On Extending the XML Engine with Query-Processing Capabilities

Klemens Böhm
Database Research Group
Institute of Information Systems
Swiss Federal Institute of Technology (ETH)
8092 Zürich, Switzerland

Abstract

We study how to efficiently evaluate queries over XML documents whose representation is according to the XML specification, i.e., XML files. The software architecture is as follows: the XML engine (i.e., XML parser) makes the structure of the documents explicit. The query processor operates directly on the output of the XML engine. We see two basic alternatives how such a query processor operates: event-based and tree-based. In the first case, the query processor immediately checks for each event, e.g., begin of an element, if it contributes to a query result or if it invalidates current partial results. In the second case, the query processor generates an explicit transient representation of the document structure and evaluates the query set-at-a-time. This work evaluates these approaches and some optimizations in quantitative terms. Our main results are as follows: the event-based evaluation scheme is approximately 10% *faster, even with all the optimizations from this article. The overhead of the query processors is small, compared to the running times of the XML engine. Finally, exploiting DTD information in this particular context does not lead to a better performance.*

1 Introduction

As information becomes digital and Digital Libraries proliferate, exchange formats for semistructured data are becoming more and more important. Semistructured data models cope with fundamental issues such as heterogeneity, irregularity, as well as incomplete information. The *Extensible Markup Language (XML)* is the most prominent representative of this class of models [25]. In short time, an impressive number of W3C standards that build on XML as well as a huge number of XML document types have become available. Clearly, declarative access to such data is important as well. Researchers have proposed a number of respective query languages [18, 21, 17, 8, 11, 2].

Abiteboul et al. have pointed out that such queries can be evaluated directly on the XML file [1]. I.e., a parser makes the structure explicit, and the query processor operates directly on the output of the parser. This *file-based approach* is useful in various situations. In particular, this is the case when maintaining materialized views/index structures over the document collection is too expensive. This current work addresses the question how to implement file-based query processing efficiently. So far, literature has not identified alternative approaches to file-based query processing and compared them systematically. Another aspect of the problem is that file-based query processing should conform to the XML specification and related standards.

The XML specification [25] not only describes the structure of XML documents, it also specifies an architecture to process such data. At its heart, there is a so-called *XML-engine*. It essentially is a parser that recognizes the inner structure of the documents, the *logical structure*, and makes it available to *applications*. According to the XML specification, an application is a system that interacts with the XML engine. Using this terminology, the objective of our work is to investigate design alternatives for a query processor that is an application. As a first contribution, we introduce two main alternatives that have not been described as such and contrasted with each other, namely *tree-based* and *event-based*: based on an algebra representation of the query, the tree-based query processor generates an explicit transient representation of the document structure and evaluates each query operator set-at-a-time. Regarding the event-based approach, if a certain event occurs, e.g., the XML engine encounters an element of type E, it invokes a callback function. Its implementation is part of the query processor: the query processor then checks if this event contributes to a query result or invalidates existing partial results.

Example 1.1 Consider the simple query "Select all documents with a caption containing the string 'millennium'.". The automaton from Figure 1 illustrates how event-based

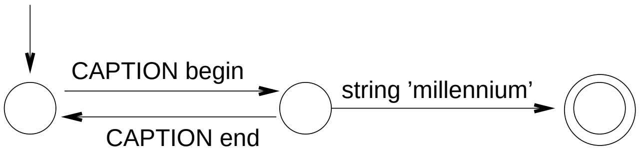

Figure 1. Automaton illustrating event-based query processing

evaluation of the query works. A tree-based query engine would generate an explicit representation of the document structure first. It might then search for all elements of type CAPTION and then check if they contain string 'millennium'.

At this level of analysis, it is not obvious how event-based query processing works for more complex queries. Furthermore, one does not see at this stage which alternative is superior. In particular, this is the case when queries are more complex than in the example, and when taking optimizations of the tree-based approach into account as well. As another contribution of this article, we develop such optimizations of the tree-based approach: the first optimization is to generate not the entire document tree, but only subtrees necessary to evaluate the current query. The body of the paper describes several variants of this optimization and proves their correctness. Some of them are generalizations of previous proposals [1]. Some of the variants require information from the *document-type definition (DTD)*. - The second optimization is to avoid the computation of joins. Instead, we use operations that traverse the tree whenever feasible.

Our main results are as follows: event-based is faster than tree-based. Typical numbers are 135 seconds (event-based) vs. 158 seconds (tree/based without any optimizations). Running the XML engine without any application over the same document collection lasts about 132 seconds, i.e., the overhead of event-based query processing is negligible. - Let us now look at the optimizations for the tree-based approach: the optimization to replace a join with traversal yields hardly any improvement. The other optimization that generates smaller trees improves performance by 10% as a rule of thumb. I.e., a typical number with that collection would be 144 seconds. This means that event-based is still a bit faster than tree-based, even with all optimizations. On the other hand, let us consider another evaluation criterion, namely ease of implementation: our experience is that implementing the tree-based query processor has required less effort. It is also easier to extend and administer, as we will explain.

In contrast to interfaces to the XML-engine, a commonly accepted standard for XML query languages has not yet emerged. The query language used in our evaluation is the PAT algebra [21]. It is roughly comparable to SPJ-queries (Select-Project-Join queries) in the relational case. The focus of this article are queries that refer to individual documents. Efficient evaluation techniques for queries that span several documents, i.e., refer to hyperlinks between documents, are orthogonal to this current work.

The remainder of this article has the following structure: first we discuss related work. Section 3 briefly reviews the XML data model and the query language used, as well as two standards that define interfaces to the XML engine (DOM and SAX). Section 4 describes the basic query processing alternatives (tree-based vs. event-based) and optimizations for the tree-based one. Section 5 discusses our experiments. Section 6 concludes. For lack of space, we have omitted the proofs of the lemmata; they are contained in the extended version of the article [4].

