Template Folding for XPath

Carl-Christian Kanne Guido Moerkotte
Department of Mathematics and Computer Science, University of Mannheim, Germany
{cc|moer}@informatik.uni-mannheim.de

ABSTRACT

We discuss query evaluation for XML-based server systems where the same query is evaluated on every incoming XML message. In a typical scenario, many of the incoming messages will be highly similar to each other. Current XML query evaluators reevaluate the query from scratch on every message.

We call substructures that occur in many input documents *template fragments*, and introduce a novel *template folding* method that allows to move the work of evaluating the query on recurring document substructures from the query execution engine into the query compiler. Similar to *constant folding*, our method avoids run-time evaluation of intermediate results whose value only depends on information that is already available at compile time. For XPath location paths, we propose a representation for such invariant intermediate results, and show how it can be incorporated into query execution plans. Such augmented execution plans improve query performance when evaluating the same query on subsequent input documents.

1. INTRODUCTION

1.1 Motivation

The Web is currently developing from a one-way medium into an active distributed system. Many of the associated protocols are based on the exchange of XML documents, for example for automatic notification about events of interest [11], more flexible application architectures [7], and Web Services [8]. Hence, no matter which platform is used for the implementation of the participating nodes, one typical ingredient is an XQuery/XPath query processor [1, 5, 6] that processes the incoming XML documents, possibly generating result documents that are being sent as reply, or forwarded to other participants. Typically, there are only a limited number of previously known queries, which are evaluated on every incoming document.

Many of the messages exchanged in such communication are highly similar, even if the underlying protocol, or schema, allows for a flexible message structure. One of the reasons is that in many application domains certain kinds of messages make up most of the traffic. For example, in a stock exchange or a marketplace environment, most of the traffic will consist of price quotes, whereas a logistics system will mostly see tracking messages as packages move through distribution centers. Another reason for similarity of XML messages is that they are generated by a small number of applications and tools that have a large market share in a particular domain. This makes the messages similar down to the level of whitespace and other formatting details.

Our work investigates how to exploit the high similarity of messages for boosting XML query processing performance. As a first result, this paper discusses the evaluation of XPath location paths on recurring document substructures.

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope">
  <Header>
    <d:statusupdate xmlns:d="http://deliver.com">
      <d:packageID>176176</d:packageID>
    </d:statusupdate>
  </Header>
  <Body>
    <d:packageinfo xmlns:d="http://deliver.com">
      <d:handlercomment>
        Package was accidentally dropped.
      </d:handlercomment>
    </d:packageinfo>
  </Body>
</Envelope>
```

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope">
  <Header>
    <d:statusupdate xmlns:d="http://deliver.com">
      <d:packageID>671176</d:packageID>
    </d:statusupdate>
  </Header>
  <Body>
    <d:packageinfo xmlns:d="http://deliver.com">
      </d:deliverystatus value="OK" />
    </d:packageinfo>
  </Body>
</Envelope>
```

Figure 1: Two similar invocations of a web service

Example 1 Fig. 1 shows two examples taken from a large number of similar XML messages sent to a package tracking server. Assume the server evaluates the query
`/Envelope/Header//d:packageID/text()`
on every incoming message.

An analysis of a large collection of sample input messages can easily detect that a large fraction of the incoming messages contain a common fragment, shown in boldface in Fig. 1. Note that this rigid message structure is not enforced by the schema, which is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
XIME-P 2006, 3rd International Workshop on XQuery Implementation, Experiences and Perspectives, June 30, Chicago, Illinois
Copyright 2006 ACM 1-59593-465-0/06/0006 ...$5.00.

assumed to also allow more flexible content in both header and body.

In our case, the common fragment includes a `Header` tag that contains a `statusupdate` with a single nested `packageID` tag. Only the actual text node containing the packageID is always different, hence not part of the repetitive pattern. To return this text node, evaluation of our example query will, for each document, navigate through the invariant initial fragment of the document, reaching the `packageID` node. The required steps are the same for each document, but are, nevertheless, reexecuted every time. □

Our goal is to amortize the evaluation cost of a query on identical, recurring substructures across a large collection of input documents, significantly improving query performance.

