discuss the efficient evaluation of the cost function within the nested loops. Obviously, for a given plan we can compute the costs and the size in (typically) linear time but there is a even more efficient way using the recurrences for these functions. If R', R'' and S', S'' are the partitions of R and S, respectively, for which the recurrence (1) assumes a minimum, we have

$$Size(R,S) = Size(R',S') * Size(R'',S'') * Sel(R',R''),$$

where

$$Sel(R',R'') := \prod_{R_i \in R', R_j \in R''} f_{i,j}$$

is the product of all selectivities between relations in R' and R''. Note that the last equation holds for *every* partition R', R'' of R independent of the root operator in an optimal plan. Hence we may choose a certain partition in order to simplify the computations. Now if $R' = U_1 \uplus U_2$ and $R'' = V_1 \uplus V_2$, we have the following recurrence for $Sel(R', R'')$

$$Sel(U_1 \uplus U_2, V_1 \uplus V_2) = Sel(U_1, V_1) * Sel(U_1, V_2) * Sel(U_2, V_1) * Sel(U_2, V_2)$$

Choosing $U_1 = \alpha(R)$, $U_2 := \emptyset$, $V_1 = \alpha(R \setminus U_1)$, and $V_2 := R \setminus U_1 \setminus V_2$, where the function α is given by $\alpha(A) := \{R_k\},\ k = \min\{i \mid R_i \in A\}$
leads to

$$\begin{aligned} &Sel(\alpha(R), R \setminus \alpha(R)) = \\ &\quad Sel(\alpha(R), \alpha(R \setminus \alpha(R))) * Sel(\alpha(R), (R \setminus \alpha(R)) \setminus \alpha(R \setminus \alpha(R))) = \\ &\quad Sel(\alpha(R), \alpha(R \setminus \alpha(R))) * Sel(\alpha(R), (R \setminus \alpha(R \setminus \alpha(R))) \setminus \alpha(R)) \end{aligned}$$

Defining the *fan-selectivity* $Fan\_Sel(R)$ as $Sel(\alpha(R), R \setminus \alpha(R))$, gives the simpler recurrence

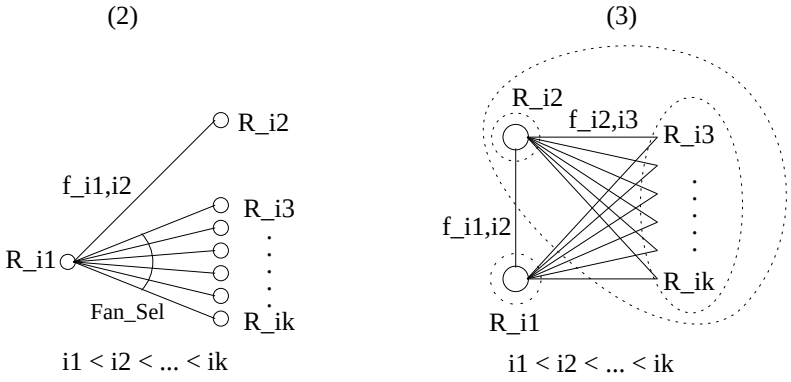
$$\begin{aligned} Fan\_Sel(R) &= Sel(\alpha(R), \alpha(R \setminus \alpha(R))) * Fan\_Sel(R \setminus \alpha(R)) \\ &= f_{i,j} * Fan\_Sel(R \setminus \alpha(R \setminus \alpha(R))) \end{aligned} \quad (2)$$

where we assumed $\alpha(R) = \{R_i\}$ and $\alpha(R \setminus \alpha(R)) = \{R_j\}$.

As a consequence, we can compute $Size(R, S)$ with the following recurrence

$$\begin{aligned} Size(R, S) = &Size(\alpha(R), S \cap \alpha(R)) * Size(R \setminus \alpha(R), (R \setminus \alpha(R)) \cap S) \\ &* Fan\_Sel(R) \end{aligned} \quad (3)$$

We remind that the single relation in $\alpha(R)$ can be computed very efficiently via the operation $a\,\&\,-a$ on the bitvector of R. Recurrences (2) and (3) are illustrated in the following figure. The encircled sets of relations denote the nested partitions along which the sizes and selectivities are computed.



