A Modular XQuery Implementation

Jan Vraný, Jan Žák

Department of Computer Science and Engineering,
FEE, Czech Technical University in Prague,
Karlovo namesti 13, 120 00, Praha, Czech Republic
{vranyj1,zakj2}@fel.cvut.cz

Abstract. *CellStore/XQuery* is modular a implementation of the XML Query Language interpreter. Primary goal of the implementation is to provide open experimental platform for implementing and testing new query optimization techniques for querying XML data. This paper describes architecture and design features of the implementation as well as possible approaches to extending the interpreter.

1 Introduction

XML is one of the most progressive technology. It has become popular as a format for data exchange between applications. However, nowadays XML is used not only for data exchange, but also for storing data. Several database management systems are specialized for storing XML documents. As more and more data were stored in the XML format, a need for querying XML data arised and thus there was a need for XML query language too.

XML query language [1] – or XQuery – is a language for querying designed by W3 Consortium. It combines features offered by two well-known and popular query languages - XPath [2] and SQL.

One of the most important and widely known constructs of SQL is *select-from-where-order by* clause. XQuery provides similar feature – so-called *FLWOR* expression. FLWOR is an acronym for *for-let-where-order by-return*.

Within an XQuery expression it is possible to use arithmetic, conditional, logical and comparison expressions, XPath expressions with predicates and function calls. It's also possible to define new functions.

XQuery is both query and transformation language, because it provides a facility for creating new XML documents using a special language construct called *constructor*.

This paper describes a XQuery implementation called *CellStore/XQuery*. The implementation was developed within the CellStore project. Primary goal of the implementation is to provide a modular implementation which is easy to understand and easy to extend. It's designed to serve as testing platform for various optimization techniques – new indexing methods, new storage models and so on.

The following text will briefly describe architecture of our XQuery implementation and possible ways how to extend the interpreter with some kind of new optimized algorithm in the future.

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2007, pp. 47–54, ISBN 80-7378-002-X.

2 XQuery interpreter architecture

Because our implementation is primarily intended as an experimental framework for testing a new approaches in XML data querying, it consists of several, well-defined and well-separated components. At figure 1 there is an architecture overview. In our implementation, each component is represented by one class and all other components communicate just through calling methods of this class. In many cases, implementation of component's functionality cannot be done within one class. In such cases, there is only one class which act as a facade for the rest. These facades together with true polymorphism makes changing of the implementations very easy.

When a query is to be executed, it's sent to a parser, which converts the query from it's textual representation to internal form. Then it the query in the internal form returned to the interpreter. The interpreter will then process the query. During query processing, various data sources might (or might not) be accessed (using the `fn:doc` or `fn:collection` functions). In such case, a document provider component is asked for particular document. For data accessing and navigating through a document, a data access component is used. There are one data access component instance per document. After processing the query, result is left in interpreter's data context.

Now we will describe each component in little bit more detail.

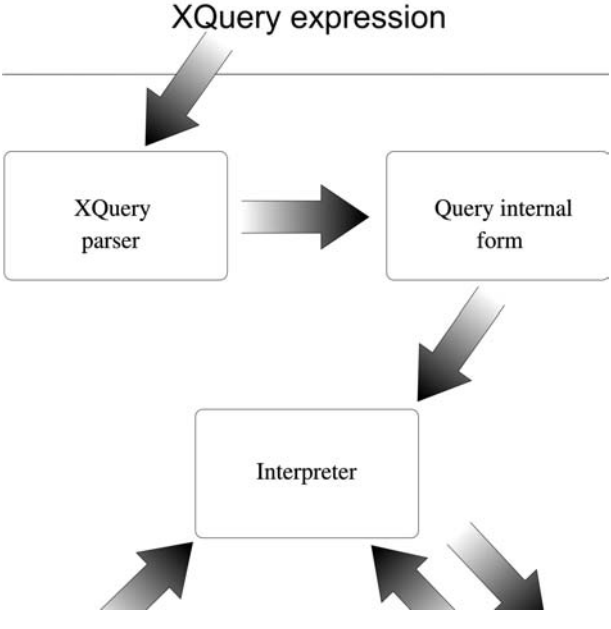

Fig. 1. XQuery architecture

2.1 Parsing XML query

The XQuery parser is implemented using SmaCC tool [5], which is able to generate LR(1) or LALR(1) parser from a grammar in the EBNF form. The grammar was taken from W3C specification. Because XQuery syntax is pretty complicated, some of its syntactic constructions cannot be easily handled by SmaCC-generated parser. This leads to some limitations of the parser which will be described in little bit more detail in the section 3.2.