1.2 Approach

We call recurring substructures that are part of many input messages *template fragments*. Our approach assumes that an analysis or training phase has discovered such template fragments in a typical input collection.

Our main contribution is a method to generate, given a particular XPath query, query execution plans that contain all information from the template fragments which is relevant with respect to the query. In other words, our method allows to move the work of evaluating the query on the template fragments from the query execution engine into the query compiler. We call our technique *template folding*, in allusion to *constant folding*. Constant folding avoids run-time evaluation of expressions whose value only depends on information that is already available at compile time. A compiler folds such constant expressions by evaluating them at compile time and emitting the result as a constant in the compiler's output.

Our situation is similar: We want to avoid a costly run-time evaluation of our query on document fragments that are already known at compile time, the template fragments. However, while we know at compile time which template fragments may occur, they do not represent true constants: We can only find out at run time how the template fragments are combined with variable fragments to represent a particular input document.

We propose to generate template-aware query execution plans in a three-stage process. First, the compiler produces a tentative execution plan. Second, it runs this plan on the template fragments using the execution engine. When all of the template fragments have been fed into the tentative plan, the state of the execution engine is "frozen" and captured. This state contains the knowledge about the template fragments in a form that is ideally suited for further query processing. In the third and final step, we augment the tentative execution plan with the captured information.

The augmented execution plan can now be used to evaluate queries on fragmented documents by using only the variable fragments of the document, since all relevant information about the template fragments is already contained in the execution plan.

1.3 Roadmap

We specify our fragmented document model in Sec. 2. In Sec. 3, we explain how to efficiently create a fragmented representation of incoming documents. An important ingredient for our method are some techniques for out-of-order evaluation of XPath expressions, which we recapitulate in Sec. 4. Sec. 5 elaborates on our *template folding* method. In Sec. 6, we discuss related work, and Sec. 7 concludes the paper.

2. DATA MODEL

In the environment under consideration in this paper, documents are delivered as streams of characters over some communication channel. XML query evaluation, however, is based on a tree data model. This section briefly sketches our logical model of XML documents. To represent the fact that documents are built from template and variable fragments, we also introduce a fragmented tree representation. This representation forms the foundation for the evaluation techniques in the remainder of the paper.

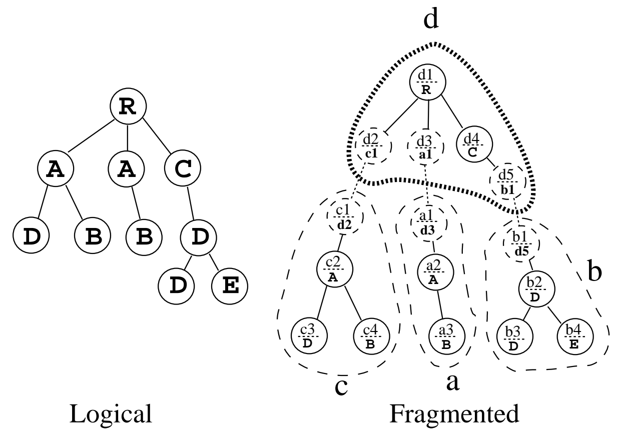

Figure 2: A logical tree and its fragmented representation

2.1 Logical Tree Model

We describe XML documents using a labeled, ordered tree model, where nodes are labeled with tags taken from a tag alphabet Σ. For brevity reasons, we do not discuss XML node types such as text nodes, attributes, namespace nodes, and entities. They can be incorporated into our path evaluation method without difficulty.

2.2 Fragmented Tree Representation

We assume that an input document consists of a number of fragments, which correspond to connected subtrees of the document tree. Some of these fragments are already known at query compilation time, and some are only known at query execution time. We call the former *template fragments* and the latter *variable fragments*.

