Optimizing Branching Path Expressions[*]

Jason McHugh, Jennifer Widom
Stanford University
{mchughj,widom}@db.stanford.edu, http://www-db.stanford.edu

Abstract

Path expressions form the basis of most query languages for semistructured data and XML, specifying traversals through graph-based data. We consider the query optimization problem for path expressions that "branch," or specify traversals through two or more related subgraphs; such expressions are common in nontrivial queries over XML data. Searching the entire space of query plans for branching path expressions is generally infeasible, so we introduce several heuristic algorithms and post-optimizations that generate query plans for branching path expressions. All of our algorithms have been implemented in the *Lore* database system for XML, and we report experimental results over a variety of database and query shapes. We compare optimization and execution times across our suite of algorithms and post-optimizations, and for small queries we compare against the optimal plan produced by an exhaustive search of the plan space.

1 Introduction

Work in *semistructured data* [Abi97, Bun97], and more recently in *data management for XML* [DFF$^+$99, GMW99], has focused on graph-based data models and on query languages that use *path expressions* to specify traversals through the data. Path expressions, or equivalent constructs, form the core of the query languages *Lorel* [AQM$^+$97], *XML-QL* [DFF$^+$99], *XQL* [RLS98], *StruQL* [FFLS97], *UnQL* [BDHS96], and others. Data encoded in XML is specified as a simple nested structure of *tagged elements*, with special attributes for cross-element references. For example, an XML data set or document that we shall refer to as `DB` might contain a `Books` element, with `Book` subelements, and further `Author` and `Title` subelements. The path expression "`DB.Books.Book.Author`" identifies all the authors of books in the database, while "`DB.Books.Book.Title`" identifies all the book titles.

Many common queries in the languages mentioned above contain *branching path expressions*, which are path expressions that begin by specifying a single sequence of nested tags, but then specify multiple subgraphs to be explored from the initial exploration. As a simple example, a query looking for all authors of books with "Database" in the title contains a branching path expression. In Lorel we would write the query:

```
Select a
From DB.Books s, s.Book b, b.Author a, b.Title t
Where t grep "Database"
```

The branching path expression appears in the `From` clause of the query. Note that linear (or *chain*) path expressions, such as the examples in the first paragraph, are a special case of branching path expressions.

The query optimization problem for branching path expressions consists of being able to find a good *query evaluation plan* for any given branching path expression, within a reasonable amount of time. In a true database system for XML (such as *Lore* [MAG$^+$97]) there will be multiple ways to access data elements, and multiple ways to traverse subelement relationships and element references. Thus, in most situations an exhaustive search of the entire plan space for a branching path expression is impractical. Although certain heuristics, such as avoiding cross-products, are effective in reducing the search space for relational query

[*]This work was supported by the National Science Foundation under grant IIS-9811947.

optimization (e.g., [OL90, SAC+79, Swa89]), and even for optimizing linear path expressions in object-oriented databases (e.g., [GGT96, SMY90]), we will see that analagous heuristics are not always a good idea for branching path expressions. On the other hand, the branching nature of our path expressions gives us new opportunities for heuristics based on query structure.

The major contributions of this paper are:

- We present six algorithms that effectively reduce the search space of query plans for a branching path expression in different ways and to different extents.
- We specify four post-optimization transformations that can be applied to a query plan. The post-optimizations move either entire branches of a path expression or individual components to more advantageous positions in the plan.
- Each algorithm and post-optimization has been implemented in the *Lore* system [MAG+97], a complete DBMS for XML data, and we present experiments showing their strengths and weaknesses. In the experiments we compare optimization and execution times across the different algorithms, and for small queries we compare their times against the optimal plan produced by an exhaustive search of the plan space.
- Database statistics and cost formulas are key to estimating the cost of a query plan. We introduce specialized statistics that are useful in optimizing path expressions. Creating the desired statistics can be prohibitively expensive for large graph-structured databases, especially when the data contains cycles. We describe how a subset of the statistics can be used to estimate the information provided by complete statistics.

The problem of optimizing branching path expressions has some obvious ties to the join order, access method, and join method selection problem in relational systems. We explore this connection in detail in Section 2, as well as comparing our results to related work in object-oriented and semistructured databases. We believe that some of the work presented here could be applied to relational or object-oriented databases. Further, although we have implemented our algorithms in Lore, they could easily be adapted to all of the semistructured and XML query languages mentioned earlier.

The rest of the paper is organized as follows. Section 2 surveys related work. Preliminary definitions and the setting for our work are given in Section 3. In Section 4 we describe the desired statistics for optimizing path expressions, and we show how a subset of these statistics can be used when the full set is too expensive to create. Section 5 then describes each of our six algorithms for generating query plans for branching path expressions, and specifies our four post-optimizations. Section 6 reports our experimental results, and we conclude in Section 7. ent

2 Related Work

Path expression optimization clearly resembles the access method and join optimization problem in relational databases [SAC+79]. If we view each path expression component (`Book`, `Author`, etc.) as a table, and the dot operator (or variable sharing) as a join condition, then the vast body of research in the relational model can be applied. There are several reasons why we chose not to simply adapt previous work in the relational model to the problem of optimizing branching path expressions in semistructured and XML databases:

- Some of the relational work has focused entirely on optimizing join order, without regard to access and join methods, e.g., [GLPK94, IK90, PGLK97, Swa89]. In our setting there is a tight coupling between evaluation order and access methods: some orders preclude certain access methods, and some access methods preclude certain orderings. Details appear in Section 3.

- Since we are considering a graph-based data model, pointer-chasing as an access method is typically cheap and supported by low-level storage. Lore also supports inverse pointers via a special index [MWA$^+$98]. These access methods typically are not supported by relational systems (although they are are similar to join indexes), and have not been considered in relational join order optimization algorithms.

- Path expression optimization benefits from path statistics that are not normally supported by relational systems. See Section 4.

- Some of the relational work has focused on specific query "shapes", e.g., *linear queries*, *star queries*, and *branching queries* [OL90]. By contrast, branching path expressions in semistructured and XML query languages have an arbitrary tree shape.

In addition to these differences, we also wanted to explore a different general tactic in plan generation. Relational optimization considers three major styles of plan space search: exhaustive bottom-up (system R style), e.g., [OL90, PGLK97, SAC$^+$79]; transformation-based search using iterative improvement or simulated annealing, e.g., [IK90, Swa89]; and random search, e.g., [GLPK94]. Our search space is even larger than the space considered by most relational optimization algorithms, yet our problem has some natural structure to it. Thus, we propose a suite of algorithms, each of which reduces the plan space in a different manner and finds the optimal plan within that space. (If we were forced to categorize our algorithms, most of them would be top-down approaches with very aggressive pruning heuristics.) The general ideas underlying most of our algorithms are transferable to the relational setting. Thus, it would be interesting to see the quality of plans generated by our algorithms (appropriately modified) in contrast to those generated by, e.g., [GLPK94, IK90, OL90, PGLK97, Swa89].

Now let us consider related work in object-oriented and semistructured databases. The closest work to ours is [ODE95], which considers optimizing a restricted form of branching path expression. Their approach handles a set of linear (*chain*) path expressions where each linear path starts with the same variable, equivalent to relational *branching queries* described in [OL90]. [ODE95] compares exhaustive search with a proposed heuristic search in the context of an object-oriented database system. In both search strategies, cross-products are not considered, and branches are treated as indivisible units in the plans. Our work extends the work of [ODE95] by considering a wider range of path expressions, query plans, and optimization strategies.

