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 responsible for consuming a lion's share of the query processing time. Thus, structural join order 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 and present an extensive experimental evaluation. 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 usually misses the best solution. Our experiments also show that a Dynamic Programming optimization with Pruning (DPP) algorithm can find the optimal solution, with low cost relative to the traditional Dynamic Programming (DP) algorithm; and an optimization technique that only considers Fully Pipelined (FP) plans 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 have 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. Structural join order selection is a critical component of an XML query optimizer, and 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.

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 the DP algorithm can be costly in terms of the optimization time, we also develop an enhanced algorithm, which is called Dynamic Programming with Pruning (DPP). DPP uses pruning techniques in the search process to eliminate partial query plans that are *guaranteed* to lead to suboptimal solutions.

Even after the pruning enhancement in DPP, the optimization process may still be expensive. More aggressive pruning techniques can be used to reduce the optimization time at the expense of a reduction in the quality of the final query plan. The Dynamic Programming with Aggressive Pruning (DPAP) algorithm uses a number of such heuristics. We consider two such heuristics. The first heuristic limits the number of intermediate results considered, which leads to the DPAP-EB algorithm. The second heuristic mirrors the choices made by many relational optimizers and only considers left-deep query plans. This optimization algorithm is called 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.

[*] H. V. Jagadish and Yuqing Wu were supported in part by NSF under grant IIS-9986030, DMI-0075447 and IIS-0208852.

[†] Jignesh M. Patel was supported in part by NSF under grant IIS-0208852, and by a gift donation from IBM.

We present extensive experimental evaluation of the proposed optimization techniques in the Timber [8] native XML database system. We demonstrate that relational rules of thumb may no longer be 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 DPP 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 many applications, such as online querying on XML data sources.

The remainder of this paper is organized as follows: Sec. 2 provides the necessary background information, and formally defines the objective of structural join order selection in XML query optimization. The five new algorithms we propose are discussed in detail in Sec. 3. The proposed algorithms are evaluated experimentally in Sec. 4. Related work is presented in Sec. 5, and Sec. 6 contains our conclusions and directions for future work.

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 enumerating alternative plans to evaluate the query, and choosing the optimal of these alternative plans. In the relational world, often the most important optimization step is join order selection. The counterpart in the XML world is structural join order selection. 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 [6]. However, considering all these aspects is beyond the scope of this paper, and we only concentrate on structural join order selection, 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 [7, 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, 19] 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 u and with v individually, a structural join outputs pairs that satisfy the required inclusion relationship.

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.*

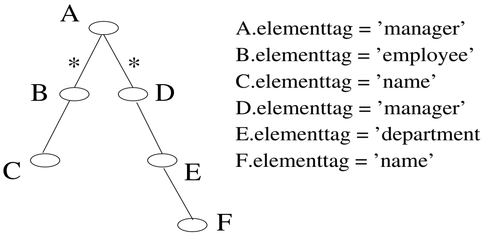

Figure 1. Example Query Pattern

*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 node 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 [19, 8]. 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*.

In this paper, we only focus on the "Stack-Tree" family of algorithms [1], because they are currently amongst the most efficient algorithms for computing binary structural joins. (The introduction of other binary structural join algorithms will not change the design of the query optimizer that is described in this paper, as new algorithms with their cost models can be incorporated into the query optimization framework that is describe in this paper.) The Stack-Tree 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 are dependent on the system implementation and machine characteristics.

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 $nlogn \times f_s$.

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 operation. 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

Example 2.2 *Consider the following query on a personnel XML database: "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.*

One possible evaluation plan for this query is to retrieve all manager *nodes (candidate for A) from an index, and then scan the sub-tree under each of these nodes, looking for other nodes that satisfy both the predicate and relationship condition. Notice that the relationships 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 entire sub-tree for each* manager *node must be scanned. The performance of this plan can be very poor, especially 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 final result is a set of 6-node tuples, corresponding to each instance in the database that matches the query pattern.*

The five joins can be evaluated in different orders. The number of alternative plans is a factorial of the number of

