

# Structural Join Order Selection for XML Query Optimization

Yuqing Wu  
Univ of Michigan  
yuwu@eecs.umich.edu

Jignesh M. Patel  
Univ of Michigan  
jignesh@eecs.umich.edu

H. V. Jagadish  
Univ of Michigan  
jag@eecs.umich.edu

## Abstract

Structural join operations are central to evaluating queries against XML data, and are typically the operations responsible for consuming a lion's share of the query processing time. Thus, structural join plan selection is at the heart of query optimization in an XML database, just as (value-based) join order selection is central to relational query optimization.

In this paper, we introduce five algorithms for structural join order optimization for XML tree pattern matching. The Dynamic Programming (DP) algorithm is similar to the traditional RDB optimization technique (including consideration of bushy plans) and selects the optimal solution. The Dynamic Programming with Pruning (DPP) algorithm prunes unpromising join plans in early stage, by introducing the notion of an upper bound on the cost of the portion of the query that is not yet optimized, and using this cost to prune intermediate query plans that are *guaranteed* to lead to suboptimal solutions. Thus it finds the optimal solution, identical to that found by DP, but at substantially lower cost. More aggressive pruning is possible at the cost of the optimum guarantee, leading to DPAP (Dynamic Programming with Aggressive Pruning) algorithms. We introduce two, each based on a different heuristic: bounding the expansion by quantity (DPAP-EB), or structurally, focusing only on left-deep plans (DPAP-LD), following the popular RDB optimization heuristic. Finally, taking the features of XML pattern matching into consideration, the Fully Pipelined (FP) algorithm considers the solution space of only fully-pipelined plans, as a new heuristic.

We present extensive experimental evaluation of our techniques for a variety of queries over a variety of XML data sets. Our experiments demonstrate that many relational rules of thumb are no longer appropriate: for instance, using dynamic programming style optimization is not efficient; limiting consideration to left-deep plans (using DPAP-LD) usually misses the best solution. Our experiments also show that the DPP algorithm can find the optimal solution, with low cost relative to the DP algorithm; and the FP algorithm can very quickly choose a plan that in most cases is close to optimal. Our recommendation is that DPP should be used in XML query optimizers where query execution time is expected to be significant, and that FP should be used where it is important to find a good (but not necessarily the best) plan quickly.

## 1 Introduction

As XML [4] has gained prevalence in recent years, the storage and querying of XML data has become an important issue. Effective query optimization is crucial to obtaining good performance from an XML database given a declarative query specification.

A join is frequently the most expensive physical operation in evaluating a relational query. Thus, selection of join order is a key task for a relational query optimizer. This observation is true for an XML query optimizer as well, but with significant twists. Perhaps the most important of these is the prevalence of *structural joins* in XML. The design of the structural join order selection algorithm, which is the kernel of an XML query optimizer, is the focus of this paper.

A join in the relational context is usually a value-based equi-join, which involves two tables and is based on the values of two columns, one in each table. In the XML context, even though there are value-based joins, *structural joins* occur much more frequently. A *structural join* focuses on the containment (ancestor-descendant or parent-child) relationship of the XML elements to be joined. The join condition is specified not on the value of the XML elements, but on their relative positions in the XML document.

In short, queries on XML data have some features that are different from queries in the traditional relational context. Therefore, the set of alternative plans, and their relative costs, in the XML context are also quite different. These structural join characteristics, and their impact on query processing, hold no matter how XML data is stored: in a native XML database, after mapping to a relational database, or after some other mapping, such as to an object-oriented database. Our task in this paper is to develop a framework for XML query optimization. The specific access methods used, and their costs, will be a function of the specific implementation.

Using simple cost models for structural join algorithms, we develop five XML query optimization algorithms for structural join order selection. The first algorithm we present uses a Dynamic Programming (DP) style optimization to exhaustively explore every plan in the search space, including bushy query plans. Since exhaustive searching of the entire search space can be costly in terms of the optimization time, we also develop enhancements to the DP algorithm. The enhanced algorithm, which is called Dynamic Programming with Pruning (DPP), considers an incomplete query plan and uses pruning techniques to eliminate partial query plans that are *guaranteed* to lead to suboptimal solutions.

Even after the pruning enhancement in DPP, the optimization process may remain costly. More aggressive pruning can be done to reduce the search space, and hence reduce the time spent in the optimization process. Choices of the additional pruning rules for the Dynamic Programming with Aggressive Pruning (DPAP) can be based on heuristics in different directions. A straightforward heuristic is to limit the number of intermediate results considered, which leads to DPAP-EB algorithm. Drawing inspiration from relational query optimizers, which frequently restrict their search to only left-deep plans, the additional pruning rule can restrict the shape of the intermediate result to be left-deep. We call this algorithm DPAP-LD.

Taking special features of XML data and XML queries into consideration, we introduce a final algorithm,

called the Fully-Pipelined (FP) algorithm, which only considers non-blocking query plans and is guaranteed to select the cheapest non-blocking query plan.

We present extensive experimental evaluation of the proposed optimization techniques in the Timber[21] native XML database system being developed at the University of Michigan. We demonstrate that many relational rules of thumb are no longer appropriate. For instance, restricting consideration to left-deep plans is not a good idea. Our experiments also show that while both DP and DPP algorithms search the entire solution space and select the optimal solution, the DPP algorithm is much more efficient. In addition, heuristic techniques (DPAP) can be introduced to speed up the DPP algorithm even further, at the cost of potentially missing out an optimal plan. The FP algorithm, which exploits features unique to XML data and XML queries, can quickly choose a non-blocking plan that in most cases is close to optimal.

The main contributions of this paper are the development of a framework for cost-based structural join order selection for XML query optimization, the development of five optimization algorithms, and an evaluation of the algorithms using an actual implementation. We show that the DPP algorithm is the algorithm of choice for XML query optimizers that want to explore the entire search space and select the optimal structural join plan. The FP algorithm very efficiently finds the optimal non-blocking plan, which could be valuable in systems that want quick and easy optimization solution, such as to support online querying on XML data sources.

The remainder of this paper is organized as follows. We begin by providing a brief tutorial, in Sec. 2, on XML queries, pattern matching, and structural join algorithms, and formally defining the objective of structural join order selection in XML query optimization. The five new algorithms we propose are discussed in detail in Sec. 3. Then, we present extensive experimental evaluation of alternative optimization techniques in Sec. 4. Finally, we conclude with a discussion of related work in Sec. 5 and directions for future work in Sec. 6.

## 2 Problem Definition

Declarative querying is a central feature of modern database systems. Given a query expressed in a declarative query language, a query optimizer has the task of developing alternative plans to evaluate the query, and then choosing one amongst these that is expected to be of the least cost. In the relational world, often the most important optimization step is join order optimization. The counterpart in XML is structural join order optimization. In the overall relational query optimization process, techniques such as selection push-down, projection push-down, etc. are also valuable, and we fully expect that such techniques will have similar uses in a complete XML query optimizer [7]. However, considering all these aspects is beyond the scope of

this paper, and we only concentrate on structural join order optimization, which, like its counterpart in the relational world, is likely to be at the heart of any complete XML query optimizer. We provide below the necessary background in structural join computation, and its role in XML query processing.

## 2.1 Pattern Matching

The XPath expressions used to bind variables in XQuery, along with the conditions in the WHERE clause, can be expressed as the matching of a query pattern tree in a database [8, 1]. In formal terms:

Given a rooted node-labelled tree  $T = (V_T, E_T)$ , representing the database.

A *query pattern* is a smaller, rooted, node-labelled tree  $Q = (V_Q, E_Q)$ . The labels at the nodes of  $Q$  are boolean compositions of predicates. Edges of  $Q$  may optionally also be labelled, with a \*, to specify the ancestor-descendant relationship between the nodes.

A *match* of a pattern query  $Q$  in  $T$  is a total mapping  $h : \{u : u \in Q\} \rightarrow \{x : x \in T\}$  such that:

- For each node  $u \in Q$ , the predicate node label of  $u$  is satisfied by  $h(u)$  in  $T$ .
- For each edge  $(u, v)$  in  $Q$ ,  $h(v)$  is a descendant (or child) of  $h(u)$  in  $T$ .

To *evaluate* a query is to find all the matches of a query pattern in the database.