Other work on cost-based optimization in object-oriented databases has considered path expressions. [GGT96] optimizes linear path expressions in a two-step process, first by heuristically choosing components of the path expression to be bound using a proposed new n-ary operator, then using any classical cost-based search strategy to assign the remaining access and join methods. In [SMY90], a dynamic programming algorithm is used to optimize a linear path expression in time $O(n^3)$, where n is the number of classes that appear in the query. Cross-products between classes are not considered and no performance results are reported. The heuristics suggested in both of these papers are not always effective for branching path expressions, so new heuristics for limiting the search space need to be considered.

A *generalized path expression*, useful in the context of semistructured databases, allows label wildcards and regular expression operators [AQM$^+$97, FFLS97, BDHS96]. Generalized path expression optimization has been studied in [CCM96, FS98, MW99a]. [FS98] and [MW99a] describe query rewrite techniques that transform generalized path expressions to simpler forms prior to optimization. In [CCM96], an algebraic optimization framework is proposed specifically to avoid exponential blow-up in the presence of closure

```
<DB>
  <Book ID="b1" Author="p1">
    <Title>The Stand</Title>
    <Keyword>Apocalypse</Keyword>
    <Keyword>Armageddon</Keyword>
  </Book>
  <Person ID="p1" AuthorOf="b1">
    <LastName>King</LastName>
    <FirstName>Stephen</FirstName>
    <Address>
      <City>Bangor</City>
      <State>Maine</State>
  </Person>
</DB>
```

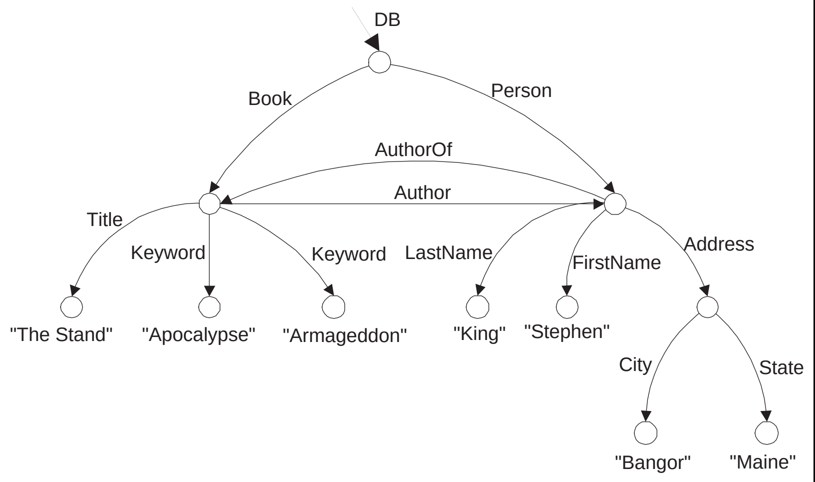

Figure 1: A tiny XML document and a graph view of the data

operators. In all three papers, the proposed techniques are complementary to the work in this paper, and could be incorporated into the algorithms that we propose.

Query optimization for the *UnQL* semistructured database query language [BDHS96, FS98] is accomplished by translating from UnQL to *UnCAL*, which provides a formal basis for optimization rewrite rules such as pushing selections down [BDHS96]. No cost-based search of a plan space is performed. Query optimization in *StruQL* [FFK$^+$99] is discussed in [FLS98], where classical top-down and bottom-up optimization search strategies are adopted. Lore's initial cost-based query optimizer is discussed in [MW99c], with branching path expressions handled by one of the six algorithms presented in this paper.

3 Preliminaries

We adopt the data model, query language, and access methods supported by the *Lore* system as the overall framework and motivation for this paper. Lore is a full-featured database management system designed specifically to store and query XML data [MAG$^+$97]. Lore's original data model, *OEM* (for *Object Exchange Model*), was a simple, self-describing, nested-object model [PGMW95]. We recently migrated and extended Lore's data model and query language to conform to the XML standard [GMW99]. For clarity in this paper, we will not digress into issues surrounding XML *attributes*, *subelements*, and *IDREF*s [BPSM98] (please see [GMW99]); it suffices to assume that path expressions can be used to navigate all three kinds of object-subobject relationships. We have purposefully kept our work on optimizing branching path expressions very general, so that it can be applied easily to other semistructured or XML-based data models and query languages (e.g., XML-QL [DFF$^+$99], XQL [RLS98]), as well as potentially to object-oriented and relational DBMSs as discussed in Section 2.

To ground our examples, let us illustrate a tiny portion of a simple database. An XML encoding is shown on the left side of Figure 1, and a graph representation appears on the right side.[1] Recall that path expressions form the basis of most query languages for graph-based data models, including Lore's query language, *Lorel* [AQM$^+$97]. We define a path expression formally as a list of *path expression steps*. A step specifies a single edge-navigation in the database. A step has the form "$x.l\ y$", for *source variable* x, *label* l, and *destination variable* y. Its semantics, in the graph-based view of semistructured or XML data, is that

[1]There are many ways to translate a given XML document into a database graph, as well as different ways to encode the illustrated database graph in XML. Again, the details are not important given the generality of the results presented in this paper, and the interested reader is referred to [GMW99].

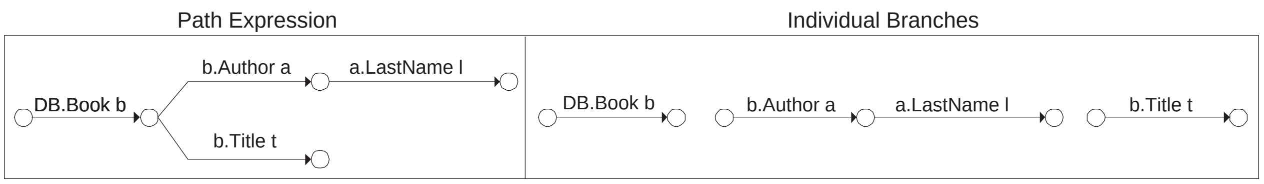

Figure 2: A branching path expression

y ranges over all l-labeled subobjects of the objects assigned to x. A *branching path expression* is a list of steps where:

1. Each source variable except the first appears as a destination variable in an earlier step. The first source variable is a *name* that identifies a distinguished database entry point [AQM+97].
2. A variable may appear as a source variable in more than one step.
3. A variable may not appear as a destination variable in more than one step (discussed below).

For example, $\langle$`DB.Book b, b.Author a, b.Title t`$\rangle$ is a branching path expression where b appears twice as a source variable. This path expression finds all authors and titles of all books reachable via the name `DB`.

We will present several algorithms that produce a *query evaluation plan* for a given branching path expression. A list of path expression steps, s, is provided as input to each algorithm, and the output is the optimal plan within the search space for that algorithm. In all of our algorithms the list s may be an arbitrarily complex branching path expression. In some situations it is necessary to isolate the individual "branches" in s. We construct a set, r, containing the individual branches. Specifically, r is a set of lists of steps created from s such that:

1. Each step in s appears in a single list in r, and each list in r contains steps only found in s.
2. Each list in r specifies a *linear path*: each step's destination variable appears as the source variable for the next step in the list (except the last).
3. If a source variable is used in more than one step in s, then each step with that source variable starts a new list in r.
4. It is not possible to combine two lists of r without violating (2) or (3).