5.3 Space Saving Measures

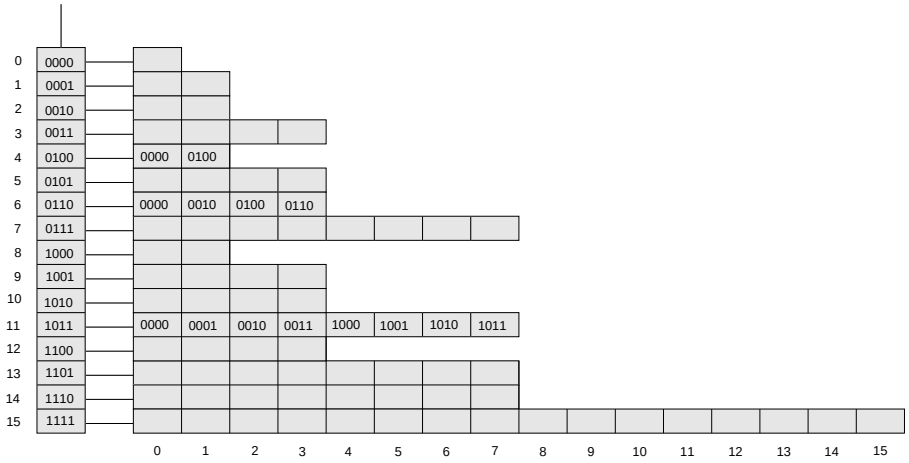
Another problem is how to store the tables without wasting space. For example, suppose $n = 10$, $m = 10$ and let r and s denote the bitvectors corresponding to the sets R and S, respectively. If we use r and s directly as indices of a two-dimensional array $cost[][]$, about 90% of the entries in the table will not be accessed by the algorithm. To avoid this immense waste of space we have to use a *contracted version* of the second bitvector s.

Definition 1. *(bitvector contraction)*
Let r and s be two bitvectors consisting of the bits $r_0 r_1 \ldots r_n$ and $s_0 s_1 \ldots s_n$, respectively. We define the contraction of s with respect to r *as follows:*

$$contr_r(s) = \begin{cases} \epsilon & \text{if } s = \epsilon \\ contr_{r_1 \ldots r_n}(s_1 \ldots s_n) & \text{if } s \neq \epsilon \text{ and } r_0 = 0 \\ s_0\, contr_{r_1 \ldots r_n}(s_1 \ldots s_n) & \text{if } s \neq \epsilon \text{ and } r_0 = 1 \end{cases}$$

Example Let us contract the bitvector $s = 0101001111$ with respect to the bitvector $r = 1110110100$. For each bit s_i of s, we examine the corresponding bit r_i in r. If r_i is zero we "delete" s_i in s, otherwise we retain it. The result is the contracted bitvector 010001.

The following figure shows the structure of the two-dimensional ragged arrays. The first dimension is of fixed size 2^n, whereas the second dimension is of size 2^k where k is the number of common nonzero bits in the value of the first index i and the bitvector of all selections *sels*. The number of entries in such a ragged array is $\sum_{k=0}^{m} \binom{m}{k} 2^k 2^{n-m} = [2(\frac{3}{2})^c]^n$. Note that this number equals 2^n for $m = 0$ and 3^n for $m = n$. A simple array would require 4^n entries. The figure below shows the worst case where the number of selections equals the number of relations.



The simplest way would be to contract the second index on the fly for each access of the array. This would slow down the algorithm by a factor of n. It would be much better if we could contract all relevant values in the course of the other computations—therefore not changing the asymptotic time complexity. Note that within our algorithm the first and second indices are enumerated by increasing values which makes a contraction easy. We simply use pairs of indices i, ic, one uncontracted index i and its contracted version ic and count up their values independently.

As mentioned before, in the two outermost loops bitvector contraction is easy due to the order of enumeration of the bitvectors. Unfortunately, this does not work in the innermost loop where we have to contract the result of a conjunction of two contracted bitvectors again. Such a "double contraction" cannot be achieved by simple counting or other combinations of arithmetic and bit operations. One possible solution is to contract the index in the innermost loop on the fly, another is to tabulate the contraction function. Since tabulation seems slightly more efficient, we apply the second method. Before entering the innermost loop we compute and store the results of all contraction operations which will occur in the innermost loop. The computation of contraction values can now be done more efficiently since we do not depend on the specific order of enumeration in the innermost loop. The values in the table can be computed bottom-up using the recurrence given in Definition 1. Due to contraction the new time complexity of the algorithm is $O((n+1)[3(5/3)^c]^n + n[2(3/2)^c]^n + (cn)^2 2^n)$ whereas the new space complexity is $O([2(3/2)^c]^n + 2^n)$.

