A Query Processor for XML Documents

by

Yidan Rogers

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2002 by Yidan Rogers

Abstract

A Query Processor for XML Documents

Yidan Rogers

Master of Science

Graduate Department of Computer Science

University of Toronto

2002

Extensible Markup Language (XML) is emerging as a standard for data representation and data exchange. It is important to have a query language suitable for XML documents and a query processor to process queries written in this query language. XQuery is a potential standard query language for XML documents and XQuery Core is a fully expressive sublanguage of it. This thesis describes a query processor that implements XQuery Core with extended features as the query language. The query processor can be used as a stand-alone application, or as a part in the ToX (Toronto XML) engine which provides tools to manipulate a repository of XML documents. Experiments are performed to show the effects of document size and query writing on the performance of the query processor. Related and future work are also discussed.

Acknowledgements

This thesis work was performed under the supervision of my supervisor, Prof. Alberto Mendelzon, and I would like to thank him for his invaluable advice and guidance. I am also grateful to Prof. Libkin for being the second reader of my thesis, and to Flavio Rizzolo, Ariel Fuxman, Cecilia Pilar, Karima Echihabi and Edward Xia for helpful discussions and advice.

Contents

1 Introduction **1**

1.1 XML 1

1.1.1 XML documents 1

1.1.2 Parsing XML documents 4

1.1.3 XML Namespaces 5

1.2 XML Query Languages 6

1.3 Related Work 12

1.4 Purpose and Outline of the Thesis 14

2 XQuery Core **16**

2.1 Overview 16

2.2 XQuery Core Declarations, Expressions and Types 17

2.3 Mapping between XQuery and XQuery Core 21

2.4 Query Examples 22

3 ToX Query Processor **26**

3.1 ToX Project Overview 26

3.2 Query Processor in ToX 29

3.2.1 Query Processor Architecture 29

3.2.2 Query Language 30

3.2.3 Query Parsing 32

3.2.4 Semantic Checking 35
3.2.5 Query Optimization 38
3.2.6 Query Evaluation 39

4 Query Processor Implementation and Results **46**
4.1 Query Processor Implementation 46
4.2 Results 47
4.2.1 Correctness 47
4.2.2 Performance 50

5 Conclusion and Future Work **63**
5.1 Conclusion 63
5.2 Future Work 63

A Grammar of the modified XQuery Core **65**

B DTD for XQCoreX **71**

C Implemented Features of XQuery Core **74**

D DTD and Sample Data of the Use Cases **78**

E The ToX Document Navigator API **83**

F Translation Rules of IRX **88**

Bibliography **94**

Chapter 1

Introduction

1.1 XML

1.1.1 XML documents

Today, data or information, and the transformation of data, have become the critical centerpiece of every application being developed. Thus a standard for data representation and data exchange between computers and computer applications is required. Extensible Markup Language (XML) is quickly emerging as such a standard because it offers flexibility, simplicity, and readability by both humans and machines. XML is a subset of the Standard Generalized Markup Language (SGML), which is a metalanguage used to define other languages. SGML is rich, powerful, but very complex. XML is much simpler and more straightforward than SGML, and the following properties make it more suitable for use on an increasingly complex and diverse Web.

- XML can be used with existing Web protocols and mechanisms.

- XML supports a wide variety of applications.

- XML is compatible with SGML, so most SGML applications can be converted into XML.

- It is easy to write programs that process XML documents.
- Information marked up in XML can be easily processed by computer programs.
- XML documents are easy to create.
- XML is based on Unicode and Unicode enables XML to handle not just Western-accented characters, but also Asian languages.
- XML is data-oriented. It describes the structure of data not the presentation.

The following is a sample XML document.

```
<?xml version="1.0"?>
<!DOCTYPE bib SYSTEM "bib.dtd">
<! − − This is a sample XML file −− >
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated < /title>
    <author><last>Stevens< /last><first>W.< /first>< /author>
    <publisher>Addison-Wesley< /publisher>
    <price> 65.95< /price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment< /title>
    <author><last>Stevens< /last><first>W.< /first>< /author>
    <publisher>Addison-Wesley< /publisher>
    <price> 65.95< /price>
  </book>
</bib>
```

An XML document basically consists of two parts: the header, which gives an XML version, XML parser and XML application information about how to handle the document; and the content, which is the XML data itself.

The typical lines in the header are <?xml version="1.0"?> and <!DOCTYPE rootelement SYSTEM dtdlocation>, which specify the version of XML and the external document type definition (DTD) file respectively.

The content of an XML document has one root element or document element, and a sequence of elements with or without attributes, comments, and processing instructions. Elements are the basic logical constructs of a document. They are bound by start tags and end tags. An attribute is a property of an element, and it is expressed as a name=value pair inside an element's start tag. A comment begins with <!−− and ends with −−>. Comments may appear anywhere in a document as long as they are outside other markup and not immediately prior to the XML header. Processing instructions are markup that provides information to be used by software applications, and they are bound by <? and ?>.

An XML document must be well-formed to be processed correctly. A well-formed document is one that has only one root element, has matching start and end tags for every element, has no tags nested out of order, and is syntactically correct in regard to the specification. An XML document can be, but is not required to be, valid. A valid document is one that is well-formed, and conforms to its document type definition (DTD). A DTD defines the grammar of the XML document, i.e., it specifies the structure and the constraints of the document. The following is the DTD [11] for the sample XML document.

```
<!ELEMENT bib (book*) >
<!ELEMENT book (title, (author+ | editor+ ), publisher, price ) >
<!ATTLIST book year CDATA #REQUIRED >
<!ELEMENT author (last, first) >
```

<!ELEMENT editor (last, first) >

<!ELEMENT title (#PCDATA) >

<!ELEMENT last (#PCDATA) >

<!ELEMENT first (#PCDATA) >

<!ELEMENT publisher (#PCDATA) >

<!ELEMENT price (#PCDATA) >

A DTD should declare every element and its attributes if there are any. An element declaration starts with the string <!ELEMENT, followed by the name of the element, and then the content specification. The content specification can be a regular expression to indicate the subelements that must occur and how they would occur, or can be PCDATA for a text value or EMPTY. Attributes of an element are declared with the string <!ATTLIST, followed by the name of the element, and a list of the attribute name, its value type or default value, and the key word REQUIRED or IMPLIED to indicate whether the attribute is optional. In the above example, the DTD says that the `author` element has two subelements: `last` and `first`; `last` is followed by `first`, and both subelements have text values.

1.1.2 Parsing XML documents

XML documents have to be parsed in order to be used by application programs. This process can be done in different ways depending on how the XML data are treated: plain text, a series of events happening, or a tree that can be navigated and manipulated. Figure 1.1 shows the application view of an XML document lifecycle [28].

SAX

Simple API for XML (SAX [32]) provides an event-based framework for parsing XML data. When an XML file is read sequentially, a series of events occur, such as encountering the start of a document, and reading the start and end tags. SAX API defines

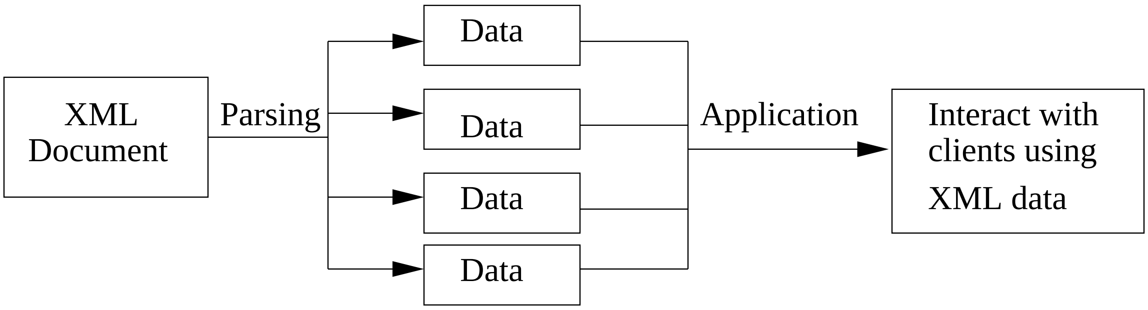

Figure 1.1: The application view of an XML document lifecycle

means to handle these events. Some XML parsers such as Xerces Java SAX parser [38] implement SAX API.

DOM

The Document Object Model (DOM [7]) sees the XML document as a tree consisting of nodes and edges. Each node represents a piece of the data (an element, an attribute, or text etc.) pulled from the original document. The node types include element node, attribute node, text node, and document node. DOM defines means to access and manipulate the nodes, such as getting the parent and children of a particular node, and deleting a node. A DOM parser (such as Apache's Xerces DOM parser [38]) usually reads an entire XML document into memory for fast access.

1.1.3 XML Namespaces

An XML namespace [8] is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names. XML namespaces provide a simple method for qualifying element and attribute names used in XML documents. They are used to handle name collisions and define data structures that allow parsers to handle collisions.

Names from XML namespaces appear as qualified names. A qualified name is a local name such as `book` or a name which contains a single colon separating the name into a namespace prefix and a local name, such as `xx:book`. The namespace prefix is associated with a URI reference. The combination of the universally managed URI namespace and the document's own namespace produces identifiers that are universally unique. The namespace prefix is needed since URI references may contain characters that are not allowed in names and cannot be used as a prefix directly. The following is a namespace example [8]:

```
<?xml version=”1.0”?>
<html:html xmlns:html=’http://www.w3c.org/TR/REC-html40’>
  <html:head> <html:title>Frobnostication < /html:title>< /html:head>
  <html:body> <html:p>Moved to
    <html:a href=’http://frob.com’>here.< /html:a> < /html:p>< /html:body>
< /html:html>
```

In the above example, `xmlns:html='http://www.w3c.org/TR/REC-html40'` is the namespace declaration, and `html:title` is a qualified name.

1.2 XML Query Languages

With the increasing popularity of e-commerce, more and more data on the Web are in the form of XML. Thus it is important to have a query language to extract information from the XML data efficiently. This language should be able to produce XML output, so that the output can be further processed by XML enabled applications. This language should be able to preserve document order and hierarchy, and also be able to support a variety of traditional database operations such as selections, joins, and groupings. Furthermore, this language needs to be able to deal with all of the information structures in XML and transform information from one structure to another. The W3C Query Working Group has published the XML query requirements [10].

Many query languages have been proposed, and the W3C Query Working Group is working to make XQuery as the standard of XML query language. Some influential query languages are the following:

- XPath

  XML Path Language (XPath) [9] is a specification that defines how a specific item within an XML document can be located. XPath models an XML document as a tree of nodes which includes element nodes, attribute nodes, and text nodes. XPath fully supports XML Namespaces, so the name of a node can be an expanded name which consists of a local part and a possibly null namespace URI.

  Expression is the primary syntax construct in XPath. An expression is evaluated to one of the four basic value objects: node-set (an unordered collection of nodes without duplicates), boolean (true or false), number (a floating-point number), and string (a sequence of characters). Expression evaluation occurs with respect to a context, which can be specified by the Extensible Stylesheet Language Transformations (XSLT) or other applications. The context consists of: a node (the context node), a pair of non-zero positive integers (the context position and the context size), a set of variable bindings, a function library, and the set of namespace declarations in the scope for the expression. One important kind of expression is a location path, which selects a set of nodes relative to the context node. Location paths can recursively contain expressions that are used to filter sets of nodes.

- XQL

  The XML Query Language (XQL) [26] is an extension of the Extensible Stylesheet Language (XSL) [31]. The key features of XQL are [19]: XML documents can be queried like a database; XQL allows powerful queries to be simply expressed and uses a string syntax that allows queries to be used in URL's or embedded in attributes; and it is easy to optimize for large-scale repositories.

- Quilt

  Quilt [5] is designed as a small and implementable language which meets the requirements identified by the W3C XML Query Working Group. The designers named it Quilt because it is like a patchwork of features from other languages and its goal is to assemble information from multiple diverse sources. From XPath and XQL, it took the path expression syntax for navigating hierarchical documents. From XML-QL, it took the notion of binding variables and the use of the bound variables to create new structures. From SQL, it took the idea of SELECT-FROM-WHERE pattern and provides a way to restructure data. From OQL, it took the notion of a functional language composed of different kinds of expressions that can be nested with full generality. Other query languages such as Lorel and YALT also have influence on Quilt.

- XQuery

  XQuery is a potential standard XML query language, designed under the influence of other languages such as SQL, OQL [22], XQL, Lorel [1], YALT [6] and XML-QL [20]. The semantics of XQuery is defined in [17]. The details of the operators and functions supported by XQuery are defined in [16].

  XQuery is a typed language. The type system of XQuery is based on XML Schema [12] and is used to describe the semantics of datatype operations and type conversion rules in XQuery. XML Schema defines a notion of validation for XML documents. When doing validation, an XML Schema processor checks if the structure verifies the constraints on the structure, and if the text content of a document verifies the constraints on values, specified in the schema. The XQuery type system is based on the structural constraints of XML schema. The main differences between the XQuery type system and XML Schema are: the XQuery type system does not capture the notion of XML Schema names (also called normalized universal names in [12]) and the XQuery type system uses a specific syntax, the static type inference.