It is easy to construct r in time linear in the length of s. As an example of the decomposition, suppose $s = \langle$`DB.Book b, b.Author a, a.LastName l, b.Title t`$\rangle$. The set r contains three elements, one for each branch in s: $r = \{\langle$`DB.Book b`$\rangle, \langle$`b.Author a, a.LastName l`$\rangle, \langle$`b.Title t`$\rangle\}$. For a graphical depiction see Figure 2. Note that our branching path expressions have a strictly tree shape—in this paper we do not consider graph-shaped path expressions, which would be generated by allowing repeated destination variables, or in Lorel by equating variables in the `Where` clause [AQM+97].

We do need to make some assumptions about the access and "join" methods that are supported by the underlying database system in order to attack the path expression optimization problem.

Access Methods. Each path expression step must be assigned an access method (or a *physical operator* in the query plan) that is responsible for providing object *bindings* to variables in the step. In this paper we consider three different access methods for a step "$x.l\ y$" as follows:

- *Extent Scan* (ES): The ES access method takes a label l as input and returns all $\langle x, y \rangle$ object pairs such that there is an l-labeled edge from object x to object y. In Lore, the ES access method is supported by the *Bindex* [MW99c, MWA$^+$98].
- *Forward Scan* (FS): The FS access method, or *pointer-chase*, takes an object x and label l as input and returns all objects y that are reachable from x via an edge labeled l. Note that x must be bound before this access method can be executed. In Lore, the FS access method is supported at the physical storage layer.
- *Backward Scan* (BS): The BS operator, or *reverse pointer-chase*, takes an object y and label l as input and returns all objects x that are parents of y via an edge labeled l. Note that y must be bound before this access method can be executed. In Lore, the BS access method is supported by the *Lindex* [MW99c, MWA$^+$98].

Lore also supports three additional access methods: a path index (*Pindex*), value index (*Vindex*), and text index (*Tindex*) [MW99c, MWA$^+$98]. The Pindex is supported by Lore's *DataGuide* [GW97]. It is specific to Lore and not always feasible to build, so we do not consider it in the general work presented in this paper. The Vindex and Tindex are similar to the ES access method, and can be used when the appropriate index exists and an appropriate predicate appears in the user's query. Only minor extensions are needed to incorporate these two access methods into the algorithms presented in this paper.

Join Methods. When two path expression steps are connected via a shared variable, then an operation similar to a join must be performed. For example, if we have the simple linear path expression $\langle$`DB.Book b, b.Author a`$\rangle$, and we choose the ES access method for both steps, then we need to perform a join on the shared b variable. Even if we choose the FS access method for both steps, a kind of join is required to pass the b bindings from the `Book` step to the `Author` step. In this paper we consider two join methods, *nested-loop join* (NLJ) and *sort-merge join* (SMJ). In many cases NLJ does not contain an explicit join condition, since we pass bound variables from left to right as in the example just given.

Recall that all path expressions begin with a *name*, which identifies an entry point and corresponds to a unique object in the database. In the Lore system, "`DB.Book`", where `DB` is a name, is actually treated as a path expression with two steps. However, this implementation detail has little importance, and in the remainder of this paper we assume that names are effectively variables whose single matching object is "prebound" to the variable.

We could use an exhaustive algorithm to enumerate plans for a given branching path expression: we consider all possible orderings of the steps, all possible access methods, and all possible join methods. The total number of left-deep plans is then $n!2^{n-1}3^n$, where n is the number of steps, and there are 3 access methods and 2 join methods; creating bushy plans of any type [OL90] increases the search space further. Because of the way our access methods must work together, many of the permutations found in the exhaustive plan space result in plans that are not *valid*. For example, consider $s = \langle$`DB.Book b, b.Author a`$\rangle$. The plan $\boxed{\texttt{FS(b.Author a) NLJ BS(DB.Book b)}}$ is invalid since the FS access method requires b to be bound. The plan $\boxed{\texttt{ES(DB.Book b) SMJ FS(b.Author a)}}$ does not violate any bound variable restrictions but is invalid because the SMJ operator does not support passing bindings for variable b from the ES method to the FS method. However, even when we eliminate the invalid plans, the size of the exhaustive plan space is prohibitively large for $n > 5$.

4 Database Statistics

A cost-based query optimizer relies on database statistics and cost formulas to estimate the cost, or predicted running time, of each plan it considers [SAC+79]. The cost formulas themselves are somewhat implementation dependent, and formulas for Lore's physical operators are given in [MW99b]. Intermediate result size estimation, however, is a key general factor that depends on statistics gathered about the data. In this section we briefly discuss statistics and result size estimation for optimizing branching path expressions.

Traditional relational and object-oriented statistics are well-suited for estimating predicate selectivities, and for estimating the number of tuples one relation (or class) produces when joined with another relation (or class). (Object-oriented statistics can be somewhat more complicated if the class hierarchy is taken into account, e.g., [CCY94, RK95, SS94, XH94].) However, these statistics are not well-suited for long sequences of joins as embodied in path expressions. A cost-based optimizer for path expressions may, for example, need to accurately estimate the number of "`Book.Author.Address.City`" paths in the database. In Lore we set a threshold k, and gather statistics for all label sequences (linear paths) in the database up to length k. We have explored several algorithms to efficiently compute these statistics, but a presentation of the algorithms is outside of the scope of this paper. Obviously for large k the cost of producing the statistics can be quite high, especially for cyclic data. A clear trade-off exists between the cost in computation time and storage space for a larger k, and the accuracy of the statistics.

The statistics we maintain, for every label sequence p of length $\leq k$ appearing in the database, include:

- The total number of instances of sequence p, denoted $|p|$.
- The total number of distinct objects reachable via p, denoted $|p|_d$.
- For each label l in the database, the total number of l-labeled subobjects of any object reachable via p, denoted $|p_l|$.
- For each label l in the database, the total number of incoming l-labeled edges to any instance of p, denoted $|p^l|$.

Consider evaluating the linear path expression $\langle$`DB.A a, a.B b`$\rangle$. If we have bindings for a from an `FS(DB.A a)` method, for example, then we may next need to estimate the average number of `B` subobjects for the a bindings. Alternatively, if we have bindings for a from an `ES(a.B b)` method, then we may next need to estimate the average number of `A` parents for these bindings. We call these two estimates *fan-out* and *fan-in*, respectively. The fan-out for a given linear path expression p and label l is computed from the statistics by $|p| * (|p_l|/|p|_d)$. Likewise, fan-in is $|p| * (|p^l|/|p|_d)$.

Our statistics are most accurate for estimating result sizes in linear path expressions of length $\leq k+1$: We store statistics about linear paths of length $\leq k$, and these statistics include information about incoming and outgoing edges to the paths—effectively giving us information about all linear paths of length $\leq k+1$. When we need statistics for linear paths of length $> k+1$, we can estimate the statistics by combining statistics of progressively smaller paths until we reach paths of size $k+1$. For example, given a path expression p of length $k+2$, we combine statistics for two overlapping paths p_1 and p_2 each of length $k+1$: p_1 is the path expression p with the last step removed, and p_2 is the path expression p with the first step removed. We combine the statistics of the two paths using the formula $|p| = |p_2| * |p_1|/|p_1 \cap p_2|$, where $p_1 \cap p_2$ is a third path expression containing all steps common to p_1 and p_2.