6 Some Generalizations

6.1 Different Join Algorithms

In addition to the problem of optimal ordering expensive selections and joins the optimizer eventually has to select a join algorithm for each of the join operators in the processing tree. Let us call processing trees where a join method from a certain pool of join implementations is assigned to each join operator an *annotated processing tree*. We now describe how our dynamic programming algorithm can be generalized to determine optimal annotated bushy processing trees in one integrated step.

The central point is that the application of a certain join method can change the physical representation of an intermediate relation such that the join costs in subsequent steps may change. For example, the application of a sort-merge join leaves a result which is sorted with respect to the join attribute. Nested loop joins are order preserving, that is if the outer relation is sorted with respect to an attribute A then the join result is also sorted with respect to A. A join or selection may take advantage of the sorted attribute. In the following discussion we restrict ourselves to the case where only nested-loop joins and sort-merge joins are available. Furthermore, we assume that all join predicates in the query are equi-joins so that both join methods can be applied for every join that occurs.

This is not a restriction of the algorithm but makes the following discussion less complex.

Consider an optimal bushy plan P for the relations in $Rel(P)$ and the selections in $Sel(P)$. Again we can distinguish between a join operator representing the root of the plan tree and a selection operator representing the root. In case of a join operator we further distinguish between the join algorithms nested-loop (nl) and sort-merge (sm). Let $C_a(R,S)$ be the costs of an optimal subplan for the relations in R and selections in S, where the result is sorted with respect to the attribute a (of some relation $r \in R$). $C(R,S)$ is similar defined, but the result is not necessarily sorted with respect to any attribute. Now, the optimality principle holds both for $C(R,S)$ and $C_x(R,S)$ for all attributes x in the result of the query (R,S) and one can easily specify two recurrences which compute $C(R,S)$ and $C_{a|b}(R,S)$ in terms of $C(R',S'), C_a(R',S')$ and $C_b(R',S')$ for "smaller" subproblems (R',S'), respectively. Here $a|b$ denotes the join attribute in the result of an equi-join with respect to the attributes a and b.

Time Complexity: The number of considered partial plans is $O(n^2[3(5/3)^c]^n + cn^3[2(3/2)^c]^n)$. Similarly, the asymptotic number of table entries in our new algorithm is $O(n^2[2(3/2)^c]^n)$.

6.2 Splitting Conjunctive Predicates

So far our algorithm does consider joins and selections over *conjunctive predicates* which may occur in the course of the algorithm as indivisible operators of the plan. For example, consider a query on three relations with three join predicates relating all the relations. Then, every join operator in the root of a processing tree has a join predicate which is the conjunction of two basic join predicates. In the presence of expensive predicates this may lead to suboptimal[2] plans since there may be a cheaper plan which splits a conjunctive join predicate into a join with high selectivity and low costs and a number of (secondary) selections with lower selectivities and higher costs. Consequently, we do henceforth consider the larger search space of all queries formed by joins and selections with *basic predicates* which are equivalent to our original query.

The approach is similar to our first approach but this time we have to take into account the basic predicates involved in a partial solution. First, we replace all conjunctive selection and join predicates in the original query by all its conjuncts. Note that this makes our query graph a multigraph. Let $p_1, \ldots, p_m$ be the resulting set of basic predicates. We shall henceforth use bitvectors of length m to represent sets of basic predicates. Let us now consider an optimal plan P which involves the relations in R and the predicates in P. We denote the costs of such an optimal plan by $C(R,P)$. Obviously, the root operator in P is either a cross product, a join with a basic predicate h_1 or a selection with a basic predicate h_2. Hence, exactly one of the following four cases holds:

[2] with respect to the larger search space defined next

$P \equiv P_1 \times P_2$ **(cross product):** $Rel(P) = Rel(P_1) \uplus Rel(P_2)$, $Pred(P) = Pred(P_1) \uplus Pred(P_2)$. The induced join graphs of P_1 and P_2 are not connected in the induced join graph of P. Besides, the subplans P_i $(i = 1, 2)$ are optimal with respect to the corresponding sets $Rel(P_i)$, $Pred(P_i)$.

$P \equiv P_1 \bowtie_{h_1} P_2$ **(join):** $Rel(P) = Rel(P_1) \uplus Rel(P_2)$ and $Pred(P) = Pred(P_1) \uplus Pred(P_2) \uplus \{h_1\}$. The induced join graphs of P_1 and P_2 are connected by a bridge h_1 in the induced join graph of P. Furthermore, the subplans P_1, P_2 are optimal with respect to the corresponding sets $Rel(P_i)$, $Pred(P_i)$.