XQuery is also a functional language in which a query is represented as an expression. The input and output of a query are instances of a data model defined in [15]. This data model is a refinement of the data model used by XPath 1.0, in which an XML document is treated as a tree of nodes. The principal forms of XQuery expressions are the following:

- path expressions

  Based on the syntax of XPath, a path expression starts with an expression that identifies a specific node or a sequence of nodes in a document, consists of a series of "steps", and returns a sequence of nodes or primitive values.

- element constructors

  An element constructor is used to generate an element node. It starts with a start tag and ends with an end tag, between which there is an optional list of expressions that provide the content of the element.

- FLWR expressions

  An FLWR expression consists of FOR, LET, WHERE, and RETURN clauses in a specific order. It binds values to one or more variables and then uses these variables to construct the result.

- expressions involving operators and functions

  XQuery supports the usual set of arithmetic operators such as $+$ and $-$, logical operators such as AND and OR, and the collection operators UNION, INTERSECT and EXCEPT. In addition, XQuery has BEFORE, AFTER, and FILTER operators. XQuery provides many built-in functions, including all the functions of the XPath core function library as well as aggregation functions such as `sum` , `avg`, `count`, `max` and `min`. Apart from the built-in functions, XQuery allows users to define their own functions.

- conditional expressions

  A conditional expression evaluates a test expression, and then depending on the result of the evaluation, returns the value of one of the two result expressions.

- quantified expressions

  Quantified expressions are used to test for the existence of some element that satisfies a condition, or to determine whether all elements of some kind satisfy a condition. The SOME expression uses an existential quantifier, and the EVERY expression uses a universal quantifier.

- expressions that test or modify data types

  XQuery's type system is based on XML Schema. By using the datatype names defined in the namespace `http://www.w3.org/2001/XMLSchema`, all the primitive and derived datatypes of XML Schema can be used in queries. Other complex types declared using XML Schema can also be referred to by their qualified names. In function declarations, type names are used to specify the types of function parameters and result. Besides, type names are also used in CAST, TREAT, and TYPESWITCH expressions and as operands of the INSTANCEOF operator.

The following are some XQuery queries taken from the XML Query Use Cases [11]. The XML documents for these queries are in Appendix D.

Query 1: Create a flat list of all the title-author pairs, with each pair enclosed in a "result" element.

```
<results> {
   FOR $b IN document("bib.xml")/bib/book,
      $t IN $b/title, $a IN $b/author
   RETURN
```

```
    <result> {$t}
      {$a}
    </result>
}</results>
```

Query 2: For each book that has at least one author, list the title and first two authors, and an empty "et-al" element if the book has additional authors.

```
<bib> {
   FOR $b IN document("bib.xml")//book
   WHERE count($b/author) > 0
   RETURN
      <book> { $b/title }
         { FOR $a IN $b/author[1 TO 2] RETURN $a }
         { IF (count($b/author) > 2) THEN <et-al/> ELSE ()}
      </book>
 }</bib>
```

Query 3: In the document "prices.xml", find the minimum price for each book, in the form of a "minprice" element with the book title as its title attribute.

```
<results> {
  LET $doc := document("prices.xml")
  FOR $t IN distinct($doc/book/title)
  LET $p := $doc/book[title = $t]/price
  RETURN
      <minprice title={$t/text()}>
        { min($p) }
      </minprice>
}</results>
```

1.3 Related Work

Though the XQuery working drafts are still a "work in progress", there are some XQuery implementations publicly available:

- Quip [2]

  Quip is an XQuery prototype implementation developed by Software AG. It is available on Windows 32 bit platforms and requires a Java virtual machine version 1.3. Quip conforms to the 7-June-2001 draft of XQuery, and can be used with queries against text-based XML files or a Tamino database [2].

  The following features are not implemented in Quip: dereference symbol; namespace functions; static type checking or schema evaluation; the test for equality on nodes compares strings (this corresponds to the implementation in XPath). And other features are partially implemented.

- XQuery Demo [29]

  XQuery Demo is Microsoft's implementation of the 7-June-2001 version of XQuery. It was meant to get developers familiar with XQuery and get feedback on XQuery specification and the implementation. It is available as a web-based tool or a download, which consists of managed classes that can be programmed against using the Beta 2 release of the .Net Framework SDK.

  There are limitations of this prototype. User defined functions are not implemented. Some functions such as `distinct`, `sortby`, `unordered`, `namespace_uri` are not supported. There is no type checking. The known issues are listed in the demo webpage.

- X-Hive [18]

  X-Hive Corporation also implemented the 7-June-2001 version of XQuery and the online demo is at [18]. This demo works with XML documents stored in a native

XML database instead of XML documents stored in the file system. Apart from the queries from the XQuery Use Cases document [11], it also includes sample queries from the XMach-1 benchmark, and users can pose their own queries on the sample databases.

The known issues are listed in the demo webpage. There are some functions such as `filter` and `bags-are-equal` that are not implemented, but most of the features of XQuery have been implemented.

- Galax [34]

  The Galax demo from Lucent Technologies includes an implementation of the XQuery 1.0 Formal Semantics, which is a subset of XQuery. Apart from query evaluation, this implementation supports static typing. The inferred type of the query is displayed along with the query result. The sample queries are from the XQuery 1.0 Formal Semantics document, and there are some other examples.

- Kweelt [25]

  Kweelt offers an evaluation engine for the Quilt query language with a lot of useful extensions. It can be run on multiple XML backends, and is extensible in the sense of supporting user-defined functions. Kweelt comes with the Kweelt Server Pages (KSP) extension, a built-in Cocoon processor[1] . KSP allows the embedding of Kweelt queries inside any XML page serviced by Cocoon.

  The main components of the Kweelt engine are the parser and evaluator. The query is parsed into an abstract syntax tree and interpreted as is rather than compiled. There is no static typing or semantics checking. Some features of Quilt are removed.

In addition to the above implementations, Fatdog Software's XML Query Engine [33] also has early support for XQuery (based on the 15-February-2001 specification). The

[1]Cocoon is an XML publishing frame work that raises the usage of XML and XSLT technologies to a new level[37]. The most common use of Cocoon is the automatic creation of HTML through the processing of statically or dynamically generated XML files.

list of XQuery implementations is changing frequently. The XML Query Working Group is now maintaining a list of XQuery implementations on its Web page at [36].

Of the above work, only Galax implements of XQuery Formal Semantics, and Kweelt implements Quilt; the rest all implement XQuery. Like Galax, this thesis does not implement XQuery, but instead, it implements XQuery Core (the core syntax of XQuery), which is a subset of XQuery. The goal of this thesis is not only to develop a query processor for XML documents, but also to integrate the query processor into the ToX (Toronto XML) framework [3] to query documents and collections in an XML repository.

1.4 Purpose and Outline of the Thesis

The purpose of this thesis is to investigate a query language suitable for XML documents, build a query processor to process queries written in this language, and also integrate the query processor into the ToX framework.

The query language chosen is XQuery Core, which was originally meant to serve as an algebra for XQuery. XQuery Core is smaller and simpler than XQuery (it is a subset of XQuery), but it is fully expressive. So implementing XQuery Core would be easier and faster, and still conform to the standard if XQuery eventually becomes the standard XML query language. However, XQuery Core excludes path expressions and this may cause some problems. Queries without path expressions need to use many nested FOR loops; thus a simple query can become complicated and hard to write and understand. The other problem is that the equivalent XQuery Core expressions to a path expression involve TYPESWITCH expressions; thus subtyping and type inference operations are needed, which would increase the complexity of the query processing. Besides, not all XML documents have type information. To avoid the above problems and to take advantage of a backend API that handles path expressions, path expressions were added to the XQuery Core.

My contributions are the following:

- modifying the XQuery Core syntax to include path expressions;
- implementing the query processor;
- integrating the query processor into the ToX engine, which manages an XML repository and poses queries over XML collections and documents;
- translating the query into XML format, and also producing intermediate representations in XML format for further processing such as optimization and/or interpretation;
- performing experiments to show the effects of document size and query writing on the performance of the query processor.

The thesis is organized as follows: Chapter Two describes the XQuery Core; Chapter Three describes the ToX query processor, including the phases of query parsing, semantic checking, and query evaluation; Chapter Four describes the implementation of the query processor and shows some example queries and experimental results; Chapter Five is the conclusion and future work.

Chapter 2

XQuery Core

2.1 Overview

XQuery 1.0 Formal Semantics document [17] defines the formal semantics of XQuery with respect to a "core syntax", which is a subset of the complete XQuery syntax and is referred to as XQuery Core. The core syntax of XQuery serves as an algebra for XQuery. An expression in XQuery can be rewritten as an expression in the core syntax, i.e., there is a mapping between XQuery syntax and the core syntax.

Types can be used to detect certain kinds of errors at query-analysis time and to support query optimization. The XQuery formal semantics uses a type system based on XML Schema [12]. One component of the formal semantics is the *static semantics*, which is defined through inference rules. The inference rules relate XQuery expressions to types, and also specify under what conditions an expression is well-typed. This semantics is static because the ill-typed expressions can be identified at the query-analysis time, i.e., before query evaluation.

Another component of the formal semantics is the *dynamic semantics*, which is defined through value inference rules. The value inference rules relate XQuery expressions to values defined in the XQuery 1.0 and XPath 2.0 Data Model [15]. This semantics guarantees that every core expression can be reduced to a value and may serve as a basis for a query interpreter or compiler.

2.2 XQuery Core Declarations, Expressions and Types

There are three kinds of declarations in XQuery Core: context declaration, type declaration, and function definition. Context declarations are used to declare namespaces. Type declarations are used to define data types. Function definitions are used to define functions. The syntax of these declarations is given in the XQuery Formal Semantics document. This document also describes the grammar of the XQuery Core expressions. The XQuery Core expression is a subset of the XQuery surface expression plus the ERROR expression. The following are the main forms of the XQuery Core expressions:

- Variables and literals

  Variables start with a \$ sign, such as `$u`. Literals include string literals such as `"book"`, boolean literals such as `true`, and numeric literals such as 34.

- Expressions involving operators and functions

  The operators are the normal arithmetic operators such as +; boolean operators such as OR; collection operators UNION, INTERSECT, and EXCEPT ; and equality operators such as LT, including NODE-EQUAL. The core built-in functions are a subset of the XQuery built-in functions. And the core syntax allows the user to define his own functions.

- LET expression

  LET Expression ::= `LET` *variable* `:=` *expression*1 `RETURN` *expression*2

  The LET expression binds a variable to the value of an expression, and then uses the bound variable to construct the result. For example, the following LET expression binds a variable to a number and returns a numerical value.

```
LET $a := 6 RETURN $a/60
```

Even though the variable can be bound to any value, semantic analyzing and type checking should ensure the binding is sound in the returned expresssion; otherwise, a semantic error will be generated. (Please see [17] for semantics of XQuery Core.)

- FOR expression

  FOR Expression ::= `FOR` *variable* `IN` *expression*1 `RETURN` *expression*2

  The FOR expression is used for iteration. The variable in a FOR expression can be thought of as iterating over the values returned by its associated expression (*expression1*). The result of a FOR expression is a sequence of values. In the following example, `$root` is a variable bound to the root element of an XML document, and the `children` function returns the child nodes of a specific node, so the FOR expression binds the variable `$a` with each child node of the root. Then for each child, the inner expression (*expression2* returns the name of the node, and finally, all the name values are appended into the result sequence.

  ```
  FOR $a IN children($root) RETURN name($a)
  ```

- Element constructor and attribute constructor

  These expressions are used to construct new attributes and elements. For example, the following expression constructs an element called review.

  ```
  <review> { "An interesting book." }< /review>
  ```

  <review> and < /review> are the start and end tags. Within the { } is the expression sequence which defines the content of the element. In this example, it is a string.

- IF expression

  IF Expression ::= `IF` *expression*1 `THEN` *expression*2 `ELSE` *expression*3

  If *expression*1 is evaluated to be true, then the value of *expression*2 is returned; otherwise, the value of *expression*3 is returned.

- SORTBY expression

  SORTBY Expression ::= *expression*1 `SORTBY` *expression*2 `(ASCENDING | DESCENDING)`

  This expression is used for sorting according to a property of a sequence of values. The built-in variable `$dot` ranges over the items in the sequence of *expression*1 and sorts those items using the key value defined by *expression*2. The following is an example, which sorts the `book` elements in `$review` by their titles:

  `$review/book SORTBY $dot/title/data() ASCENDING`[1]

- Type casting expressions

  There are two kinds of type casting expressions. One uses the syntax *Expression:Type*, and the other is the CAST AS expression in XQuery. The first kind of type casting casts type statically, while CAST AS casts type dynamically.