Essentially, fragments are represented as logical trees as explained above. In addition to the *core nodes* representing logical document nodes, fragments may also contain *border nodes*. Border nodes are used to represent the inter-fragment edges that connect template and variable fragments to form a complete document. Border nodes must be either root nodes or leaf nodes in a fragment. One leaf and one root border node form a pair that represents an inter-fragment edge.

Example 2 Fig. 2 shows a logical tree and a corresponding fragmented tree. The fragments `a`–`d` are represented by the dashed lines. Each node contains a node ID above the dashed line. The core nodes have their tag name below the dashed line, while the border nodes at the ends of inter-fragment edges have their companion border node's NodeID below the dashed line.

In our example, fragment `d` is a template fragment, corresponding to the invariant envelope structure from Example 1, marked in boldface in Fig. 2. Fragments `a`–`c` are variable fragments that are different for every new message. □

A *fragmented tree* is a set of fragment subtrees. We designate with C the set of all core nodes and with B the set of all border nodes. One of the nodes from C is the designated *root node* r of the fragmented tree (in our example, r =d1). Further, we have a function $target(x)$, which returns for all border nodes $x \in B$ the

node ID of a companion border node.

A fragmented tree may be mapped to a logical tree by simply replacing border node pairs by a simple edge. Note that in some cases, some of the available template fragments are not part of a concrete input document. In these cases, there may be border nodes without a companion border node, i.e. the $target()$ function refers to a nonexistent node ID. The nodes of fragments containing such border nodes do not belong to the corresponding logical tree and can be ignored.

3. TEMPLATE PARSING

Our method relies on a *fragmented* representation of the input documents, i.e. the input documents must be partitioned into the template fragments that are known at compile time, and occur in many documents, and the variable fragments that are different for every document. The challenge is to efficiently detect the template fragments in the incoming message, and to create a fragmented representation.

A correct, but inefficient method would be to parse the input document into a logical tree (as defined in Sec. 2.1), and then search the tree for occurences of any template fragments using pattern matching. If template subtrees are found, appropriate border nodes are inserted. Unfortunately, this method would be so expensive that it outweighed the performance gains of our template folding method.

We claim that it is much more efficient to generate the fragmented representation using the parser. Instead of using a regular, generic XML parser, we propose to specify a grammar that incorporates the template fragments in its productions. Using a compiler construction tool, we can generate a custom parser to find the template fragments.

Example 3 In our example from Fig. 1, there are some character sequences that occur in both documents. Only the contents of `s:packageID` and the `s:packageinfo` tag vary. This corresponds to a single template fragment which looks like the logical tree of any of the documents, with the variable subtrees replaced with border nodes.

To generate a fragmented representation with such a template fragment, we can use a parser based on a grammar containing a production like

```
packagestatus ::= A packageid B packageinfo C
```

where A, B and C are terminal symbols that correspond to the strings that represent our template fragment, i.e.

```
A = <Envelope xmlns="....com">\n <d:packageID>
B = </d:packageID>\n ...</Header>\n <Body>
C = </Body>\n </Envelope>
```

The nonterminals `packageid` and `packageinfo` correspond to rules that match the content of those tags, for the former a string, and for the latter some kind of (possibly recursive) XML schema.

Attributing the grammar with proper tree construction actions results in a fragmented representation of the document. The production `packagestatus` would correspond to one template fragment, whereas the productions for packageid and packageinfo must create variable fragments, consisting of a logical tree for the contents with an additional border node at the top that references the border nodes in the template fragment. □

If the input document cannot be parsed by the custom parser, it is parsed from scratch using a generic XML parser, as usual. Note that this means that we are processing an "uncommon" type of message, and hence the overhead of a failed parsing attempt can be tolerated. In addition, in a typical case the lexer will not even recognzie the inital A token, and the custom parser will stop quite early. The result of the regular parser can be fed to our fragmented evaluation algorithm, as a fragmented document with just a single variable fragment referencing no template fragments.

