Horizontal Query Optimization on Ordered Semistructured Data [*]

Hartmut Liefke
Dept. of Computer and Information Science
University of Pennsylvania
liefke@seas.upenn.edu

Abstract

The exchange and storage of XML data is becoming increasingly important. In contrast to conventional semistructured data [4, 1], the labels in a document-oriented representation such as XML are ordered. Furthermore, regular expressions (DTDs) describe the horizontal (and vertical) structure of the data. Traditional query languages for semi-structured data ignore the horizontal order and are therefore limited in their expressiveness and optimizability.

We describe a query language for querying ordered semistructured data. This query language provides primitives for specifying more powerful queries on ordered semistructured data. Furthermore, we describe how horizontal type information in DTDs is used to optimize queries based on finite automata.

1 Introduction

The exchange of data over the Internet has increased rapidly in the last few year. XML [3] has become an important standard for representing such data. The data model underlying XML is very similar to the semistructured data model [4, 1] representing a database as a directed graph with labeled edges. Many important issues about semistructured data have been studied, in particular query languages [7, 5, 2], optimization techniques [9, 13], auxiliary data structures (e.g. DataGuides [10] and T-indices [14]), and extraction of type information [15].

In contrast to the graph-based semistructured data model [1, 5], data in XML is *ordered*. The element labels of a given XML node are ordered. We consider the order of labels to be an important part of XML data itself. Furthermore, the order of labels is an important aspect of *document type descriptions* (DTD), which describe the structure of lists by regular expressions. Therefore, query optimizations based on DTDs must consider order.

In this paper, we will propose an intuitive extension of XML-QL to allow the querying of order on labels. Furthermore, we show how many queries can be optimized based on the information in the DTD and by *query unnesting*. Our running example is a publication database where a typical publication entry has the following XML format:

```
<Publ>  <Title> Theorems for free! </Title>
        <Author> P. Wadler </Author>
        <Year> 1989 </Year>
        <Conf> Functional Programming </Conf>
        ...
        <RTitle> Proofs and Types </RTitle>
        <RAuthor> J.-Y. Girard </RAuthor>
        <RYear> 1989 </RYear>
        <RTitle> Types, abstraction, ... </RTitle>
        <RAuthor> J.C. Reynolds </RAuthor>
        <RYear> 1983 </RYear>
        ...
</Publ>
```

Each publication has a title, an author, a publication year, and information about the conference (or journal). Furthermore, a (simplified) bibliography is stored by representing each reference as a triple of labels RTitle, RAuthor, and RYear. The following regular expression within a DTD describes the structure of a publication:[1]

```
<!ELEMENT Publ ( Title,Author+,Year?,(Journ | Conf),
                 (RTitle,RAuthor+,RYear?)*)>
```

Each publication has a title, at least one author, an optional publication year, and a journal or conference title. Furthermore, a publication has a list of references. Each reference has a title, a list of authors, and an optional publication year.

Note that in practice, each reference (i.e. the RTitle, RAuthor, and RYear elements) is often stored in a separate element Ref. However, we cannot generally assume that references are nested. For example, many conventional text formats, such as EMBL [11], and their straightforward XML representation have a flat representation for every entry.[2]

Several query languages for querying XML data have been proposed [7, 16]. We adopt an SQL-like select–from–where syntax. The following query $Q1$ extracts information about conferences in the database. For each conference publication, the query extracts the conference title and the publication year:

```
Q1: select <Publ> <CTitle>$c</> <Year>$y</></>
    from   <Publ> <Conf> $c </>,
                  <Year> $y </>,
           </> in db
```

For each binding of conference title c and year y within a publication, a new value <Publ>...</> is created.

The question of storing XML data efficiently is an area of active research. In principle, two different approaches are considered:

[*] This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO AASERT DAAH04-93-G0129, and ARPA N00014-94-1-1086.

[1] We omit unimportant details about the type of other elements.

[2] Note that any query languages with this nesting assumption would be restricted to XML data with a DTD that does not contain regular expressions of the form ...(A,B,...)*....