2.2 Internal query representation

A parsed query is internally represented as a parse tree. For example the parse tree for the query at figure 2 is depicted at figure 3.

```
let $ a := 1
return  $ a + 1
```

Fig. 2. Example of an XQuery expression

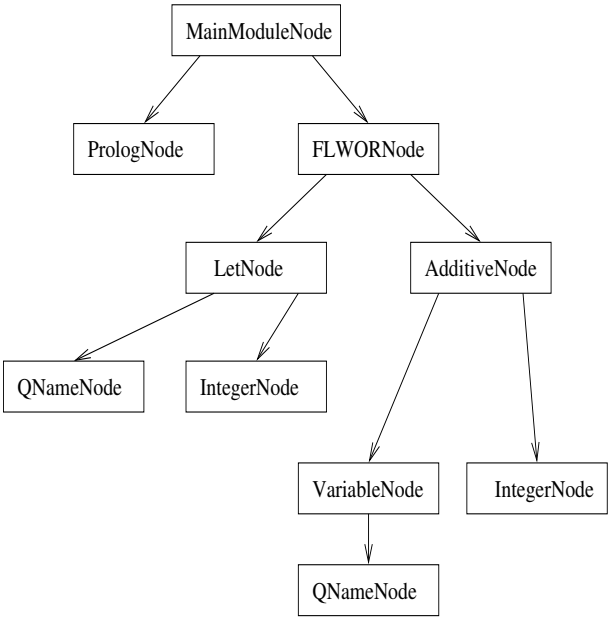

Fig. 3. Parse tree for the query at fig. 2

XQuery specifications defines so-called *XQuery Core*, which is a subset of XQuery. Every XQuery expression could be converted to XQuery Core expression, which has same semantics as the original query. Our implementation doesn't perform such conversion. A query is represented in the same form as it was written by user.

2.3 Accessing data sources

XQuery allows user to access more than one XML document per query using `fn:doc` and `fn:collection` built-in functions. These two functions loads document with given uri at current context[1], so the user can access and query data in th document. These functions could be used as many times as user wish.

Whenever a document is to be loaded into the current context, a *document provider* is asked to return the document in form in which it could be used by *interpreter* for further processing.

2.4 Accessing and navigating through XML data

Once a XML document is loaded in interpreter's context, data in the document are (not necessarily) accessed. Because we want to make our implementation as general as possible, we should have the possibility of different data source type and data models in mind. For example, documents could be stored as DOM nodes, as DOM-like nodes, as XML infoset or in data structures as they are used by various XPath accelerators.

It's important for any practical XQuery interpreter to access data stored in various data models, because user could access data in a XML database together with data stored on disk in one query.

To solve this problem, we developed a concept of *document adaptor*. It provides methods for data accessing (*xpathLocalNameOf:*, *xpathValueOf:* for example) and for navigating through the document structure in XPath axis manner (*xpathChildOf:*, *xpathParentOf:* for example). Those methods get so-called *node-id*, which identifies a node within a document, and return a set of node-ids or value represented as a string.

Full list of methods provided by the adaptor is in table 1. It's obvious that presented set of methods is not the minimal one. For example, method *xpathAncestorOf:* could be implemented using *xpathParentOf:*. It is also obvious, that such implementations might not be the most efficient ones for given storage model. Our implementation defines several so-called *primitive methods* that must be supported by the adaptor and every other method could be assembled as applications of primitive ones. However, user can override any of such methods and use the most efficient algorithm for given storage model.

Node-id could be any kind of object, it has no meaning for the rest of the system. It's used just for identifying XML nodes. For example in our implementation node-ids for DOM document tree adaptor are DOM nodes itself, whereas

[1] In fact, these function load just the root node. The rest of the document is loaded lazily.

Table 1. Document adaptor methods

method name	parameters	return value
xpathDocument		*node-id*
xpathIsAttribute:	*node-id*	`Boolean`
xpathIsDocument:	*node-id*	`Boolean`
xpathIsElement:	*node-id*	`Boolean`
xpathIsText:	*node-id*	`Boolean`
xpathLocalNameOf:	*node-id*	`String`
xpathNameOf:	*node-id*	`String`
xpathNameSpaceOf:	*node-id*	`String`
xpathValueOf:	*node-id*	`String`
xpathAncestorOf:	*node-id*	ordered set of *node-ids*
xpathAncestorOrSelfOf:	*node-id*	ordered set of *node-ids*
xpathAttributeOf:	*node-id*	ordered set of *node-ids*
xpathChildOf:	*node-id*	ordered set of *node-ids*
xpathDescendantOf:	*node-id*	ordered set of *node-ids*
xpathDescendantOrSelfOf:	*node-id*	ordered set of *node-ids*
xpathFollowingOf:	*node-id*	ordered set of *node-ids*
xpathFollowingSiblingOf:	*node-id*	ordered set of *node-ids*
xpathPrecedingOf:	*node-id*	ordered set of *node-ids*
xpathPrecedingSiblingOf:	*node-id*	ordered set of *node-ids*
xpathParentOf:	*node-id*	*node-id*

for CellStore XML database adaptor node-ids are pointers to CellStore's cell-space [6]

2.5 Interpreting the query

The XQuery interpreter is implemented as interpreter design pattern [3]. It traverses the parse tree and evaluates nodes. In order to a evaluate node, all its subnodes have to be evaluated. A node is evaluated within a *context* (which stores a data set, variable bindings and some auxiliary informations like *focus*) and as result of evaluation new context with new data is created and stored in the interpreter. There is one method per one parse tree node type in the `Interpreter` class. This has significant impact on implementation flexibility – see section 4.