Note that at this time we are not gathering statistics about branching path expressions, which would be extremely expensive in the general case, even given a threshold k. Instead, at "branch points" we combine statistics for individual branches using standard formulas similar to [SAC+79].

```
Procedure Exhaustive(s)→Plan
1  Cost leastCost = COST_MAX;
2  Plan bestPlan;
3  foreach $s'$ possible ordering of $s$ do
4      foreach assignment $a$ of access methods steps in $s'$ do
5          foreach assignment $j$ of join methods to adjacent steps in $s'$ do
6              Plan current = BuildPlan($s'$, $a$, $j$);      // Build the actual plan
7              Cost c = GetCost(current);
8              if  ($c <$ leastCost)
9                  leastCost = c;
10                 bestPlan = current;
11 return bestPlan;
```

Figure 3: Pseudocode for the exhaustive algorithm

5 Plan Selection Algorithms

Assuming left-deep query plans only, a plan is characterized by the order of the steps, the assignment of an access method to each step, and the assignment of join methods connecting the access methods. An exhaustive algorithm searches the entire space, estimates the cost of each plan, and returns the predicated optimal plan. In this section we present six additional algorithms that heuristically reduce the search space in a variety of ways. The running time for each algorithm is dominated by the size of the plan space that is searched. We present the algorithms roughly in decreasing order of running time, and thus in decreasing amount of plan space explored. However, the search space is pruned in different ways for each algorithm, and usually the search space for an algorithm is not a subset of the search space for the previous algorithm. We also present four post-optimizations that can be applied to a plan generated by any of our algorithms, although we focus on their effectiveness when applied after two of our six algorithms.

Most of our algorithms generate left-deep plans only, and we are not searching the plan space for alternative plan shapes. The exceptions are Algorithm 2, which may swap left and right subplans in some situations, and Algorithm 5 which, although it searches a relatively small amount of the plan space, can produce some bushy plans.

The algorithms we have designed and the plan spaces they explore were inspired by our observation of queries posed to the Lore system. There are many other ways to reduce the search space and many ways to combine our algorithms. We believe the algorithms and post-optimizations presented here are an interesting representative sample, as confirmed by our experiments presented in Section 6.

5.1 Algorithm 0: Exhaustive

As a measure against which we can compare plans produced by the other algorithms, we consider an exhaustive search of the plan space (Figure 3). Recall that the total number of plans considered by the exhaustive algorithm is $n!m^nj^{n-1}$, for n steps, m access methods, and j join methods. However, some of these plans are not valid since they violate constraints imposed by the selected access or join methods and the step order (Section 3). Although not shown explicitly, each of our algorithms checks the validity of each plan considered (e.g., within procedure `BuildPlans` in Figure 3). Recall that all algorithms take as input a branching path expression expressed as a list s of steps.

5.2 Algorithm 1: Semi-exhaustive

The motivation for our "semi-exhaustive" algorithm is to continue generating all possible step orderings, but reduce the number of access method permutations. The algorithm considers all possible step orderings and combinations of join methods, but assigns access methods greedily for each ordering and join method

```
Procedure Exponential($s$)→Plan
1    // Create a structure to track the bound variables, initially empty
2    Bindings b;
3 return RecOpt(s, b);

Procedure RecOpt($s$, Bindings $b$)→Plan
1    // If $s$ has a single step then choose the best access method
2    int l = lengthof(s);
3    if (l==1)
4        return OptimalAccessMethod(s[1],b);      // Modifies bindings in b
5    // Otherwise, create a plan for the left-then-right order by optimizing s[1..l-1] and then s[l]
6    Bindings b1 = b;
7    Plan p1LHS = RecOpt(s[1..l-1], b1);          // Modifies bindings in b1
8    Plan p1RHS = RecOpt(s[l], b1);               // Modifies bindings in b1
9    Plan p1 = OptimalJoin(p1LHS, p1RHS);
10   // Create a plan for the right-then-left order by optimizing s[l] then s[1..l-1]
11   Bindings b2 = b;
12   Plan p2LHS = RecOpt(s[l], b2);               // Modifies bindings in b2
13   Plan p2RHS = RecOpt(s[1..l-1], b2);          // Modifies bindings in b2
14   Plan p2 = OptimalJoin(p2LHS, p2RHS);
15   if (GetCost(p1) < GetCost(p2)) return p1 else return p2;
```
Figure 4: Pseudocode for the exponential algorithm

permutation. This approach replaces the m^n term in the exhaustive search with 1, resulting in $n!j^{n-1}$ plans. The algorithm chooses access methods by performing a single scan of the steps, in order, assigning to each the best access method given the bindings of variables that came before it. The pseudocode is obtained by replacing line #4 in Figure 3 with a procedure that performs a linear scan of s', keeping track of bound variables and assigning the least-cost access methods.

While a significant portion of the plan space is pruned in the semi-exhaustive algorithm, the running time may still be prohibitively large due to the $n!$ term. Also, the locally optimal access method decisions are not always globally optimal. For example, the cost of a single step in isolation is never lower for ES than for FS or BS (when FS or BS can be used). However, there are situations where a more expensive ES followed by SMJ with the rest of the plan has lower overall cost than using a FS or BS as the first access method.

5.3 Algorithm 2: Exponential

In this algorithm we reduce the $n!$ term by considering a subset of the possible step orderings. Algorithm 2 generates different step orderings by swapping the order between the first $n-1$ steps and the last step, recursively over the input list s. This approach reduces the step ordering term to 2^{n-1} [MW99b]. Figure 4 shows precisely how the search space is reduced. Procedure `RecOpt` accepts a list of steps and a list of variables currently bound. Two plans are produced. *p1* is the plan where s without its last step is optimized via a recursive call, then joined with the best access method for the last step. *p2* is the converse: an access method for the last step in s is chosen, then joined with the selected plan for the remainder of s. Key to constructing the subplans recursively is the bound variable structure b, which tracks the variables that are currently bound and has a strong influence over the selected access methods for later steps. Besides reducing the number of orderings considered, this algorithm also reduces the permutations of join and access methods considered by making locally optimal decisions with respect to a given set of bound variables. Note that when plan $p2$ is chosen over $p1$, then a non-left-deep plan is constructed. This algorithm is similar in spirit to the original Lore cost-based optimizer. The full technique as applied over the entire Lore language is described in [MW99c].

```
Procedure Polynomial($s$)$\rightarrow$Plan
    Bindings b;
    Plan finalPlan;
    while (!empty(s)) do
        Cost leastCost = COST_MAX;
        Step bestStep;
        Plan bestPlan;
        // Find the step currently in s with the least-cost access method
        foreach $e$ in $s$ do
            Bindings bTemp = b;
            Plan p = OptimalAccessMethod($e$,bTemp);      // Modifies bindings in bTemp
            Cost c = GetCost(p);
            if ($c < leastCost$)
                bestStep= e;
                bestPlan= p;
                leastCost= c;
        // Remove the chosen step
        s $-=$ bestStep;
        // Add the bindings and add the chosen step to the final plan using the best join method
        AddBindings(b, bestStep);
        finalPlan = OptimalJoin(finalPlan, bestPlan);
return finalPlan;
```

Figure 5: Pseudocode for the polynomial algorithm

Note that this algorithm is sensitive to the order that the steps appear in input list s. The post-optimizations described in Section 5.8 specifically address this issue.