1. XML data is mapped into relational [8] or object-oriented databases. The advantage of this approach is that traditional solutions for transaction management, query optimization or indices can be adopted. Since sequences of elements are typically mapped into sets or relations, the order is ususally ignored.

2. XML data is stored in specific databases designed for semi-structured databases [2].

In this paper, we consider the second approach where XML data is internaly represented as a ordered tree (or graph). In the absence of auxiliary data structures, such as indices [14, 10], lists are processed by iterating over them. In query $Q1$, for example, labels in each publication are processed in two nested loops. In the outer loop, elements with label Conf are found and c is bound to the value. In the inner loop, the year variable y is assigned.

The first consequence of considering order is the possibility of new query optimizations based on the DTD. One can observe from the given publication DTD that Year and Conf occur at most once each and that the Conf always comes after Year. Intuitively, query $Q1$ could be answered by only *one single* iteration over each publication: First, the list is traversed to find label Year. From that position, the search continues to find label Conf. In this paper, we will show how query $Q1$ can be optimized accordingly.

A second issue about conventional semistructured query languages is that certain queries based on the order of labels are impossible to formulate. For example, consider the following query for the publications database: Extract the titles and publication years of all references:

```
select <Ref><Title> $t </> <Year> $y </></>
 from <Publ><RTitle> $t </> ... <RYear> $y </>
      </> in db
```

Pattern <RTitle>$t</> ... <RYear>$y</> should match each pair of reference title and reference year. However, "..." is not possible to express in semistructured languages. Informally it means that there can be an any number of elements between the two labels, but there should not be any element labeled RTitle or RYear. To formulate this precisely, we introduce a construct to bind a variable to the elements *between* two labels:

```
Q2: select <Ref><Title> $t </><Year> $y </></>
     from <Publ><RTitle> $t </> $g <RYear> $y </>
          </> in db,
          not exists (<RTitle></> in $g),
          not exists (<RYear></> in $g)
```

Pattern <RTitle> $t </> $g <RYear> $y </> matches each possible pair of RTitle and RYear elements such that the elements between RTitle and <RYear> are bound to variable $g. Thus, $g is a list itself. Conditions not exists (<RTitle></> in $g) and not exists (<RYear></> in $g) check whether there is no occurrence of RTitle or RYear in $g.[3] A naive and inefficient evaluation would consider all possible bindings for $t, $g, and $y and check conditions not exist ... for each of them. We will illustrate how this query can be answered using one single iteration over the labels of each publication.

[3] A user-friendly query language might also allow the user to specify regular expression for such conditions.

In Section 2, we describe a data model and the query algebra *OPAL* for semistructured lists. Section 3 explains how DTDs and expressions are represented as finite state automata. In Section 4, we show how query expression are typed based on automata intersection. The optimization of nested automata is described in Section 5, we show some preliminary experimental results in Section 5 and we conclude in Section 6.

2 The XML Data Model and the *OPAL* Algebra

An XML document can be modeled as am ordered edge-labeled tree. The edge label represents the element tag and a node is either an element itself (pointing to subnodes) or a text value. Elements in XML can be annotated with a set of attributes. We ignore attributes for the purposes of this paper. Furthermore, we only consider tree-structured databases instead of a more general graph-based data model as found in semistructured data [4, 1].

Our XML data-model is based on lists whose elements are pairs of labels and lists. We use $[a_1 : l_1, \cdots, a_n : l_n]$ to denote a list with n elements such that the element at position i ($1 \leq i \leq n$) has label a_i and value l_i. For example, $[b : [x : [], x : [3 : []]]]$ denotes a valid list. For simplicity, we omit details about base values, such as integers and booleans and their base operations.