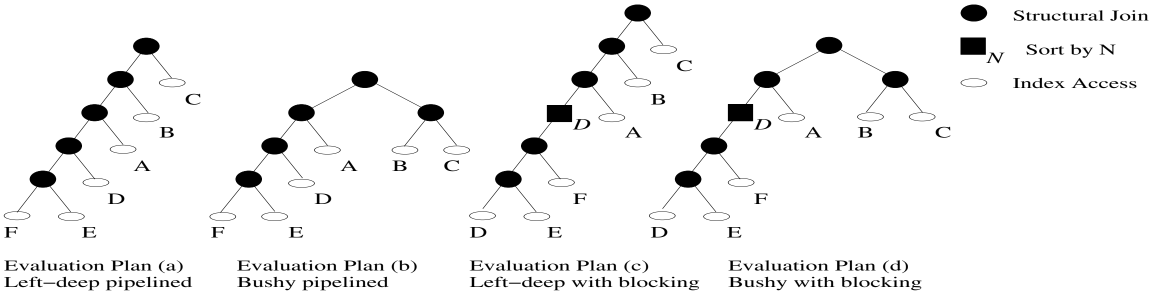

Figure 2. A Few Plans that Evaluate the Example Query Pattern

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).

For any given query pattern, there are a number of different ways of evaluating the query. The query optimizer's task is to pick the optimal (or at least a good) evaluation plan, based on the estimated cost of each alternative plan it considers.

More formally, 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 requirement to evaluate the query using this plan.

Our goal in this paper is to design and evaluate algorithms that find the cheapest evaluation plan for a given *query pattern*. In addition to picking good plans, the time required to choose a query plan (query optimization time) should be only a small percentage of the time required to actually execute the chosen query plan.

3 Join Order Selection Algorithms

In this section, we introduce five different algorithms for join order selection, and develop cost formulae to estimate the number of plans evaluated by each algorithm.

3.1 Exhaustive Dynamic Programming (DP)

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 adapted for the XML context.

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 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* $Q \Rightarrow w \in N_S$;

Since the join algorithms that we consider are Stack-based join algorithms, the output of any intermediate plan will have the intermediate results sorted by one of the inputs in the intermediate plan (see Section 2.2.1). This physical ordering property can be useful in a later step of the query evaluation, and is recorded in the status node. Note that the query may specify an explicit set of attributes for ordering the final result, and in such cases, additional sort operations may be required to produced the final query plan.

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_0. When $V_S = \{\{V_Q\}\}$, the status is called *final status*, represented by S_f. All other statuses are called *intermediate statuses*.

Example 3.1 *Figure. 3 illustrates 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_{30}, 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_{10} *has two pattern tree nodes, A and B, and the AB result is ordered by A. (The ordered-by node is presented in bold in the figure.)*

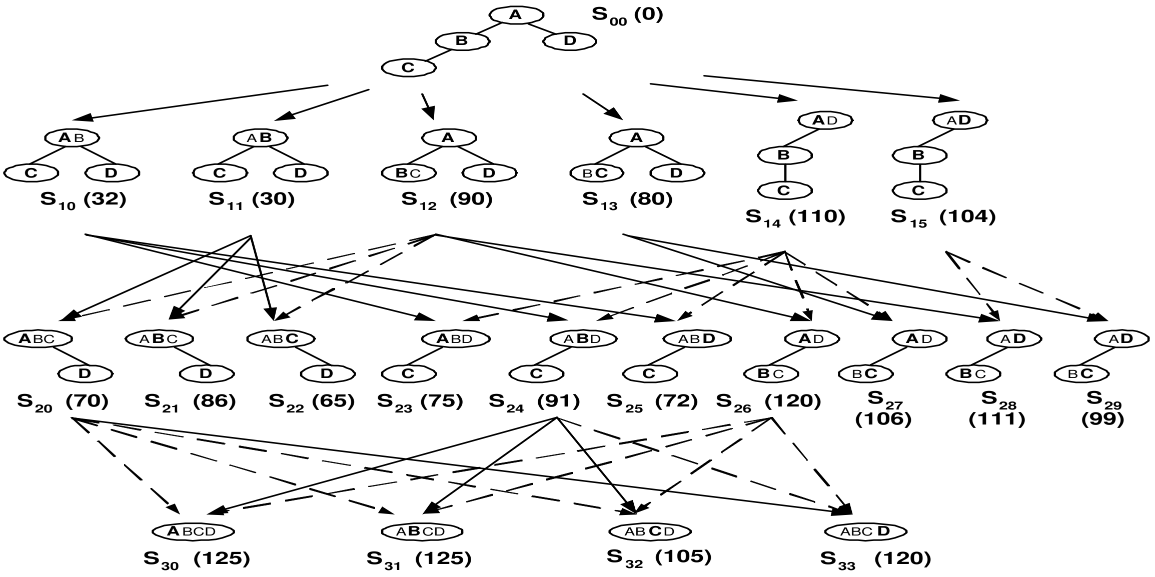

Figure 3. Example Optimization Process Using DP algorithm

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, 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 is only required if the intermediate results are not already in the sort order that is required by the next operator in the query plan.