5.4 Algorithm 3: Polynomial

Our next algorithm reduces the plan space even more aggressively than Algorithms 1 and 2. It combines step order, access method, and join method selection into an $O(n^2)$ operation. The algorithm, shown in Figure 5, makes a greedy decision about which step is next and which access and join methods are chosen through each iteration of the `while` loop. The inner `foreach` loop finds the cheapest access method for each remaining step, based on the current bound variables. The step with the least cost is then added to the plan, its variables are marked as bound, and the step is removed from further consideration. For example, given $s = \langle$`DB.Book b, b.Author a, a.LastName l,`$\ldots\rangle$, the step with the least cost access method may be an ES over `LastName`. In the next iteration a and l are bound. At that point a BS over `Author` might have least cost; if so, b becomes bound, and a join method for variable a is selected.

Obviously, this very greedy approach can produce nonoptimal plans in some situations. For example, consider $\langle\ldots$`, x.Author a, a.PhoneNumber p,`$\ldots\rangle$. Suppose there are many `PhoneNumber`'s and `Author`'s in the database, but very few authors have given their phone numbers. The optimal plan may include an ES for `PhoneNumber` and then a BS for `Author`, but the polynomial algorithm probably would not consider this plan since the BS cannot be chosen before the ES (due to the bound variable restriction), and the ES is unlikely to be cheapest at any point during the iteration.

5.5 Algorithm 4: ES-Start

Because the ES access method requires no bound variables, it is possible to use an ES to "start" the evaluation of a path expression at any point, then use the FS and BS access methods to "spread out" and bind the remaining steps. The heuristic behind our next algorithm is to first identify those steps in s that make good ES starting points. Let us defer for a moment the definition of "good" starting points and the mechanism

```
Procedure ES-start($s$)→Plan
1    Plan finalPlan;
2    Set⟨Step⟩ $p$;
3    SortBasedOnSize($s$);
4    $p$ = ChooseStartingPoints($s$);
5    // Connect each adjacent pair via all FS or all BS methods (depending on cost).
6    foreach adjacent pair $\langle e_1, e_2 \rangle$ in $p$ do
7        Plan p1 = AssignFSandJoin($s$,$p$,$e_1$,$e_2$);
8        Plan p2 = AssignBSandJoin($s$,$p$,$e_1$,$e_2$);
9        if   (GetCost(p1) $<$ GetCost(p2))
10           finalPlan = OptimalJoin(finalPlan, p1);
11       else
12           finalPlan = OptimalJoin(finalPlan, p2);
13   // Assign FS to remaining steps
14   foreach $e$ in $s$ but not in finalPlan do
15       Plan temp = AssignFS($e$);
16       finalPlan = OptimalJoin(finalPlan, temp);
17 return finalPlan;
```

Figure 6: Pseudocode for the ES-start algorithm

by which we choose them. Once we have the ES starting points, we make a simple linear-time decision for each pair of starting points of whether to use a complete FS-based or complete BS-based plan between them.

The pseudocode for this algorithm appears in Figure 6. The starting points are selected (discussed below) and the chosen steps are copied into the set p. The first `foreach` loop in Figure 6 considers each adjacent pair of starting points in p, where steps e_1 and e_2 in p are considered adjacent if there is a sequence of steps in s that leads from the destination variable of e_1 to the source variable of e_2 without using another step in p (i.e., without going through another starting point). For $\langle e_1, e_2 \rangle$ we generate two subplans: the first assigns FS to every step connecting e_1 and e_2, and the second assigns BS to every connecting step. The best join methods are selected, and the subplan with the lower cost is added to the final plan. Note that if a step is shared by multiple connecting paths then it keeps the first access method selected. Finally, remaining unassigned steps are assigned the FS access method in sorted order according to extent size, respecting bound variable restrictions.

Key to the success of this algorithm is identifying those steps that make good ES starting points. Procedure `ChooseStartingPoints` is shown in Figure 7. Recall from Figure 6 that when this procedure is called, the steps in s have been sorted by the size of their extents. The procedure selects a k, $0 \leq k \leq n$, such that the first k steps in s are the starting points. It does so by incrementing k until the ratio between the sizes of the kth and ($k-1$)st extents is below some threshold. That is, we accept the kth step as a good starting point as long as the increase from the size of the previous extent isn't too large. We denote the size of the kth extent as z_k, and set $z_0 = 1$. The procedure is complicated by three details. First, the initial increase from $z_0 = 1$ to a $z_i > 1$ can be very large, so we define a special threshold for this case. Second, if the extents grow at a steady rate below our ratio threshold, then `ChooseStartingPoints` will determine that all steps should be assigned the ES access method. Thus, we set an absolute maximum on starting point extent size based on the first $z_i > 1$. Third, recall that names are variables "prebound" to a single object. For the ES-start algorithm, all names are automatically assigned as starting points, although in the actual Lore implementation it isn't necessary to use an ES method to find the named objects since they are handled in a special manner. Note that the ES-start algorithm will always choose at least one starting point, since all path expressions begin with a name.

Again, choosing a good set of ES starting points is crucial. Note that the constants in Figure 7, `INITIAL_CUTOFF`, `RATIO_CUTOFF`, and `TOTAL_CUTOFF` are "tuning knobs", and they required some ad-

```
Procedure ChooseStartingPoints($s$)→Set⟨Step⟩
int k = 0;
Boolean first = TRUE;
int nontrivial;
for($i = 1$; $i$ <lengthof($s$); $i$++)
    if (first)
        if ($z_i$!=1)
            first = FALSE;
            nontrivial = $z_i$;
            if ($z_i >$ INITIAL_CUTOFF) break;
    else
        if ($z_i$ / $z_{i-1} >$ RATIO_CUTOFF) break;
        if ($z_i >$ TOTAL_CUTOFF * nontrivial) break;
    k++;
// Copy the first $k$ steps into the result
Set⟨Step⟩ result;
for($i = 1$; $i \leq k$; $i++$)
    result.Add(s[i]);
return result;
```
Figure 7: `ChooseStartingPoints` used by the ES-start algorithm

justing before appropriate settings were obtained. However, our current settings result in good performance for a wide variety of database shapes and queries.

The complexity of the ES-start algorithm is $O(n \log n)$, and as we will see in Section 6 it tends to perform well in overall (optimization plus execution) time.

5.6 Algorithm 5: Branches

Our next algorithm optimizes each branch in s in isolation. Optimal subplans for each branch are then combined into a final plan in order of subplan costs, using the cheapest join method between subplans. Pseudocode is shown in Figure 8. `Decompose` identifies the individual branches in s, as described in Section 3. We have chosen our polynomial algorithm (Algorithm 3, Section 5.4) to optimize the individual branches, although any of the other algorithms could be used. Note that we are not concerned about one branch relying on bindings passed from another, since each branch is optimized separately. A disadvantage to this approach is an overreliance on the ES access method, since at least one ES must appear in the subplan for each branch except the first.