2 Related Work

Considering the wealth of recent work on semistructured data and query processing, [1] seems to be the closest piece of work. They use lex/yacc-like technology to process queries on a file of arbitrary structure. This corresponds to our tree-based evaluation scheme. From one perspective, their work is more general, as a Yacc grammar describes the file structure. On the other hand, [1] considers the relational algebra, which does not contain primitives for semistructured data. Some optimizations of our tree-based evaluation scheme are not applicable in the context of the relational algebra. But our BOTTOM optimization is a generalization of their approach to avoid generation of unnecessary subtrees.

Schema-based query optimization for semistructured data is the topic of [12]. The authors use knowledge from the schema to replace one operator from the PAT-algebra with another, cheaper one whenever possible. This replacement reduces query-execution costs under a specific cost model. [6] also uses schema knowledge and information on existing index structures for query rewriting. Assumptions regarding the cost model are much weaker.

Furthermore, much work has been done on query languages for semistructured data, and this overview is not complete. [18] presents a query language for hierarchical data that incorporates notions such as position, order, and aggregates. [21] introduces the PAT algebra, [17] introduces tree-inclusion queries, both of which we will describe subsequently. [8] observes that paths in structured documents may be subject to queries and introduces a respective query language. [11] develops a query language that allows to query both the logical and the layout structure of the document. With LORE [2], queries have an SQL structure, and path expressions may occur in any part of the query. - Fi-

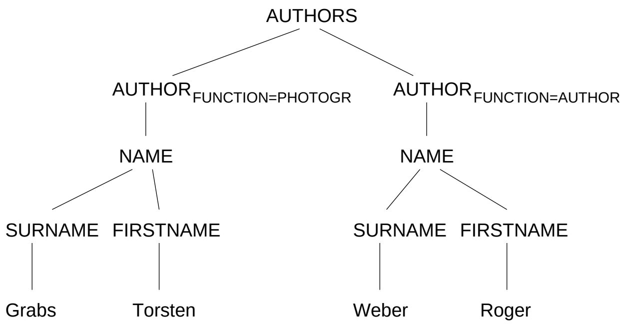

Figure 2. Logical Structure of Document Fragment from Figure 3

nally, there is a large variety of work on index structures and database representations of semistructured data, e.g., [15, 16, 19, 5, 23]. However, such work is orthogonal to this current work, and we do not dwell on it any further.

3 Preliminaries

Subsection 3.1 reviews XML and introduces our running example. Subsection 3.2 describes the query language used. Subsection 3.3 reviews two specifications of interfaces to the XML engine, namely *DOM* and *SAX*. The different interaction models introduced by these standards correspond to our two basic query evaluation alternatives.

3.1 Extensible Markup Language (XML)

XML Documents. The logical document components, the so-called *elements* form a hierarchical structure (cf. Figure 2). In our terminology, the root of the hierarchy has depth 1. Going down the hierarchy increases the depth. The content of the leaves is so-called *parsed character data*. A particular instance of a string is typically contained in more than one element; e.g., the string 'Grabs' is contained in a `SURNAME`-element as well as in a `NAME`-element. As usual, a *path in the document* is a sequence of elements $e_1.e_2.\dots.e_n$ such that the i-th element directly contains the $i+1$-th element.

XML Document-Type Definitions (XML DTDs). An XML DTD specifies the types of the elements that may occur in documents of a particular type, and it restricts the possible order of elements. Figure 4 contains an XML-DTD, the MMF-DTD from [24]. It is the running example of this article. The document from Figure 3 conforms to this DTD. - A line starting with '`<!ELEMENT`' is an *element-type definition*. The expression behind the element-type name is the so-called *content model*: it specifies the order of elements directly contained in an element of the current type. While an XML document does not have to have a DTD, the concept is important: XML-DTDs have been proposed for virtually any application scenario, and state-of-the-art mediation technology is now DTD-based [20].

Definition 3.1 Element type A contains element type B according to DTD D *if there is a document that conforms to* D *and that contains an element of type* A *which contains an element of type* B.

Even though the XML specification allows it, we assume for the sake of presentation that an element type does not contain itself.

[6] says how to compute the containment-relation on element types for a given DTD. Subsequently, $above(ET)$ denotes the set of element types containing ET according to a given DTD, and $below(ET)$ denotes the ones contained in ET. Further, $above(ET_1, \dots, ET_n)$ is short for $\bigcup_{i=1,\dots,n} above(ET_i)$.

Observation 3.1 *Let element type* A *contain element type* B. *Then the following holds:* $above(A) \subseteq above(B); below(A) \supseteq below(B)$.

3.2 Querying XML Documents

In this article, we will use the PAT algebra [21].

Definition 3.2 *The following grammar, where* T, T_1, *and* T_2 *denote a set of elements, generates PAT algebra terms:*

$T \rightarrow$ <element-type name> | CONTENT_SELECT (T, r) | ATTR_SELECT (T, A, r) | T_1 INCLUDS T_2 | T_1 INCL-IN T_2 | (T_1)

The term <element-type name> *stands for the set of all elements of the respective type.* CONTENT_SELECT *takes a set of elements and returns those whose textual content satisfies regular expression* r. ATTR_SELECT *takes a set of elements and returns those where attribute* A *satisfies regular expression* r. *The remaining operators take two sets of elements* T_1 *and* T_2 *and return the set of elements*

T_1 INCL-IN $T_2 = \{e_1 \in T_1 \mid \exists e_2 \in T_2 \textit{ s.t. } e_1 \textit{ is contained in } e_2\}$

T_2 INCLUDS $T_1 = \{e_2 \in T_2 \mid \exists e_1 \in T_1 \textit{ s.t. } e_2 \textit{ contains } e_1\}$

Example 3.1 The query SURNAME retrieves all `SURNAME`-elements from a set of documents. It does not return a set of identifiers, it rather returns the elements themselves, e.g., `<SURNAME>Boehm</SURNAME>`, ..., `<SURNAME>Grabs</SURNAME>`. The query

```
<AUTHORS>        -- '<AUTHORS>' identifies begin of AUTHORS-element --
     <AUTHOR FUNCTION=PHOTOGR><NAME><SURNAME>Grabs</SURNAME>
             -- SURNAME-element contained in NAME-element --
     <FIRSTNAME>Torsten</FIRSTNAME></NAME></AUTHOR>
     <AUTHOR FUNCTION=AUTHOR><NAME><SURNAME>Weber</SURNAME>
     <FIRSTNAME>Roger</FIRSTNAME></NAME></AUTHOR>
</AUTHORS>        -- '</AUTHORS>' identifies end of AUTHORS-element --
```