Queries $Q1$ and $Q2$ in Section 1 are given in a surface syntax that resembles standard database query languages such as XML-QL [7], and OQL [6]. Similarly to the well-known relational algebra, queries are transformed into a more manipulable (i.e. optimizable) algebraic form, called *OPAL* (Ordered list Processing ALgebra). For simplicity, we only present the relevant fragment of *OPAL* :

$$
\begin{aligned}
e \;\; ::= \;\; & v \mid [] \mid [a : e] \mid e_1 + e_2 \mid \cdots \mid \\
& [e_2 \mid \langle l + a : v + r \rangle \leftarrow e_1] \mid \exists \langle l + a : v + r \rangle \in e : c
\end{aligned}
$$

The empty list is denoted as $[]$. $[a : e]$ denotes a singleton list whose element has label a and value e, and $e_1 + e_2$ denotes the concatenation of two lists. The most important construct for operating on lists is the *list iteration* $[e_2 \mid \langle l + a : v + r \rangle \leftarrow e_1]$, which iterates over the elements of list e_1. The construct resembles the comprehension construct, which is well known for structured [17] and unstructured data models [5]. For each element x, the label and the value are bound to a and v, respectively. The elements of list e_1 preceding element x are bound to list l and the elements *after* x are bound to r. For example, the list iteration $[\cdots \mid \langle l+a:v+r \rangle \leftarrow [x : [], y : [3 : []]]]$ has two possible bindings, $(l = [], a = x, v = [], r = [y : [3 : []]])$ and $(l = [x : []], a = y, v = [3 : []], r = [])$. Finally, for each element binding $\langle l+a:v+r \rangle$, e_2 is evaluated and the resulting lists are concatenated.

Lastly, the existential quantification $\exists \langle l + a : v + r \rangle \in e : c$ tests whether condition c evaluates to true for some element of list e. Condition c is evaluated under the binding pattern $\langle l+a:v+r \rangle$. Note that we omit other important construct, such as operators for dividing a given list into an head element and a tail list, for "deeply" traversing trees or a group-by operator. The techniques described in this paper also apply for more powerful sets of constructs.

Figure 1 shows the algebraic representation of queries Q1 and Q2 from Section 1. We use abbreviation $[e_2 \mid \langle l+\mathsf{label}:v+r \rangle \leftarrow e_1]$ for $[\mathsf{if}\ a = \mathsf{label}\ \mathsf{then}\ e_2\ \mathsf{else}\ [] \mid \langle l + a : v + r \rangle \leftarrow e_1]$. Variable v in Q1 is sequentially bound to each publication. There

$$\text{Q1:}\quad [[[[\mathsf{Publ} : [\mathsf{CTitle} : c, \mathsf{Year} : y]] \mid \langle l'' + \mathsf{Year}{:}y + r''\rangle \leftarrow v] \mid \langle l' + \mathsf{Conf}{:}c + r'\rangle \leftarrow v] \mid \langle l + \mathsf{Publ}{:}v + r\rangle \leftarrow \mathsf{db}]$$

$$\begin{aligned}\text{Q2:}\quad & [[[\mathsf{if\ not}(\exists\langle l''' + a{:}x + r'''\rangle \in g : a = \mathsf{RTitle} \vee a = \mathsf{RYear})\ \mathsf{then}\ [\mathsf{Ref} : [\mathsf{Title} : y, \mathsf{RYear} : y]]\ \mathsf{else}\ [] \\ & \quad \mid \langle g + \mathsf{RYear}{:}y + r''\rangle \leftarrow r'] \mid \langle l' + \mathsf{RTitle}{:}t + r'\rangle \leftarrow v] \mid \langle l + \mathsf{Publ}{:}v + r\rangle \leftarrow \mathsf{db}]\end{aligned}$$