5.7 Algorithm 6: Simple

Finally, we consider for comparison purposes a very simple $O(n \log n)$ algorithm that searches only a tiny fraction of the plan space. The algorithm, shown in Figure 9, first sorts the steps in s by the size of their extents, and this becomes the join order. A single pass through the sorted list assigns the best access and join methods, in a greedy fashion, based on the current bound variables.

5.8 Post-Optimizations

We now introduce four post-optimizations that transform complete plans into equivalent plans with the same or lower cost by moving access methods to more advantageous positions within the plan, and reassigning join methods as appropriate. The four post-optimizations are divided into two pairs based on the granularity at which they operate. *Branch post-optimizations* move entire subplans that correspond to complete branches in the original path expression. *Step post-optimizations* move individual access methods.

```
Procedure Branches($s$)→Plan
    Plan finalPlan;
    int numBranches;
    r = Decompose(s, numBranches);
    // One subplan for each branch optimized using Algorithm 3
    Plan subPlan[numBranches];
    int count = 0;
    foreach $l$ in r do
        subPlan[count] = Polynomial(l);
        count++;
    // Sort the array of subplans based on their costs
    SortBasedOnCost(subPlan);
    // Join the subplans together
    for i = 1 to numBranches
        finalPlan = OptimalJoin(finalPlan, subPlan[i]);
return finalPlan;
```

Figure 8: Pseudocode for the branches algorithm

```
Procedure Simple($s$)→Plan
    Plan finalPlan;
    Bindings b;
    SortBasedOnSize(s);
    // Assign access and join methods in single scan
    foreach $e$ in $s$ do
        Plan tempPlan = OptimalAccessMethod(e,b);       // Modifies bindings in b
        finalPlan = OptimalJoin(finalPlan, tempPlan);
return finalPlan;
```

Figure 9: Pseudocode for the Simple algorithm

5.8.1 Branch Post-optimizations

Let us assume that we have our set r of branches of s (computed as described in Section 3), and let l be the size of r, i.e., l is the number of branches in s. Note that the access methods corresponding to the steps of a given branch may not be adjacent in the plan we start with, but we can collect the access methods for a branch and place them elsewhere in the plan as long as bound variable restrictions are met. When bound variable restrictions are not met, the corresponding reorderings are not considered.

Post-optimization A. A simple greedy heuristic, running in $O(l^2)$, reorders the branches in the plan. The heuristic estimates the cost of the subplan for each branch in r, and appends to a new final plan the cheapest subplan that does not rely on a branch not yet in the new final plan. This procedure repeats until all branches are in the final plan.

Post-optimization B. This post-optimization is more thorough and therefore more expensive. It constructs and costs all possible reorderings of the branches. There are $O(l!)$ such orderings, but l is usually small in comparison to n (the number of steps), and many of the reorderings may be invalid since the subplan for a branch may depend on other branches being executed before it.

5.8.2 Step Post-optimizations

As with the branch post-optimizations, there are two ways to search the additional plan space.

Post-optimization C. Analogous to post-optimization A but operating at the step level, in $O(n^2)$ time we repeatedly find the step with the smallest cost that does not rely on a step not yet in the new final plan, and append the access method associated with that step to the new final plan. The process repeats, with new cost estimates for the remaining steps, until all steps have been placed.

Post-optimization D. Analogous to post-optimization B but operating at the step level, all possible valid reorderings of the steps are considered. In general this can add an additional $n!$ to the running time, but in practice, since access methods have already been assigned to the steps, the number of valid reorderings is limited.

We will evaluate the effectiveness of these post-optimizations when applied to plans generated by Algorithms 2 and 3. Algorithm 2 (the exponential algorithm) can benefit greatly from these post-optimizations, because the quality of the initial plan produced is sensitive to the order of the steps in input s. Since Algorithm 3 combines step order and access method selection into a single pass, the post-optimizations provide a "second chance" to reorder the steps without also deciding the best access methods.

6 Performance Results

We implemented the six algorithms and four post-optimizations presented in Section 5 in the Lore system, and we performed a variety of experiments over data and path expressions of varying shapes. We report on the times required to construct query plans along with query execution times. The setting for our experiments is described in Section 6.1. A summary of results for Algorithms 1–6 and Post-optimizations A–D is provided in Section 6.2. We examine specific results in more detail in Section 6.3. In Section 6.4, we focus on the improvement that the post-optimizations produce when applied to plans produced by Algorithms 2 and 3. Finally, in Section 6.5 we compare some results against optimal plans generated by exhaustive search.

6.1 Setting

We used a synthetic XML database containing information about movies, stores that rent and sell the movies, companies that own the stores, and people that work for the companies or have participated in making the movies.

There are over 12,000 movies in the database. Each movie has as subobjects (among other things) people who acted in the movie, locations where the movie was shot, and stores where the movie is available for rent. Each of the 256 store objects has as subobjects (among other things) store location and the company that owns the store. There are only 13 companies that own stores, although the database contains more than 150 companies (companies that don't own stores are assumed to relate to the movie industry in other ways). Companies contain as subobjects (among other things) the people who work for that company. Each person has a subtree containing personal information, including things that they like and dislike.

The shape of the data is very important. It is is highly graph-structured, with a unique entry point named `DB`. There is a very small first-level fan-out to distinguish between different categories in the data (e.g., all movies in the database are reachable via "`DB.Movies`", and all companies are reachable via "`DB.Companies`"). The data then fans out rapidly since there are thousands of movies, hundreds of companies, thousands of people, etc. The data then gets even wider or narrows substantially, depending on the path taken. For example, the data narrows when we look for all the stores that rent movies because there are only 256 of them, although note that the number of "`DB.Movies.Movie.AvailableAt`" paths is huge. The data narrows even further if we consider "`DB.Movies.Movie.AvailableAt.OwnedBy` ", since franchises own many stores. However, the data fans out again if we explore the franchise employees via the path "`DB.Movies.Movie.AvailableAt. OwnedBy.Employee`". Our experience is that this "narrow-wide-narrow" pattern appears commonly in nested data.

1. DB.Movies x, x.Movie m, m.Actor a, m.AvailableAt t
2. DB.People x, x.Person p, p.Name n, p.Phone z, p.Likes l, l.Thing t
3. DB.People x, x.Person p, p.Likes l, l.Thing t2, p.Dislikes d, d.Thing t1
4. DB.Stores x, x.Store s, s.Name n, s.Location l, l.City c
5. DB.Movies x, x.Movie m, m.Sequel s, s.AvailableAt a, a.OwnedBy o, o.Affiliated f, f.Phone p
6. DB.Movies x, x.Movie m, m.Sequel s, s.AvailableAt a, a.OwnedBy o, o.Affiliated f, f.Name n
7. DB.Movies x, x.Movie m, m.Actor a, a.Likes l, l.Thing t, a.Address d, m.Title z
8. DB.Companies x, x.Company c, c.Affiliated a, x.Name n

Figure 10: Sample set of 8 branching path expressions

Algorithm	1	2	2A	2B	2C	2D *	3	3A	3B	3C	3D	4 **	5	6
Execution Time Rank	11	14	9	10	3	1	6	8	7	5	3	2	13	12
Total Time Rank	14	13	9	11	5	11	2	4	6	2	9	1	7	8

Table 1: Overall results

All experiments were conducted using Lore on an Intel Pentium II 333 mhz machine running Linux. The database size was 12 megabytes, and the buffer size was set to 40 pages of 8K each, or about 2% of the size of the database.