Figure 3. Fragment of sample XML document (Comment enclosed in '--')

```
<!ELEMENT MMFDOC   (LOGBOOK, MONOMED, ...)>
      -- MMFDOC-elem. contains LOGBOOK-element, followed by MONOMED-element --
<!ELEMENT LOGBOOK  (AUTHORS, EDITORS, ...)>
      -- '(AUTHORS, EDITORS, ...)'  is the content model --
<!ELEMENT MONOMED  (PARAGRAPH|REFERENCE)*>
      -- MONOMED-elem. contains arbitrary number of PARAGRAPH-elements and
<!ELEMENT AUTHORS  (AUTHOR*)>
<!ELEMENT AUTHOR   (NAME)> -- 'NAME' is the content model --
<!ATTLIST AUTHOR   FUNCTION    (AUTHOR | CAMERA | COMPOSER | EDITOR| PHOTOGR | PRODUCER |
                               REGISS| OTHERS )  "AUTHOR">  -- AUTHOR is the default value --
<!ELEMENT NAME     (SURNAME, CHRNAME?, ADDITION?)>
      -- NAME-elem. contains SURNAME-element, followed by optional CHRNAME-element,
         followed by optional ADDITION-element --
<!ELEMENT (CHRNAME | SURNAME | ADDITION|PARAGRAPH|REFERENCE)   (♯PCDATA)>
      -- CHRNAME, SURNAME and ADDITION have the same content model, namely PCDATA --
```

Figure 4. Fragment of sample DTD

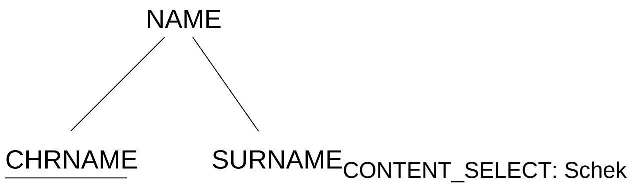

Figure 5. Sample Query Tree

(CHRNAME INCL_IN (NAME INCLUDS CONTENT_SELECT(SURNAME, 'Schek'))) selects all CHRNAME-elements, i.e., all first names, that are contained in the same NAME-element as the SURNAME-element with value 'Schek'. ∎

It is fairly straightforward to extend the algebra to take the secondary structure of documents into account, i.e., ID/IDREF-relationships or hyperlinks between the documents. We omit this for lack of space. Furthermore, there is a graph representation of a query, which we will use in the following: the nodes are the element types from the query, and the edges correspond to INCLUDS/INCL-IN relationships. Figure 5 is the graph of the query from Example 3.1. The underline identifies the query result, called *result node* in the following, and select conditions are labels of the respective graph node. - In what follows, we will only consider queries whose graphs are trees. We will refer to the graph representation of a query Q as *the tree of* Q or simply *query tree*, and we will speak of *the root of query* Q.

3.3 Interfaces to the XML Engine

DOM vs. SAX. The XML specification does not only specify the format of XML documents. It also introduces the notions of an *XML engine* and an *application* and specifies the tasks of an XML engine. The XML engine recognizes the structure of an XML document and makes it available to the application. The XML specification describes the interaction of the XML engine with applications in abstract terms, i.e., it does not define an interface. But there are two specifications that fill this gap, the *Document Object Model (DOM)* [14] and the *SAX callback interface* [22].

DOM. The Document Object Model essentially is a set of interfaces whose implementation is part of the XML engine. It provides an object-oriented view of the XML document. Every construct appearing in the XML document is an object, not only elements as in Figure 2. For instance, there is a DOM object corresponding to each attribute. An application invokes methods that are part of these interfaces. The

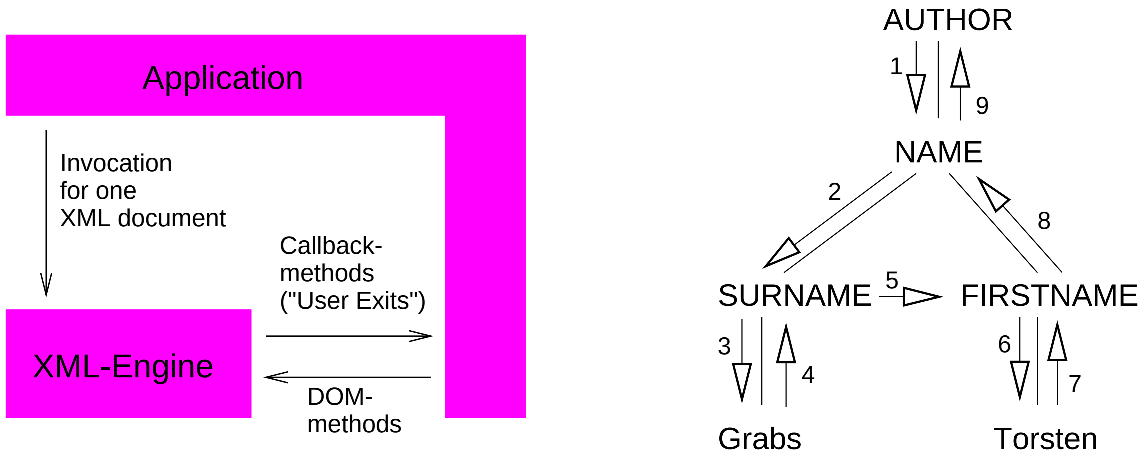

Figure 6. Left Half: Interaction between XML Engine and Application, Right Half: ESIS order in which XML engine goes through the document

left half of Figure 6 serves as an illustration.