Horizontal Loop Fusion:

$$[e_3 \mid \langle l + a{:}v + r\rangle \leftarrow e_1 + e_2] \implies \begin{array}{l}[e_3\{r \setminus r + e_2\} \mid \langle l + a{:}v + r\rangle \leftarrow e_1] + \\ [e_3\{l \setminus e_1 + l\} \mid \langle l + a{:}v + r\rangle \leftarrow e_2]\end{array}$$

$$\begin{aligned}\text{Q1':}\quad [[\quad & [[\mathsf{Publ} : [\mathsf{CTitle} : c, \mathsf{Year} : y]] \mid \langle l'' + \mathsf{Year}{:}y + r''\rangle \leftarrow l'] + \\ & [[\mathsf{Publ} : [\mathsf{CTitle} : c, \mathsf{Year} : y]] \mid \langle l'' + \mathsf{Year}{:}y + r''\rangle \leftarrow r'] \mid \langle l' + \mathsf{Conf}{:}c + r'\rangle \leftarrow v] \mid \langle l + \mathsf{Publ}{:}v + r\rangle \leftarrow \mathsf{db}]\end{aligned}$$

Figure 1: The *OPAL* representation of queries $Q1$ and $Q2$ and horizontal loop fusion of $Q1$

are two nested loops over v: The outer loop finds elements with label Conf and the inner loop searches for labels Year. The singleton list [Publ : [Conf : c, Year : y]] is constructed for each pair of those elements and the generated results are concatenated to form the result of the query.

Similarly, Q2 binds variable v to each publication and the outer loop searches for elements with label RTitle. The inner loop, however, iterates over the elements *after* element RTitle, represented by r'. For each occurrence of RYear in r', the variable g is bound to the left part of r' and therefore represents the list between the pair of RTitle- and RYear-elements. Finally, condition $\mathsf{not}(\exists\langle l''' + a : x + r'''\rangle \in g : \cdots)$ tests whether g does not contain any labels RTitle or RYear. In this case, the result [Ref : [Title : y, RYear : y]] is constructed. Otherwise, the empty list is returned. The concatenation of all singleton lists [Publ : $\cdots$] is the result of the query.

2.1 Rewriting of *OPAL* expressions

OPAL provides a framework for various powerful optimizations. Many of the optimizations resemble well-known rewrite rules for comprehension syntax [17]. Most importantly, it is possible to rewrite $[e_2 \mid \langle l + a : v + r\rangle \leftarrow e_1]$ where e_1 is a complex list construct itself. *Horizontal loop fusion* (Fig. 1) allows us to reduce list iterations on the list concatenation $l_1 + l_2$ by iterating over l_1 and l_2 separately. Another important rewrite rules, called *vertical loop fusion*, allows us to optimize compositions of list iteration constructs: $[\cdots \mid \langle l' + a'{:}v' + r'\rangle \leftarrow [\cdots \mid \langle l + a{:}v + r\rangle \leftarrow \cdots]]$.

A somewhat more subtle rewriting is based on the following observation: The pattern $\langle l + a{:}v + r\rangle$ in construct $[e \mid \langle l + a{:} v + r\rangle \leftarrow s]$ divides lists s into three parts, l, $[a : v]$, and r. Thus, each occurrence of s in e can be replaced by expression $l + [a : v] + r$.

We can replace v in the innermost loop of Q1 by $l' + [\mathsf{Conf} : c] + r'$. Thus, we can rewrite the innermost loop using horizonal loop fusion. We obtain three separate iterations over l', [Conf : c] and r', respectively. Query Q1' in Figure 1 shows transformed query Q2. The iteration over [Conf : c] can be eliminated, since label Conf does not match Year.

Intuitively, we obtain two separate iterations over the publication v, one on the fragment of v *left* of [Year : y] and one on the part of v *right* of [Year : y]. As we will describe later, it is possible to eliminate the second iteration using the DTD, since label Year does never occur *after* Conf.

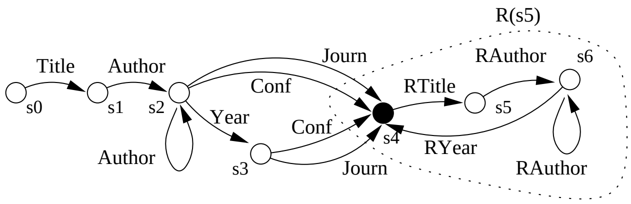

Figure 2: The DTD automaton for Publ

3 Processing Queries through Finite State Automata

In order to answer the queries given above, it is necessary to iterate over the elements of lists. Auxiliary data structures such as indices can help to avoid iterating over the entire list. In this paper, however, we ignore the interesting issues connected to indices and we will focus solely on a simple model where XML is stored as an ordered tree.