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).*

For a given query pattern, the number of moves needed to transform the start status to the final status equals 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 $pM(S)$.

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 $pM(S)$, and estimating 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 explored using dynamic programming.

Definition 5 *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 (presented in the parenthesis following 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, a 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. For such statuses, only the minimum of these alternative ways of expanding is retained as the cost of that status, and only the path with this minimum cost 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, but with different ordering of the final result. The costs of the final statuses are compared and the one with minimum cost is picked. If the query has an explicit order-by clause, then additional sorting costs are added to the status that don't produce plans in the required order.

For example, let's assume that S_{32} has the minimum cost among all the final statuses. Tracing back the moves that lead to this final status (S_{32}) all the way back to the start status (S_{00}), we obtain the optimal solution to evaluate the query. The structural join plan selected is to first join node A with node B, producing an intermediate result ordered by node A. The exact algorithm to use for this join would be recorded with the status node in S_{10}. The next step is to join with node D, producing the next intermediate result ordered by node B, and then finally joining with node C producing the final result, which is 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)

Example 3.4 *In Figure 3, as we can see, lots of statuses 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. The goal of the DPP algorithm 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 estimate of the cost needed for transforming the status to the final status. 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 (more accurate $ubCost$ computation methods can be used to improve the pruning property of DPP, though any upper-bound estimate guarantees the correctness of the algorithm).

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. Thus DPP achieves efficiency by expanding promising plans first, which usually results in producing a "good" full plan quickly. As soon as the cost of a full plan becomes available, further pruning is used to eliminate sub-plans with a $Cost$ greater than the $Cost$ of the full plan.

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 6 *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 statuses can be prevented by looking ahead one step at the time of expanding a status.

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.

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

To analyze the complexity of DPP, 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 $S_{DPP}(lv) = f_{S_{DPP}} \times \left(\begin{array}{c} lv \\ |E| \end{array} \right)$, where $f_{S_{DPP}} = O(lv^2)$ is the number of distinct nodes orderings that are possible for each status. The upper-bound for the total number of statuses evaluated in the searching is $\sum_{lv=0}^{|E|} S_{DPP}(lv)$.

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.

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 unexpanded 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)

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 DPAP with Expansion Bound (DPAP-EB)

We considered several 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.*

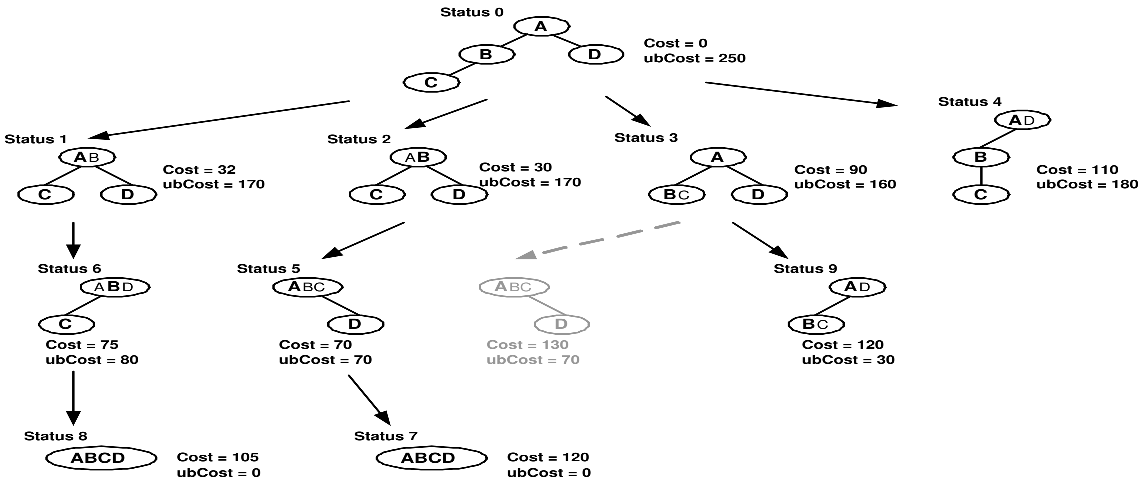

Figure 4. Example Optimization Process Using DPP algorithm

3.3.2 DPAP on Left-Deep Plans (DPAP-LD)

Relational query optimizers frequently restrict consideration to left-deep plans. It is straightforward to restrict the DPP algorithm using the following rule to only consider left-deep plans: In any status, only one status node is allowed to comprise of multiple pattern tree nodes. We call this status node the *growing node*.