4. PARTIAL PATH PROCESSING

Before we elaborate on our template folding technique, we give a short introduction into *partial path processing*, a technique originally introduced in the native XML data store Natix [9] to avoid random access and allow efficient physical access patterns for navigational queries without sacrificing an algebraic approach. The original technique is a good starting point to implement our template folding method, because in Natix, persistent XML documents are stored in a fragmented way similar to our representation from Sec. 2.2. In Natix, a fragment represents a physical storage unit of limited capacity, whereas our fragments are derived from patterns in the input documents.

In Sec. 4.1, we discuss the notion of *partial path instances* that describe partially evaluated paths. In Sec. 4.2, we present an algebra capable of expressing execution plans that allow to separately evaluate a query on the different fragments of a document.

4.1 Path Instances

Query evaluation on the fragments produced by our approach from Sec. 3 require a representation of a path evaluation that was interrupted after a few steps because a fragment border was encountered, and the target fragment is not yet available. We use *partial path instance* to represent such intermediate results. They are needed (1) for evaluating the queries on the template fragments during query compilation, (2) for evaluating the queries on the variable fragments during query execution, and (3) to transfer the preprocessed results from the compilation to the execution phase by means of the query execution plan.

4.1.1 Location Paths

Let π be a location path expression in XPath. We denote with $|\pi|$ the number of location steps in π. π_i is the ith location step ($1 \leq i \leq |\pi|$), $axis_{\pi_i}$ is the step's axis, and nt_{π_i} is its node test. In our model, node tests are specified as a subset of the tag alphabet Σ, describing the node tags that are allowed at this step. This limits our method to a subset of XPath predicate types, but as we will see later, our physical algebra expressions are compatible with a a more expressive algebra that provides full XPath support, although more advanced language constructs cannot benefit from our template folding technique yet.

4.1.2 Full Path Instances

A *full path instance* is a map p from the steps $0, 1, \ldots, |\pi|$ of a location path π to the set C of core nodes, mapping each step i to a core node $p_i \in C$. The 0th location step π_0 represents the context nodes on which to evaluate π, so that p_0 is the context node where the path instance originates.

Of course, we require for each $0 < i \leq |\pi|$ that p_i is reachable from p_{i-1} using step π_i.

Let π be a location path, c a context node, and r a result node reached by π starting from c. Then a full path instance p with $p_0 = c$ and $p_{|\pi|} = r$ can be considered a certificate for the fact that r is indeed a result node of p, describing how r can be reached from c.

4.1.3 Partial Path Instances

A *partial path instance* is a map p representing a fragment of a path instance.

Given the set of border nodes B (see definition in Sec. 2.2), and ϵ as a null value, p is a map $p : \{0, 1, \ldots, |\pi|\} \longrightarrow C \cup B \cup$

No	π_0 Context	π_1 /A	π_2 //B	l	r	F	L	R	C
1	d1	ϵ	ϵ	0	0	-	+	+	+
2	d1	a2	ϵ	0	1	-	+	+	+
3	d1	c2	ϵ	0	1	-	+	+	+
4	d1	c2	c4	0	2	+	+	+	+
5	d1	a2	a3	0	2	+	+	+	+
6	d1	d2	ϵ	0	1	-	+	-	-
7	d1	d3	ϵ	0	1	-	+	-	-
8	c1	c2	c4	0	2	-	-	+	-
9	a1	a2	a3	0	2	-	-	+	-

F=full, L=left-complete, R=right-complete, C=complete

Table 1: Path instances for `/A//B` in the tree from Fig. 2

$\{\epsilon\}$ which maps each step number i to p_i, and which satisfies the condition

$$\exists l, r : 0 \leq l \leq r \leq |\pi| \text{ such that}$$

$$\forall i : \text{all of} \left\{ \begin{array}{rcl} i < l & \Rightarrow & p_i = \epsilon \\ i = l & \Rightarrow & p_i \in C \cup B \\ l < i < r & \Rightarrow & p_i \in C \\ i = r & \Rightarrow & p_i \in C \cup B \\ i > r & \Rightarrow & p_i = \epsilon \end{array} \right\} \text{hold}$$