## 2.2 Structural Join Algorithms and Cost Models

A structural join operation can be evaluated by a database system using a number of different algorithms. There already exist a wide array of access methods for structural join computation [1, 5, 17] and we expect new ones to be invented in the future. From the perspective of the design of the query optimizer algorithm the actual choice of these algorithms is not crucial. (In fact, it does not even matter whether the algorithms are implemented in a native XML database or through mapping to a relational system). Of course, the cost models for these algorithms and the effect of any physical properties (such as producing sorted outputs) are needed to compute the cost of intermediate query plans and to pick the optimal query plan.

### 2.2.1 Structural Join Algorithms

Given a query pattern, it is usually reasonable to assume that candidate matches for individual query nodes can be found efficiently, for instance, through an index scan. Consider an edge  $(u, v)$  in a query pattern. This edge represents a structural inclusion relationship between the elements represented by nodes  $u$  and  $v$ . This inclusion can be specified to be either immediate ( $u$  is the parent of  $v$ ) or arbitrary ( $u$  is an ancestor of  $v$ ). Having obtained two lists of candidate nodes that satisfy any predicates associated with each of  $u$  and with  $v$  individually, a structural join outputs pairs that satisfy the required inclusion relationship.

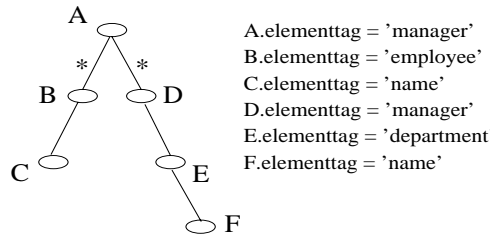


Figure 1: Example Query Pattern

**Example 2.1** Consider the query in Figure. 1, accessing an index built on the element tag names gives us a list of candidate data nodes for each node in the query pattern. Performing structural join operation on a pair of pattern tree nodes, for instance, node A (manager) and node B (employee), with the relationship specified (\*), the results are pairs of data nodes (e.g. manager and employee) where each manager node in a pair is the ancestor of the employee node in the same pair, in the XML database.

A critical issue for the physical join operator is the order in which data sets are input and output. For most structural join algorithms, it is most useful to order inputs by the structural positions of the nodes in the original data set. A common structural ordering scheme that is often used is to number the nodes using a depth-first pre-order node numbering scheme [17, 21]. The output can be ordered by either node participating in the join. For convenience, we call the output *ordered by ancestor* when it is ordered by the including node ( $u$ ), even if the structural join is based on direct parent-child relationship. When the output is ordered by the included node ( $v$ ), we say it is *ordered by descendant*.

We focus on the “Stack-Tree” family of algorithms[1], because they are currently amongst the best algorithms for computing binary structural joins. (The introduction of other binary structural join computation algorithms will not change the design of the query optimizer: rather, it will simply be folded in as a new algorithm with its associated costs.) These algorithms are applicable to both relational implementations of XML storage and native XML systems. These algorithms have properties that are similar to the relational sort-merge join algorithm. Like the relational sort-merge join algorithm, these algorithms take as input two data sets that are sorted by the pre-order node number, and can produce output that is ordered by either the ancestor or descendent. The Stack-Tree-Ancs algorithm, in this family, produces output ordered by ancestor, whereas the Stack-Tree-Desc algorithm produces output ordered by descendant.

Irrespective of the implementation specifics, the cost of an access method implementing either algorithm is a linear function of the sizes of the inputs and the size of the output. In other words, if we have estimates of intermediate result sizes, we can obtain cost estimates for these access methods as linear functions of these sizes. The specific constants used in the linear functions depend on system specifics.

### 2.2.2 Cost Models

In addition to the two algorithms of focus for structural joins, there are two other key physical operations: *index access* and *sorting*. The cost formulae for these operations are as follows:

**Index Access** The cost for accessing index and retrieving  $n$  items is  $f_I \times n$ .

**Sort** The cost for sorting a list of  $n$  items is  $n \log n \times f_s$ .

**Structural Join** The cost for joining two nodes A and B (where node A is ancestor of node B) in the pattern depends on the join algorithm used. In addition to disk I/O, the majority of the CPU cost goes towards operations on an in-memory stack, as discussed in [1]. The cost formulae of the Stack-Tree algorithms are as following (with cardinality of the node A expressed as  $|A|$ , the join results of A join B expressed as  $AB$ ):

- Stack-Tree-Ancs:  $2 \times |AB| \times f_{IO} + 2 \times |A| \times f_{st}$ .
- Stack-Tree-Desc:  $2 \times |A| \times f_{st}$ .

Each implementations of an XML database would have different constants associated with the cost of each physical operations. We use a set of factors ( $f_I$  for index access,  $f_s$  for sorting,  $f_{IO}$  for disk IO, and  $f_{st}$  for stack operations) to normalize the cost of different operations when these are to be compared or added.

### 2.3 Structural Join Order Selection

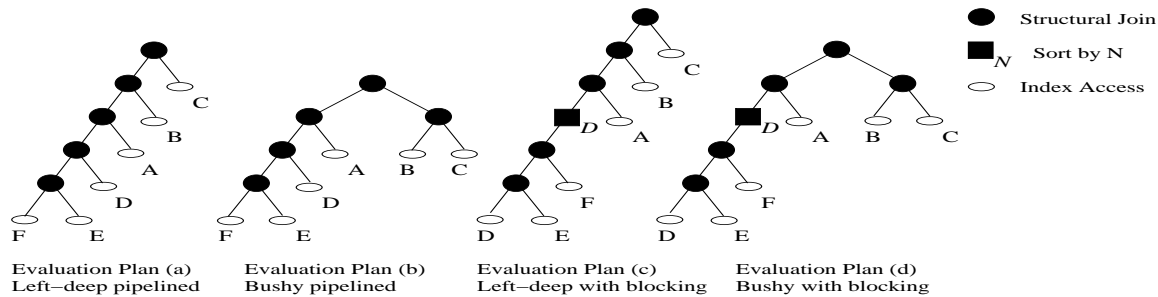


Figure 2: A Few Plans that Evaluate the Example Query Pattern

**Example 2.2** Consider a query in a personnel XML document, which asks for “for each manager A, list the names of the employees supervised by A, and the name of any department that is directly supervised by another manager, who is a subordinate of A.” The query defines a pattern tree, shown in Figure. 1, to be matched.

One possible evaluation plan for this query is to get all manager nodes (candidate for A) from an index, then, scan the sub-tree of each of these nodes in database, looking for other nodes that satisfy both the predicate and relationship condition. Notice that the relationship specified for A-B and A-D pair in the

pattern are all ancestor-descendant relationships. This means, to find all the pattern matches, the whole subtree for each manager node, no matter how deep it is and how many elements it contains, will be scanned thoroughly. The performance can be very bad when the XML data is deep and nested.

Evaluation of this query using structural join progresses as follows. First, one identifies all manager nodes (candidate matches for nodes A and D in the pattern), all employee nodes (candidates for B), all department nodes (candidates for E) and name nodes (candidates for C and F) in the database. Then, five structural joins are computed, one for each edge in the pattern, between these six sets of candidates. The results are a set of 6-node tuples. Each node in such a tuple satisfies the node predicate of one node in the pattern, and the relationship between the nodes satisfies the relationship specified in the pattern between corresponding nodes.

The five joins can be evaluated in different orders. The number of alternative plans is to the factorial order of the number of nodes in the pattern. Figure. 2 shows a few of these evaluation plans. A plan can be left-deep, as in Figure. 2(a,c), or bushy, as in Figure. 2(b,d); fully pipelined, as in Figure. 2(a,b), or with blocking, as in Figure. 2(c,d).

Given a query pattern, there are many ways to find matches in the database. The shape of the query pattern determines how many alternative evaluation plans it can have, and what they are. Given a pattern, it is not clear which plan has the minimum cost. It is the task of the query optimizer to pick the optimal (or at least a good) evaluation plan, based on the estimated cost of each alternative plan it considers.

An *evaluation plan* is a rooted, labeled tree  $P = (V_P, E_P)$ . Each node in the tree is a physical operation using a specified access method. Each *evaluation plan* has a *cost* associated with it, which represents the time and space requirement to evaluate the query using this plan.

Our goal is to define an algorithm that can find the evaluation plan for a given *query pattern* which has the minimum *cost* among all the plans that can evaluate the query pattern, using only some summary information about the database. The time required to choose an evaluation plan (run this algorithm) should be only a small percentage of the time required to execute the plan chosen.