One of the most natural concepts for extract information from sequences are finite state automata. In particular, (finite state or pushdown) automata are used to verify that a given word is a part of a (regular or context-free) language[12]. In this paper, we consider finite state automata (FSM) as the main primitive for iterating over lists for two reasons:

- FSMs have many desirable characteristics and they are relatively easy to manipulate.
- The DTD specifies for each element a regular language that describes the acceptable element sequences with such element. There is a one-to-one correspondence between regular languages and FSMs.

We consider two types of automata: *DTD automata* and *query automata*. A DTD automaton represents the regular expression corresponding to an element in the DTD. The edges of DTD automata are annotated with the corresponding element labels. Furthermore, each element label has an associated DTD automaton or is marked as a text (i.e. PCDATA) element. Figure 3 shows the DTD automaton of the given publication DTD in Section 1.

A DTD automaton only accept lists that comply to the DTD. Since DTDs are deterministic regular expressions, the transformation into a deterministic finite state machine does not

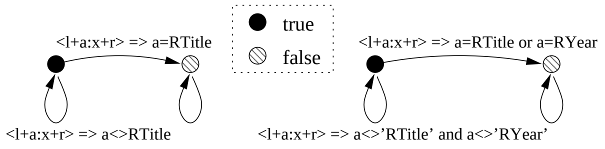

Figure 3: Condition automata for query Q2

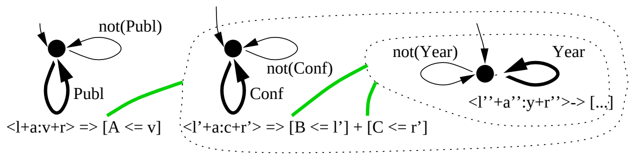

Figure 4: The list automaton for query Q1

lead to exponential growth as in the general case of general regular expressions.

A *query automaton* generates a new value for a given list. The result can be used in complex *OPAL* queries and the corresponding expression is denoted as $\lfloor \mathcal{A} \Leftarrow l \rfloor$, where l is a list that is traversed by query automaton $\mathcal{A}$. We distinguish two types of query automata: *condition automata* represent boolean expressions over lists and *list automata* compute new lists from a given list. In contrast to DTD automata, edges are not labeled with names, but with condition expressions of the form $\langle l+a:v+r \rangle \Rightarrow c$. Similar to the *OPAL* constructs, the pattern $\langle l+a:v+r \rangle$ binds variables l, a, v, r to the currently traversed list element and its preceeding and following list. Condition c determines whether for a given element the automaton should move to the successor state. The automaton is deterministic the condition of at most one outgoing edge for a given state can be true.

A condition automata is a query automaton with each final state marked as true or false. The automaton expression evaluates to true or false, if the final state of the iteration is marked as true or false, respectively. It is possible to compose condition automata by conjunction, disjunction, or negation. Figure 3 shows the condition automata for conditions $\mathsf{not}(\exists \langle l+a:x+r \rangle \in g : a = \mathsf{RTitle})$ and conjunction $\mathsf{not}(\exists \langle l+a: x+r \rangle \in g : a = \mathsf{RTitle}) \wedge \mathsf{not}(\exists \langle l+a:x+r \rangle \in g : a = \mathsf{RYear})$.

A *list automaton* traverses the given list and constructs a new list by evaluating and concatenating expressions that are associated with the edges. Edges are optionally annotated with some expression $\langle l+a:v+r \rangle \Rightarrow e$. In the rest of the paper, we represent these edges as *bold* arrows. If the condition of a bold edge is true, then e is evaluated under the current binding. The result of e is concatenated to the already computed lists from previous bold edges. Figure 4 shows the query automaton for Q1. The automaton on the left side iterates over all publication. For each publication, automaton A finds all Conf labels. For each Conf label, the list left (right) of this label is traversed by automaton B (C, respectively). Note that B and C are equivalent in this example.