Informally, a partial path instance maps only a consecutive subsequence of location steps to document nodes, where the two ends of the subsequence may be incomplete navigations represented as border nodes.

p is said to be *left-incomplete* iff $p_l \in B$ and *left-complete* otherwise. *Right-complete* and *right-incomplete* are the equivalents for the right path end. A path instance that is left- and right-complete is called *complete*. Full path instances are a special case of partial path instances: A path instance p is *full* iff it is complete, and $l = 0 \wedge r = |\pi|$.

Example 4 Tab. 1 shows some path instances based on the sample tree (Fig. 2), and the location path `/A//B` with context node d1. Note that a path instance can be non-full, but complete.

The path instances represent knowledge about the document tree with respect to the query. This information is used in our operators. For example, path instance 3 can be interpreted as "If d1 is a context node, we can reach a2 after step 1". Path instance 7 says "if we have d1 as a context node, we can reach d3 while processing step 1". Path instance 9 implies "if we can reach a1 while processing step 1, we have a3 as a result node". □

4.2 Algebra

Our approach to evaluate XPath queries on fragmented documents is based on a translation of XPath location paths into algebraic expressions. These algebraic expressions represent the query execution plans that are executed by the query execution engine.

The operators in these expressions operate on and return sequences of partial path instances. Note that a result node set of a location path can be represented as a sequence of full path instances whose right ends correspond to the nodes that can be reached by the location path.

As an introduction, we briefly describe how to evaluate location paths with just one kind of operator, provided that there are no border nodes, i.e. the whole document is in a single fragment. This corresponds to a nested-loop evaluation strategy on unfragmented documents, as employed by many XPath processors. We call this method the *in-order* method because the fragments are accessed in the order they are needed for nested step evaluation.

We show how to extend this strategy to documents with multiple fragments. Only a few operators are sufficient to allow processing a fragmented document *out-of-order*, i.e. to access the fragments in any order that is beneficial from a performance point of view. In the original article on partial path evaluation [9], the order was determined by efficient physical access patterns.

4.2.1 In-Order Evaluation

The straightforward approach to evaluate location paths uses nested loops, one for each location step, navigating through the tree structure to enumerate the intermediate results for each step. In addition, final duplicate elimination and sorting may be required. This is easily expressed as a chain of UnnestMap [3] operators (one for each location step), where each operator reads a context node from its producer and then enumerates all result nodes for a step. The context nodes are fed into the chain by a producer expression at the bottom.

In our terminology, such an approach passes non-full, but complete path instances from operator to operator. The context nodes on which the location path is supposed to be evaluated are fed into the step chain by a producer expression. For each input context node x, a path instance p with $p_0 = x$ and $p_i = \epsilon$ (for $0 < i \leq |\pi|$) is delivered as input to the bottommost UnnestMap operator.

Each UnnestMap operator is then responsible for extending the path instance by one step. The final UnnestMap operator has full path instances as a result, whose right ends correspond to the result node set. Our focus is not on completeness, but on the template folding technique, so we do not specify a complete translation function and omit discussion of duplicate elimination and order issues, referring the reader to [3] for details.

Example 5 The location path `child::A/descendant::B` is translated into the algebraic expression

$$\text{UnnestMap}_{p_2:p_1/\texttt{descendant::B}}(\text{UnnestMap}_{p_1:p_0/\texttt{child::A}}(e))$$

where e is an algebraic expression returning the input context as explained above. The inner UnnestMap operator navigates through the document tree to enumerate all `A` descendants of every p_0 context node, producing a sequence of partial path instances, each containing one descendant node as p_1. The outer UnnestMap then produces the final result by similarly enumerating all `B` children of the p_1 nodes in p_2. □

4.2.2 Out-of-Order Evaluation

The in-order method requires that fragments can be visited in the order resulting from step traversal. To allow separate evaluation on template and variable fragments, we want to determine the order in which the fragments are accessed. Such an out-of-order query execution plan must produce a complete query result, and is not allowed to randomly access fragments. Hence, when visiting each fragment, it must extract all data that may be relevant for evaluation of the query. This includes partially evaluated paths that cannot be completely evaluated because a border node occurs on the path, and the fragment with the target border node is not available yet.