SAX Callback Interface. With SAX, the application implements a set of *callback functions*. Their interface is standardized. The XML engine invokes these methods when a corresponding event occurs while processing the document. In other words, the XML engine reports parsing events. An assumption behind SAX is that the XML engine steps through a document in a well-defined order, namely depth-first. It returns to an element a second time after it has seen the subdocument rooted at this element. Figure 6 depicts this order. - The following callback functions are important in this context:

- startElement(String element-type-name, AttributeList attrs) - When the XML engine sees an element for the first time, it invokes this function.
- endElement(String element-type-name) - dto. second time.
- pcdata(String pcdata) - The engine invokes this function when it encounters parsed character data.

4 File-Based XML Query Processing: Alternatives and Implications

4.1 Tree-Based Query Processing

4.1.1 General Approach

Tree-based query processing requires a tree representation of the document and an operator-tree representation of the query (cf. Definition 3.2). As usual, each node of the query tree implements a function evaluate that returns the result of its subquery. The implementation of evaluate of an internal node invokes evaluate of its child node. evaluate of a leaf node collects the elements of the respective type. The only marginal difference to set-at-a-time query evaluation in the relational setting is that evaluate now returns a set of pointers to tree objects, rather than a set of elements. We dereference only the pointers that are in the result set of evaluate of the root node.

4.1.2 Query Evaluation over Subtrees of the Document Tree

The following examples illustrate the different variants of the optimization idea to leave aside irrelevant subtrees of the document tree.

Example 4.1 Reconsider the query from Example 3.1. Instead of generating the entire document tree, it is sufficient to look at subtrees whose roots are elements of type `NAME`. ∎

Example 4.2 Suppose that we are interested in the authors of all documents from the MMF collection having to do with person ’Böhm’. This corresponds to the following query: “AUTHOR INCL_IN (MMFDOC INCLUDS CONTENT_SELECT(PERSONS, ’Böhm’))”. Large parts of an MMF document are typically contained in an element of type `MONOMED`. But according to the MMF-DTD, an element of type `MONOMED` neither contains an element of type `AUTHOR` nor an element of type `PERSONS`. Hence, to evaluate the query, we do not need subtrees whose root is of type `MONOMED`. ∎

Example 4.3 Consider again the query from Example 4.2. To evaluate it, we only need to consider elements that contain string 'Böhm', or that contain an element of type `DOCID`, or that are contained in an element of type `DOCID`. (The last case is necessary because the query results are of type `DOCID`. If an element qualifies, the query processor must assemble its content and output it.) ∎

Note that these examples are different: in the fist one, we have identified subtrees that are sufficient for query evaluation. In the second and the third case, we have ruled out subtrees that are not needed for query evaluation. We will refer to the different optimizations as *TOP optimization*, *OUT optimization* and *BOTTOM optimization*, respectively. Figure 7 depicts the situation: the grey part corresponds to the subtree actually generated. Figure 7 (a) illustrates the TOP optimization, Figure 7 (b) the OUT optimization, and Figure 7 (c) the BOTTOM optimization.

TOP-Optimization. The following paragraphs formalize the notions illustrated in Example 4.1. Recall that we have a graph representation of each query Q that is a tree.

Definition 4.1 *The* top element type $TOP(Q)$ of a query Q *is the element type of the root of* Q.

Example 4.4 In Example 4.1, `NAME` is the top element type of the query. ∎

Lemma 1 *Let a query Q be given. Evaluating Q only on the subtrees of the document tree whose root element type is in $TOP(Q)$ yields all query results.*

The corresponding optimization, subsequently referred to as *TOP optimization*, only generates the relevant subtrees of the document tree.

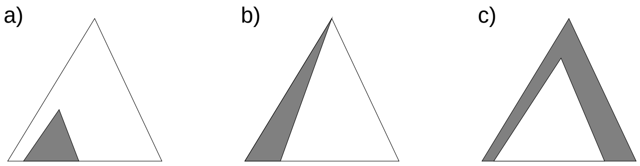

Figure 7. TOP Optimization, OUT Optimization, and BOTTOM Optimization

OUT Optimization. In the following, we are interested in all types of elements that are not needed to evaluate the query, such as element type `MONOMED` from Example 4.2. We call these types *out element types*. However, we cannot merely prune subtrees whose root element type does not contain an element type from the query. Instead, we must consider that the query engine might need subtrees (1) to evaluate string predicates or (2) to build the output elements, after having identified them in the document tree.

Example 4.5 Consider again the query "AUTHOR INCL_IN (MMFDOC INCLUDS CONTENT_SELECT(PERSONS, 'Böhm'))". The query engine needs the subtrees rooted at the `PERSONS`-elements to evaluate the 'Böhm'-predicate. It also needs the subtrees rooted at the `AUTHOR`-elements to assemble the query results. Consider now the query "NAME INCL_IN (AUTHOR INCL_IN (MMFDOC INCLUDS CONTENT_SELECT(PERSONS, 'Böhm')))". We now need the subtrees rooted at the `PERSONS`-elements and the ones rooted at the `PERSONS`-elements, but not the complete subtrees rooted at `AUTHOR`-elements. ∎

Let $ETYPES(D)$ denote the set of element types defined in DTD D. Given this, we define:

Definition 4.2 *The* set of out element types $OET(n)$ of node n of a query tree, given a DTD D, *is defined by recursion over the structure of the query tree.*

- *Node n of the query tree is a leaf node.*

$$OET(n) = \begin{cases} \textit{ETYPES(D)-above(ET(n))-}\{\textit{ET(n)}\} & (*) \\ \textit{ETYPES(D)-above(ET(n))-} & \\ \textit{below(ET(n))-}\{\textit{ET(n)}\} & (**) \end{cases}$$

  $(*)$ *if n is not the result node, and there is no string condition associated with n,*

  $(**)$ *otherwise.*

- *Node n is inner node of the query tree. - Let n_1, ..., n_m denote the children of n.*

$$OET(n) = \begin{cases} \bigcap_{i=1,\ldots,m} OET(n_i) & (\star) \\ \textit{ETYPES(D)-above(ET(n))-} & \\ \textit{below(ET(n))-}\{\textit{ET(n)}\} & (\star\star) \end{cases}$$

  $(\star)$ *if n is not the result node, and there is no string condition associated with n,*

  $(\star\star)$ *otherwise.*

We define the set of out element types $OET(Q)$ of query Q, given a DTD D, *as the set of out element types of the root node of the tree.*

Lemma 2 *Let a query Q and the document tree be given, without the subtrees whose root element type is in $OET(Q)$. Evaluating Q on this tree yields all query results.*

A difference between the TOP optimization and the OUT optimization is that the OUT optimization requires DTD knowledge. It is necessary to guarantee that a subtree does not contribute to a query result. The TOP optimization in turn only requires information from the query.