This rule implies that 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 $O(|E|^{fd})$.

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 intermediate results are ordered by the node that is 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. ∎

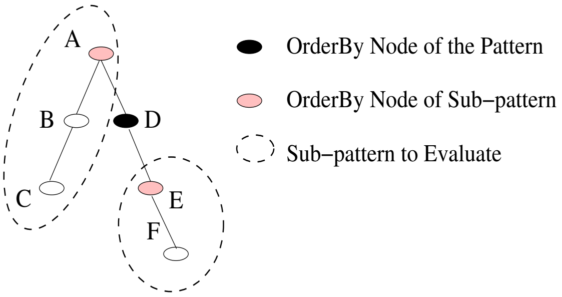

Figure 5. Pattern Tree Example in FP algorithm

The set of fully-pipelined evaluation plans for a given query pattern is a small subset of all the evaluation plans for the query pattern. Furthermore, fully-pipelined plans have the property of producing the initial result tuples quickly, which is desirable in many applications. The FP algorithm only considers fully-pipelined plans as follows: For a given query pattern, a fully-pipelined query plan could produce the final result ordered by any of the nodes in the query pattern. The FP algorithm examines each of these possibilities. Effectively for each node N in the query pattern tree, the pattern tree is "picked up" at that node. Thus the node N becomes the root of the pattern tree and it divides the pattern tree into sub-pattern trees. Plans for each of the sub-pattern trees are considered (by recursive application of the algorithm) so that the best fully-pipelined plan that produces intermediate results ordered by the root of each sub-pattern tree is generated. Then the algorithm considers the order in which the plans corresponding to the sub-pattern trees is joined first with node N. Please refer to [18] for the pseudo-code of the FP_optimization algorithm.

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 500MHz Intel Pentium III processor, 512MB of memory and a 40GB ATA Compaq disk drive. All experiment were carried out in Timber [8]. Timber uses the SHORE storage manager, and for these experiments the SHORE buffer pool size was set 16MB. All estimates for the join results were made using positional histograms [17].

4.1 Data Set and Queries

The data sets that we used in the experiments are: a) Mbench [13], an XML benchmark developed at University of Michigan, b) the popular DBLP data set [20] 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.

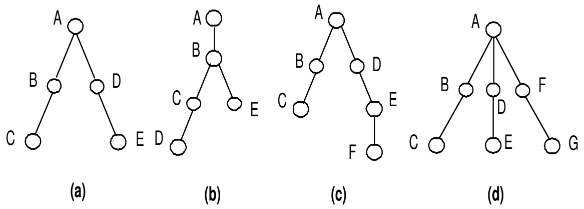

Figure 6. Sample Pattern Trees for Experiments

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 Figure 6).

4.2 Quality of Plans and Optimization Time

In this section, we examine 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. 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.

By examining the plan execution times for the algorithms in Table 1 (see the columns under *Eval.*), we observe that the query plan execution times varies dramatically across different evaluation plans. 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 se-

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

Table 1. Query Optimization and Query Plan Evaluation Times (in seconds)

lect 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

In this section, we examine the time taken by each algorithm to optimize the queries. These times are shown in Table 1 under the column **Opt.**. 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.

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

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

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 linearly proportional to the number of alternative plans that are considered. The DP algorithm not only considers many plans, but also considers 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

This experiment investigates 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", generating data sets that are 10, 100 and 500 times larger than the original data sets. 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.