6.2 Overall Results

We ran each algorithm except the exhaustive one, including Algorithms 2 and 3 augmented with Post-optimizations A–D (denoted 2A, 2B, etc.), on the sample set of 8 branching path expressions shown in Figure 10. For each of the 8 experiments, we ranked the algorithms based on the time to execute the chosen plan, and also the total time to both select and execute the plan. We then added together the ranks for each algorithm across all 8 experiments, treating each query as equally important. The results are shown in Table 1.

Algorithm 4, the ES-start algorithm (marked by ** in Table 1), performs the best. In terms of plan execution speed it ranks second, just behind Algorithm 2D (marked by *). Algorithm 4 ranks first for total time, which includes the time required for optimization. Note that Algorithm 2D is ranked eleventh in total time: Algorithm 2 (the exponential algorithm) explores a fairly large portion of the search space, and Post-optimization D is the most expensive post-optimization. (Further experimental results for Post-optimization D are reported in Section 6.4.)

In two experiments Algorithm 4 created the fastest plan, but in other instances it ranked in the top three or four. Its strength is that it consistently selected good plans in a reasonable amount of time. Overall the plans produced by Algorithms 5 and 6 (the branches and simple algorithms) performed poorly, as shown in the last two columns of Table 1. Although both algorithms did produce very good plans for a small number of queries, the results were inconsistent. Unfortunately, we have not been able to characterize the situations in which these algorithms perform well—it appears to depend on complex interactions between query shape and detailed statistics about the data.

Another interesting result from Table 1 is the poor overall performance of Algorithm 2, the exponential algorithm, without post-optimizations. Recall from Section 5.3 that this algorithm reduces the step orderings considered from $n!$ to 2^{n-1}, and is similar to Lore's cost-based optimization strategy for the full Lorel language [MW99c].

The high overall times for Algorithm 1 were expected since optimization time is prohibitively large.

Rank	Algorithm	Optimization time	Execution Time	Total Time
1	5	0.445	41.770	42.215
2	3	0.099	48.573	48.672
3	4	0.145	48.573	48.718
4	1	1.180	48.573	49.753
5	2	0.108	60.643	60.751
6	6	0.318	108.600	108.918

Table 2: Experiment 1 – simple branching path

However, the slow plans produced by Algorithm 1 were unexpected. Apparently making a local access method decision for a given step order ignores the global situation too often.

Note the anomaly in the results of Table 1 for the execution times of Algorithms 2A and 2B, reported as 9th and 10th respectively. Since 2B explores a strictly larger plan space than 2A we would expect it to produce strictly better plans. We attribute this slight inconsistency to somewhat imperfect statistics and/or cost estimates, although our costing is quite accurate in general as shown in [MW99b].

6.3 More Detailed Results

In this section we look in more detail at some of the experiments from Section 6.2, focusing on Algorithms 1–6 without considering post-optimizations.

6.3.1 Experiment 1: Simple Branching Path

In our first experiment $s = \langle$`DB.Movies x, x.Movie m, m.Actor a, m.AvailableAt v`$\rangle$. This expression contains three branches, similar to the example shown in Figure 2. In our database, on average there are more actors that acted in a movie than stores that carry that movie. Thus, it is usually beneficial for a plan to evaluate the branch "`m.AvailableAt v`" before "`m.Actor a`" (to keep intermediate results smaller). Table 2 shows the optimization, execution, and total time (in seconds) for each of the algorithms, ranked by total time.

Algorithm 5, the branches algorithm, generates the best plan and does so quickly. This plan uses ES for `AvailableAt`, then SMJ with an FS-based plan for `DB.Movies.Movie`. A final SMJ with an ES for `Actor` completes the plan. This plan performs well in this particular case because most of the data discovered by each branch independently actually contributes to the final result. Thus, optimizing branches independently does not cause significant irrelevant portions of the database to be explored. Algorithm 6, the simple algorithm, does very poorly. It first selects ES for `AvailableAt`, then BS for `Movie` and `Movies`, then FS for `Actor`. The better plans verify that an object has both `AvailableAt` and `Actor` subobjects before working backwards to match `DB.Movies.Movie`. Algorithms 1, 3, and 4 all produced the same plan for this experiment, so here and in subsequent results where the plans were the same, we averaged their slightly deviating execution times.

6.3.2 Experiment 2: More Branches

In our second experiment $s = \langle$`DB.People x, x.Person p, p.Name n, p.Phone h, p.Likes l, l.Thing t`$\rangle$. In our database each person has a single name, and roughly half of the people have things that they like. On average, those with likes have four of them. Most people in the database do not have a phone number. The results of this experiment are shown in Table 3.

Rank	Algorithm	Optimization time	Execution Time	Total Time
1	6	0.0741	0.0729	0.147
2	4	0.104	0.127	0.231
3	3	0.1108	0.136	0.247
4	5	0.085	1.241	1.326
5	2	0.26	1.996	2.256
6	1	174.749	1.38	176.129

Table 3: Experiment 2 – more branches

Rank	Algorithm	Optimization Time	Execution Time	Total Time
1	6	0.07	2.117	2.1875
2	4	0.085	6.932	7.017
3	2	0.264	6.932	7.196
4	5	0.143	7.098	7.241
5	3	0.096	19.551	19.647
6	1	161.274	5.354	166.628

Table 4: Experiment 3 – longer branches

Algorithm 6 happened to do well in this case, in contrast to the first experiment where it had the worst execution time. It first chose ES for `Phone` (because there aren't many in the database), then FS for `Likes` which immediately narrows the search to people that have both a phone number and some likes. Other algorithms did not find this plan for various reasons. Algorithm 4, the ES-start algorithm, also did well. It chose `DB` and `Phone` as starting points (recall from Section 5.5 that names are always chosen as starting points) with a BS-based plan between them, and FS's for `Name` and `Likes`.

6.3.3 Experiment 3: Longer Branches

In our third experiment $s = \langle$`DB.People x, x.Person p, p.Likes l, l.Thing t1, p.Dislikes d, d.Thing t2`$\rangle$. Most people in the database have either likes or dislikes, but few have both, so this is a situation in which treating branches as indivisible units results in poor plans. Results are shown in Table 4.

Algorithm 6 again produces a good plan (the same plan is produced by Algorithm 3C, not shown in the table). In this plan, an ES for `Dislikes` followed by an FS for `Likes` narrows the search to people that have both likes and dislikes, without bothering yet with the actual things that they like/dislike. It is the interleaving of the execution of branches in the plan that results in good execution times. Poor decisions are made by Algorithms 2 and 3, which choose FS-based plans. Algorithm 5 does poorly because it requires branches to be executed indivisably.

6.3.4 Experiment 4: Weakness of ES

Our fourth experiment illustrates the weakness inherent in overusing the ES access method. While several ES operators joined using SMJ's can be competitive against multiple FS operators with NLJ's, a major drawback is that ES always considers the entire extent for a given label. Consider $s = \langle$`DB.Stores x, x.Store s, s.Name n, s.Location l, l.City c`$\rangle$. An ES for `Location` fetches not only the locations for stores, but also locations where movies were filmed. By contrast an FS for `Location` using

Rank	Algorithm	Optimization Time	Execution Time	Total Time
1	3	0.071	0.312	0.389
2	2	0.144	0.312	0.456
6	5	0.111	7.122	7.232

Table 5: Experiment 4 – weakness of ES

Post-optimization	Optimization Time	Execution Time	Total Time
None	0.26	1.996	2.256
A	0.342	0.623	0.965
B	0.364	0.62	0.984
C	0.311	0.24	0.551
D	2.383	0.229	2.612

Table 6: Post-optimizations for Algorithm 2 on Experiment 2

bindings for stores does well, since the number of stores in comparison to the number of locations in the database is small. Table 5 presents a few results for this experiment.