BOTTOM optimization. Example 4.3 illustrates another way to prune the document tree. Furthermore, the stumbling block is building the query result, as the example shows: we must know if the current element is within a potential result element when building the document tree.

With the BOTTOM optimization, the leaves of the tree generated are the elements that may be part of a query result, that are needed to check dependency constraints, or that contain textual content satisfying a regular expression from the query. The following definitions reflect this.

Definition 4.3 *The element type of a leaf of the query tree is a* BOTTOM element type *if the leaf is not the result node of the query, and if there is no string condition corresponding to the leaf.*

Definition 4.4 *The so-called* BOTTOM optimization *works as follows:*

- *There is a tree object representing a leaf element if the leaf meets a string condition from the query, or if it is contained in an element of the result type;*
- *there is a tree object representing a non-leaf element if the element is an instance of a BOTTOM element type, or there is a tree object representing a child of the element.*

Lemma 3 *Let a query Q be given. Evaluating Q only on the subtrees of the document tree specified in Definition 4.4 yields all query results.*

Implementation. We now say how we have implemented the optimizations presented before. Recall that the XML engine visits each element twice, when it sees the start tag, and when it sees the end tag (cf. Figure 6). At both occasions, we could generate the objects being part of the tree representation of the documents. The first alternative corresponds to top-down, the second one to bottom-up. Top-down allows to implement the TOP optimization and the OUT optimization, but not the BOTTOM optimization. This is because the decision whether an object is generated depends on the child elements. But bottom-up in isolation does not allow to implement the BOTTOM optimization as well, as that decision also depends on the parent elements. On the other hand, the XML engine sees both the parent elements and the child elements before visiting an element the second time. Hence, while 'climbing downwards', the query processor checks if the current element is in a relevant subtree according to the TOP/OUT/BOTTOM optimizations. This is only a necessary condition with the BOTTOM optimization. Generating objects only occurs while 'climbing upwards', after having checked all conditions of the BOTTOM optimization.

4.1.3 Avoiding Join Computations

So far, the optimizations have addressed generation of the document tree. We now turn to the query evaluation process, after the document tree has been generated.

Operators INCLUDS and INCL_IN of the PAT-algebra are join operators. In a set-at-a-time implementation, they inspect all pairs of elements where the first one is a result of the first subquery, and the second one is a result of the second subquery. We now describe another evaluation scheme whose complexity is less than $O(n \cdot m)$, with n being the size of the left input and m being the size of the right input. The idea is as follows: in order to find pairs of elements where the first one contains the second one (or vice versa), we can also start at one element, traverse the document tree downwards or upwards, and inspect each element found.

Example 4.6 We can evaluate the query from Example 3.1 by evaluating CONTENT_SELECT(SURNAME, 'Schek'); from each `SURNAME`-element, we go up in the document tree to find an element of type `NAME`. We then search all subtrees rooted at such elements for elements of type `CHRNAME`. ∎

However, it is not straightforward to replace a join operator with a navigation operation, as a join operator may combine complex subqueries. Furthermore, if both subqueries contain selective predicates, the intermediate results might already be fairly small, and that replacement might not pay off. Consequently, the optimization investigated here is as follows: we only evaluate subqueries by traversal that do not contain predicates. The full version of the article describes the respective query evaluation techniques in more formal terms.

4.2 Event-based Query Evaluation

The previous subsection has proposed query evaluation techniques that are based on the existence of a document tree. This subsection describes an alternative that directly operates on the events. The idea is to check for each event if it contributes to the query result, or if it rules out partial results that have been constructed so far. The query processor does not have to construct the document tree any more, but it now has to deal with more auxiliary structures. The question addressed in Section 5 is how this affects the performance of the query engine.

Describing our event-based query engine in full would be rather technical. For the sake of presentation, we limit the following description to a restricted case:

- Queries do not contain predicates on attributes or on the textual content of elements.
- The query engine only tells us if a document contains a result, but does not compute it.

Our evaluation will be based on the full query engine.

The query engine directly operates on the query tree. Figure 8 contains it in pseudo-code. The main data structures are the set of query nodes that currently match an element (matchedNodes in the following) and the set of query tree nodes which the query engine currently tries to match to elements (nodesToBeMatched). Initially, matchedNodes is empty, and nodesToBeMatched contains the root node (Line (2)). If the XML engine reports that it has encountered an element of a type that is to be matched (Line (6)), it moves the respective query node from nodesToBeMatched to matchedNodes and now tries to match its children (Lines (7)-(10)). If the XML engine reports the end of an element that matches a tree node, we must remove the corresponding tree node n and its direct and indirect children from matchedNodes (Lines (19), (20)) and must try again to match n (Line (21)).

4.3 Implementation Aspects of the Query Evaluation Alternatives

Since ease of implementation and manageability of the system are important evaluation criteria next to performance, we now report on our implementation experiences. The main point is that we found the tree-based variant much easier to implement. The problem with event-based is that predicates such as string patterns cannot be checked immediately when they arise. Instead, we must keep track of all the current predicates, which essentially may change with any event. We also found the tree-based version to be easier to extend. E.g., introducing a new operator requires

- extending the component that builds the query tree (lex/yacc-based),
- providing an implementation of evaluate for the new operator,
- checking how the new operator affects the optimizations TOP/OUT/BOTTOM; possible alterations of the routines that generate the document tree are limited to well-defined portions of the code.