### 3 Structural Join Order Selection Algorithm

For any given query, there usually are a very large number of possible query plans. We need an effective means to choose the best of these. Following the dynamic programming approach used for relational join order selection, we first develop a Dynamic Programming (DP) algorithm. While this algorithm follows the same style as the relational dynamic programming algorithm, and searches the entire solution space,

including bushy plans, there are substantial differences on account of structural joins being different than relational value joins.

Based on these special features, it is possible to identify non-promising structural join plans early in their evolution. Thus, some plans can be eliminated before they are fully developed, and the search limited to a narrow band along the optimal path. The resulting Dynamic Programming with Pruning (DPP) algorithm still selects the optimal solution in the entire solution space, since it only prunes out consideration of paths guaranteed not to lead to the optimum.

More aggressive pruning rules can be introduced to reduce the search space, based on heuristics. We call algorithms in this category Dynamic Program with Aggressive Pruning (DPAP). Faster optimization times can be obtained, but the optimum is no longer guaranteed.

A very straightforward heuristic is to assume it unlikely that a bad intermediate plan will evolve into a good final plan. Based on this heuristic, we limit the number of intermediate plans generated and develop the DPAP-EB algorithm. (An extreme case of DPAP-EB, where the number of intermediate plans is limited to one, reduces to the greedy heuristic).

Relational query optimizers frequently restrict themselves to left-deep plans since, in the relational context, the optimal evaluation plan is usually left-deep. Imposing such a restriction is straightforward even in the XML context. (Whether the resulting solutions are good is a different question, and we will argue later that this restriction is not a good choice). Using this heuristic, we present the DPAP-LD algorithm by making appropriate modifications to the DPP algorithm.

Pipelining is an important feature of database query evaluation. Query optimizers attempt to minimize the number of “blocking” operators – trying to build a pipelined evaluation plan in as far as is possible. In fact, in the relational context, one reason why left-deep plans are preferred is because most sequences of joins involve joining on different attributes, and hence require ordering the data differently for each join; in a left-deep plan, at least the right-hand join operands are all base data, and can be ordered/prepared in advance so as to minimize blocking in the main pipeline.

It turns out that it is *always* possible to find a bushy non-blocking join plan for XML structural joins. In Sec. 3.4 we introduce the FP algorithm and show how to enumerate plans with no blocking, and choose the best from amongst them.

For each of the five algorithms, we develop cost formulae to estimate the number of plans evaluated. We show that the first four have to consider a number of plans that is exponential in the query size, whereas the last one is much cheaper.



### 3.1 Searching the Entire Solution Space Using Dynamic Programming (DP) Technique

The traditional dynamic programming algorithm searches the entire solution space by computing the minimum cost evaluation plan for each combination of the relations participating in the query. It does so progressively, using the minimum cost plans for smaller combinations to determine the minimum cost for larger ones. The same idea can be applied in the XML context with some twists.

#### 3.1.1 Defining Status

In order to keep track of each partial structural join plan, we introduce the notion of *status*. A *status* defines an intermediate stage of query evaluation, in which the structural relationships between some pattern tree nodes have been evaluated, while others remain unresolved. This partitions the nodes in the pattern tree into some subsets, each representing a sub-pattern that has been evaluated. A *status node* is used to represent each such joined sub-pattern.

**Definition 1** Given a query pattern  $Q = (V_Q, E_Q)$ , a status node  $N_S$  in a status derived from  $Q$  is a cluster of nodes in  $V_Q$  that satisfies the following formulae:

- $N_S \subseteq V_Q$ ;
- $\forall u, v \in N_S \wedge w$  on path from  $u$  to  $v$  in  $Q \Rightarrow w \in N_S$ ;

In each status node, one pattern tree node is called the *SortBy node*, which specifies by which node the intermediate join results of the sub-pattern are ordered. Note that this *SortBy* node may be different from the *OrderBy* node of the last structural join operation in the sub-plan which evaluates the sub-pattern represented by the status node.

**Definition 2** Given a query pattern  $Q = (V_Q, E_Q)$ , A status is a tree  $S = (V_S, E_S)$ , where

- $V_Q = \{v | v \in N_S\}$ .
- $\forall N_S, N'_S \in V_S \Rightarrow N_S \cap N'_S = \phi$ ;
- $\bigcup_{N_S \in V_S} N_S = V_Q$ ;
- $E_S \subseteq E_Q$ .
- $\forall N_S \in V_S, \forall u, v \in N_S \Rightarrow (u, v) \notin E_S$ .

The query pattern  $Q$  is a status itself, and is called *start status*, represented by  $S_s$ . When  $V_S = \{\{V_Q\}\}$ , the status is called *final status*, represented by  $S_f$ . All other statuses are called *intermediate statuses*.

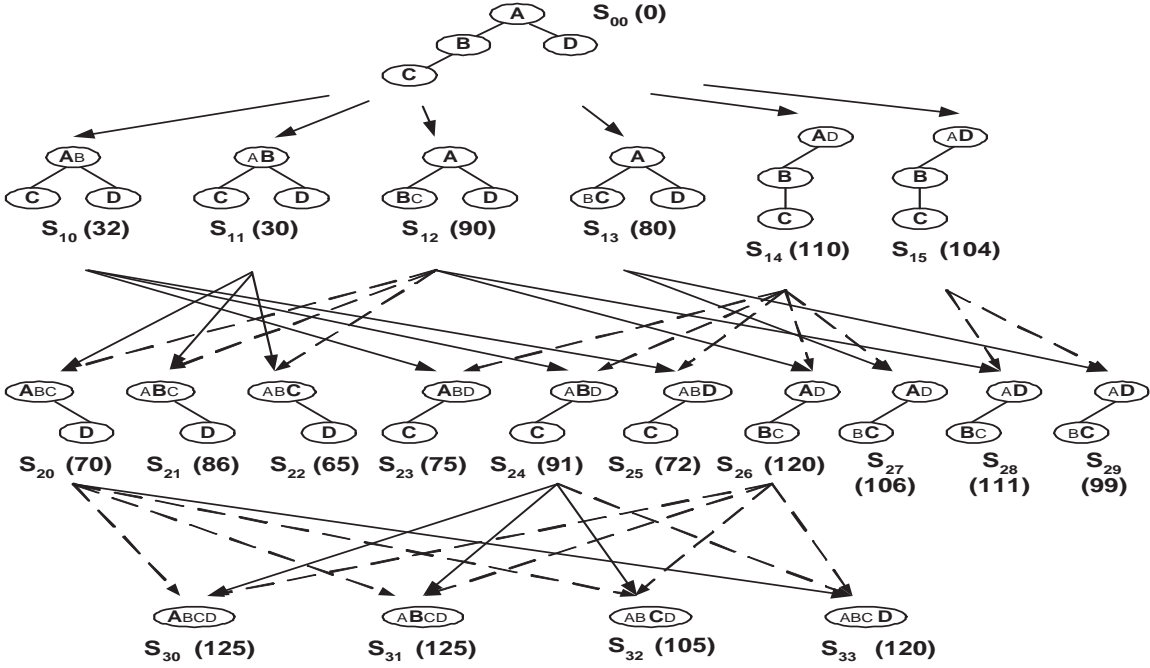


Figure 3: Example Optimization Process Using DP algorithm

**Example 3.1** Figure. 3 is a very simple example of the optimization process using the DP algorithm. Status  $S_{00}$  is the start status, each status node in it contains one pattern tree node. Status  $S_{00}$ ,  $S_{31}$ ,  $S_{32}$ ,  $S_{33}$  are all final statuses. In each of those statuses, there is only one status node, which contains all the pattern tree nodes in the original query pattern. In an intermediate status, some status node may have more than one pattern tree nodes. For example, the root node in status  $S_{00}$  has two pattern tree nodes, A and B, and the SortBy node is A. (The SortBy node is presented in bold in the figure.)

**Definition 3** Associated with each status  $S$  is a Cost value. The Cost is the accumulated cost of the operations needed to evaluate all the sub-patterns represented by the status nodes in  $S$ .

In other words, for a given status  $S$ , the Cost is the accumulated cost needed to transform from the start status  $S_0$  to  $S$ .

There are many intermediate statuses between the start status and the final status. A *move* connects two statuses that differ in only one edge. The physical operations associated with each move include: (1) A structural join operation, which joins the two nodes at the end of the edge and produces a larger joined sub-pattern. The choice of join algorithm determines by which node the intermediate results is ordered. (2) An optional sort operation, which sorts the intermediate results by a node other than the *Orderby* node specified by the structural join algorithm used.