$P \equiv \sigma_{h_1}(P_1)$ **(secondary selection):** $Rel(P) = Rel(P_1)$ and $Pred(P) = Pred(P_1) \uplus \{h_1\}$. The subplan P_1 is optimal with respect to the corresponding sets $Rel(P_1)$ and $Pred(P_1)$.

$P \equiv \sigma_{h_2}(P_1)$ **(single table selection):** $Rel(P) = Rel(P_1)$ and $Pred(P) = Pred(P_1) \uplus \{h_2\}$. The subplan P_1 is optimal with respect to the corresponding sets $Rel(P_1)$ and $Pred(P_1)$.

Again, it is not difficult to see that the optimality principle holds and one can give a recurrence which determines an optimal plan $opt(R, P)$ for the problem (R, P) by iterating over all partitions of R applying the corresponding cost function according to one of the above cases. More details can be found in the technical report [10].

Complexity Issues: Let $m > 0$ be the number of different basic predicates (joins and selections) and n the number of base relations. We derived the following upper bound on the number of considered partial plans $m2^m3^n - 3m2^{m+n-1} + m2^{m-1} = O(m2^m3^n)$.
As to the space complexity, one can easily give an upper bound of 2^{n+m} on the number of table entries used by the algorithm.

6.3 Using Structural Information from the Join Graph

So far we concentrated on *fast enumeration* without analyzing structural information about the problem instance at hand. To give an example, suppose our query graph to be 2-connected. Then every cut contains at least two edges; therefore the topmost operator in a processing tree can neither be a join nor a cross product. Hence, our second algorithm iterates in vain over all 2^n partitions of the set of all n relations.

We will now describe the approach in our third algorithm which uses *structural information* from the join graph to avoid the enumeration of unnecessary subproblems. Suppose that a given query relates the relations in R and the predicates in P. We denote the number of relations in R by n, the number of predicates in P by m and the number of selections by s. Consider any optimal processing tree T for a subquery induced by the subset of relations $R' \subseteq R$ and the subset of predicates $P' \subseteq P$. The join graph induced by the relations R and the predicates P is denoted by $G(R, P)$. Obviously, the root operator in T is either *a cross product, a join, a secondary selection* or *a single table selection.* Depending on the operation in the root of the processing tree, we can classify

the graph $G(R', P')$ as follows. If the root operation is a cross product, then $G(R', P')$ decomposes into *two components*. Otherwise, if it is a join with (basic) predicate h then $G(R', P')$ contains a *bridge* h. If it is a secondary selection with (basic) predicate h then $G(R', P')$ contains an *edge* h. And, if it is a single table selection with predicate h then $G(R', P')$ contains a *loop* h.
Consequently, we can enumerate all possible cross products by building all partitions of the set of components of $G(R', P')$. All joins can be enumerated by considering all bridges of $G(R', P')$ as follows. Suppose that the bridge is the basic predicate h and denote the set of components resulting from removing the bridge by C. Now, every possible join can be obtained by building all partitions of the components in C and joining them with the predicate h. All secondary selections can be enumerated by just considering every edge (no loops) as selection predicate. And, if we step through all loops in $G(R', P')$ we enumerate all single table selections. In a sense we are tackling subproblems in the reverse direction than we did before. Instead of enumerating all partitions and analyzing the type of operation it admits we consider types of operations and enumerate the respective subproblems.

Complexity: One can show that the number of considered partial plans is at most $2^{m-1}(3^n - 1) + 2m(3/2)^{m-1}(3^n - 1) + m2^{m-1}(2^n - 1)$ and the asymptotic worst-case time complexity is $O(2^m 3^n + m(3/2)^m 3^n + (m+n)2^{m+n} - m2^m)$. Furthermore, the number of table entries used by the algorithm is 2^{n+m}.

The above time and space complexities are generous upper bounds which do not account for the structure of a query graph. For an acyclic join graph G with n nodes, $m = n - 1$ edges (no loops and multiple edges) and s loops (possibly some multiple loops) the number of considered partial plans is bound by $2^s[4^n(n/2 - 1/4) + 3^n(2n/9 - 2/9 + s/6)]$ and the asymptotic time complexity is $O(2^s(n^2 4^n + (n^2 + s)3^n))$.