For event-based, such a list is not feasible, and each operator typically must be treated differently.

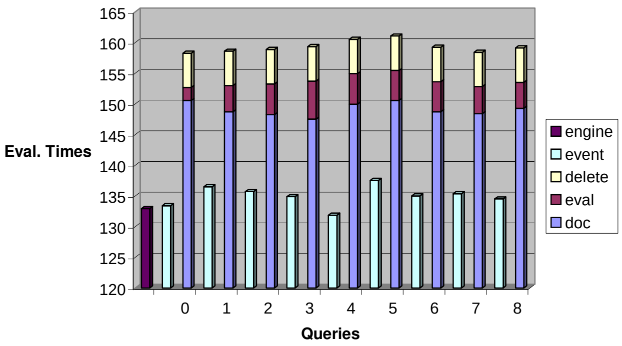

Figure 9. Event-Based vs. Tree-Based Evaluation Times

5 Experiments

This section evaluates the query evaluation alternatives proposed in this article. We have conducted a number of experiments with different document collections, i.e., documents of different size and structure, different platforms, and different parsers, namely SP [10] and expat [9]. We limit the discussion to experiments from one particular scenario, as the other experiments did not provide additional insight.

Experimental Setup. We have used an IBM ThinkPad with Windows NT, 363 MHz, 130 MB. The document explicitly considered in the following is the MMF document collection [24]. The MMF document collection is a set of SGML documents created at GMD-IPSI. We have converted them to XML using a feature of the SP. The collection used for the experiments contained 11,763 documents (with documents replicated several times). This number is sufficiently large to avoid buffering effects, i.e., a document is not in memory any more with a new run of the query engine over the document collection. For the experiments described here, we have used a set of nine queries that are contained in Appendix A. While we have formulated some of the queries to evaluate specific characteristics of the query engines, e.g., effect of query result size, other queries model typical information needs over the MMF document collection (cf. Appendix A).

5.1 Tree-Based vs. Event-Based

The objective of the first series of experiments is to compare the two basic alternatives, i.e., tree-based vs. event-based, and to determine the overhead of our query processors. We have first tried to identify the overhead by mea-

```
init()
    {nodesToBeMatched := {root of query tree}; matchedNodes := {};}

startElement() {
    forall (i ∈ nodesToBeMatched)
        if (i has the same type as the current element) {
            insert i into matchedNodes, together with the depth of the current element;
            delete i from nodesToBeMatched;
            if (i has children)
                insert the children of i into nodesToBeMatched;
        };
    if (nodesToBeMatched is empty) { print "Solution found."; exit; }
};

endElement() {
    forall (i ∈ nodesMatched)
        if (i -> depth == currentDepth)
            if ((!(i -> subtreeCurrentlyMatched)) || (i -> containsResultNode)) {
                remove i and all successors of i from nodesMatched;
                remove all successors of i from currentNodes;
                insert i into currentNodes;
            };
};

close() { print "Solution not found."; exit; };
```

Figure 8. Event-based query engine in pseudo-code

suring the durations of our callback routines, but they are typically too short to allow for exact measurements. Hence, the yardstick is to run the XML engine over the documents without any application. Figure 9 contains the respective numbers. The leftmost column denotes the time for parsing the documents, without query processing. For the tree-based query-evaluation scheme, we have split the execution times as follows: (1) XML engine steps through the document and generates the tree representation, abbreviated as doc in Figure 9 and in the following, (2) evaluation of the query (eval) and (3) deletion of the tree representation (delete). For the event-based scheme, there is no comparable way to split execution times, and we only give the overall duration. event is the respective abbreviation. Note that the minimum value on the y-axis is larger than zero.

First, we observe that event-based wins over tree-based without optimizations. Comparing overall execution times, event-based query evaluation is by $15-20\%$ faster than the tree-based alternative. Comparing pure query-evaluation times, i.e., the differences between the overall query execution times and the run time of the XML engine without query processing (the leftmost column in Figure 9), there is a factor of 9. I.e., $\frac{T(\text{tree})-T(\text{XML doc})}{T(\text{event})-T(\text{XML doc})} = \frac{158-132}{135-132} \approx 9$, where $T(\text{tree})$ is the time for tree-based query processing, $T(\text{event})$ is the time for event-based query processing and $T(\text{XML doc})$ is the time for the XML engine going through the document collection.

Another result is that the query-evaluation effort is moderate in relative terms in both cases, compared to parsing the documents. Finally, the most time-consuming aspect of tree-based query evaluation is the creation and deletion of the tree representation of the documents. This holds true even though we have implemented a memory management of our own that allocates space for a number of objects at a time. The actual query-evaluation times (eval) are relatively small.

The experiments also tell us that the effect of the query size and the one of result size are negligible with both evaluation schemes. The result of Query 4 contains hundreds of items, as opposed to Query 5 with only four result items, but the response times are not different. Query 3 contains four joins, but its response time is not any bigger than of the other queries.

Finally, Query 6 has been designed to investigate the effect of results with a complex structure: its results are elements of type `TEXT`. In the MMF collection, these elements are relatively large and have such a complex inner structure. The event-based query engine assembles all `TEXT`-elements, using the parameter values that are passed on via handler invocations. This is necessary since we do not know at the beginning of a `TEXT`-element if the string pattern 'DOCKING' occurs later in the element. However, the experiments show that this additional effort does not lead to longer query evaluation times.

5.2 Optimizations for the tree-based evaluation scheme

The first series of experiments has told us that the event-based evaluation scheme outperforms the basic tree-based one. However, the tree-based scheme is more flexible: adding a new query operator to the algebra, e.g., 'directly-included-in' or 'directly-includes' [12] or aggregation operators [18], requires only some well-defined extensions of the query engine. On the other hand, our experience from implementing the event-based query engine tells us that this does not exactly hold for this case. Hence, it is worthwhile to investigate the possible optimizations for the tree-based evaluation scheme.