**Definition 4** A move  $M$  from status  $S$  is a vector  $(aN, dN, Algo, St, Cost)$ , where  $aN$  and  $dN$  are pattern

tree nodes and  $(aN, dN) \in E_S$  is the edge to be evaluated; *Algo* specifies the physical operator; *St* is the node to be sorted by, if extra sorting is needed; and *Cost* is the estimated cost of the join (plus sorting cost, if *St* is specified).

The relationship of a *move* and its two end *statuses* can be stated as following:

**Definition 5** A move  $M = (aN, dN, Algo, St, Cost)$  transforms status  $S_1 = (V_{S_1}, E_{S_1})$  to status  $S_2 = (V_{S_2}, E_{S_2})$  iff

- $(aN, dN) \in E_{S_1}$  and  $E_{S_2} = E_{S_1} - \{(aN, dN)\}$ ;
- $\exists u, v \in V_{S_1}, aN \in u \wedge dN \in v \Rightarrow V_{S_2} = (V_{S_1} - \{u, v\}) \cup \{u \cup v\}$ .
- status node  $N_S \in V_{S_2} - V_{S_1} \Rightarrow N_S.SortBy = St$ ;
- $S_2.Cost = S_1.Cost + Cost$ .

For a given query pattern, the number of moves needed to transform the start status to the final status equals to the number of edges in the pattern. Each move processes one edge in the query pattern and takes one step towards the final status, in which all edges have been processed.

Starting from a given status  $S$ , there is a set of possible moves that transform the status into a set of statuses that are one step closer to the final status. We call this set of moves *possible moves* and represent it using  $pM(S)$ .

**Definition 6** Given a status  $S = (V_S, E_S)$ , possible moves  $pM(S)$  are a set of moves defined as  $pM(S) = \{(aN, dN, Algo, St, Cost) | \exists N_a \in S(aN = N_a.SortBy) \vee \exists N_d \in S(dN = N_d.SortBy)\}$ . *Algo* is the algorithm chosen to carry out the move, *St* can be left empty, or be the node in  $N_u \cup N_d$ , and is not specified as the *OrderBy* node in the *Algo*.

*St* can be left empty, which means the ordering of the result is determined by the *Algo*, or can be set to be any other node in the new joined-sub-pattern generated after this move.

**Example 3.2** In Figure. 3, the six moves from status  $S_{00}$ , each deals with one edge in the status, transform status  $S_{00}$  into status  $S_{10} \cdots S_{15}$ , respectively.

### 3.1.2 Search Structure

$pM(S)$  contains all the possible moves that can be made from a given status  $S$ . The consideration of all moves in  $pM(S)$ , to estimate the desirability of each resulting new status is called *expanding* status  $S$ .

Our goal is to find a sequence of moves that transforms the start status  $S_0$  to the final status  $S_f$ , with the total cost of the moves the lowest among all sequences of moves that can achieve the same transformation. The set of possible sequences of moves is searched using dynamic programming technique.

**Definition 7** *Statuses are arranged in a topological graph, with start status as the root. Statuses that are  $k$  moves from the start status are said to be on level  $k$ .*

The searching is done one level at a time. No status on level  $k$  is generated until all possible solutions for reaching each status on level  $k - 1$  are checked and the best join plan for each status on level  $k - 1$  is found.

After the best plan for each status on level  $k - 1$  is found, the statuses on level  $k - 1$  will be picked and expanded one by one. All statuses on the same level have equal priority for picking and expansion. For each status on level  $k - 1$  that is picked, all possible moves are considered, and new statuses, which belong to level  $k$  are generated. A unique status on level  $k$  may be generated more than once, based on different statuses from level  $k - 1$ . The costs of all these plans are compared and only the one with minimum cost is kept. All others are eliminated from further consideration.

**Example 3.3** *Figure 3 demonstrates the search process using DP algorithm. The statuses on the same level are arranged in a row. A Cost value is associated with each status. It is presented in the parenthesis following each status ID.*

*After the search is done, all statuses have been checked and expanded according to the possible moves of the statuses. For example, status  $S_{12}$  has 5 possible moves. The new statuses generated are  $S_{20}$ ,  $S_{21}$ ,  $S_{22}$ ,  $S_{26}$  and  $S_{28}$ . Also, one status can be generated from more than one status on the level just above it. For instance,  $S_{30}$  can be generated from  $S_{20}$ ,  $S_{24}$ , or  $S_{26}$ . For a given status, the Cost can be different if it is generated from different statuses. The costs associated with same status are compared against each other and only the minimum becomes the cost of the status, and the path is kept (we represent these kind of paths with a solid line and the ones eliminated with a dotted line in the figure).*

*There is more than one final status when the search is completed. Each has the desired pattern match result, ordered differently, due to the SortBy node being different. The costs of the final statuses are compared and the one with minimum cost is picked.*

*For example, let's assume that  $S_{32}$  has the minimum cost among all the final statuses. Tracking back along all the moves, starting from the final status picked ( $S_{32}$ ), all the way back to the start status ( $S_0$ ), we obtain the optimal solution to evaluate the query pattern. The structural join plan selected is to join node A with node B first, ordered by node A, then, join node D, sort the results by node B, finally, join the intermediate result with node C and ordered by node C.*

Consider a pattern with  $n$  nodes. The number of statuses generated during the searching process is  $O(n \times 2^n)$ . The number of alternative plans considered is  $O(n^2 \times 2^n)$ .

### 3.2 Dynamic Programming with Pruning (DPP) Algorithm

**Example 3.4** *In Figure 3, as we can see, lots of nodes that are not potentially good are expanded, only because they can be generated from a status one level above. For example, status  $S_{26}$  does not contribute to any path that leads to the final status, in fact, its cost is larger than that of the final status with minimum cost. So, there is no way that a good plan can be generated from  $S_{26}$ . However, in the optimization process using DP algorithm,  $S_{26}$  is generated and all its possible moves are tried.*

The complexity of the DP algorithm is exponential in the number of nodes in the pattern. As we can see from the example above, some of the structural join plans are far more costly than the optimal; the cost to carry out a part of the plan, even a small portion of it, may be more than the total cost of the optimal plan. Taking this observation into consideration, we propose the DPP algorithm, Dynamic Programming with Pruning.

Our goal here is to find the optimal solution in a more efficient way. Conceptually, we still do exhaustive search in the set of possible sequences of moves. However, we want to limit the search to a narrow band along the optimal path. This is done by (i) setting a priority list for the statuses and no longer requesting that one level be fully developed before the searching is moved on to the next level; (ii) pruning intermediate plans that are *guaranteed* to lead to suboptimal solutions.

Besides *Cost*, the actual cost of the operations, we introduce, for each status, another cost value, *ubCost*, which is the upper-bound estimation of the cost needed for transforming the status to the final status. All un-expanded statuses are arranged in a priority list, ordered by the value of  $Cost + ubCost$ . The Status at the head of the priority list, which has the lowest  $Cost + ubCost$  is always expanded first. A sub-plan is no longer expanded when other sub-plan with lower cost is found to reach the same status. The expansion of a sub-plan is terminated when the Cost of the status it reaches exceeds the lowest cost of a final status that has been reached.

It is a reasonable assumption that a sub-plan with lower cost is likely to lead to low cost solution. By always expanding sub-plans with lowest  $Cost + ubCost$ , promising plans grow much faster than plans with larger  $Cost + ubCost$ . Therefore, plans for complex sub-patterns can be generated much earlier than in DP algorithm. Since the plans for complex sub-patterns are generated based on cheap sub-plans for smaller sub-patterns, it is likely that the plans for the complex sub-patterns are also cheap. This speeds up the generation of the first full structural join plan for a query pattern. Also, whenever a full plan is generated, its total cost

is computed and used to eliminate more bad plans before they have a chance to grow larger.

Another thing worth noticing is that many statuses generated in the DP algorithm have  $pM(S) = \phi$ . That is, they do not lead to any possible solution, let alone a good solution.

**Definition 8** A status  $S$  is a deadend if the possible moves  $pM(S) = \phi$ .

A status is deadend if none of the edges left in the status has the status nodes at both ends of it sorted by the end node of the edge. In the DP algorithm above, these deadend nodes are generated but are useless for the optimization. The generation of deadend nodes can be prevented by looking ahead one step at the time of expanding a status.

**Example 3.5** In Figure 3, more than half of the nodes on the level above the last level have no outgoing move. These nodes are all deadends. With the Lookahead Rule, all these nodes could be detected and not be generated at all.