- TYPESWITCH expression

  TYPESWITCH expression has branches each of which is a CASE clause, and each branch is labeled with a type (explained later in this section). The following is an example:

```
DEFINE FUNCTION foo ( xs:AnyTree $x) RETURNS HOO {
  TYPESWITCH ($x) AS $z
    CASE xs:AnySimpleType RETURN $z
    CASE xs:AnyAttribute RETURN
      <b> { name($z) }< /b>
    DEFAULT RETURN ERROR
}
```

  In this example, HOO is a type defined previous to this function, and its type definition could be:

```
TYPE HOO = (xs:AnySimpleType | ELEMENT b (xs:string))*
```

[1] Path expressions are not in the XQuery Core. They are used here only for simplicity. Please see the next section for the mapping between XQuery and XQuery Core expressions.

The first branch of the TYPESWITCH expression checks (at run time) whether the value of `$x` is a subtype of `xs:AnySimpleType`, and if it is then `$z` is bound to that value. So if this branch is taken, then `$z` is the same as `$x` but with a more precise type. The second branch checks (at run time) whether the value of `$x` is a subtype of `xs:AnyAttribute`. If it is, as before, `$z` is the same as `$x` but with a more precise type. This branch returns a `b` element containing the name of the attribute.

- ERROR expression

  The ERROR expression is not in XQuery. It indicates an error as in the example of the TYPESWITCH expression.

The XQuery Formal Semantics document also defines the abstract syntax for XQuery's type system. The built-in simple datatypes of the XQuery Core are defined in XML Schema Part 2 [13], in which *xs* is the namespace prefix associated with the URL http://www.w3.org/2001/XMLSchema that describes the XML Schema namespace. However, for simplicity, XQuery Core does not provide a way to define datatypes with facets. Corresponding to the four kinds of schema components: elements, attributes, simple types and complex types, there are four general types: AnyElement, AnyAttribute, AnySimpleType, and AnyComplexType. The user can also define his own data type using a type declaration such as:

```
TYPE Book = ELEMENT book ( ATTRIBUTE year (xs:integer),
            ELEMENT title (xs:string),
            (ELEMENT author (xs:string))+ )
```

2.3 Mapping between XQuery and XQuery Core

The semantics of the full XQuery can be obtained by mapping the full XQuery expression (in the full XQuery grammar) into the core syntax. The XQuery Core does not contain path expressions. In order to capture simple navigation in path expressions, it uses a combination of the `children` function, iteration, and the TYPESWITCH expression. For recursive navigation, expressed by "//" in the abbreviated path expression [9], it uses the `descendant-or-self` function, which returns the current node as well as all its descendants. The following mapping applies when `y` is a legal name in the path expression.

XQuery	XQuery Core

`expression/y`	`FOR $v1 in expression RETURN`
	   `FOR $v2 in children($v1) RETURN`
	     `TYPESWITCH ($v2) AS $v3`
	       `CASE ELEMENT y (xs:AnyComplexType) RETURN $v3`
	       `DEFAULT RETURN ()`

`expression//@y`	`FOR $v1 in descendant-or-self(expression) RETURN`
	   `For $v2 in attributes($v1) RETURN`
	     `TYPESWITCH ($v2) AS $v3`
	       `CASE ATTRIBUTE y (xs:AnyComplexType) RETURN $v3`
	       `DEFAULT RETURN ()`

In XQuery, FLWR expressions can have complex forms where LET and FOR clauses may have several variables. In contrast, XQuery Core only allows elementary FOR and LET clauses (one variable in one clause) independent from each other, and terminated by a RETURN clause. Moreover, a WHERE clause of XQuery is translated into a IF expression of XQuery core. Operators in XQuery are mapped to corresponding operators in the XQuery Core. The XQuery Formal Semantics document provides details of the mapping.

2.4 Query Examples

The following are some example queries expressed in XQuery Core, based on accessing XML data shown in Appendix D.

1. Projection

   Query 2.0: Get all `author` elements contained in `book` elements contained in `$bib`.

   ```
   FOR $x IN children($bib) RETURN
     TYPESWITCH $x AS $a
       CASE ELEMENT book(xs:AnyComplexType) RETURN
         ( FOR $d0 IN children($a) RETURN
           TYPESWITCH ($d0) AS $d1
             CASE ELEMENT author(xs:AnyComplexType) RETURN $d1
             DEFAULT RETURN () )
       DEFAULT RETURN ()
   ```

   Using path expression, this query is just `$bib/book/author`.

   Query 2.1: Get the year the book was published for all the book elements in `$bib`.

   ```
   FOR $x IN children($bib) RETURN
     TYPESWITCH $x AS $a
       CASE ELEMENT book(xs:AnyComplexType) RETURN
         ( FOR $d0 IN attributes($a) RETURN
           TYPESWITCH ($d0) AS $d1
             CASE ATTRIBUTE year(xs:AnyComplexType)
               RETURN typed-value($d1)
             DEFAULT RETURN () )
   ```

```
DEFAULT RETURN ()
```

Using path expression, this query is just `$bib/book/attribute year/data()` where `data()` is the key word to access simple data (strings, integers or booleans).

2. Selection

Query 2.2: List all `book` elements in `$bib` that were published after 1998.

```
FOR $b0 IN ( FOR $x0 IN children($bib) RETURN
                 TYPESWITCH $x0 AS $y0
                   CASE ELEMENT book(xs:AnyComplexType) RETURN $y0
                   DEFAULT RETURN () )
             ) RETURN
              IF ( FOR $b1 IN children($b0) RETURN
                TYPESWITCH $b1 AS $y1
                CASE ATTRIBUTE year(xs:AnyComplexType)
                   RETURN typed-value($y1)
                DEFAULT RETURN () )
             ) > 1998 THEN $b0 ELSE ()
```

As we have seen, without the path expressions, the query becomes "verbose" and complex in XQuery Core. Thus, in the following examples, we still use the path expressions just for simplicity and ease of reading.

3. Iteration and Element Construction

Query 2.3: For each book in `$bib`, create a `book` element with only the authors and the title. Put the results in a `result` element.

```
LET $bib:=document("file:bib.xml") RETURN
   <result> {
      FOR $b IN $bib/book RETURN
         <book> { $b/author, $b/title } </book>
```

```
} </result>
```

4. Join

   In appendix D, bib.xml contains the information about books, while reviews.xml contains the book reviews. The following query joins the two data sources on title values.

   Query 2.4: For the books in bib.xml, find each book's title, authors and reviews, and put them in a book element.

```
LET $bib:=document("file:bib.xml") RETURN
 LET $review:=document("file:reviews.xml") RETURN
  <result> {
    FOR $b IN $bib/book RETURN
     FOR $x IN $review/book RETURN
       IF ($b/title/data() = $x/title/data()) THEN
         <book> { $b/author, $b/title, $x/review} </book>
       ELSE ()
  } </result>
```

5. Sorting and Aggregation

   Query 2.5: From bib.xml, select books with more than one authors, and report them in an ascending order by their titles.

```
LET $bib:=document("file:bib.xml") RETURN
  <result> {
    (FOR $b IN $bib/book RETURN
      IF count($b/author) > 1 THEN $b ELSE ()
    ) SORTBY $dot/title/data()
  } </result>
```

6. User-defined Functions

Functions can make queries modular and concise. The following defines a function that takes a string `$s` and a book `$b` as arguments, and returns true if book `$b` does not have an author with name `$s`. `BOOK` is the type defined for books.

```
DEFINE FUNCTION notauthor (xs:string $s, BOOK $b)
RETURNS xs:boolean {
    FOR $a IN $b/author RETURN
       IF ($a/data() != $s) THEN true ELSE ()
}
```

Then the following query can be expressed using the above function:

Query 2.6: Find all books that do not have "Suciu" as an author.

```
  For $b IN $bib/book RETURN
     IF notauthor("Suciu", $b) THEN $b ELSE ()
```

Chapter 3

ToX Query Processor

3.1 ToX Project Overview

The ToX (Toronto XML) project [3] is a work in progress in the Department of Computer Science at the University of Toronto. The purpose of this project is to develop a repository for XML data and tools to manage the data. The ToX engine can manage the XML documents that may be stored in a file system, a relational database, or on the Web. ToX clients can transparently register and remove documents, query document contents or structure, and define logical views of distributed data sources.

The ToX architecture is shown in Figure 3.1. The main components of the ToX engine are ToX Manager, Catalog Manager, Query Processor, Query Optimizer, Type Manager, Index Manager, Storage Manager, and Document Navigator. A brief description of each component is as follows:

- ToX Manager

  This module is visible to the user. It interacts with the Catalog Manager to store, retrieve or delete document collections and documents, and also interacts with the Query Processor to pose queries and display the query results to the user.

- Catalog Manager

This module is responsible for maintaining the metadata about the collections and documents, and manipulating the metadata and the XML documents, so it interacts with the ToX Manager, the Storage Optimizer, the Index Manager, and the Document Navigator. It also interacts with the Type Manager for schema checking.

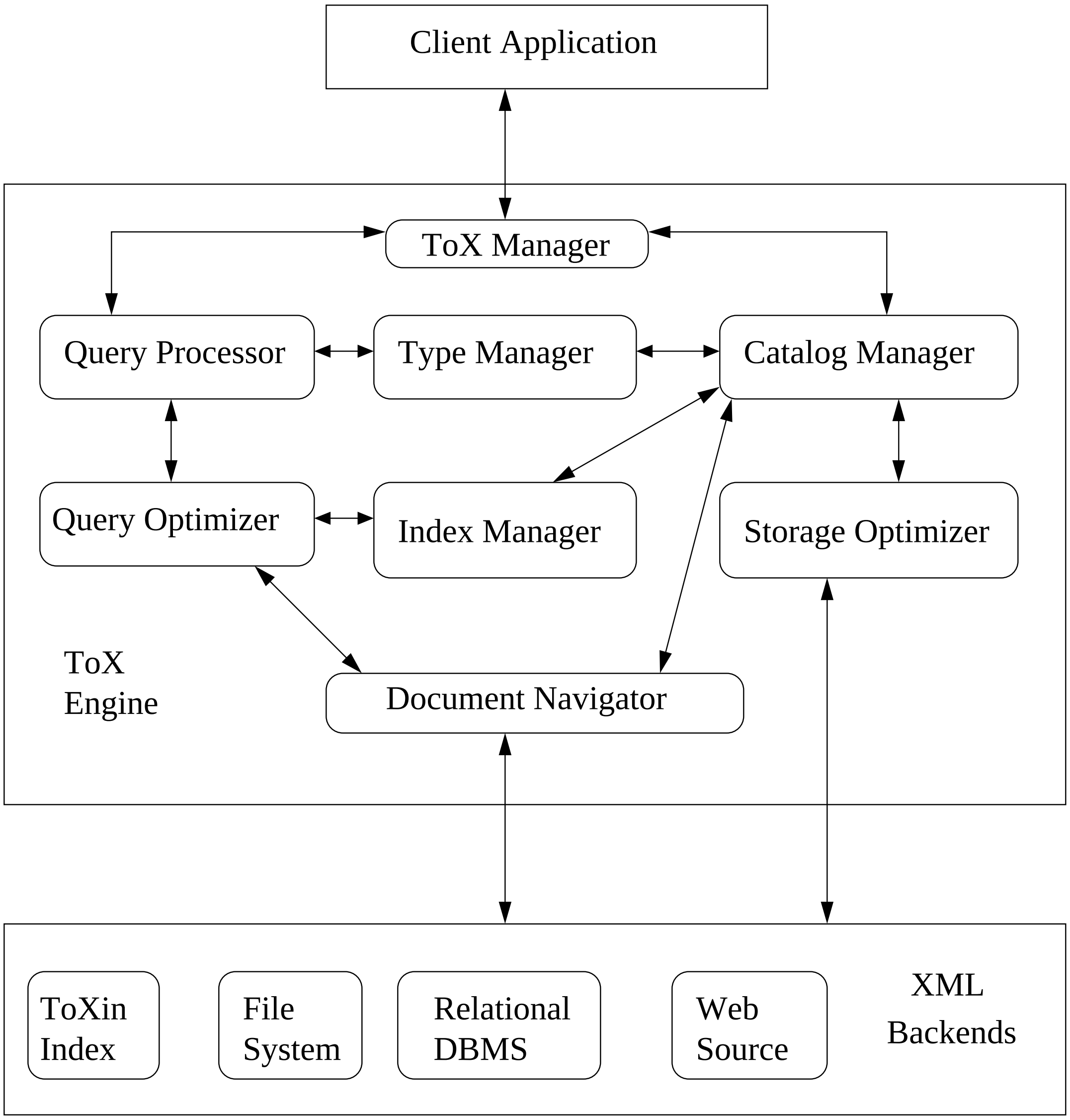

Figure 3.1: ToX Architecture

- Query Processor

  This module is responsible for query evaluation. It gets the query from the ToX

Manager, and returns the result to the ToX Manager. During query processing, it interacts with the Type Manager to perform type checking and type inference, and interacts with the query optimizer to optimize the query. It can also directly interact with the Document Navigator for query evaluation.