As a first step, we have compared the tree-based 'set-at-a-time' evaluation scheme with the variant that does navigation instead. It turns out that this optimization has an effect with a relatively small number of queries only, and even then it is not substantial (numbers omitted from this short version for lack of space). This is not exactly surprising after having learned that the most time-consuming part of tree-based query evaluation is generation and deletion of the document tree. Our conclusion is that this optimization does not deserve further investigation, and the experiments to follow will always use the 'set-at-a-time' variant of the tree-based evaluation scheme.

Figure 10 compares the execution times of queries with and without the various optimizations, i.e., TOP, OUT, and BOTTOM. Some of the queries have been formulated to evaluate exactly these optimizations, i.e., the rationale is to identify the maximal benefit to be obtained with these optimizations. E.g., the top element type of Query 1 is `NAME`, and elements of this type typically contain two to three elements only. Hence, if the TOP optimization does not yield a significant improvement with this query, it is unlikely to be useful for any query.

The results are as follows: TOP and OUT give a small improvement. BOTTOM leads to response times that are slightly below these times for TOP and OUT. We could not observe a difference between queries whose string patterns have different selectivities. E.g., the string pattern from Query 5 ("Elisabethenstift") is much less frequent than the one from Query 4 ("Frankfurt"), otherwise the queries are identical.

Finally, we can combine the optimizations. Another series of experiments evaluates such combinations, and Figure 10 graphs respective results, labeled with 'TOP & BOT.' and 'OUT & BOT.', respectively. It turns out that the combination of BOTTOM and OUT does not only not pay off, it is also detrimental. The combination of TOP and BOTTOM in turn does lead to an improvement. This is actually a positive result, since TOP and BOTTOM are the optimizations that do not require information from the DTD. However, tree-based does not fully level up to event-based even with this performance gain.

6 Conclusions

With the increasing amount of semistructured data available in Digital Libraries, notably XML data, declarative access to such data becomes more important. In many situations, generating a redundant representation of such data for query processing is not feasible or is too expensive. Hence, a query engine must operate on the original XML representation. This motivates the investigation and quantitative assessment of file-based XML query processing.

We have identified two basic alternatives: the first one generates the parse tree for each document and evaluates the query 'set-at-a-time'. The second alternative operates directly on the event stream; each event may trigger a transition of the query engine to another state. From an implementation perspective, the tree-based approach is preferred because of higher flexibility. The tree-based scheme lends itself to various optimizations. We have conducted an extensive experimental evaluation of the alternatives and the optimizations. The main results are as follows: tree-based is slower than event-based by approximately 10%. The overhead imposed by the query engine, as compared to a run of the XML engine without any application, is relatively small ($2-3\%$ for event-based and around 10% for tree-based).

Some optimizations (OUT) require knowledge from the document-type definition, others do not (TOP, BOTTOM). The objective of using DTD information is to say when elements of a certain type cannot occur. Our analysis of the DTD to this purpose is rather intricate. Furthermore, a DTD is not always available. However, the optimization schemes that do not deploy information from the DTD yield the same performance as the one that does.

This work is part of a larger effort to facilitate efficient declarative access to XML documents that may be subject to modifications. A future issue is to extend the file-based query engine to efficiently compute large numbers of views over XML document collections.

A The List of Sample Queries Used for the Experiments

We have used the following sample queries for the experiments.

0. "Select all elements of type `PIPAPO`." Returns the empty set.

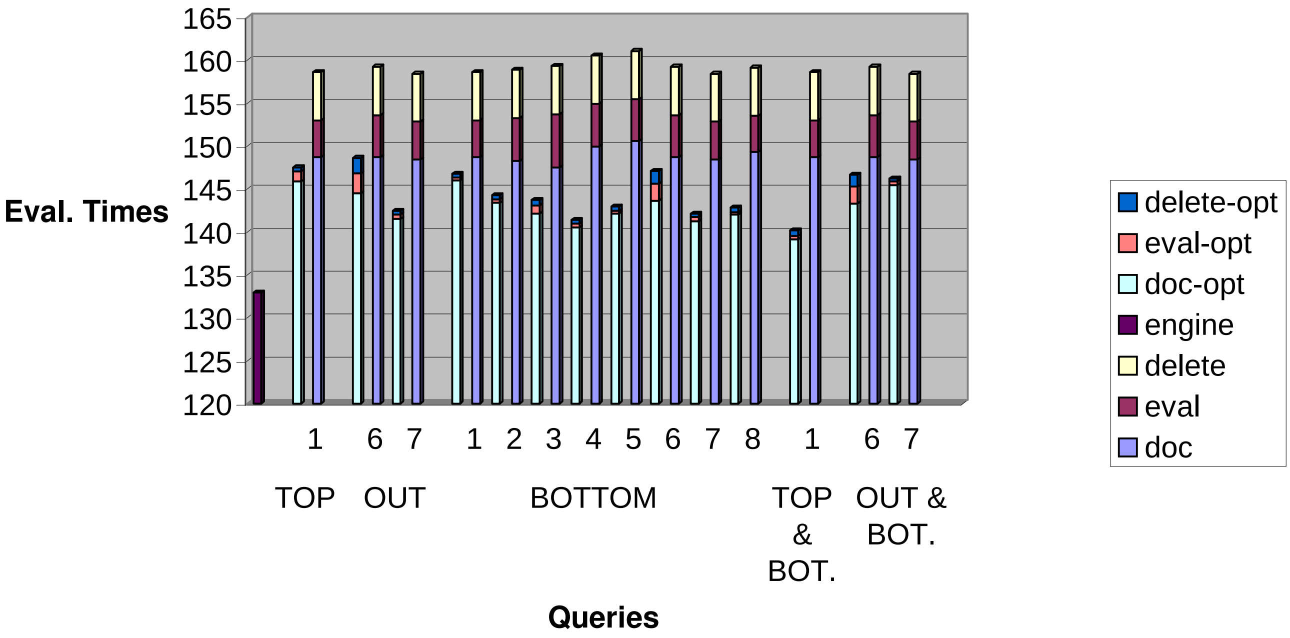

Figure 10. Evaluation Times for Optimizations with the Tree-Based Approach

1. "Select the first name of 'Boehm'." The documents contain a NAME-element which in turn contain a CHRNAME-element and a SURNAME-element. The query evaluates the TOP-optimization.