The partial path instances introduced in Sec. 4.1 provide a way to represent such partial results. We will now explain which additional operators are needed to process partial path instances in an out-of-order execution plan.

There are three operators out of which we build our plans. The first operator is a descendant of the UnnestMap operator, called XStep. It can interrupt navigation when it encounters a border node, hence representing incomplete computations. The XStep operators form a chain as in the simple in-order approach. At the bottom of each chain, we have the XScan operator, which produces the initial path instances on which to operate. In addition to the input context nodes, these initial instances also include left- and

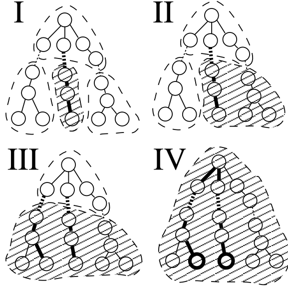

Figure 3: Out-of-Order Path processing

right-incomplete partial path instances for every border node at every step, to guarantee that all partial paths that may be relevant are generated by the XStep chain. The outermost operator for each location path execution plan is the XAssembly operator, which merges incomplete path instances, creating longer, and, eventually, full path instances. We now briefly sketch the execution of such an out-of order plan using an example, before going into more detail about the operators' semantics.

Example 6 The query `child::A/descendant::B` can be evaluated with the out-of-order plan represented by the expression
XAssembly(
$\text{XStep}_{p_2:p_1/\texttt{descendant::B}}(\text{XStep}_{p_1:p_0/\texttt{child::A}}(\text{XScan}(e)))$
)

We show how this plan is executed on the document from Fig. 2 with d1 as input context node. We can determine an arbitrary order in which XScan should access the fragments, and choose the fragment sequence `a, b, c, d`. Hence, the plan proceeds as shown in Fig. 3, with the shaded area representing those parts of the tree that have already been processed.

(I) First, fragment `a` is examined. a3 is identified as a candidate result node for the path, on condition that the parent fragment `d` contains a proper input context node. Hence, a left-incomplete path instance is created and stored (indicated by bold lines), which will be processed in step IV (see below). (II) Fragment `b` is visited during the scan. However, the XStep operators do not find any nodes in `b` matching the node tests of our path, so no action is taken. (III) Fragment `c` is visited, and a left-incomplete path instance is generated for node c4, as above. (IV) Finally, the scan arrives at fragment `d`. The XStep operators produce right-incomplete paths leading to the border nodes d2 and d3, pointing to `a` and `c`. XAssembly detects that these right-incomplete paths can be merged with the two left-incomplete paths created earlier, so the result nodes a3 and c4 can be returned. □

4.2.2.1 XStep.

The XStep operator evaluates single location steps, and is based on the UnnestMap operator. As the UnnestMap operator, it is parameterized with the axis and node test of the associated step, and with the step number, i.e. the position of the location step within the location path.

The difference between UnnestMap and XStep is that XStep can also operate on and return incomplete path instances, as follows. The XStep operator for a given step i checks for each input instance p whether $p_{i-1} \in C \wedge p_i = \epsilon$, i.e. whether the input path is right-complete and ends at step $i-1$. If not, the path is forwarded unchanged to the next XStep because it does not represent a proper context node for this step, but is some kind of intermediate information that is processed by another operator (see below).

If $p_{i-1} \in C$, XStep produces all core nodes from the current fragment reachable by the specified location step. In addition, it also returns all border nodes which are reached by the axis specification as they may lead to further result nodes.