The Expanding Rules and Pruning Rules of DPP algorithm can be summarized as following:

- **Expanding Rule** Always expand the status with lowest  $Cost + ubCost$ ;
- **Pruning Rule** A status  $S$  is “dead” if the cost of the path from  $S_0$  to  $S$  exceeds the lowest path cost from  $S_0$  to  $S_f$  (recall  $S_f$  is a final status) . No status is “dead” before one such path is found. A status is eliminated from further consideration when it is found to be “dead”.
- **Lookahead Rule** At the time of expanding one status, a new status would not be generated if it is a deadend.

Since  $ubCost$  only provides a guideline for the expanding priority of statuses, the quality of the estimate does not effect the optimal property of the result solution, but only the efficiency of the optimization algorithm itself. The better the estimate, the narrower the search band along the optimal path. The DP Algorithm is a special case of the DPP algorithm with  $ubCost$  set to 0 for all statuses. For DPP, the  $ubCost$  for a status can be obtained easily and quickly by computing the cost of the join operations for each un-joined edge in the status in a bottom-up fashion, plus sorting cost, when necessary.

Consider a pattern that is a complete tree with depth  $d$  and fixed fan-out  $f$ . The total number of edges is  $|E| = \sum_{k=1}^d f^k$ . The number of possible statuses on level  $lv$  is  $S_{A^*}(lv) = f_{S_{A^*}} \times \binom{lv}{|E|}$ , where  $f_{S_{A^*}} = O(lv^2)$  is the number of OrderBy nodes in each status nodes. The upper-bound for the total number of statuses evaluated in the searching is  $\sum_{lv=0}^{|E|} S_{A^*}(lv)$ .

### 3.2.1 Example Optimization Process

Let's look at the example shown in Figure. 4 and see how the DPP algorithm finds the optimal evaluation plan for the query. In Figure. 4, we number the statuses in the order they are generated. The lookahead rule is applied to reduce the number of statuses generated.

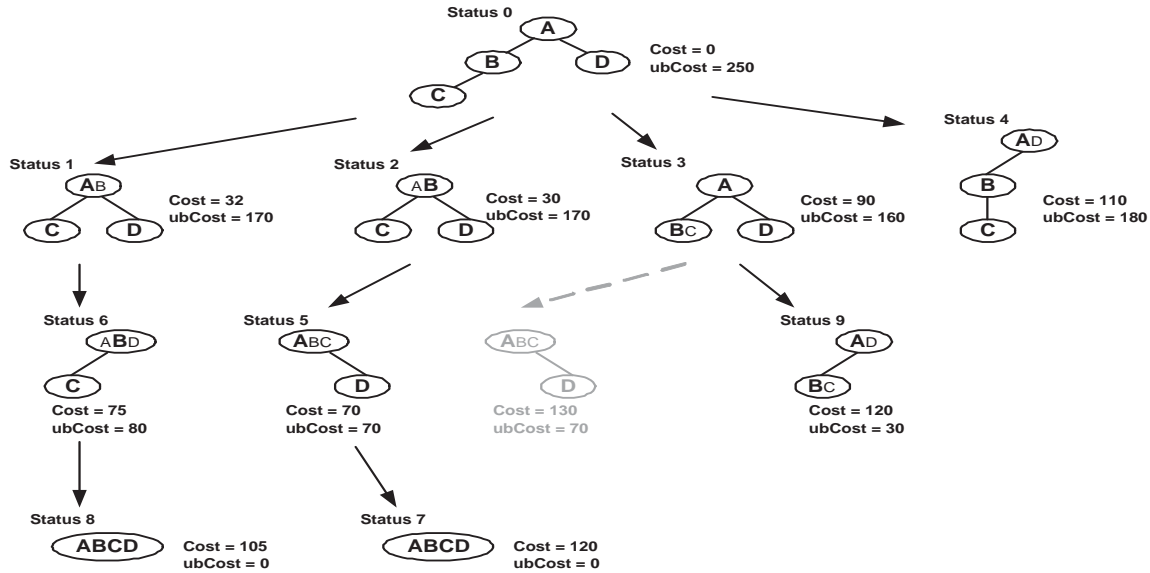


Figure 4: Example Optimization Process Using DPP algorithm

**Example 3.6** Status0 has four possible moves, which transform status0 to status1,2,3,4, respectively. Note that in Figure 3, there are other possible moves, which transform status0 to a “deadend”. No “deadend” node is generated here since we are using DPP algorithm with Lookahead Rule.

Among the new statuses, status2 has the least  $Cost + ubCost$ . So, it is expanded next and Status5 is generated. Then, the status with the lowest  $Cost + ubCost$  is status1. Expanding status1 produces status6. Now, the status with lowest  $Cost + ubCost$  is status5. Expanding status5, we get status7. Note that only one status is generated in this expansion, since the new status is the final status and we don't care about the ordering any more.

Now, we have found one path from the start status to the final status. The Cost of status7 is recorded as the current minimum cost ( $MinCost$ ). But the search has not finished, there are still other statuses that may lead to a better solution. Status6 is selected next. Expanding status6, we reach another final status, status8. We find that the Cost of status8 is smaller than that of status7. Therefore, status7 is eliminated and  $MinCost$  is set to 105, the Cost of status8.

Next, we expand status3. Two new statuses could be generated. However, we detected that one of the newly generated status is exactly the same as status5, except that its Cost is higher. This status is discarded

right away. Status9 is generated since no such status has been generated so far.

The next status on the priority list is status9. Before expanding it, we find that the Cost of status9 is larger than the MinCost. This means, status9 is “dead”. Status4 is checked next, it is also “dead”. Now, there is no un-expanded statuses. The search is over.

The evaluation plan chosen for the query is the moves along the path from status0 to status8, the only final status left in the search process. As we can see, the structural join plan selected by DPP algorithm is exactly the same as the one selected by DP algorithm.

### 3.3 Dynamic Programming with Aggressive Pruning (DPAP) Algorithm

Additional pruning rules can be introduced into the DPP algorithm to eliminate less promising portions of the search space, thereby decreasing the optimization cost considerably in return for, hopefully, a small risk of eliminating the optimum solution. Various heuristics can be devised for this purpose. We describe two possibilities below.

#### 3.3.1 Dynamic Programming with Aggressive Pruning on Expansion Bound (DPAP-EB) Algorithm

One straightforward heuristic is to limit the total number alternatives evaluated. We considered several such heuristic pruning parameters, including the depth of the expanding, the number of statuses created at a level, and so on. Of these, we describe here only the one parameter that we empirically found to be most effective.

The parameter  $T_e$  restricts how many statuses can be expanded at each level. This is based on the heuristic that a good sub-plan has a higher chance of leading to an approximation of the optimal solution for evaluating a query pattern. In other word, if a sub-plan is costly, it is less likely it can be expanded to a good plan. With parameter  $T_e$ , when the number of statuses expanded at level  $lv$  reaches the limit  $T_e$ , there is no point creating any more statuses at this level, so no more statuses will be expanded at any level less than  $lv$ . This restriction brings the upper-bound for the total number of statuses considered down to  $\sum_{lv=0}^{|E|} T_e \times (|E| - lv) \times lv$ .

**Example 3.7** Considering the optimization process in Figure 4. If parameter  $T_e$  is set to be 2, status3 and status 4 will not be generated in the expansion process of status0. Therefore, status9 will not be considered in the search process, too. In this case, with  $T_e$  setting to 2 can still results in the optimal solution. However, it is not always true for other queries and other settings.



### 3.3.2 Dynamic Programming with Aggressive Pruning on Left-Deep Solution Space (DPAP-LD) Algorithm

Relational query optimizers frequently restrict consideration to left-deep plans. It is straightforward to modify the DPP algorithm with a restriction to the expanding rule so that only left-deep plans are considered.

- **Left-deep Expanding Rule** In any status, only one status node is allowed to comprise multiple pattern tree nodes. We call this status node the *growing node*.

This means a potential move can only be based on evaluating an edge with one end in the *growing node* and one end out of it.

**Example 3.8** *In the example we presented for the DPP algorithm above (Figure. 4), when the left-deep expanding rule is applied, the optimization process remains the same, except that status9 is not legal (not left-deep) and would not be produced. In this simple example, DPP algorithm and DPAP-LD algorithm find the same solution.*

Consider a pattern that is a complete tree with depth  $d$  and fan-out  $f$ . The upper-bound for total number of statuses considered in the search is  $O(|E|^f d)$ .