4 Query Typing and Pruning

One of the key aspects of query optimization is to take advantage of available information about the structure of the given data. In the context of XML, the regular expressions described in the DTD play an important role for optimization. For example, consider the DTD in Figure 3. One can easily observe that element Year can only occur directly before element Conf. Therefore, automaton C in $\lfloor C \Leftarrow r' \rfloor$ in Figure 4 could be eliminated since r' denotes the elements after element Conf.

In this section, we will describe the key ideas of query typing. Due to space restrictions, we will only illustrate the mechanism of query typing using query $Q2$. The same technique, however, can be applied to query Q1. For the moment, consider the left column of Figure 5. Similar to Q1 in Figure 4, it represents the query automaton of query $Q2$. The root automaton iterates over the elements of a publication to find element RTitle. For each RTitle element, automaton A searches through the rest of the list (r') to find label RYear. Finally, automaton A' verifies that the elements between RTitle and RYear do not have labels RTitle or RYear themselves.

We can make the following observation: The root automaton *traversing* over the elements of a publication can be "merged" with the DTD automaton (Fig. 3) *describing* the elements of a publication. The new automaton (shown in the middle column of Figure 3) is construct by *intersecting* both automata: Each edge in the new automaton corresponds an edge in the DTD and an edge in the original query automaton. Intuitively, the neq query automaton performs the same operations as the untyped automaton, but emulates the DTD automaton at the same time.

The core operation needed is called *automata intersection*. Two automata $\mathcal{A}$ and $\mathcal{A}'$ are intersected by forming the crossproduct of the states of the original automata. Two states (s_1, s'_1) and (s_2, s'_2) are connected through an edge, if there exists an edge from s_1 to s_2 in $\mathcal{A}$ and an edge from s'_1 to s'_2 in $\mathcal{A}'$. Intuitively, the edge conditions and expressions of the new automaton are taken from the original query automaton. Since the specific label name is given by the DTD edge, we can substitute the corresponding variable a with the label name and the annotation expressions can often often be simplified. Most importantly, if an edge condition (simplified) can never evaluate to true, it can be removed.

In order to type the automaton A in $\lfloor A \Leftarrow r' \rfloor$, we must determine the type of list r'. Recall that r' represents the elements of the publication that come after the RTitle edge. In the DTD automaton in Figure 3 the RTitle edge leads to state s_4. Therefore, the list r' must be parsable by a DTD automaton with the new start state s_4. In other words, the DTD automaton of r' can be constructed by identifying all states and edges in the original DTD automaton that are reachable from state s_4 (see Figure 3). We denote this automaton as the *right-adjoined automaton* of s_4, or $R(s_4)$.

Similarly, variable g in $\lfloor A' \Leftarrow g \rfloor$ must be typed by the *left-adjoined automaton* of s_6 – the start state of edge RYear in the DTD automaton. The left-adjoined automaton contains all edges and states that lead from the start state to state s_6. To be more correct, the left-adjoined automaton of s_6 must be determined in the previous DTD automaton $R(s_4)$, since all bindings $\langle g+a'':y+r'' \rangle$ are with respect to list r'.

It is possible to type query Q1 in a similar fashion. As mentioned earlier, the iteration over the elements after Conf can be eliminated, since no Year element is expected. The cornerstone of this elimination is called **automata pruning**. Intuitively, the iteration over a given list can be stopped at some state s, if the output of the automaton will not change anymore. For condition automata, this the case if all final states reachable