Example 7 In Tab. 1, given path instance 1 as single input instance, the operator $\text{XStep}_{p_1:p_0/\texttt{child::A}}$ would return the right-incomplete path instances 6 and 7 as output. The child axis evaluated on d1 returns the nodes d2, d3 and d4. However, d4 fails the node test as it is not an `A` node. d2 and d3 are not immediate results, but may lead to further results in their target fragments, and hence are returned. □

4.2.2.2 XScan.

The objective of the XScan operator is to provide potential starting points for partial path instances. It is parameterized with the location path length $|\pi|$ and a collection of fragments to scan.

XStep returns a partial path instance for each location step and for every border node that occurs in the specified fragments. Hence, in generates all path instances of the form (x, i) with $x \in B, 0 \leq i \leq |\pi|$.

These initial path instances are extended by the XStep operators to generate all partial path instances that may be part of a query result. In addition, the XScan operator receives all input context nodes for the path evaluation as input and copies them unchanged into its output.

4.2.2.3 XAssembly.

The XAssembly operator merges partial path instances produced by the XStep operator chain and returns the final query result.

The algorithm implemented by XAssembly consumes input path instances and tries to merge them with previously seen instances to produce new, longer instances. Any full path instance produced by the algorithm is returned as a result.

The operator maintains two major data structures in its state, called S and R. R is the *reachable set* of pairs (x, i), which describe that node x can be reached after step i of the path. S is a set of yet-unprocessed left-incomplete partial path instances, and is maintained using an associative data structure with the left end of the path as key. Hence, it is efficient to find all partial path instances in S given a pair (x, i) that represents a border node x and a step i.

For an input instance p, XAssembly proceeds according to one of three cases:

1. If p is a full path instance, it is returned as part of the result.
2. If p is left-incomplete, we check whether its left end (x, i) has a companion border node in R. If no, we store p in S. If yes, we reprocess it as if it were a left-complete input instance.
3. If p is left-complete and the right end (x, i) of p is not already in R, (x, i) is added to R. In addition, all instances in S that have the matching border node for (x, i) as left end are processed as if they were input instances to XAssembly.

Example 8 Assume that XAssembly has already consumed the left-complete path instance 6 from Tab. 1, hence $(\text{d2}, 1) \in R$. Now, XAssembly consumes path instance 8, called p. p is left-incomplete, and its left end is $(\text{c1}, 0)$. The reachable right end $(\text{d2}, 1)$ is in R, which means that p can be treated as a left-complete instance. As a left-complete instance, p is equivalent to path instance 4, and node c4 can be returned as a result. □

5. TEMPLATE FOLDING

Our goal is to generate execution plans for queries that have to be evaluated against a large number of similar messages that are available in a fragmented representation created by a custom parser as explained in Sec. 3.

Partial path instances and the out-of-order method for path evaluation are ideally suited as tools to implement our template folding approach. In fact, the abstract components from our overview in Sec. 1.2 can be easily mapped to the components of the out-of-order approach explained in Sec. 4.2.2. The intuition behind our approach is simple: For a complete answer to the query, it must be evaluated on all fragments. The out-of-order approach allows us to evaluate the fragments in any order we choose. In particular, we can choose an order which first accesses all template fragments, and then the variable fragments. Further, we can "freeze" the execution state after the template fragments have been processed, and resume later when the variable fragments are known. To avoid reconstruction of the execution state for every new input document, we provide a way to fold the state reached after processing the templates into the execution plan.

To do this, we only need to extend the XAssembly operator by parameterizing it with initial values for the sets S and R. In the original XAssembly operator, S and R are empty when the execution starts. In our new operator called XAssemblyFold, we allow the operator to start with predefined values for S and R, which are specified in the execution plan.

To fold template fragments into a query execution plan, the query compiler generates an out-of-order plan QEP as shown in Sec. 4.2.2 for the specified path. Both parameters of XScan are left as free variable of the plan.