### 3.4 Fully-Pipelined Solution Space

By choosing an appropriate structural join algorithm, the results of a structural join can be output ordered by either of the two nodes involved in the join. No extra sorting is needed, and no blocking points created in the pipeline, if the *OrderBy* node in one join is a node involved in the next join. This leads to the following:

**Theorem 3.1** *Any XML pattern match can be evaluated with a fully-pipelined evaluation plan to produce results ordered by any node in the pattern tree.*

**Proof Sketch:** Prove by induction on  $n$ , the total number of edges in a pattern. For the base case, the theorem obviously goes through for a query pattern with a single node and zero edges. For the inductive case, we can show that there is at least one pipelined plan, whose last join involves a sub-pattern which contains the result *OrderBy* node  $r$ , and a sub-pattern which contains one of its neighbors  $u$ . Each of these sub-patterns has less than  $n$  edges. By the inductive assumption, there is a pipelined plan for the first sub-pattern with results ordered by  $r$  and a pipelined plan for the second sub-pattern with results ordered by the neighbor node  $u$ . ■

The set of fully-pipelined evaluation plans for a given query pattern is a very small subset of all the evaluation plans for the query pattern. Furthermore, full-pipelining is a good feature we look for in relational

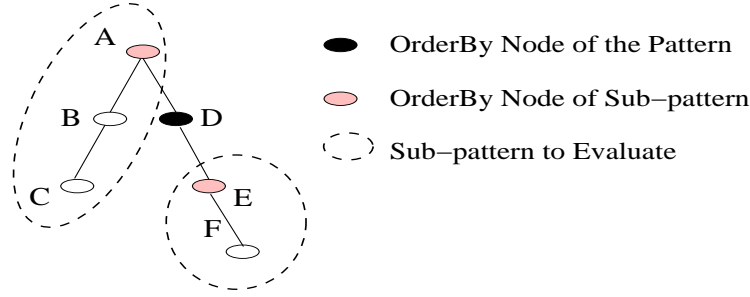


Figure 5: Pattern Tree Example in FP algorithm

query optimization. It is not unreasonable to make the assumption that the best plan in this subset is an approximation of the best plan in the whole solution space. Figure. 9, in the appendix, is the pseudo-code of the FP\_optimization algorithm, which looks for the best fully-pipelined evaluation plan for a given query pattern by recursively breaking it down into smaller sub-patterns.

**Example 3.9** Consider the pattern tree in our running example, as shown in Figure. 5. To find the best fully pipelined join plan to evaluate the query and guarantee the output is ordered by the node in black, the best join plans to evaluate two sub-patterns, with output ordered by the nodes in grey, need to be generated. Then, the order of the two sub-patterns joining with the node in black is selected by enumerating all the possible permutations. The best plan to evaluate the whole pattern is chosen from a set of plans, each of which evaluates the whole pattern and guarantees that results are ordered by one node in the pattern.

Consider a pattern that is a complete tree with depth  $d$  and fixed fan-out  $f$ . The total number of alternative plans considered in searching for plans to evaluation the pattern, without specifying by which node the results should be ordered by, is  $O((\sum_{k=0}^{d-1} f^k)^2 \times f!)$ .

Frequently, an *OrderBy* node is specified for a query pattern, which requires that the result be ordered by a certain node, to facilitate other operations following the pattern matching. In this case, the total number of alternative plans considered in the FP algorithm is only  $O(\sum_{k=0}^{d-1} f^k \times f!) = O(|E| \times (f - 1)!)$ .

## 4 Experimental Evaluation

In this section, we present an experimental evaluation of the various query optimization techniques discussed. All experiments were run on a machine with a 500MB Intel Pentium III processor, 512MB of memory and a 40GB ATA Compaq Hard Disk Drive hard drive. The buffer pool size used by SHORE (the storage manager that Timber uses) was set to 16MB.

After describing the data sets and queries for this experimental evaluation, we will present a broad assessment of the quality of the plans and the optimization time of each optimization technique. Then, we will

use a few specific queries for a more detailed analysis to gain a better understanding of the performance and behavior of the different optimization techniques. All optimizations are based on the answer sizes estimated using position histograms [15].

## 4.1 Data Set and Queries

While we experimented with a number of different popular XML data sets, in the interest of space, we only present the results using the following three data sets: a) Mbench [22], an XML benchmark developed at University of Michigan, b) the popular DBLP data set [18] and c) Pers, the synthetic personnel data set from AT&T [1]. The size of these three data sets are 740K nodes (about 535MB) for the Mbench data set, 500K nodes (about 9MB) for the DBLP data set, and 5K nodes (about 113MB) for the Pers data set.

To test the effectiveness of the optimization techniques we used a number of queries of varying complexity. In this section, we present experimental results for queries conforming to the patterns shown in Figure 6. For an actual query conforming to a pattern, an edge in the pattern tree could be either a parent-child relationship or an ancestor-descendant relationship.

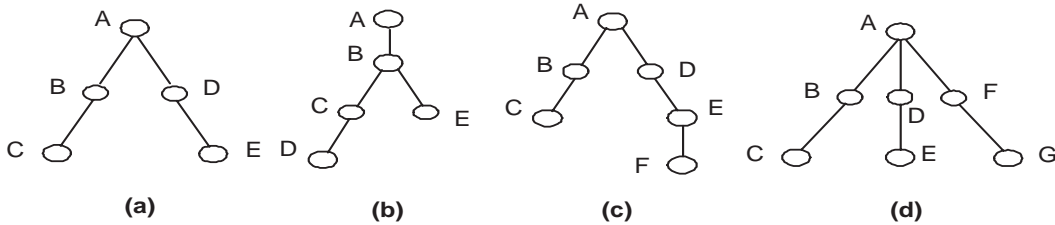


Figure 6: Sample Pattern Trees for Experiments

We limit our presentation of the experimental results to a small subset of the queries that we have actually used. In this section, we consider two queries on the Mbench data set, two queries on the DBLP data set, and four queries on the Pers data set. In our presentation, we label the queries using the form Q.DataSet.QueryNum.Pattern. For example, Q.DBLP.1.b is the first query on the DBLP data set, and the pattern of this query is b in the Figure 6.

## 4.2 Quality of Plans and Optimization Time

In this section, we examined the time taken to execute the query plans produced by the optimization algorithms, and the time taken by each algorithm to optimize the queries (the total query evaluation time is the sum of these two times). Both these results are presented in Table 1. In this table, the query optimization time is shown in a **boldface** font, and the plan execution time is shown in an *italics* font. We analyze each of

these components in the following sections. We note that for the DPAP-EB algorithm, in this experiment, the value of the tuning parameter is set to be the same as the number of edges in the pattern.

#### 4.2.1 Quality of Plans

To put the query plan execution times in perspective, we randomly (but not exhaustively) generated a number of query plans for each query, and picked the worst of these plans. This “bad” plan, which is shown in the last column of Table 1, is not necessarily the worst plan for a query. It is simply shown here to quantify the impact of a good query optimization algorithm.