- Query Optimizer

  This module performs query optimization.

- Type Manager

  This module performs type-related tasks such as type checking and type inference for the query processor and the catalog manager.

- Storage Optimizer

  This module decides the optimum storage method for the document and accesses the storage. It interacts with the Catalog Manager and the XML backends.

- Index Manager

  This module manages the indices of collections and documents. It interacts with the Query Optimizer and the Catalog Manager.

- Document Navigator

  This module interacts with the XML backends to perform navigation tasks. It provides an interface for other modules in the ToX engine, so that the backends can be transparent to other modules.

An XML document can be stored as a text file in a file system, or be mapped into a relational database. According to different storage, ToX supports several backends, such as DOM and relational backends, as well as ToXin [30], which is an index mechanism for fast access to the elements in the XML tree.

3.2 Query Processor in ToX

3.2.1 Query Processor Architecture

The query processor is an important part of the ToX engine. Its architecture is shown in Figure 3.2. The main components of the query processor are the parser, the semantic

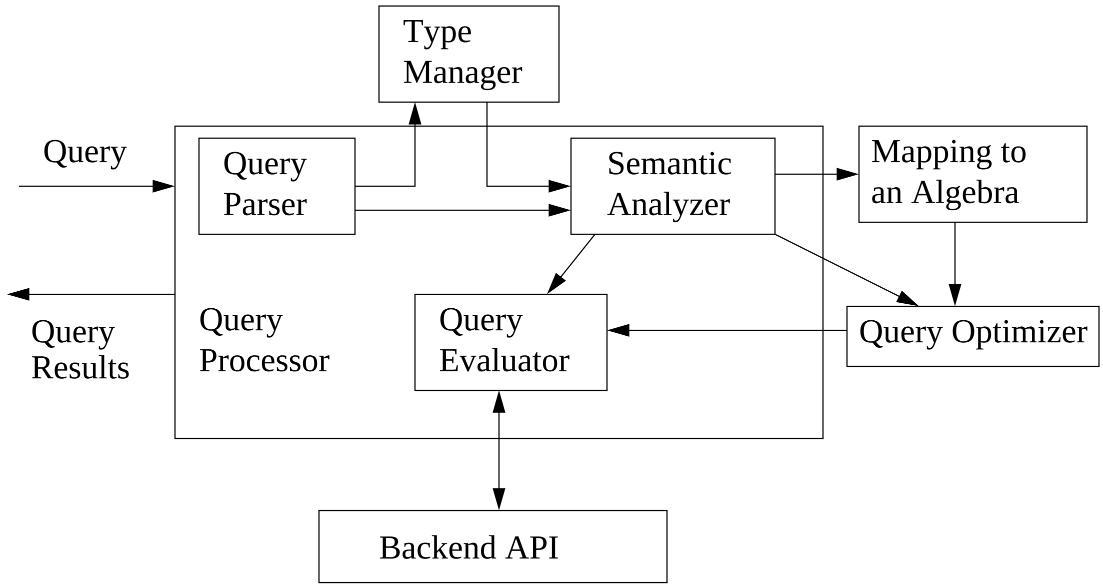

Figure 3.2: Query Processor Architecture

analyzer, and the evaluator. The input to the query processor is the query, and the output from the query processor is the query result expressed as an XML document. After the semantic checking of the query, the query can optionally be translated into an algebra for further optimization. My work is the query parser, the semantic analyzer, and the query evaluator. The type manager is the work of Karima Echihabi [21], and the mapping to an algebra is the work of Edward Xia [35].

The backend API (See Appendix E) provides an interface between the query evaluator and the backend implementation. Thus, whether the XML data is represented by a DOM tree, or a ToXin index [30], or a relational database, is transparent to the query processor. The query evaluator accesses the data through the functions of the API.

3.2.2 Query Language

Any query over the database needs to be expressed by a query language. The query language used in ToX is the XQuery Core (described in Chapter 2), a working draft by the W3C XML Query working group. XQuery is meant to be a standardized query language for XML data, and XQuery Core is a relatively small, but fully expressive sub-language of it. Thus, XQuery Core is relatively easier to implement and still conforms to the potential standard.

The XQuery Formal Semantics document [17] describes the grammar of XQuery Core's expressions and types. However, we made the following modifications to it:

(1) The rule in XQuery Core $Expr ::= ExprSequence$ is changed to $Expr ::=$ "(" [$ExprSequence$] ")", where $ExprSequence ::= Expr$ ("," $Expr$)*. Parentheses are added to indicate grouping, which is common and useful in expressions. Furthermore, the empty sequence is expressed as () and added to the set of expressions.

(2) The abbreviated path expression is added to the XQuery Core expressions. The XQuery Core does not include path expressions, but uses a combination of `children` and `descendant-or-self` functions, iterations and TYPESWITCH expressions to perform navigation of the document, as we show in Section 2.3. There are some problems with this approach. One is that a simple path expression can become very complicated in XQuery Core because the equivalent expressions usually involve nested loops. Another problem is the use of the TYPESWITCH expression. Since XML data may not necessarily have a schema or a DTD, the type information required in the TYPESWITCH expression may not be always known. In addition, to process the TYPESWITCH expression, subtyping and type inference are needed, and these operations may add complexity to the query processor. Besides, the Document Navigator in ToX provides an API that can handle path expressions. Therefore, in order to take advantage of the Document Navigator's API and to simplify the query, we added abbreviated path expressions to the XQuery Core. The abbreviated path expression is based on the abbreviated syntax of XPath's

Location Path, but it does not include *predicate* which eliminates nodes that fail to satisfy a certain condition, since this type of predicate can be expressed by the *IF* expression of XQuery Core. The abbreviated path expression starts with an expression identifying a specific node or a sequence of nodes in the XML document and consists of a sequence of *steps*. Each *step* represents a movement through the document along a specified *axis*, either the *child* axis or the *descendant-or-self* axis as specified in the XPath. The *child* axis is denoted by / and specifies a parent-child relationship between two nodes, while the *descendant-or-self* axis is denoted by // and specifies an ancestor-descendant relationship between two nodes at any depth. The following are some examples:

- The XPath expression `$a/child::book` selects the `book` elements of the context node associated with `$a`. In the abbreviated form, it is expressed as `$a/book`.

- The XPath expression `$a/descendant-or-self::author` selects the `author` descendants of the context node associated with `$a`, and if the context node is `author`, the context node as well. In the abbreviated form, it is expressed as `$a//author`.

- The XPath expression `$a/child::chapter[child::title]` selects the `chapter` children of the context node (associated with `$a`) that has one or more `title` children. The query has a predicate. The XQuery Core expression is:

  ```
  FOR $x IN $a/chapter RETURN
    IF ( count($x/title) != 0 ) THEN $x ELSE ()
  ```

- The XPath expression `$a/child::chapter[child::title='Introduction']` selects the `chapter` children of the context node (associated with `$a`) that has one or more `title` children with string value equals to `Introduction`. In XQuery Core with the abbreviated path expression, it is:

  ```
  FOR $x IN $a/chapter RETURN
    FOR $y IN $x/title RETURN
  ```

```
IF ( $y/data()= "Introduction" ) THEN $x ELSE ()
```

The `data()` keyword is used to access simple data (strings, numbers, or booleans).

The grammar defined by the XQuery Formal Semantics document is expressive and concise, but in order to implement it, operator precedence needs to be considered and ambiguity needs to be eliminated. The modified grammar is shown in Appendix A.

In XQuery Core, keywords such as `FOR` and `LET` are case insensitive whereas identifiers such as `book` are case sensitive. And a query can contain a comment, which is ignored by the query processor. A comment starts with the symbol "#" and ends with the newline character.

3.2.3 Query Parsing

Query parsing is the first step of query processing. The parser takes the query as input, checks it syntactically, and generates an abstract syntax tree (AST) of the query as its output. Furthermore, the query processor can express the AST as an XML document. Parsing the query into an XML representation of the AST is useful under several circumstances:

- Querying queries

  If queries are expressed in XML format, then queries can be queried and can be transformed into new queries.

- Generating queries

  In some XML-oriented programming environment, XML tools can be used to build a query in its XML format, which may be more convenient than building a query in the corresponding XQuery or XQuery Core format.

- Reusing parser

  In heterogeneous data environment, various systems or programs may be used to execute a query. One parser can generate an XML representation of the query for all these systems.

- Embedding queries in XML

  If the query is in XML format, it can be embedded in an XML document directly.

XQueryX [14] is the XML syntax for XQuery 1.0. The ToX query processor uses XQuery Core, so our XML representation of the query, which is called XQCoreX, is different from XQueryX, but we try to be as close as possible. The DTD of XQCoreX is shown in Appendix B.

The following is an example query in our extended XQuery Core syntax (regarding the sample XML document in Chapter 1):

List all the books published after 1991, including their year and title.

```
LET $root:= document("file:bib.xml") RETURN
 <result> { FOR $a IN $root/book RETURN
      IF ($a/attribute year/data()>1991 ) THEN
          <book>{ $a/attribute year, $a/title }</book>
      ELSE ()     } </result>
```

The following is the query's XML representation.

```
<Start>
 <QueryModule>
  <ExprSequence>
    <LetExpr variable="$root">
      <Function name="document">
        <ExprSequence>
          <StringLiteral> "file:bib.xml"</StringLiteral>
        </ExprSequence>
      </Function>
      <Return>
        <ElementConstructor>
          <TagName>
            <QName><NCName> result</NCName></QName></TagName>
          <ExprSequence>
```

```
      <ForExpr variable="$a">
        <Step axis="CHILD" >
          <Variable>$root</Variable>
          <NameSet><LocalName> book</LocalName></NameSet>
        </Step>
        <Return>
          <IfThenElseExpr>
            <Operator name="gt" >
              <Step axis="CHILD" >
                <Step axis="CHILD" >
                  <Variable>$a</Variable>
                  <Attribute>
                    <NameSet><LocalName> year</LocalName></NameSet>
                  </Attribute>
                </Step><Data/>
              </Step>
              <IntLiteral> 1991</IntLiteral>
            </Operator>
            <ElementConstructor>
                <TagName>
                  <QName><NCName> book</NCName></QName></TagName>
                <ExprSequence>
                  <Step axis="CHILD" >
                    <Variable>$a</Variable>
                    <Attribute>
                      <NameSet><LocalName> year</LocalName></NameSet>
                    </Attribute>
                  </Step>
                  <Step axis="CHILD" >
                    <Variable>$a</Variable>
                    <NameSet><LocalName> title</LocalName></NameSet>
                  </Step>
                </ExprSequence>
                <EndTagName>
                  <QName><NCName> book</NCName></QName></EndTagName>
            </ElementConstructor>
            <EmptySequence/>
          </IfThenElseExpr>
        </Return>
      </ForExpr>
    </ExprSequence>
    <EndTagName>
      <QName><NCName> result</NCName></QName></EndTagName>
  </ElementConstructor>
</Return>
```

```
      </LetExpr>
    </ExprSequence>
  </QueryModule>
</Start>
```

3.2.4 Semantic Checking

The purpose of (static) semantic checking is to detect semantic errors in the query as soon as possible, before query execution. It is always desirable for errors to be detected at the query-analysis time since computational resources can be saved. Semantic checking mainly consists of two parts: type checking and identifier checking.

Type Checking

Unlike the relational or object-oriented data models, XML data does not require a prior schema. However, if such a schema is available, it can be used to express types of XML data, since the type system of XQuery and XQuery Core are based on XML Schema. The XQuery type system takes as given the *built-in simple data types* from XML Schema Part 2 [13]. In the sample XML document in Chapter 1, the string content of the element *last* has the type of *xs:string*, and the element *last* itself has the type of *ELEMENT last (xs:string)*, where *xs* is the name prefix associated with the URL http://www.w3.org/2001/XMLSchema which describes the XML Schema namespace. Using the type declaration in XQuery Core, the sample XML data and schema can be expressed as the following:

```
NAMESPACE xs = "http://www.w3.org/2001/XMLSchema"
TYPE Author = ELEMENT author ( ELEMENT last (xs:string),
                                ELEMENT first (xs:string))
TYPE Editor = ELEMENT editor ( ELEMENT last (xs:string),
                                ELEMENT first (xs:string))
```

```
TYPE Book = ELEMENT book ( ATTRIBUTE year (xs:string),
                           ELEMENT title (xs:string),
                           (Author+ | Editor+),
                           ELEMENT publisher (xs:string),
                           ELEMENT price (xs:string))
TYPE Bib = ELEMENT bib (Book*)
```

In the context of query processing, the structure information or the types of data can be useful for:

- Detecting certain errors at the query-analysis time

  Given the types of input values, a query applied on these values, and the expected type of the query's output value, the type system can detect at the query-analysis time if the query's output value has the expected output type. For example, in a simple expression `IF ($a/book/author = "John") THEN $a/book/title ELSE ()`, the path expression `$a/book/author` has an ELEMENT type which cannot be used for the equal operator ("=" is used to compare simple values such as string, numbers or boolean values), so there should be an error and this error can be detected at the query-analysis time.