This tentative plan is then run using the query execution engine. For this run, the XScan parameters are a temporary representation of the template fragments as collection of fragments, and the empty set as the set of input context nodes. Once XAssembly has consumed all of its input path instances, we capture the current values of the sets S and R from the execution state of XAssembly. Then, a new plan QEP' is constructed, which is based on QEP, but with XAssembly substituted by XAssemblyFold, with the values of S and R obtained from the execution of QEP as parameters. This new plan QEP' is our desired plan with embedded preevaluated information about the template fragments. Given a fragmented input document, the final query result can be obtained by running QEP', using only the variable fragments and the input context nodes as parameters to XScan.

6. RELATED WORK

Schema-aware XQuery processing [2, 12] is an orthogonal technique to optimize queries in the presence of knowledge about document structure. In our Example 1, it would be possible, given some schema knowledge, to replace the descendant step `//` with a (cheaper) child step `/d:statusupdate`. However, this only rewrites the query, all steps of which still have to be evaluated. We avoid evaluation on previously known fragments.

Another related area is the *incremental recomputation* of query results [4]. Given a result of the same query on some original data, and a set of updates to the original data, incremental evaluation systems try to compute the query result on the updated data using only the original result and possibly some extra information generated during the original evaluation. We can express our problem as an instance of incremental recomputation: We consider the template fragments the original data and the variable fragments *updates* to the templates. The extra information required by the approach of [4] correspond to our S and R sets in the XAssembly operator. However, we are able to prune these auxiliary information by specifying a limited class of updates using our border nodes. Updates of the original tree may only happen at border nodes, and we can discard information that refers to updates at other places in the tree.

Creating custom parsers for particular XML schemas instead of using general-purpose XML parsers has been studied in [10], whose results let us expect the template-aware custom parsers from Sec. 3 to be faster than generic XML parsers.

7. CONCLUSION AND FUTURE WORK

For query processors that have to evaluate the same query over a large number of incoming XML messages, we have shown how to avoid repeated evaluation of intermediate results for recurring input substructures. We aim to increase query performance for XPath evaluation by moving repeated evaluation work into the query compilation phase, such that query execution only needs to evaluate the query on those parts of input messages that vary from message to message.

In the future, we plan to extend our technique to language constructs that go beyond simple location paths.

8. REFERENCES

[1] Serge Abiteboul, Ioana Manolescu, and Emanuel Taropa. A framework for distributed xml data management. In *EDBT*, pages 1049 – 1058, 2006.

[2] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *SIGMOD Conference*, 2001.

[3] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged algebraic XPath processing in Natix. In *ICDE*, pages 705–716, 2005.

[4] Guozhu Dong and Rodney Topor. Incremental evaluation of datalog queries. In *Materialized views: techniques, implementations, and applications*, pages 229–240. MIT Press, Cambridge, MA, USA, 1999.

[5] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: a platform for web services. In *CIDR*, 2003.

[6] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan, and Geetika Agrawal. The bea/xqrl streaming xquery processor. In *VLDB*, 2003.

[7] Jesse James Garrett. Ajax: A new approach to web applications. Technical report, Adaptive Path, February 2005. http://www.adaptivepath.com/publications /essays/archives/000385.php.

[8] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2. Technical report, The World Wide Web Consortium (W3C), 2003.

[9] Carl-Christian Kanne, Matthias Brantner, and Guido Moerkotte. Cost-sensitive reordering of navigational primitives. In *SIGMOD Conference*, pages 742–753, 2005.

[10] Welf M. Löwe, Markus L. Noga, and Thilo S. Gaul. Foundations of fast communication via XML. *Ann. Softw. Eng.*, 13(1-4):357–379, 2002.

[11] Mark Nottingham and Robert Sayre. The atom syndication format. Technical Report RFC4287, The Internet Engineering Task Force, December 2005.

[12] Chang-Won Park, Jun-Ki Min, and Chin-Wan Chung. Structural function inlining techniques for structurally recursive xml queries. In *VLDB Conference*, 2002.