Query	DP		DPP		DPAP-EB		DPAP-LD		FP		Bad Plan
	Op	Ex	Op	Ex	Op	Ex	Op	Ex	Op	Ex	
<b>Q.Mbench.1.a</b>	<b>0.67</b>	<i>2.61</i>	<b>0.12</b>	<i>2.61</i>	<b>0.09</b>	<i>2.61</i>	<b>0.10</b>	<i>3.17</i>	<b>0.07</b>	<i>2.92</i>	<b>76.76</b>
<b>Q.Mbench.2.b</b>	<b>0.69</b>	<i>1.03</i>	<b>0.12</b>	<i>1.03</i>	<b>0.11</b>	<i>1.17</i>	<b>0.11</b>	<i>1.69</i>	<b>0.10</b>	<i>1.12</i>	<b>124.22</b>
<b>Q.DBLP.1.b</b>	<b>0.75</b>	<i>5.77</i>	<b>0.14</b>	<i>5.77</i>	<b>0.12</b>	<i>5.98</i>	<b>0.12</b>	<i>6.96</i>	<b>0.11</b>	<i>5.77</i>	<b>156.71</b>
<b>Q.DBLP.2.c</b>	<b>2.21</b>	<i>0.14</i>	<b>0.53</b>	<i>0.14</i>	<b>0.43</b>	<i>0.18</i>	<b>0.30</b>	<i>0.14</i>	<b>0.10</b>	<i>0.18</i>	<b>18.60</b>
<b>Q.Pers.1.a</b>	<b>0.69</b>	<i>0.50</i>	<b>0.13</b>	<i>0.50</i>	<b>0.09</b>	<i>0.50</i>	<b>0.10</b>	<i>0.57</i>	<b>0.07</b>	<i>0.50</i>	<b>15.90</b>
<b>Q.Pers.2.c</b>	<b>2.34</b>	<i>11.39</i>	<b>0.56</b>	<i>11.39</i>	<b>0.44</b>	<i>12.1</i>	<b>0.29</b>	<i>17.62</i>	<b>0.09</b>	<i>12.1</i>	<b>520.90</b>
<b>Q.Pers.3.d</b>	<b>6.32</b>	<i>0.37</i>	<b>1.62</b>	<i>0.37</i>	<b>1.37</b>	<i>0.42</i>	<b>0.90</b>	<i>0.37</i>	<b>0.35</b>	<i>0.42</i>	<b>9.77</b>
<b>Q.Pers.4.d</b>	<b>5.78</b>	<i>1.89</i>	<b>1.71</b>	<i>1.89</i>	<b>1.39</b>	<i>1.89</i>	<b>0.87</b>	<i>4.13</i>	<b>0.39</b>	<i>1.89</i>	<b>96.34</b>

Table 1: Query Optimization (Op) Time and Query Plan Evaluation (Ex) Time (in seconds)

By examining the plan execution times for the algorithms in Table 1 (see the columns under *Ex*), we observe the same major trends for all queries on all data sets. As expected, the query plan execution time for different evaluation plans are very different. In some cases, a bad plan is 10,000 times slower than a good plan! All five algorithms serve the purpose of avoiding really bad plans, but the quality of the plans chosen by different algorithms are still different. The DP and DPP algorithms always select the optimal evaluation plan for the query, as expected, while DPAP and FP algorithms only do so sometimes. Both DPAP-EB and FP do quite well, finding a plan close to optimum in the cases the optimal plan is missed. DPAP-LD fares significantly worse.

#### 4.2.2 Optimization Time

Being able to find the best plan is not always the end goal of the optimizer (though it usually is for static queries that are pre-compiled). For dynamic queries, the goal of the optimizer is to minimize the time spent in evaluating a query, which is the summation of the time spent optimizing the query and the time spent in

	Query OpTime	# of Plans
<b>DP</b>	6.32	396
<b>DPP'</b>	3.01	122
<b>DPP</b>	1.62	71
<b>DPAP-EB</b>	1.37	57
<b>DPAP-LD</b>	0.90	39
<b>FP</b>	0.35	14

Table 2: The Query Optimization Time and Number of Alternative Plans Considered by different algorithm for Query Q.Pers.3.d

	Folding Factor			
	×1	×10	×100	×500
<b>DP</b>	0.37	3.11	26.1	110.97
<b>DPP</b>	0.37	3.11	26.1	110.97
<b>DPAP-EB</b>	0.42	3.21	28.9	292.86
<b>DPAP-LD</b>	0.37	3.56	56.1	702.89
<b>FP</b>	0.42	3.21	28.9	110.97
<b>bad plan</b>	9.77	103.66	879.59	> 4000

Table 3: Data Size and Query Plan Execution Time (in sec.) for Plans Selected by Different Optimization Algorithms (Q.Pers.3.d)

executing the optimized query plan. In this section, we examine the optimization times for all the queries.

Table 1 shows the query optimization time for the queries under the column **Op**. These results show that even though both the DP and DPP algorithms search the entire solution space to select the optimal solution, the DPP algorithm can eliminate bad plans in an early stage, and consequently, is more efficient. The DPAP algorithms, which employ addition restrictions on the status expansion, eliminate more plans, and consume less time than the DPP in optimizing the queries. Interestingly, the FP algorithm usually spends the least amount of time optimizing the query, and still generally produces plans that are close to optimal.

The source of the difference in the optimization time spent by the algorithms is the number of alternative plans considered by each algorithm. Besides keeping track of the optimization time and query plan execution time for each query we tested, we also kept record of the number of plans considered. In the interest of space, we present this result for only one query, Q.Pers.3.d, in Table 2. In this table, DPP' represents the DPP algorithm without the Lookahead Rule.

The results in Table 2 show that the time spent on optimization is linear to the number of alternative plans considered. The DP algorithm not only consider a lot of plans, but also consider the same plan several times, e.g. starting from different branches for a bushy plan. The DPP algorithm eliminates this redundant consideration and a large number of non-promising plans, and is much faster than the DP algorithm. The DPAP algorithms are even faster, considering even fewer plans. The FP algorithm explores the least number of plans and is the fastest. The results in this table also demonstrate the effectiveness of the lookahead rule in DPP.

### 4.3 Effect of Data Size

In this experiment, we investigate the effect of increasing the data set size on the plans that are produced by the various algorithms. To produce larger data sets, we replicated each data set by a “folding factor”. We generated data sets that were 10, 100 and 500 times larger than the data sets presented before. In the interest of space, we only present the effect of increasing the data set size for the time taken to evaluate plans for one

representative query, Q.Pers.3.d, in Table 3.

Note that the optimization time for the algorithms remains the same even when the data set sizes are increased. However, the larger the data sets, the larger is the plan evaluation time. Consequently, in large data sets, a more expensive optimization algorithm is worthwhile. However, there is a more interesting effect of data size, as follows:

When the data size is small, the performance of the execution plans chosen by different algorithms is not very different. The optimal plan chosen by the DP and DPP algorithm is left-deep, and is the same as the one chosen by the DPAP-LD algorithm. For larger data sets, the optimal plan chosen by the DP and DPP algorithm becomes a fully-pipelined bushy plan, and is the same as what the FP algorithm chooses. As the data size increases, the gap between the performance of the best plan and the left-deep plan gets larger. The reason is that with larger data sets, the intermediate results sizes also increase. Sorting these intermediate results becomes a big part of the plan evaluation cost and drags down the performance of plans, like the left-deep plans, that need to materialize intermediate results. Fully pipelined plans start becoming more attractive as they never have to sort any intermediate results.

#### 4.4 Effect of the Tuning Parameter $T_e$ in DPAP-EB

In this section, we evaluate the effect of the tuning parameter  $T_e$  on the performance of the DPAP-EB algorithm. The DPAP-EB algorithm uses the tuning parameter  $T_e$  to limit the number of statuses to expand on each level. This in turn limits the total number of statuses considered in the optimization process. We ran each query with different values of  $T_e$ . In the interest of space, here we only present the results for the query Q.Pers.3.d on databases with folding factors of 100 and 1 (in Figures 7 and 8 respectively); the same conclusions can be drawn for the other queries too. Along the X-Axis in these figures, we show runs of the DPAP-EB algorithm for various values of the  $T_e$  parameter (shown in parentheses). We increased the value of  $T_e$  from 1 to the number of nodes in the pattern, since by then the optimal solution has already be selected. Data for other optimization techniques is also included to facilitate comparison. Along the Y-axis, we show the optimization time and execution time as components of the total query evaluation time.

Let's begin by looking at Figure 7. From this figure, we observe that the time spent on optimization increases monotonically when the value of the parameter  $T_e$  increases, while the execution time of the evaluation plan selected decreases rapidly and becomes optimal quickly around  $T_e = 5$ . At this point the total query evaluation time (optimization time plus plan execution time) is minimized for DPAP-EB. Beyond this point, for  $T_e > 5$ , the query execution time remains the same, but the optimization time increases, until it eventually becomes the same as the optimization time for full DPP. For this query, with this data set, the

query plan execution time (even for the best plan) is much larger than the optimization time (even for the most costly DP algorithm). In this sort of situation, rather than try to guess the optimum value for the  $T_e$  parameter, one can simply use DPP, and expect it to do only slightly worse than the best case for DPAP-EB.