- Optimizing query evaluation

  Knowing the schema of data or types can guide the query processor to search only a small fragment of the data. For example, considering the following query:

  ```
  FOR $a IN $root//author RETURN
      IF ($a/last/data() = "Smith") THEN parent($a) ELSE ()
  ```

  Suppose that `$root` refers to the root element `bib`, and in addition to the `book` element, `bib` also has some other subelements such as `journal`, but only the `book` element contains the `author` element. Using // in the query will result in searching all the children of the `bib` element, and recursively scanning all their subelements to search for the `author` elements. Type information can help to restrict the search, since the query can be rewritten by an optimizer as:

```
FOR $a IN $root/book/author RETURN
    IF ($a/last/data() = "Smith") THEN parent($a) ELSE ()
```

Typical type checking tasks (static) include the following:

- Checking the types of the operands of operators. For unary operators, the operand should be of the correct type. For some binary operators, the types of the operands should be the same and of the correct type. For example, the operands of boolean operators AND and OR should be of the simple type xs:boolean.

- Checking the IF expression. The expression following IF should have the type xs:boolean.

- Checking the static type casting expression. In the expression *Expr1 : Type1*, the value of *Expr1* is statically cast to the given type *Type1*. The type of *Expr1* should be a subtype of *Type1*.

- Checking the types of functions. XQuery and XQuery Core allow users to define their own functions. The parameters' types are declared in the function definition. So when the function is called later, the types of the arguments should be the subtypes of the types of the corresponding parameters.

Type checking depends on type inference. Type inference is to derive the most specific type of an expression, according to some inference rules. For example, in the IF expression: *IF expr1 THEN expr2 ELSE expr3*, expr1 should have the type of xs:boolean. Suppose expr2 and expr3 have the types *t2* and *t3* respectively; then the resulting type of this IF expression is the choice $(t2|t3)$. The type inference rules for XQuery are presented in [17].

Identifier Checking

The identifier checking in the query processor is similar to that in a programming language, which is to use the scope rules to associate applied occurrences of identifiers (such as function calls or variables in an expression) with their declarations. If there is no corresponding declaration of an applied occurrence of an identifier, an error report should be generated. Each declaration has a *scope*, which is the portion of the program/query over which the declaration takes effect. Specifically for XQuery Core, the variable declaration may occur in the LET and FOR expressions. Consider the following query:

```
LET $a := document("http://abcc.com/a.xml") RETURN children($a)
```

The variable $a is declared before the RETURN clause, and its scope is the body of the RETURN clause. In XQuery Core, a variable cannot be declared more than once in its effective scope, and once it is defined, its value does not change.

The XQuery Core allows users to construct new elements and attributes. The name of a new element can be a qualified name, which is either a local name, or a namespace prefix joined to a local name by a colon. If the qualified name has a namespace prefix, then the prefix has to be declared by the context declaration at the beginning of the query. And the start and end tag names have to be the same. For the attributes, an element cannot have two attributes with the same name.

3.2.5 Query Optimization

After the query is checked against the syntax and static semantics, it can be optimized to obtain better performance. The general approach is query rewriting, taking into account data access structures (such as index) and knowledge about the data, the system, and the application.

First, the query can be translated into an algebraic form using an algebra. In the case of SQL to query relational data, relational algebra is used. As for XML data, the algebra has not been fully developed; however, there are some proposals such as SAL [4] and TAX [24]. For one query, there may be more than one logically equivalent expressions. These equivalent algebraic expressions of the query can be evaluated and the query can be rewritten in order to have a low cost. Even though the role of the algebra is to support the query optimization, it is optional, especially in the case of XML data since it is not well developed yet. In the absence of an algebra, query optimization depends on other rewriting rules, which is an on going research. Currently, the ToX query processor does not include the query optimizer.

3.2.6 Query Evaluation

Taking the original query or the optimized query, the query evaluator executes the query and returns the result. The Document Navigator in ToX provides an API (developed by Flavio Rizzolo and given in Appendix E) for the query evaluator to access the XML data. In this way, the implementation of the backends, i.e., whether it is a DOM tree or a relational database, is transparent to the query processor, which increases the modularity of the system and decreases the complexity of the query processor.

The evaluator can work in different ways. One way is that it traverses the AST of the query, and executes it directly. Another way is that the query processor traverses the AST of the query, translates the query into an intermediate representation, performs optimization, and then interprets the intermediate representation into another high level programming language such as Java. Finally, the interpreted program is executed and the query result is returned. Currently, the query processor can evaluate the query directly and give the result, and it also can translate the query into an intermediate representation; however, the translation from the intermediate representation to another program has not been implemented yet.

IRX (Intermediate Representation in XML)

For a programming language, the intermediate representation is a language between the source language and the machine code. For a query, it is an intermediate representation between the query language and the language that executes the query; it is like a query plan. This representation should have instructions with simple format which can be analyzed easily and quickly. Furthermore, translation from the original query into the intermediate representation should be easy and fast. Considering the above requirements and that XML is a data transformation standard, it is natural to want to use XML as the format for the intermediate representation of the instructions for query evaluation. If the query evaluation instructions can be expressed in XML format, then any XML enabled application can process them. Here we call our intermediate representation IRX (Intermediate Representation in XML).

The following are some examples of how the expressions in XQuery Core can be translated into IRX. For path expressions, the navigation tasks are represented by function calls to the Document Navigator's API. The more formal translation rules from XQuery Core expressions to IRX are given in Appendix F.

- Literals

  `"a"` is translated into: `<string>a</string>`

  `45.5` is translated into: `<number>45.5</number>`

  `true` is translated into: `<boolean>true</boolean>`

- Operators

  Postfix notation is used for operators.

  `-8` is translated into: `<number>8</number><uminus/>`

  `3+4/5` is translated into:

  ```
  <number>3</number>
  <number>4</number>
  <number>5</number>
  <divide/> <plus/>
  ```

- Variables

  Variables are associated with their scope to produce unique variable names for the query. So `$a` in scope 1 is translated into: `<variable>$a1</variable>`

- Path expressions

  Path expressions imply navigation through the document, so they are translated using functional calls to Document Navigator's API. The frequently used API functions are `getDescendants`, `getAttribute`, and `getValue`.

  `$root/book/title` is translated into:

```
<pathExpr>
  <startNode><variable>$root0</variable></startNode>
  <backendAPIcall name="getDescendants">
    <arguments><string>book.title</string></arguments>
  </backendAPIcall>
</pathExpr>
```

  `$root/book/attribute isbn` is translated into:

```
<pathExpr>
    <startNode><variable>$root0</variable></startNode>
    <backendAPIcall name="getDescendants">
      <arguments><string>book</string></arguments>
    </backendAPIcall>
    <iteration>
      <backendAPIcall name="getAttribute">
        <arguments><string>isbn</string></arguments>
      </backendAPIcall>
    </iteration>
</pathExpr>
```

  `$root/book/attribute isbn/data()` is translated into:

```
<pathExpr>
    <startNode><variable>$root0</variable></startNode>
    <backendAPIcall name="getDescendants">
      <arguments><string>book</string></arguments>
    </backendAPIcall>
    <iteration>
      <backendAPIcall name="getAttribute">
        <arguments><string>isbn</string></arguments>
      </backendAPIcall>
      <backendAPIcall name="getValue">
        <arguments></arguments>
```

```
        </backendAPIcall>
      </iteration>
  </pathExpr>
```

- IF expression

  `IF ($a>4) THEN $a-4 ELSE ()` is translated into:

```
<conditionalExpr>
  <condition>
    <variable>$a1</variable><number>4</number> <lt/>
  </condition>
  <trueBranch>
    <variable>$a1</variable><number>4</number> <minus/>
  </trueBranch>
  <falseBranch>
    <emptySequence/>
  </falseBranch>
</conditionalExpr>
```

- Function call

  `parent($z)` is translated into:

```
<funCall name="parent">
  <arguments>
    <variable>$z1</variable>
  </arguments>
</funCall>
```

- Expression sequence

  `3, 4, 5` is translated into:

```
<sequence>
  <number>3</number>
  <number>4</number>
  <number>5</number>
</sequence>
```

- Let expression

  `LET $x:=$root//book RETURN $x/year` is translated into:

```
<letExpr>
  <pathExpr>
    <startNode><variable>$root0</variable></startNode>
    <backendAPIcall name="getDescendants">
      <arguments><string>_*.book</string></arguments>
    </backendAPIcall>
  </pathExpr>
  <varBinding name="$x1"/>
  <pathExpr>
    <startNode><variable>$x1</variable></startNode>
    <backendAPIcall name="getDescendants">
      <arguments><string>year</string></arguments>
    </backendAPIcall>
  </pathExpr>
</letExpr>
```

- Element constructor

  `<book> {$root//book/title, $root//book/author}<book>` is translated into:

```
<elementConstructor>
  <backendAPIcall name="createElement">
    <arguments><string>book</string></arguments>
  </backendAPIcall>
  <sequence>
    <pathExpr>
      <startNode><variable>$root0</variable></startNode>
      <backendAPIcall nam="getDescendants">
        <arguments><string>_*.book.title</string></arguments>
      </backendAPIcall>
    </pathExpr>
    <pathExpr>
      <startNode><variable>$root0</variable></startNode>
      <backendAPIcall name="getDescendants">
        <arguments><string>_*.book.author</string></arguments>
      </backendAPIcall>
    </pathExpr>
  </sequence>
  <iteration>
      <backendAPIcall name="appendChild">
        <arguments></arguments>
      </backendAPIcall>
  </iteration>
</elementConstructor>
```

- Attribute constructor

  The following expression creates a new attribute for an element:

  `ATTRIBUTE isbn ("1990-29")`, and it can be translated into:

```
<attributeConstructor>
    <backendAPIcall name="createAttribute">
      <arguments><string>isbn</string></arguments>
    </backendAPIcall>
    <backendAPIcall name="setValue">
      <arguments><string>1990-29</string></arguments>
    </backendAPIcall>
</attributeConstructor>
```

- For expressions

  `FOR $p IN $root/book RETURN`

  `<book> {$p/title, $p/year} </book>`

  is translated into:

```
<forExpr>
  <pathExpr>
    <startNode><variable>$root0</variable></startNode>
    <backendAPIcall name="getDescendants">
      <arguments><string>book</string></arguments>
    </backendAPIcall>
  </pathExpr>
  <iteration>
    <varBinding name="$p1"/>
    <elementConstructor>
      <backendAPIcall name="createElement">
        <arguments><string>book</string></arguments>
      </backendAPIcall>
      <sequence>
        <pathExpr>
          <startNode><variable>$p1</variable></startNode>
          <backendAPIcall name="getDescendants">
            <arguments><string>title</string></arguments>
          </backendAPIcall>
        </pathExpr>
        <pathExpr>
          <startNode><variable>$p1< /variable></startNode>
          <backendAPIcall name="getDescendants">
```

```
              <arguments><string>author</string></arguments>
            </backendAPIcall>
          </pathExpr>
        </sequence>
        <iteration>
          <backendAPIcall name="appendChild">
            <arguments></arguments>
          </backendAPIcall>
        </iteration>
      </elementConstructor>
    </iteration>
  </forExpr>
```

Dynamic Semantics

The query processor basically follows the XQuery's dynamic semantics (operational semantics), which is defined in [17]. The dynamic semantics specifies the order in which an XQuery expression is evaluated and guarantees that every expression can be reduced to a semantics object. XQuery 1.0 and XPath 2.0 Data Model defines five types of semantics objects: error, simple values, sequences, and schema components. There is one error value called ERROR. Simple values are the union of all the value spaces of XML Schema simple types. A node is one of the following kinds: element, attribute, namespace, comment, processing-instruction, text, and reference. A sequence is an ordered collection of nodes, simple values, or any mixture of nodes and simple values, and a sequence cannot be a member of another sequence. There is no distinction between a unit value (i.e., a simple value or a node) and a singleton sequence containing that value.

Chapter 4

Query Processor Implementation and Results

4.1 Query Processor Implementation

The parser is generated by JavaCC [23], a Java parser generator which generates top-down parsers.

The Type Manager module provides an interface for the query processor. Type checking tasks are performed through this interface. The Type Manager is under development by Karima Echihabi [21].

The semantic analyzer can perform the semantic checking, as described in Chapter 3. However, type information is not always available for XML documents, so type checking is an option which can be turned off. And since the Type Manager has not been fully developed yet, some type checking functions are not available.

The evaluator now includes an interpreter that executes the query directly, and a translator which translates the query into IRX, described in Chapter 3.

The query processor can be run on the UNIX command line, or through the ToX engine. Through the ToX engine, the user can query not only the documents in the file system and remotely on the Web, but also the documents registered in the ToX repository. The query processor can query over one specific ToX document, or over a ToX collection.