An outline of method which interprets *additive node* at figure 3 is at figure 4. In fact, the real method is little bit more complicated, because XQuery defines process called *atomization*, which should be applied to every operand before addition is performed. Context handling is also a little bit tricky and should be cleaned-up.

3 Limitations

Our implementation has also some limitations, which will be described in this section.

```
visitAdditiveNode( additiveNode ) {
    var myContext, leftNodeContext, rightNodeContext;
    /* we have to save current context because it's overriden by sub-node */
    myContext = this.context;
    /* evaluate left node */
    this.evaluate( additiveNode.getLeftNode() );
    /* save its context */
    leftNodeContext = this.context;
    /* restore original context before right node's evaluation */
    this.context = myContext;
    /* evaluate right node */
    this.evaluate( additiveNode.getRightNode() );
    /* save its context */
    rightNodeContext = this.context;
    /* store result context */
    this.context = new Context( leftNodeContext.getData()
        + rightNodeContext.getData() )
}
```

Fig. 4. Method for evaluating additive node

3.1 Type system

XQuery language contains set of language constructs for typing and validating expressions against W3C XML Schema [4]. Our implementation does not support types. Unsupported constructs are not included in the grammar and thus using such constructs will result in a parse error.

Although no typing is supported in the query itself, our implementation internally uses six data types:

- `node`
- `xs:boolean`
- `xs:number`
- `xs:string`
- `xs:NCName`
- `xs:QName`

This is because XQuery contains constructs for arithmetic, conditional (*if-then-else*), logical and comparison expressions, XPath expressions with predicates and function calls.

3.2 Parser limitations

XML query language syntax contains some problematic parts, which cannot be handled by SmaCC-generated parsers.

First sort of problems is related to fact, that keywords (`if`, `div`, `return`, etc.) are not reserved words. That means, that whether some token is treated

```
element element {}
```

Fig. 5. XQuery expression with keyword as element name

as keyword token or *NCName* token depends on parsing context. Consider the query at figure 5.

Although it's perfectly legal according to the XQuery specification, our parser (generated by SmaCC) will raise a parse error because after "element" keyword (first "element") an *QName* is expected.

Another sort of problems is related to direct constructors. Direct constructors allows user to construct new nodes during the query processing using standard XML syntax. In fact, this means that every well-formed XML document is also valid XQuery expression and thus XQuery parser should parse any XML document. XML grammar itself is too complex to be handled by SmaCC. Our implementation currently supports direct constructors in a very limited way.

4 Extending the intepreter

In this section we will outline how to extend the XQueryinterpreter and add and test new optimization techniques. Basically, there are two levels at which our implementation could be extended and/or improved.

First one is the level of data access layer. To add a new type of data source, a new document adaptor has to be implemented. A complete set of tests is provided, so it's easy to find out whether new adaptor fulfils interpreters requirements. Because document adaptor implements axis-based navigation, any method that improves axis accesses can be used.

However, there are also several approaches that speeds-up not only per-axis access, but whole XPath expressions. Such techniques can be easily implemented and tested as well. Because whole query is represented as a parse tree, embedded XPath expression is represented as one parse tree node (whichh contains nodes for every part of expression) and because there is one method for each parse tree node type, one can simply override method responsible for evaluating XPath expression node and use any kind of optimization method. One can use either structural indices,value-based indices or both, if applicable. It's also possible to speed-up just some parts of XQuery expressions (for example, just parts without predicates). In fact, one can optimize any XQuery construct by this way.

5 Conclusion and future work

A prototype of the XQuery interpreter has been implemented. This implementation is currently very limited, but it supports all basic constructs: FLWOR expressions, full XPath expressions with predicates, conditionals, arithmetic, logical, comparison and quantified expressions, XML nodes construction and

function calls (both built-in and user-defined ones). It can be seen as a platform for further experiments with XML query optimization techniques using both indices (value-based and structural ones) and special techniques like structural joins.

Further development will be focused on the following topics:

- improvement of the XQuery parser
- implementing full set of built-in functions
- simplifying the context machinery in the interpreter
- improving document adaptor for CellStore/XML database by using some XML-specific optimization techniques
- creating interface to the XQuery Test Suite [7]

References

1. S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C, 1st edition, 2006. http://www.w3.org/TR/xquery/.
2. J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. W3C, 1999. `http://www.w3.org/TR/xpath`.
3. K. B. Sherman R. Alpert and B. Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1st edition, 1998.
4. C. M. Sperberg-McQueen and H. Thompson. *XML Schema 1.1*. W3C, 2006. `http://www.w3.org/XML/Schema`.
5. The Refactory Inc. *SmaCC compiler-compiler*. The Refactory Inc., 1st edition, 2000. http://www.refactory.com/Software/SmaCC.
6. J. Vraný. Cellstore - the vision of pure object database. In *Processings of DATESO 2006*, pages 32–39, 2006.
7. W3C XML Query Working Group. *XML Query Test Suite*. W3C, 1st edition, 2006. http://www.w3.org/XML/Query/test-suite/.