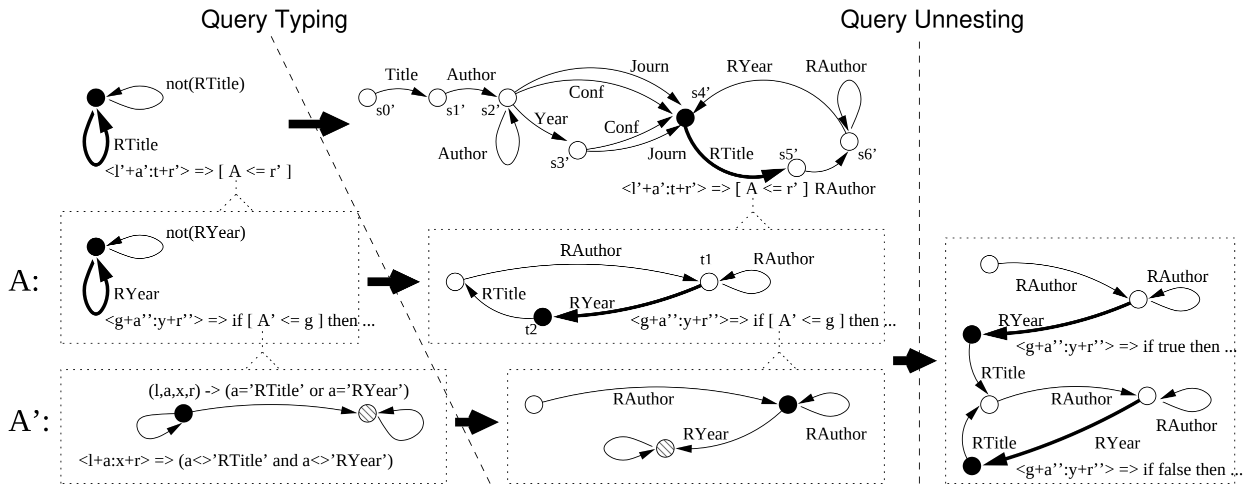

Figure 5: Typing of Query Q2

from s have the same boolean value. For list automata, all reachable edges must return an empty list as its output. The typed automaton for $\lfloor C \Leftarrow r' \rfloor$ in Figure 4 will not contain any bold edges. Therefore, the automaton can stop at the start state or be completely eliminated.

5 Optimizations by Automata Unnesting

XML queries often have the characteristic that the same list is accessed and traversed multiple times. For example, consider automata A and A' in Figure 5. Automaton A iterates over r' to identify elements with label RYear. Automaton A' iterates over the prefix g of list r' in order to check that there is no label RYear and RTitle before the current element RYear.

Intuitively, it is desirable to compute the result of automaton A' on g *while* automaton A traverses list r' to find element RTitle. Thus, the result of condition automaton A' is already computed when the element RTitle is found. The automaton in the right column of Figure 5 shows the corresponding unnested automaton. Note that expression $\lfloor A' \Leftarrow g \rfloor$ is replaced by true or false - depending on the final state reached in A'.

Similar to query typing, the basic priniple of automata unnesting is the intersection of automata. In the example, automaton A' is intersected with the left-adjoined automaton of state t_1 in automaton A. Each final state in the intersection is either marked as true or false depending on the final state reached in A'. The iteration expression $\lfloor A' \Leftarrow g \rfloor$ at edge RYear is substituted with the boolean value of the corresponding final state.

Similar to unnesting of condition automata, it is possible to unnest list automata. The unnesting of automata on variables representing the left part of a list is denoted as *left-unnesting*. It is also possible to unnest an automata on the right part of a list. Due to space limitations, we omit the technical details of unnesting. The final results of right-unnesting for queries Q1 and Q2 are shown in Figure 6 and 7. One can observe that both queries only consist of one single iteration over the labels within a given publication.

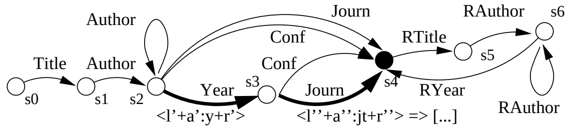

Figure 6: Query Q1 after left- and right-unnesting

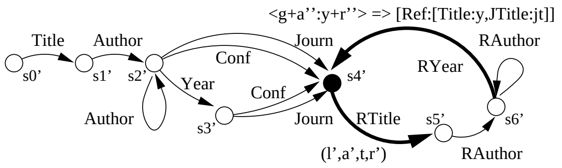

Figure 7: Query Q2 after right-unnesting

6 Experimental Results (Preliminary)

We have implemented a prototype in Java that optimizes *OPAL* queries by query typing, pruning, and automata unnesting. As a sample database, we used a subset of the DBLP publication database [4] containing the conference publications of 1996 for SIGMOD, VLDB, and ICDE. We used an extended version of *OPAL* to extract the corresponding information from the bibliography XML data sources. The generated XML file contains 214 publications with an average of about three authors. Each publication has an average of 7.4 references[5] and the file has a size of about 400KByte.