XQuery Core supports many built-in functions. Currently, not all of the functions are implemented. The implemented features are listed in Appendix C.

4.2 Results

All the experiments in this section were run on a Sun SunBlade 1000 workstation (UltraSPARC3 600MHz CPUs) with 2048MB memory running Solaris 8, and the DOM backend was used.

4.2.1 Correctness

The following are some selected use cases from [11][1]. These queries cover projection, selection, join, the aggregation and other built-in functions, restructuring and sorting. The returned answers are the same as the expected results (They are shown in Figure 4.1 to Figure 4.5). The XML data and DTDs used for these queries are shown in Appendix D.

1. Q1 (1.1.9.3 in [11]): For each book in the bibliography, list the title and authors, grouped inside a "result" element.

   Solution in XQuery Core:

```
<results> {
    for $a in document("file:bib.xml")/book return
      <result> { $a/title, $a/author} </result>
```

[1]Since some features of XQuery Core have not been implemented yet, the query processor cannot perform all the use cases. We have tried most queries in the Use Case "XMP", "TREE", "SGML" and "REF".

```
    }
   </results>
```

2. Q2 (1.1.9.5 in [11]): For each book found at both bn.com and amazon.com, list the title of the book and its price from each source.

   Solution in XQuery Core:

```
<books-with-prices> {
  for $b in document("file:bib.xml")//book return
    for $a in document("file:reviews.xml")//entry return
      if ($b/title/data() = $a/title/data())
      then <book-with-prices> {
           $b/title,
           <price-amazon>$a/price/data()</price-amazon>,
           <price-bn>$b/price/data()</price-bn> }
          </book-with-prices>
      else ()
} </books-with-prices>
```

3. Q3 (1.1.9.7 in [11]): List the titles and years of all books published by Addison-Wesley after 1991, in alphabetic order.

   Solution in XQuery Core:

```
<bib> { ( for $b in document("file:bib.xml")/book return
      if ((string-value($b/publisher) = "Addison-Wesley") and
          (string-value($b/attribute year) > 1991) )
          then
            <book> { $b/attribute year, $b/title }</book>
          else ()
        ) sortby $dot/title/data() ascending
```

```
    }</bib>
```

4. Q4 (1.1.9.9 in [11]): In the document “books.xml”, find all section or chapter titles that contains the word “XML”, regardless of the level of nesting.

   Solution in XQuery Core:

```
let $root := document("file:books.xml") return
    <results>{
      for $t in $root//chapter/title union $root//section//title
      return
        if (contains(string-value($t), "XML")) then $t else ()
    }</results>
```

5. Q5 (1.1.9.10 in [11]): In the document “prices.xml”, find the minimum price for each book, in the form of a “minprice” element with the book title as its attribute.

   Solution in XQuery Core:

```
<results> {
    let $doc := document("file:prices.xml") return
      for $t in distinct-value($doc/book/title/data()) return
        let $q := ( for $p in $doc/book return
          if ($p/title/data() = $t) then $p/price/data() else ()
        )
        return
          <minprice>{attribute title ($t), minimum($q)}</minprice>
}</results>
```

4.2.2 Performance

In order to study the relation between the query processing performance and the size of XML data, and the relation between the query processing performance and the query structure, the following experiments were performed, in which we measured compilation time and execution time. The compilation time starts from parsing the query and ends with the completion of the translation to IRX. The execution time is the period of the query evaluation.

1. Query processing versus query size

   The compilation time depends on the size of the query, as we expected. Figure 4.6 shows how the compilation time increases with the size of the query. The queries are based on the XML data in Appendix D.

2. Query processing versus data size

   The XML data used is the courses in the University of Washington (www.cs.washington.edu/homes/anhai/lsd/lsd.html) and the DTD is as follows, provided by Marcelo Arenas.

   <!ELEMENT courses (course*)>

   <!ELEMENT course (code,title,section*)>

   <!ELEMENT code (#PCDATA)>

   <!ELEMENT title (#PCDATA)>

   <!ELEMENT section (schedule_line,section_code,credits,session*,restrictions?)>

   <!ELEMENT schedule_line (#PCDATA)>

   <!ELEMENT section_code (#PCDATA)>

   <!ELEMENT credits (#PCDATA)>

   <!ELEMENT session (days,time,place,instructor)>

   <!ELEMENT days (#PCDATA)>

   <!ELEMENT time (start_time,end_time)>

```
<!ELEMENT start_time (#PCDATA)>
<!ELEMENT end_time (#PCDATA)>
<!ELEMENT place (building,room)>
<!ELEMENT building (#PCDATA)>
<!ELEMENT room (#PCDATA)>
<!ELEMENT instructor (#PCDATA)>
<!ELEMENT restrictions (#PCDATA)>
```

The following query is used on the course data, and it was run on documents with different sizes but the same DTD.

Query 1: Find the courses on Wednesdays; return each such course's code as an attribute, together with the title, and the session information.

Solution in XQuery Core:

```
<courses-W> {
   for $co in document("file:courses1.xml")/course return
     for $a in $co//days return
       if (contains(string-value($a), "W" )) then
         <course> { attribute code ($co/code/data()),
           $co/title, parent($a) }
         </course>
       else ()
 }</courses-W>
```

The performance results with different document sizes are shown in Table 4.1. The compilation time is about 0.2s, and this does not change with the document size since the query is the same, while the execution time increases as the document size increases. This is what we expected. Figure 4.7 shows how the execution time increases with the document size.

	Document Size				
	0.115 MB	0.366 MB	1.087 MB	2.040 MB	3.069 MB
execution time (s)	3.6	12.9	80.1	263.3	546.3

Table 4.1: Execution time vs. document size for Query 1.

The following query is run on the DBLP document [27] provided by Flavio Rizzolo, and Figure 4.8 shows the execution time versus the document size.

Query 2: List the authors, each with the name, publication titles and year.

Solution in XQuery Core :

```
let $root := document("file:dblpf1.xml") return
  let $auth := $root//author return
    <authors> {
      for $a in distinct-value( $auth/data()) return
        <author>attribute name ($a),
          for $x in $auth return
            if ($x/data() = $a ) then
              <paper>{ parent($x)/title, parent($x)/year}
              </paper>
            else ()
    }</author>
</authors>
```

We can see that when the document size is increased to 1.2MB, the execution time becomes too long. So for Query 2, it is impractical to run it on a document larger than 1.2MB.

We performed some experiments to see how the time is spent on parsing the XML document, evaluating the path expression, and copying the nodes to form the new element in the element constructor. The following is a simple example query, and Table 4.2 shows the evaluation time on different tasks for different document sizes. We denote Time0 to be the time for evaluating the document function (parsing the XML document); Time1 to be the time for evaluating the path expression in the Element Constructor; and Time2 to be the time for copying nodes to construct the new element.

Example 1: `let $root:= document("file:courses.xml") return`

```
<results> { $root/course/title }</results>
```

	Document Size			
	0.366 MB	1.087 MB	1.577 MB	2.040 MB
Time0 (s)	1.06	1.85	2.47	2.94
Time1 (s)	0.24	0.64	1.01	1.40
Time2 (s)	0.04	0.09	0.13	0.23
Total execution time (s)	1.42	2.76	3.80	4.96

Table 4.2: Evaluation time for different tasks vs. document size (Example 1)

The following is another example showing the evaluation time on different tasks for different document sizes. Here Time0 and Time2 are defined as in the previous example, Time1 is the time for evaluating the path expression in the FOR expression, and TimeI is the time iterating over the RETURN clause in the FOR expression.

Example 2: `let $root:= document("file:courses.xml") return`

```
<results> {
  for $x in $root/course return $x/title
```

```
}</results>
```

	Document Size			
	0.366 MB	1.087 MB	1.577 MB	2.040 MB
Time0 (s)	1.06	1.85	2.47	2.94
Time1 (s)	0.16	0.35	0.59	0.84
TimeI (s)	1.06	1.26	2.02	2.94
Time2 (s)	0.04	0.10	0.14	0.23
Total execution time (s)	2.46	3.75	5.54	7.33

Table 4.3: Evaluation time for different tasks vs. document size

We found that copying takes a relatively small amount of time. The execution time is mainly spent on parsing the document, evaluating the path expression, and the iterations. If the query is fixed, as the size of the document increases, the time spent on parsing the document increases, and so does the time to evaluate the path expressions. If there are FOR loops, since there are more iterations (more nodes are returned) for a larger document, the time spent on a FOR loop will increase. So the total time will increase significantly. Figure 4.9 shows how the time spent on different tasks (Example 1) changes with the document size. This figure shows that for a simple query like Example 1, as the document size increases, the execution time does not increase as dramatically as for a complex query. This is because a complex query usually involves nested for loops, which can be very costly for a large document.

3. Query processing versus query structure

   The query processing performance largely depends on how the query is written, i.e., the structure of the query. The following experiments illustrate how the query structure affects the performance.

In Query 1, the path expression "$co//days" is used. Query 3 is the same as Query 1, except that it uses the full path "$co/section/session/days" instead of "$co//days". Table 4.4 is the performance comparison of these two queries:

	Document Size				
	0.115 MB	0.366 MB	1.087 MB	2.040 MB	3.069 MB
Query 1 (s)	3.6	12.9	80.1	263.3	546.3
Query 3 (s)	2.1	7.7	35.3	113.5	221.3

Table 4.4: Execution time vs. document size for Query 1 and Query 3

The compilation time of these two queries is about the same, 0.2s. The execution time shows a big difference between the two queries. The results show that, though it is convenient to use "//", it is an expensive operation as far as the execution time is concerned, especially when the document has a large size. The execution time saved by not using "//" increases as the document size increases.

The following are three queries, which produce the same results (all the `title` elements followed by all the `code` elements of the courses data). The first query uses the document function repeatedly while the others use the LET expression so that the document function is only used once; the first two queries use path expressions only while the third one uses a combination of FOR expressions and path expressions.

Query 4:

```
<courses> {
    document("file:courses1.xml")/course/title,
    document("file:courses1.xml")/course/code
```

```
    }</courses>
```

Query 5:

```
let $root := document("file:courses1.xml") return
  <courses> {
     $root/course/title,$root/course/code
  }</courses>
```

Query 6:

```
let $root := document("file:courses1.xml") return
  <courses> {
     (for $a in $root/course return $a/title),
     (for $b in $root/course return $b/code)
  }</courses>
```

The following tables are the performance comparisons of the above queries on documents of different sizes. As the results show, when the document is not very large, there is not a large difference in the execution time between the three queries; however, when the document size increases, the difference in the execution time among the queries increases. The most economic way to write this query is to use the full path expression and only use the `document` function once. We also can see that the compilation time is relatively very small compared to the execution time, especially when the document is large. Therefore, if a query can be written to save execution time but look complex, it is all right since the total time (compilation plus execution) can be reduced.

Mapping a query into an algebra can help with query optimization, and hence greatly improve the performance of query processing, as demonstrated in Edward Xia's work [35]. In [35], an XQuery Core query is first mapped into the modified (and extended) Beeri algebra, and then several optimization techniques are employed to improve the

	Document Size				
	0.115 MB	0.366 MB	1.087 MB	2.040 MB	3.069 MB
Query 4 (s)	0.14	0.14	0.14	0.14	0.14
Query 5 (s)	0.15	0.15	0.15	0.15	0.15
Query 6 (s)	0.17	0.17	0.17	0.17	0.17

Table 4.5: Compilation time vs. document size for Query 4,5,6.

	Document Size				
	0.115 MB	0.366 MB	1.087 MB	2.040 MB	3.069 MB
Query 4 (s)	1.2	2.3	5.2	9.8	14.6
Query 5 (s)	1.0	1.7	3.5	6.7	9.8
Query 6 (s)	1.2	3.1	7.5	13.3	18.6

Table 4.6: Execution time vs. document size for Query 4,5,6.

performance of the query evaluation. The experiments in [35] showed that as the document size increases, the query evaluation time increases sharply for XQuery Core queries without algebra mapping, which is in accordance with our results; however, the algebra and some optimization techniques greatly reduced the query evaluation time, and the time increase versus the document size is much more gradual than that of XQuery Core queries without algebra mapping.

```
<results>
  <result>
   <title >TCP/IP Illustrated</title>
   <author>
     <last>Stevens</last>
     <first>W.</first>
   </author>
  </result>
  <result>
    <title>Advanced Programming in the Unix environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
  <result>
  <result>
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
  </result>
  <result>
    <title>Tne Economics of Technology and Content for Digital TV</title>
  </result>
</results>
```

Figure 4.1: Result of Q1.

```
<books-with-prices>
  <book-with-prices>
    <title>TCP/IP Illustrated</title>
    <price-amazon>65.95</price-amazon>
    <price-bn>65.95</price-bn>
   </book-with-prices>
  <book-with-prices>
    <title>Advanced Programming in the Unix environment</title>
    <price-amazon>65.95</price-amazon>
    <price-bn>65.95</price-bn>
  </book-with-prices>
  <book-with-prices>
    <title>Data on the Web</title>
    <price-amazon>34.95</price-amazon>
    <price-bn>39.95</price-bn>
  </book-with-prices>
</books-with-prices>
```

Figure 4.2: Result of Q2.

```
<bib>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
  </book>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
  </book>
</bib>
```

Figure 4.3: Result of Q3.

```
<results>
  <title>XML</title>
  <title>XML and Semistructured Data</title>
</results>
```

Figure 4.4: Result of Q4.

```
<results>
   <minprice title="Advanced Programming in the Unix environment">
      <price>65.95</price>
   </minprice>
   <minprice title="TCP/IP Illustrated">
      <price>65.95</price>
   </minprice>
   <minprice title="Data on the Web">
      <price>34.95</price>
   </minprice>
<results>
```

Figure 4.5: Result of Q5.