The best plan in this situation happens to be one with all FS access methods, and all of the algorithms except Algorithm 5 generate this plan. Since Algorithm 5 must optimize each branch separately, it is forced to use ES for `Location`. Notice that the query shape is actually very similar to Experiment 1 (Section 6.3.1), where Algorithm 5 produced the optimal plan, but the shape and distribution of the data being accessed is very different.

6.4 Post-Optimizations

In general, the post-optimizations improve query execution time at the expense of increased optimization time. As we saw in Section 6.2 (Table 1) with the good performance of the plans produced by Algorithm 2D, the net effect can be a win.

Recall that Post-optimization D is the most thorough, since it operates at the step granularity and doesn't apply any heuristics in its search. It is also the most expensive: it can add a second or even more to the optimization time. In our experiments, it decreased query execution time by an average of 22%, ranging from 0% faster (no change to the plan) to 88.5% faster. Obviously the benefit of post-optimization thus depends on whether the query itself is expected to be expensive.

To be more concrete, let us consider as an example the impact of each of our four post-optimizations on the plan produced by Algorithm 2 for Experiment 2 (see Section 6.3.2). Results are shown in Table 6. Algorithm 2 without post-optimization does very poorly in this experiment, and after applying Post-optimization D the new plan is almost an order of magnitude faster. However, the trade-off between better query performance and longer optimization time is evident with an increase in total time after post-optimization. In this situation, and in many others, we found that Post-optimizations B and C produce tangible improvements at a reasonable cost.

6.5 Comparison Against Exhaustive Search

We implemented the exhaustive search strategy described in Section 5.1 in order to compare the true lowest (predicted) cost plan against plans chosen by our six algorithms. Since exhaustive search is so expensive,

Algorithm	1	2A	2B	2C	2D	3A	3B	3C	3D	4	5	6
Average Times Optimal	1.23	1.35	1.12	2.38	1.08	2.19	1.26	2.30	1.29	2.05	2.25	2.60

Table 7: Summary of the average times worse than optimal

we were limited to considering path expressions with fewer than 6 steps, and even 5-step expressions were very slow to optimize. Overall our algorithms produced plans that were competitive with the optimal plan. We ran four representative experiments and calculated how much slower each plan was when compared to the plan selected by the exhaustive algorithm. Table 7 shows the average multiplicative increase in query execution time over all experiments when compared with the optimal plan.

We also considered some extreme points. For simple linear path expressions our algorithms did very well. In one case, all of our algorithms except Algorithms 4 and 6 produced the same plan as the exhaustive algorithm, and Algorithms 4 and 6 produced plans that were only 1.05 times slower. In the worst experiment, none of the algorithms generated the same plan as the exhaustive algorithm, some of the plans were 2 to 3 times slower than the optimal, and Algorithm 5 produced a plan that was nearly 6 times slower. However, as can be seen in Table 7, overall our algorithms do produce competitive plans. Furthermore, they do so in a small fraction of the optimization time.

7 Conclusions and Future Work

We investigated the query optimization problem for branching path expressions in semistructured and XML query languages. We introduced six algorithms that reduce the large search space of plans in a variety of ways, and four post-optimizations that transform complete plans into better ones at the expense of further optimization time. We implemented all of the algorithms and post-optimizations in the Lore system, and experimentally confirmed their strengths and weakness. Overall, we found that the best results were obtained by Algorithm 4 (ES-start), which generates plans in $O(n \log n)$ time.

Although we have considered a wide variety of algorithms already, based on the initial results reported here we plan to investigate some additional algorithms and combinations of techniques. We also would like to compare our algorithms against more traditional search strategies, such as top-down, dynamic programming style, or transformation-based search (e.g., simulated annealing or iterative improvement). Finally, since the algorithms were designed for branching path expressions in isolation, there is further work to be done to fold the techniques into a complete optimizer for Lorel queries.

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Delphi, Greece, January 1997.

[AQM$^+$97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, April 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.

[BPSM98] Editors: T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0, February 1998. W3C Recommendation available at http://www.w3.org/TR/1998/REC-xml-19980210.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, Tucson, Arizona, May 1997. Tutorial.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–422, Montreal, Canada, June 1996.

[CCY94] S. Chawathe, M. Chen, and P. Yu. On index selection schemes for nested object hierarchies. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 331–341, Santiago, Chile, September 1994.

[DFF$^+$99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. In *Proceedings of the Eight International World-Wide Web Conference*, Toronto, Canada, May 1999.

[FFK$^+$99] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 414–425, Seattle, Washington, June 1999.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[FLS98] D. Florescu, A. Levy, and D. Suciu. Query optimization algorithm for semistructured data. Technical report, AT&T Laboratories, June 1998.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Orlando, Florida, February 1998.

[GGT96] G. Gardarin, J. Gruser, and Z. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pages 390–401, Bombay, India, 1996.

[GLPK94] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, randomized join-order selection – why use transformations? In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 85–95, Santiago, Chile, September 1994.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, Philadelphia, Pennsylvania, June 1999.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[IK90] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 312–321, Atlantic City, New Jersey, May 1990.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MW99a] J. McHugh and J. Widom. Compile-time path expansion in Lore. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Isreal, January 1999.

[MW99b] J. McHugh and J. Widom. Query optimization for semistructured data. Technical report, Stanford University Database Group, September 1999. Extended version of [MW99c], available at ftp://db.stanford.edu/pub/papers/qo.ps.

[MW99c] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases*, Edinburgh, Scotland, September 1999.

[MWA$^+$98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University Database Group, 1998. Available at ftp://db.stanford.edu/pub/papers/semiindexing98.ps.

[ODE95] C. Ozkan, A. Dogac, and C. Evrendilek. A heuristic approach for optimization of path expressions. In *Proceedings of the International Conference on Database and Expert Systems Applications*, pages 522–534, London, United Kingdom, September 1995.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 314–325, Brisbane, Australia, August 1990.

[PGLK97] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 306–315, Athens, Greece, August 1997.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[RK95] S. Ramaswamy and P. Kanellakis. OODB indexing by class-division. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 139–150, San Jose, California, May 1995.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *QL '98 – The Query Languages Workshop*, Boston, MA, December 1998. Papers available online at http://www.w3.org/TandS/QL/QL98/.

[SAC$^+$79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, MA, June 1979.

[SMY90] W. Sun, W. Meng, and C. T. Yu. Query optimization in object-oriented database systems. In *Proceedings of the International Conference on Database and Expert Systems Applications*, pages 215–222, Vienna, Austria, August 1990.

[SS94] B. Sreenath and S. Seshadi. The hcC-tree: An efficient index structure for object oriented databases. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 203–213, Santiago, Chile, September 1994.

[Swa89] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–376, Portland, Oregon, May 1989.

[XH94] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 522–533, Santiago, Chile, September 1994.