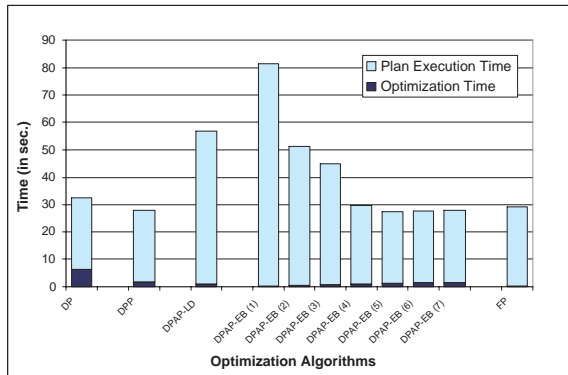


Figure 7: Comparison of Query Plan Evaluation Times for Query Q.Pers.3.d, Folding Factor = 100

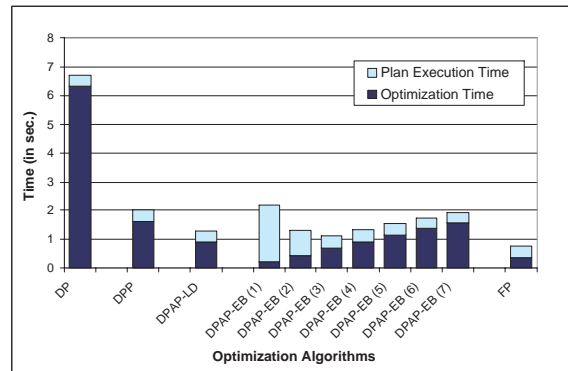


Figure 8: Comparison of Query Plan Evaluation Times for Query Q.Pers.3.d, Folding Factor = 1

Now let us examine Figure 8, which represents a different scenario in which the query plan evaluation time is comparable to the optimization time. As a result, the optimization time becomes a significant portion of the total query evaluation time. As Figure 8 shows, now the FP algorithm is the most efficient overall algorithm. The figure also shows a more obvious “U” shape pattern for the DPAP-EB plans as the value of  $T_e$  is increased. For queries with small plan evaluation time, it may not be worth the effort to use a large value of  $T_e$ , and in fact a smaller value of  $T_e$  is preferable.

## 5 Related Work

### Join Plan Order Optimization:

Query optimization is central to modern databases, and has been extensively studied since the classic work by Selinger et al [13]. Ideas proposed in [13] are still common practice in relational optimizers: Use statistics about the database instance to estimate the cost of a query evaluation plan; consider only plans with binary joins in which the inner relation is a base relation (left-deep plans); postpone Cartesian product after joins with predicate. [9] proved that under some circumstance, if sorting is not required, a pipelined evaluation plan is the optimum solution.

Bushy plans have been shown to be preferred in many circumstances [16, 14]. The Starburst optimizer [12] permits consideration of selected bushy plans, and shows that the complexity of optimizing a query is largely dependent on the shape of the query graph, instead of the number of relations involved.

### XML Query Optimization:

While XML query processing is relatively new, there already has been at least some work in this regard.

The “classic” work on XML query optimization is [11]. The idea proposed in this paper is to break branching path expressions into single path expressions (without branching). Several algorithms are proposed and evaluated. However, this work is applicable in the context of navigational access methods only, and these have since been found not to be the methods of choice (for most cases).

In even earlier work, [10] proposed a technique to specify and optimize queries on ordered semi-structured data using automata. It uses automata to present the queries and optimize the query using query typing and automata unnesting.

More recently, [3] optimizes XML queries by minimizing the pattern tree specification, using schema information. This sort of rewriting optimization is complementary to, and can be applied before, the cost-based access plan optimization that we consider.

## 6 Conclusions and Future Work

XML query processing is important, irrespective of how XML data is stored. In this paper, we have developed a framework for cost-based optimization of structural join order selection, a central issue in XML query processing. While the spirit of this optimization is the same as for relational optimization, there are significant differences on account of the tree-structure of XML data and the concept of structural join in XML query processing.

Based on the special features of XML data and XML queries, we developed five different techniques for performing query optimization in this XML framework. The performance and efficiency of the proposed algorithms are analyzed theoretically and evaluated experimentally. A significant finding is that consideration of left-deep plans alone, which is the rule-of-thumb for relational optimizers, is not a good idea in the XML context. A reasonable heuristic for quickly finding a good plan is to focus on fully pipelined plans. The FP algorithm is a quick and easy solution for queries expected to run fast. For queries with long evaluation time, one can afford more optimization time and we recommend that the DPP algorithm be used to pursue the best solution.

The work presented in this paper has been performed in the context of developing a query optimizer for the Timber native XML database system and is an important first step towards cost-based optimization for XML queries, but there is much left to be done. We are currently working on enhancing our techniques to work where not every node predicate is indexed, and with new access methods for merged operators as in [2] and multi-way structural joins as in [5]. In the future, we will also consider expensive operations beyond structural pattern matching, such as value-based joins and grouping.



## References

- [1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE Conf.*, to appear, Mar. 2002.
- [2] S. Al-Khalifa, H. V. Jagadish. Combining Operators in XML Query Processing. Technical Report. University of Michigan, 2002.
- [3] S. Amer-Yahia, S. Cho, L. Lakshmanan, et al. Minimization of Tree Pattern Queries *SIGMOD* 2001.
- [4] T. Bray, J. Paoli, C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation. Available at <http://www.w3.org/TR/1998/REC-xml-19980210>, Feb. 1998.
- [5] N. Bruno, D. Srivastava, and N. Koudas. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of SIGMOD*, 2002.
- [6] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simon, M. Stefanescu XQuery 1.0: An XML Query Language. *W3C Working Draft*, <http://www.w3.org/TR/xquery/>, June 7, 2001.
- [7] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. *PODS* 1998: 34-43.
- [8] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL Conf.*, Sep. 2001.
- [9] R. Krishnamurthy, H. Boral, C. Zaniolo. Optimization of Nonrecursive Queries. *VLDB*, 1986, pages 128–137.
- [10] H. Liefke. Horizontal Query Optimization on Ordered Semistructured Data. *WebDB (Informal Proceedings)* 1999: 61-66
- [11] J. McHugh, J. Widom. Optimizing Branching Path Expressions *VLDB* 1999: 315-326.
- [12] K. Ono, G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. *VLDB* 1990: 314-325
- [13] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. Access Path Selection in a Relational Database Management System. *SIGMOD* 1979: 23-34.
- [14] M. Steinbrunn, G. Moerkotte, A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal* 6(3): 191-208 (1997)
- [15] Y. Wu, J. M. Patel, H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proc. EDBT Conf.*, to appear Mar. 2002.
- [16] B. Vance, D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. *SIGMOD* 1996: 35-46
- [17] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. *SIGMOD* 2001.
- [18] DBLP data set. Available at <http://www.informatik.uni-trier.de/ley/db/index.html>.
- [19] Department of Chemistry, Imperial College of Science, Technology and Medicine, UK. Chemical Markup Language (CML). Available at <http://www.ch.ic.ac.uk/chimeral/>.
- [20] ibiblio Organization. XML dataset for Shakespeare drama. Available at <http://sunsite.unc.edu/pub/sun-info/xml/eg/shakespeare.1.10.xml.zip>.
- [21] TIMBER Group. TIMBER Project at Univ. of Michigan. Available at <http://www.eecs.umich.edu/db/timber/>.
- [22] DB Group at University of Michigan. The Michigan Benchmark Available at <http://www.eecs.umich.edu/db/mbench>.
- [23] W3C. XML Path Language. Available at <http://www.w3.org/TR/xpath>.

## A FP Algorithm

```
Algorithm FP.Optimization (PatternTree)
// Inputs: The query pattern to be evaluated,
// Output: Processing tree to evaluate the query.
minCost =  $\infty$ 
For (each node  $V_i$  in PatternTree)
    cost = FP.OrderBy(PatternTree,  $V_i$ , joinplan)
    if (cost < minCost)
        minCost = cost; JoinPlan = joinplan;
output (JoinPlan);

Algorithm FP.OrderBy(PtTree, OdNode, JoinPlan)
// Inputs: PtTree: The query pattern to be evaluated;
//          OdNode: The node to be ordered by in output.
// Output: Processing tree and its cost
Neighbors = neighbors of OdNode
For each Neighbors[i]
    subpattern = SubPattern(PtTree, Neighbors[i], OdNode)
    cost = FP.OrderBy(subpattern, Neighbors[i], subplan);
Enumerate all the possible permutations of subpatterns to join with OdNode.
Store the best plan in JoinPlan. Store its cost in cost.
output (cost);

note:
Function SubPattern(PatterTree, Node1, Node2) partitions the PatternTree by cutting the edge
between Node1 and Node2, and returns the subpattern contains Node1.
```

Figure 9: Algorithm *FP.Optimization* for Finding the Best Fully-Pipelined Evaluation Plan