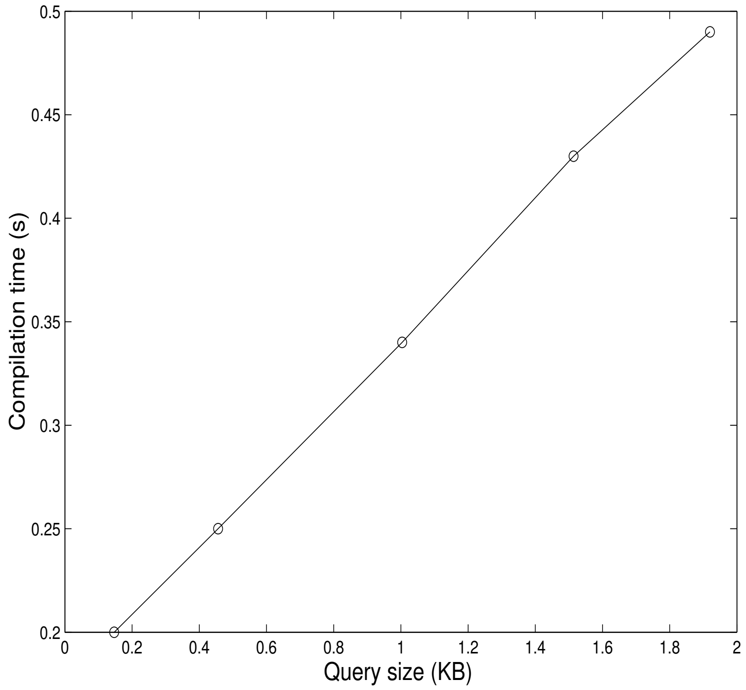

Figure 4.6: Compilation time vs. Query size

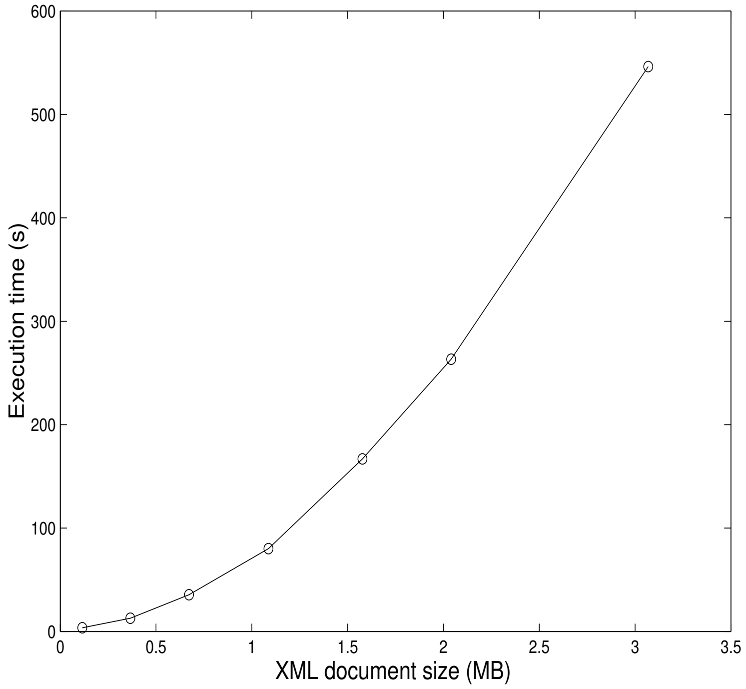

Figure 4.7: Query execution time vs. XML document size for Query 1

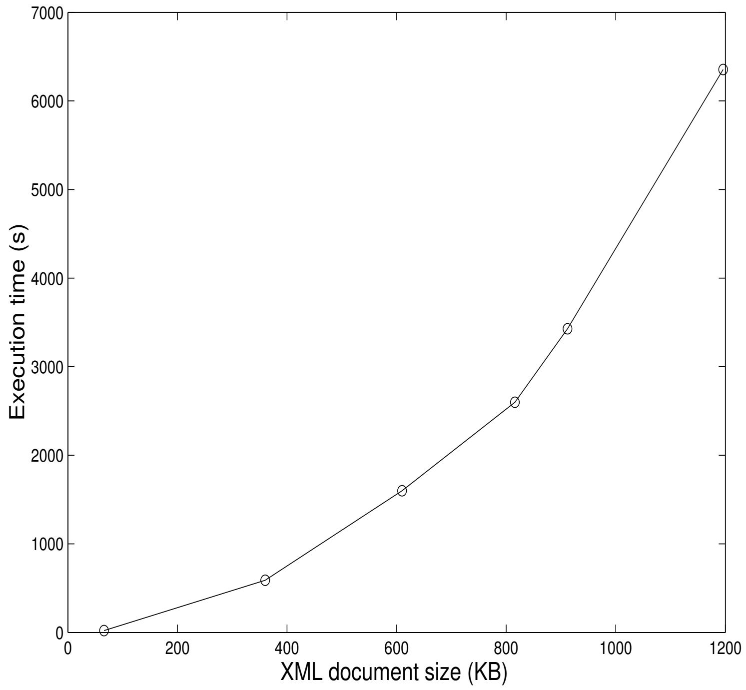

Figure 4.8: Query execution time vs. XML document size for Query 2

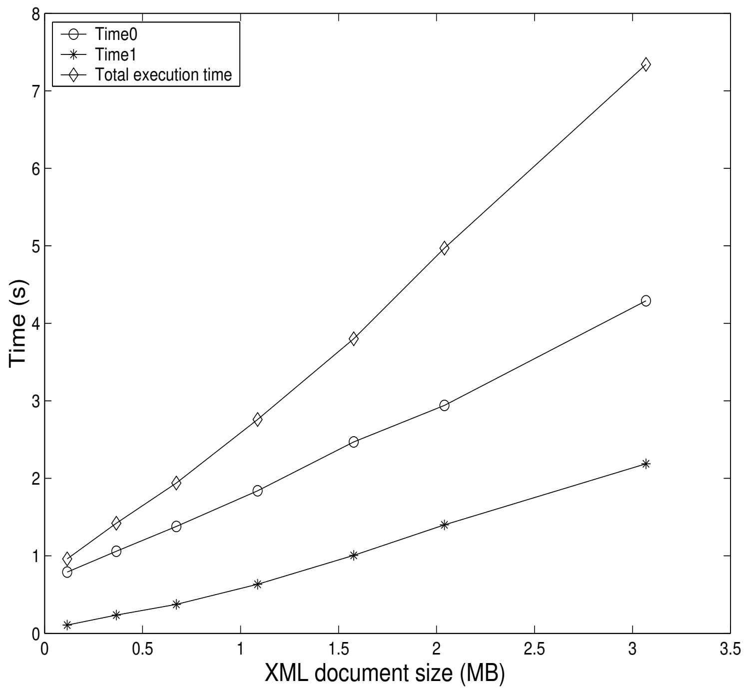

Figure 4.9: Time spent on different tasks (Example 1) vs. XML document size

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The query processor is an important part of any data management system. For XML documents, XQuery is a potential standard query language and XQuery Core is a fully expressive sublanguage of it. XQuery Core with extended features has been chosen as our query language for the query processor to query XML documents. The query processor has been developed such that it can be used as a stand-alone application or as part of the ToX engine. Through a common API, it can process queries over documents stored in the file system, or a relational database, or the ToX collection, or on the Web. The experimental results have shown that the query processing performance largely depends on the document size and the way the query is written. The compilation time is relatively insignificant compared to the execution time for large documents; thus it is important and beneficial to capture errors at the compilation stage rather than at run time.

5.2 Future Work

The following is the potential future work for the development of the query processor.

- Implementing more built-in functions

  Currently, some built-in functions such as `index`, `ref`, `dereference`, and `comment` are not implemented yet. Completing the built-in function library is future work.

- Adding user defined functions

  Currently, the query processor can allow the user to define his own functions, but it just supports the function calls to the built-in functions. Adding the function calls to the user defined functions will make the query processor more flexible and powerful.

- Implementing namespace features

  The namespace declaration declares the namespace prefix associated with a URI, and the namespace prefix can then be used in the tag names. The query processor now just treats the prefix as part of a tag name, but it seems more actions should be taken to include the semantics of the namespace, which may need more research.

- Code generation

  The query processor generates the intermediate representation of the query, the next step would be to translate the intermediate representation into a program in another language, such as Java, and then this program can be compiled and executed to get the query results.

- Developing an automatic tool for translating type information

  Now the type information is input to XQuery Core by the type declaration, which is not convenient if the document has long and complex schema. So a tool to transform the document schema into the format of the type declaration of XQuery Core or directly into the type nodes representing the types will be useful.

Appendix A

Grammar of the modified XQuery Core

NON-TERMINALS:

QueryStart	::=	Query <EOF>
Query	::=	QueryModule (<SEMICOLON> QueryModule)*
QueryModule	::=	(ContextDecl)* (TypeDecl)* (ExprSequence)?
ContextDecl	::=	<NAMESPACE> NCName <EQ> <STRING_LITERAL>
	|	<DEFA> <NAMESPACE> <EQ> <STRING_LITERAL>
NCName	::=	<ID>
TypeDecl	::=	<TYPE> <ID> <EQ> Type

```
NameSet       ::=  NameSetUnit ( ( <OR> NameSetUnit | <DIFF> NameSetUnit ))*
NameSetUnit   ::=  ( NamePrefix <COLON> )? LocalName
               |   "(" NameSet")"
LocalName     ::=  <ID> | <STAR>
NamePrefix    ::=  <STAR> | <ID> | "{" StringL "}"
Type          ::=  Type1 ( <COMMA> Type1 )*
Type1         ::=  Type2 ( <INTERLEAVE> Type2 )*
Type2         ::=  Type3 ( <CHOICE> Type3 )*
Type3         ::=  Type4 ( ( ( <MIN> ( LBound | <STAR> ) <MAX>
                   ( UBound1 | <STAR> ) ) | <STAR> | <PLUS>|
                   <QUESTIONMARK> ) )?
LBound        ::=  <INTEGER_LITERAL>
UBound1       ::=  <INTEGER_LITERAL>
Type4         ::=  <NULL>
               |   <EMPTYSEQ>
               |   <PIC> "(" Type ")"
               |   UnitType
               |   "(" Type ")"
UnitType      ::=  <ELEMENT> NameSet "(" Type ")"
               |   <ATTRIBUTE> NameSet "(" Type ")"
               |   <COMMENT>
               |   <PI>
               |   <REFERENCE> "(" Type ")"
               |   AtomicType
               |   NCName
```

AtomicType	::=	<INTEGER> | <STRING> | <FLOAT> | <BOOLEAN> | <URI> |
		<S> | <UID>|<IDREF> | <TIME> | <ANYSIMPLETYPE>
		| <ANYCOMPLEXTYPE>
Expr	::=	Expr1 ((<SORTBY> sortField | <COLON>Type))?
	|	<ERROR>
sortField	::=	Expr1 (<ASCENDING>|<DESCENDING>)
Expr1	::=	Expr2 (<OR> Expr2)*
Expr2	::=	Expr3 (<AND> Expr3)*
Expr3	::=	Expr4 | SpecialExpr
SpecialExpr	::=	ForExpr
	|	IfThenElseExpr
	|	LetExpr
	|	TypeSwitchExpr
	|	CastExpr
Expr4	::=	Expr5 ((<EQ> Expr5 | <NODEEQ> Expr5 | <NEQ> Expr5))*
Expr5	::=	Expr6 ((<LT> Expr6 | <LTE> Expr6 |<GT> Expr6 |
		<GTE>Expr6))*
Expr6	::=	Expr7 ((<PLUS> Expr7 | <MINUS> Expr7))*
Expr7	::=	Expr8 ((<STAR> Expr8 | <DIV> Expr8 | <MOD> Expr8))*
Expr8	::=	(<NOT> | <PLUS> | <MINUS>) Expr9
	|	Expr9
Expr9	::=	Expr10 (<UNION> Expr10)*
Expr10	::=	Expr11 ((<INTERSECT> Expr11 | <EXCEPT> Expr11))*
Expr11	::=	BasicExpr ((<SLASH> | <SLASHSLASH>) AtomicStep)*
AtomicStep	::=	<DATA> | NameSetUnit | <ATTRIBUTE> NameSetUnit

BasicExpr ::= <STRING_LITERAL>

| <INTEGER_LITERAL>

| <FLOATING_POINT_LITERAL>

| <TRUE>

| <FALSE>

| <ATTRIBUTE> QName “(” Expr “)”

| ElementConstructor

| Function

| Variable

| <EMPTYSEQ>

| “(” ExprSequence “)”

Function ::= FuncName (<EMPTYSEQ> | “(” ExprSequence “)”)

FuncName ::= QName

ElementConstructor ::= StartTag (“/>” | (<GT> “{”ExprSequence“}” EndTag))

StartTag ::= <LT> Tagname

EndTag ::= “</” Tagname <GT>

Tagname ::= QName | “{” Expr “}”

ExprSequence ::= Expr (<COMMA> Expr)*

Variable ::= <VAR>

QName ::= NCName (<COLON> NCName)?