We tested queries Q1 and Q2 in four different optimization stages: 1) unoptimized, 2) after query typing, 3) after automata pruning, and 4) after automata unnesting. For each query and each variation, we measure the overall number of element accesses. An element access occurs whenever an element is visited through an edge in the automaton and whenever we must evaluate some expression for this element depending on

[4] `http://www.informatik.uni-trier/ ley/db/about/instr.html`

[5] ICDE'96 publications did not include any references, thus lowering the average.

the element label or value.[6] The following table shows the overall number of element accesses:

	Q1	Q2
Unoptimized	17625	215063
Typed	13389	477022
Pruned	1669	185501
Unnested	1049	8492

The query typing itself without query pruning does not improve the cost substantially. The transformation of construct $\exists\langle l + a : v + r\rangle \in e : c$ (which only iterates over the list e until condition c is satisfied) into an automaton which iterates over the *entire* list e leads to the higher cost for query Q2.

The effect of automata pruning is substantial for query Q1. Intuitively, the traversal can stop after the label Conf has been found. Therefore, many elements that follow element Conf do not have to be accessed. The pruning is not as effective for query Q2. This is the case because query Q2 must fully traverse each publication until the last reference is reached. Therefore, it is not possible to stop the traversal earlier.

Query unnesting reduces the query cost of query Q2 substantially. Naturally, the nested iteration for finding RTitle and RYear can be merged into one single, much more efficient iteration over each publication. In contrast to Q2, the query unnesting for Q1 only leads to a rather small cost improvement.

7 Conclusion

In this extended abstract, we developed novel techniques for specifying and optimizing queries on ordered semistructured data. We illustrated how automata representing the queries can be optimized using *query typing* and *automata unnesting*.

The query optimization substantially reduces the number of objects accessed. In practice, however, the cost improvement depends on the chosen storage model, caching and prefetching stategies. Therefore more experiments on the effect of the optimization are necessary.

Horizontal query optimization is orthogonal to existing vertical optimizations [9]. In future work we want to investigate the interaction between auxiliary data structures, such as T-indices [14] and horizontal query optimization. Furthermore, the combination of ordered and unorder data must be investigated. Often, the order of the input (or the output) is not relevant, which provides space for more optimizations.

Acknowledgments: Many thanks to the Penn Database Research Group for numerous helpful discussions.

References

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

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1996.

[3] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. *W3C Recommendation REC-xml-19980210*, 1998. Available as `http://www.w3.org/TR/1998/REC-xml-19980210`.

[4] P. Buneman. Tutorial: Semistructued data. In *Proceedings of ACM Symposium on Principles of Database Systems*, 1997.

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

[6] R. G. G. Cattell and et.al., editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Mateo, California, 1997.

[7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: a query language for XML. *W3C Note Note-xml-ql-19980819*, 1998. Available as `http://www.w3.org/TR/NOTE-xml-ql`.

[8] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data*, Philadelphia, May 1999.

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

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

[11] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[12] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[13] J. McHugh and J. Widom. Query optimization for XML. *To appear in VLDB'99*, September 1999.

[14] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, pages 277–295, Jerusalem, Israel, January 1999.

[15] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of ACM Symposium on Principles of Database Systems*, Philadelphia, June 1999.

[16] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *W3C Query Languages Workshop (QL'98)*, Boston, December 1998.

[17] Limsoon Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

[6] If a state has one outgoing edge, then the condition need to be evaluated and the access to the element can be avoided (if the edge does not have a list construction expression).