   (CHRNAME INCL-IN (NAME INCLUDS CONTENT_SELECT(SURNAME, 'Boehm')))

2. "Select the PERSONS from documents whose text contains string 'DOCKING'." (PERSONS INCL-IN (MMFDOC INCLUDS CONTENT_SELECT(TEXT, 'DOCKING')))

3. "Select the DOCID of documents containing a NAME-element which in turn contains a CHRNAME-element with content 'Klemens' and a SURNAME-element with content 'Boehm'."

   DOCID INCL-IN (MMFDOC INCLUDS ((NAME INCLUDS CONTENT_SELECT(CHRNAME, 'Klemens')) IN CLUDS CONTENT_SELECT(SURNAME, 'Boehm')))

4. "Select the DOCID of documents containing string 'Frankfurt'." The result of this query is relatively large. The query evaluates the FLAT-optimization.

   DOCID INCL-IN CONTENT_SELECT(MMFDOC, 'Frankfurt')

5. "Select the DOCID of documents containing string 'Elisabethenstift'." This query has the same structure as the one before. But there is only one document containing the string.

   DOCID INCL-IN CONTENT_SELECT(MMFDOC, 'Elisabethenstift')

6. "Select all TEXT-elements containing string 'DOCKING' contained in a document whose DOCID-element contains string 'Boehm'." The query evaluates the OUT-optimization.

   (CONTENT_SELECT(TEXT, 'DOCKING') INCL-IN (MMFDOC INCLUDS CONTENT_SELECT(DOCID, 'Boehm')))

7. The query evaluates the OUT-optimization as well. CONTENT_SELECT(INSTITUT, 'I4') INCL-IN (MMFDOC INCLUDS CONTENT_SELECT (DOCID, 'Boehm'))

8. "Select the DOCID of documents containing an element PERSON that contains string 'Boehm', and that has an attribute FUNCTION with value 'DEALT'."

   DOCID INCL-IN (MMFDOC INCLUDS CONTENT_SELECT(ATTR_SELECT(PERSON, FUNCTION, 'DEALT'), 'Boehm'))

References

[1] S. Abiteboul, S. Cluet, and T. Milo. Querying and Updating the File. In R. Agrawal, S. Baker, and D. Bell, editors,

Proceedings of the International Conference on Very Large Data Bases, pages 73–84. VLDB Endowment, 1993. Dublin, Ireland.

[2] S. Abiteboul et al. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1), 1997.

[3] G.E. Blake et al. Text / Relational Database Management Systems: Harmonizing SQL and SGML. In *Proceedings of the First International Conference on Applications of Databases*. Lecture Notes in Computer Science, Springer Verlag, June 1994.

[4] Klemens Böhm. On Extending the XML Engine with Query-Processing Capabilities, November 1999. available from the author.

[5] Klemens Böhm, Karl Aberer, Erich J. Neuhold, and Xiaoya Yang. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. *VLDB Journal*, 6(4):296–311, 1997.

[6] Klemens Böhm, Karl Aberer, M. Tamer Özsu, and Kathrin Gayer. Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition, 1998. Proceedings of IEEE Advances in Digital Libraries 1998 (ADL).

[7] Chen-Chuan K. Chang, Hector Garcia-Molina, and Andreas Paepcke. Predicate Rewriting for Translating Boolean Queries in a Heterogeneous Information System. *TOIS*, 17(1):1–39, 1999.

[8] V. Christophides et al. From Structured Documents to Novel Query Facilities. In *Proceedings ACM SIGMOD*. ACM Press, May 1994.

[9] James Clark. expat - XML Parser Toolkit. http://www.jclark.com/xml/expat.html.

[10] James Clark. SP – an SGML System Conforming to International Standard ISO 8879 – Standard Generalized Markup Language. http://www.jclark.com/sp/.

[11] C.L.A. Clarke, B.V. Cormack, and F.J. Burkowski. An Algebra for Structured Text Search and a Framework for its Implementation. *The Computer Journal*, 38(1):43–56, 1995.

[12] Mariano Consens and Tova Milo. Optimizing Queries on Files. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23, pages 301–312. ACM Press, May 1994. Minneapolis, Minnesota.

[13] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *Proceedings ACM SIGMOD*. ACM Press, 1999.

[14] Document Object Model (DOM). http://www.w3c.org/DOM/.

[15] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval*. Prentice Hall.

[16] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of Conference on Very Large Databases*, pages 436–445, 1997.

[17] P. Kilpeläinen and H. Mannila. Retrieval from Hierarchical Texts by Partial Patterns. In *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222. ACM Press, 1993.

[18] I.A. Macleod. A Query Language for Retrieving Information from Hierarchic Text Structures. *The Computer Journal*, 34(3):254–264, 1991.

[19] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *ICDT 1999*, pages 277–295, 1999.

[20] Yannis Papakonstantinou and Pavel Velikhov. Enhancing Semistructured Data Mediators with Document Type Definitions. In *Proceedings of the Fifteenth International Conference on Data Engineering*. IEEE Computer Society, 1999.

[21] A. Salminen and F.W. Tompa. PAT Expressions: an Algebra for Text Search. *Acta Linguistica Hungarica*, 41(1):277–306, 1994.

[22] SAX: The Simple API for XML. http://www.megginson.com/SAX/Index.html.

[23] Jayavel Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 1999. Edinburgh, Scotland.

[24] Klaus Süllow et al. MultiMedia Forum - an Interactive Online Journal. In Christoph Hüser, Wiebke Möhr, and Vincent Quint, editors, *Proceedings of Conference on Electronic Publishing*, pages 413–422. John Wiley & Sons, Ltd., April 1994.

[25] XML. http://www.w3c.org/xml/.