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

Table 3. Data Size and Query Plan Execution Time (in sec.) for Query Q.Pers.3.d

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

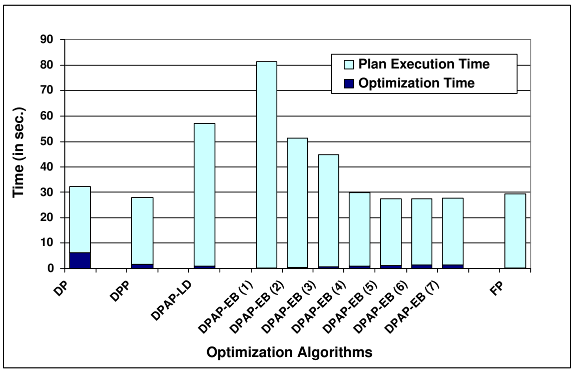

Figure 7. Comparison of Query Plan Evaluation Times for Query Q.Pers.3.d, Folding Factor = 100

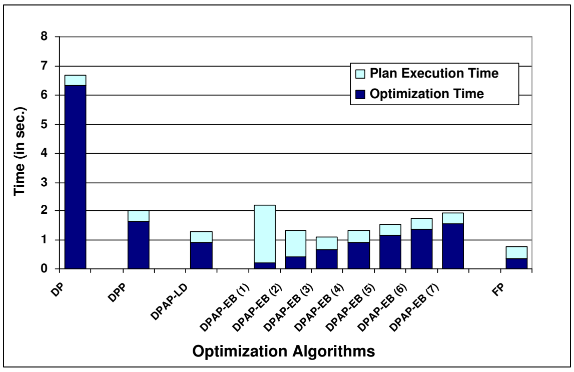

Figure 8. Comparison of Query Plan Evaluation Times for Query Q.Pers.3.d, Folding Factor = 1

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 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. DPAP-EB uses the tuning parameter T_e to limit the number of statuses that are expanded at each level, which 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 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.

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

Relational Join Order Optimization:

Query optimization is central to modern databases, and has been extensively studied since the classic work by Selinger et al. [14]. Ideas proposed in [14] 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. Krishnamurthy et al. [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, 15]. 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 by McHugh and Widom [11]. The idea proposed in this paper is to break branching path expressions into single path expressions (without branching). Several algorithms are also proposed and evaluated. However, this work is applicable in the context of navigational access methods only, which are usually not very efficient for evaluating structural joins [1, 19].

Even earlier, Liefke [10] proposed a technique to specify and optimize queries on ordered semi-structured data using automata. Automata is used to present the queries and optimize the query using query typing and automata unnesting.

More recently, techniques for optimizing XML queries by minimizing the pattern tree specification and using schema information has been proposed [3]. This sort of rewrite 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 a native XML database, after mapping to a relational database, or after some other mapping, such as to an object-oriented database. 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 finds such plans very efficiently. For queries with long evaluation times, one can afford to spend more time optimizing the query, and in such cases, we recommend that the DPP algorithm be used to produce the optimal query plan.

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. However, there are number of additional directions for future work. We are currently working on enhancing our techniques to consider cases where every node predicate is not evaluated using an index, 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. *ICDE* 2002: 141–152.

[2] S. Al-Khalifa, H. V. Jagadish. Multi-level Operator Combination in XML Query Processing. *CIKM* 2002.

[3] S. Amer-Yahia, S. Cho, L. Lakshmanan, et al. Minimization of Tree Pattern Queries. *SIGMOD* 2001: 497-508.

[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. *SIGMOD* 2002: 310–321.

[6] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. *PODS* 1998: 34-43.

[7] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL Conf.*, Sep. 2001.

[8] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, Y. Wu, C. Yu. TIMBER: A Native XML Database. *VLDB journal* 2002.

[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] K. Runapongsa, J. M. Patel, H. V. Jagadish, S. Al-Khalifa. The Michigan Benchmark Available at http://www.eecs.umich.edu/db/mbench.

[14] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. Access Path Selection in a Relational Database Management System. *SIGMOD* 1979: 23–34.

[15] M. Steinbrunn, G. Moerkotte, A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal* 6(3), 1997.

[16] B. Vance, D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. *SIGMOD* 1996: 35-46

[17] Y. Wu, J. M. Patel, H. V. Jagadish. Estimating Answer Sizes for XML Queries. *EDBT* 2002: 590–608.

[18] Y. Wu, J. M. Patel, H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. Tech Report. Available at http://www.eecs.umich.edu/ yuwu/SJOS.pdf.

[19] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. *SIGMOD* 2001: 425-436.

[20] DBLP data set. Available at http://www.informatik.uni-trier.de/ley/db/index.html.