7 Performance Measurements

In this section we investigate how our dynamic programming algorithms perform in practice. Note that we give no results of the quality of the generated plans, since they are optimal. The only remaining question is whether the algorithm can be applied in practice since the time bounds seem quite high. We begin with algorithm Optimal-Bushy-Tree(R,P) without enhancements. The table below shows the timings for random queries with $n = 10$ relations and query graphs with 50% of all possible edges having selectivity factors less than 1. Join predicates are cheap (i.e. $c_p = 1$) whereas selection predicates are expensive (i.e. $c_p > 1$). All timings in this section were measured on a lightly loaded Sun SPARCstation 20 with 128 MB of main memory. They represent averages over a few hundred runs with randomly generated queries.

10 relations *sel* :	0	1	2	3	4	5	6	7	8	9	10
time [s]:	< 0.1	< 0.1	< 0.1	0.1	0.2	0.5	0.9	2.1	4.3	8.1	13.5

Let us now consider algorithm Optimal-Bushy-Tree(R,P) enhanced by splitting join predicates.

The next table shows running times for random queries on $n = 5$ relations. We varied the number of predicates from 0 to 16, the fraction of selections was held constant at 50%. For this example we used a modified "near-uniform" random generation of queries in order to produce connected join graphs whenever possible, i.e. we did only produce queries where the number of components equals $\max(n - p + s, 1)$.

$n = 5$ p :	0	1	2	3	4	5	6	7	8
time [s]:	< 0.001	< 0.001	< 0.001	0.001	0.001	0.002	0.003	0.006	0.011
p :	9	10	11	12	13	14	15	16	
time [s]:	0.022	0.043	0.084	0.177	0.364	0.801	1.763	3.899	

Finally, we considered queries with 10 predicates, out of which 5 are selections. By varying the number of relations from 1 to 10, we obtained the following timings.

$p = 10$, $s = 5$ n :	1	2	3	4	5	6	7	8	9	10
time [s]:	-	0.004	0.008	0.018	0.043	0.122	0.374	1.204	4.151	13.927

More performance measurements can be found in the technical report [10].

8 Conclusion

We presented a novel dynamic programming algorithm for the problem to determine optimal bushy processing trees with cross products for conjunctive queries with expensive selection and join predicates. Several enhancements were discussed giving our algorithm a leading edge over the only correct algorithm existing so far [3]. Our main focus was on developing *asymptotically efficient algorithms* that also admit extremely *efficient implementations* which make the algorithms competitive in operation.

Our algorithm and its enhancements compare to the algorithm of Chaudhuri and Shim [3] as follows. It works on arbitrary cost functions whereas the algorithm of Chaudhuri and Shim is confined to regular cost functions—excluding e.g. sort-merge joins. Our algorithm incorporates cross products, different join and selection algorithms and predicate splitting, none of them is supported by the algorithm of Chaudhuri and Shim. Furthermore we describe how to use structural information from the join graph to cut down the size of the search space. This is orthogonal to the cost-driven pruning techniques described in [3]. We also derived tight bounds on the asymptotic worst-case time complexity.

Acknowledgement We thank S. Helmer and B. Rossi for careful reading of a first draft of the paper.

References

1. S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 529–542, Dublin, Ireland, 1993.
2. S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 87–98, Bombay, India, 1996.
3. S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. Technical report, Microsoft Research, Advanced Technology Division, One Microsoft Way, Redmond, WA 98052, USA, 1997.
4. R. Gamboa D. Chimenti and R. Krishnamurthy. Towards an open architecture for $\mathcal{LDL}$. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 195–203, Amsterdam, Netherlands, August 1989.
5. J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–277, Washington, DC, 1993.
6. J. M. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 325–335, Minneapolis, Minnesota, USA, May 1994.
7. A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimization of boolean expressions in object bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 79–90, 1992.
8. M. Minoux. *Mathematical Programming. Theory and Algorithms.* Wiley, 1986.
9. T. L. Morin. Monotonicity and the principle of optimality. *J. Math. Anal. and Appl.*, 1977.
10. W. Scheufele and G. Moerkotte. Efficient dynamic programming algorithms for ordering expensive joins and selections. Forthcoming Technical Report, Lehrstuhl für Praktische Informatik III, Universität Mannheim, 68131 Mannheim, Germany, 1998.
11. C. E. Leiserson T. H. Cormen and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, USA, 1990.
12. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II: The New Technologies. Computer Science Press, 1989.
13. B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, Toronto, Canada, 1996.