ForExpr ::= <FOR> <VAR> <IN> Expr Return

Return ::= <RETURN> Expr

IfThenElseExpr ::= <IF> “(” Expr “)” <THEN> Expr <ELSE> Expr

LetExpr ::= <LET> <VAR> <ASSIGN> Expr Return

TypeSwitchExpr	::=	<TYPESWITCH> "(" Expr ")" <AS> <VAR> CaseRules
CaseRules	::=	<CASE> Type Return CaseRules
	|	<DEFA> Return
CastExpr	::=	<CAST> <AS> Type "(" Expr ")"

TERMINALS (ignore case)

<TYPE>	::=	"type"	<QUERY>	::=	"query"
<DEFINE>	::=	"define"	<FUNCTION>	::=	"function"
<LET>	::=	"let"	<IN>	::=	"in"
<FOR>	::=	"for"	<IF>	::=	"if"
<THEN>	::=	"then"	<ELSE>	::=	"else"
<RETURN>	::=	"return"	<RETURNS>	::=	"returns"
<DO>	::=	"do"	<MATCH>	::=	"match"
<CASE>	::=	"case"	<CAST>	::=	"cast"
<AS>	::=	"as"	<DEFA>	::=	"default"
<TYPESWITCH>	::=	"typeswitch"	<BY>	::=	"by"
<NAMESPACE>	::=	"namespace"	<SORTBY>	::=	"sortby"
<ASCENDING>	::=	"ascending"	<DESCENDING>	::=	"descending"
<ATTRIBUTE>	::=	"attribute"	<ELEMENT>	::=	"element"
<REFERENCE>	::=	"reference"	<COMMENT>	::=	"comment"
<NULL>	::=	"null"	<PIC>	::=	"pic"
<MIXED>	::=	"mixed"	<ERROR>	::=	"error"
<S>	::=	"xs:NCName"	<INTEGER>	::=	"xs:integer"
<STRING>	::=	"xs:string"	<FLOAT>	::=	"xs:float"

<BOOLEAN> ::= “xs:boolean” <URI> ::= “xs:uriReference”

<TIME> ::= “xs:time” <IDREF> ::= “xs:IDREF”

<UID> ::= “xs:ID” <NOT> ::= “not”

<OR> ::= “or” <AND> ::= “and”

<DIV> ::= “div” <MOD> ::= “mod”

<MAX> ::= “max” <MIN> ::= “min”

<TRUE> ::= “true” <FALSE> ::= “false”

<DIFF> ::= “diff” <UNION> ::= “union”

<INTERSECT> ::= “intersect” <EXCEPT> ::= “except”

<PI> ::= “processing-instruction”

<ANYSIMPLETYPE> ::= “xs:AnySimpleType”

<ANYCOMPLEXTYPE> ::= “xs:AnyComplexType”

<COMMA> ::= “,” <COLON> ::= “:”

<INTERLEAVE> ::= “&” <CHOICE> ::= “|”

<STAR> ::= “*” <QUESTIONMARK> ::= “?”

<SEMICOLON> ::= “;” <ASSIGN> ::= “:=”

<EQ> ::= “=” <NODEEQ> ::= “==”

<NEQ> ::= “!=” <LT> ::= “<”

<LTE> ::= “<=” <GTE> ::= “>=”

<GT> ::= “>” <PLUS> ::= “+”

<MINUS> ::= “-” <SLASH> ::= “/”

<SLASHSLASH> ::= “//” <EMPTYSEQ> ::= “(” “)”

<DATA> ::= “data()”

Appendix B

DTD for XQCoreX

<!ENTITY % Literals “StringLiteral | IntLiteral | FloatLiteral | BoolLiteral ”>

<!ENTITY % Expression “ %Literals; | Variable | ExprSequence | Attribute | ElementConstructor | Function | Step | ForExpr | IfThenElseExpr | LetExpr | TypeSwitchExpr | CastExpr | EmptySequence | Error | Operator | sortBy | TypeCasting ”>

<!ENTITY % type0 “ Type | Operator ”>

<!ELEMENT Start (QueryModule+)>

<!ELEMENT QueryModule ((ContextDeclaration*, TypeDeclaration*, FunctionDefinition*, ExprSequence?)+)>

<!ELEMENT ContextDeclaration (NCName?, StringLiteral)>

<!ELEMENT TypeDeclaration (%type0;)>

<!ATTLIST TypeDeclaration TypeName CDATA #REQUIRED>

<!ELEMENT FunctionDefinition (ParameterList?, (%type0;), (%Expression;))>

<!ATTLIST FunctionDefinition name CDATA #REQUIRED>

<!ELEMENT ExprSequence ((%Expression;)+)>

<!ELEMENT ParameterList (((%type0;) , Variable)+)>

<!ELEMENT NCName (#PCDATA) >

<!ELEMENT QName (NCName, NCName?) >

```
<!ELEMENT Type ( ElementType | AttributeType | %type0; | PIC | QName | Reference | AtomicType )>
<!ATTLIST Type min CDATA #IMPLIED
               max CDATA #IMPLIED >
<!ELEMENT PIC ( %type0;) >
<!ELEMENT Reference ( %type0;) >
<!ELEMENT AtomicType (#PCDATA) >
<!ELEMENT ElementType ( (NameSet | Operator), (%type0;)) >
<!ELEMENT AttributeType ( (NameSet | Operator), (%type0;)) >
<!ELEMENT NameSet ( (NamePrefix?, LocalName) | NameSet ) >
<!ELEMENT NamePrefix (#PCDATA) >
<!ELEMENT LocalName (#PCDATA)>
<!ELEMENT Variable (#PCDATA) >
<!ELEMENT sortBy ( (%Expression;) , sortField ) >
<!ELEMENT sortField ( %Expression;) >
<!ATTLIST sortField order CDATA #REQUIRED>
<!ELEMENT Error EMPTY >
<!ELEMENT EmptySequence EMPTY >
<!ELEMENT TypeCasting ( (%Expression;) , (%type0;) ) >
<!ELEMENT ForExpr ( (%Expression;) , Return ) >
<!ATTLIST ForExpr variable CDATA #REQUIRED>
<!ELEMENT Return ( %Expression; ) >
<!ELEMENT LetExpr ( (%Expression;), Return ) >
<!ATTLIST LetExpr variable CDATA #REQUIRED>
<!ELEMENT IfThenElseExpr ( (%Expression;), (%Expression;), (%Expression;) ) >
<!ELEMENT TypeSwitchExpr ( (%Expression;), Variable, Case ) >
<!ELEMENT Case ( ((%type0;) , Return, Case) | Return ) >
```

<!ELEMENT CastExpr ((%type0;) , (%Expression;)) >

<!ELEMENT ElementConstructor (TagName, (ExprSequence, EndTagName)?) >

<!ELEMENT TagName (QName | %Expression;) >

<!ELEMENT EndTagName (QName | %Expression;) >

<!ELEMENT Function (ExprSequence?) >

<!ATTLIST Function name CDATA #REQUIRED>

<!ELEMENT Attribute (NameSet | (QName, (%Expression;)))>

<!ELEMENT Step (((%Expression;) | Step), (NameSet | Attribute | Data))>

<!ATTLIST Step axis CDATA #REQUIRED>

<!ELEMENT Data EMPTY >

<!ELEMENT Operator (((Type | Operator), Type) | (((%Expression;) | Operator), (%Expression;)) | ((NameSet | Operator), NameSet))>

<!ATTLIST Operator name CDATA #REQUIRED>

<!ELEMENT StringLiteral (#PCDATA)>

<!ELEMENT IntLiteral (#PCDATA) >

<!ELEMENT FloatLiteral (#PCDATA)>

<!ELEMENT BoolLiteral (#PCDATA) >

Appendix C

Implemented Features of XQuery Core

The following operators and expressions have been implemented in ToX query processor: arithematic operators, boolean operators, collection operators, equality opertors, literals, variable, element constructor, attribute constructor, error, grouping of expression sequence, let expression, for expression, if-then-else expression, static type casting and sortby expression.

The following built-in functions have been implemented in ToX query processor.

Function Name	Precondition	Functionality
`attributes`(Expr)	Expr is evaluated to an element node	returns attributes of the element
`children`(Expr)	Expr is evaluated to a node	returns children of the node
`parent`(Expr)	Expr is evaluated to a node	returns the parent element of the given node
`name`(Expr)	Expr is evaluated to a node	returns the tagname of the given node
`descendant-or-self`(Expr)	Expr is evaluated to a node	returns the given node and its descendants in document order
`typed-value`(Expr)	Expr is evaluated to a node	returns the simple typed value of the given node
`string-value`(Expr)	Expr is evaluated to a node	returns the string-value* of the given node
`document`(Expr)	Expr is evaluated to a string**	returns the root node of the referenced document
`avg`(Expr)	Expr is evaluated to a sequence of numbers	returns the average value of the sequence
`sum`(Expr)	Expr is evaluated to a sequence of numbers	returns the summation of the sequence

Function Name	Precondition	Functionality
`minimum`(Expr)	Expr is evaluated to a sequence of numbers	returns the minimum value of the sequence
`maximum`(Expr)	Expr is evaluated to a sequence of numbers	returns the maximum value of the sequence
`count`(Expr)	none	returns the number of elements in the sequence
`distinct-node`(Expr)	none	returns the distinct nodes of the sequence
`distinct-value`(Expr)	none	removes duplicated values from a sequence
`lower-case`(Expr)	Expr is evaluated to a string	translates every upper-case letter to its lower-case correspondent
`contains`(Expr1, Expr2)	Expr1 and Expr2 are evaluated to strings	returns true if the string from Expr1 contains the string from Expr2

* The string-value of a node is the concatenation of the string-values of its descendants. If a node has a simple typed value, the simple typed value's string representation is its string-value.

** The valid string to specify the document location has the following syntax:

document-location ::= "tox:" collection-name ["/" document-name]
| "file:" filepath
| URL

where collection-name and document-name are the names of the ToX collection and document respectively, and filepath is the path to locate the file in the file system. The following are some valid document locations: tox:movies; tox:movies/movie; file:reviews.xml; http://www.abc.com/abc.xml.

The functions `document`, `contains` and `lower-case` are not in the W3C's XQuery Formal Semantics document.

Appendix D

DTD and Sample Data of the Use Cases

The DTD for the document bib.xml:

```
<!ELEMENT bib (book*)>
<!ELEMENT book (title, (aythor+|editor+), publisher, price )>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author ( last, first)>
<!ELEMENT editor ( last, first, affiliation)>
<!ELEMENT title (#PCDATA) >
<!ELEMENT last (#PCDATA) >
<!ELEMENT first (#PCDATA) >
<!ELEMENT affiliation (#PCDATA) >
<!ELEMENT publisher (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

The DTD for the document reviews.xml:

```
<!ELEMENT reviews (entry*) >
```

```
<!ELEMENT entry (title, price, review) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT price (#PCDATA) >
<!ELEMENT review (#PCDATA) >
```

The DTD for the document books.xml:

```
<!ELEMENT chapter (title, section*) >
<!ELEMENT section (title, section*) >
<!ELEMENT title (#PCDATA) >
```

The DTD for the document prices.xml:

```
<!ELEMENT prices (book*) >
<!ELEMENT book (title, source, price) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT source (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

```xml
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV </title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation></editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

Figure D.1: bib.xml.

```
<reviews>
  <entry>
    <title>Data on the Web</title>
    <price>34.95</price>
    <review>A very good discussion of semi-structured database systems and XML</review>
  </entry>
  <entry>
    <title>Advanced Programming in the Unix environment</title>
    <price>65.95</price>
    <review>A clear and detailed discussion of UNIX programming.</review>
  </entry>
  <entry>
    <title>TCP/IP Illustrated</title>
    <price>65.95</price>
    <review>One of the best books on TCP/IP.</review>
  </entry>
</reviews>
```

Figure D.2: reviews.xml.

```
<chapter>
  <title>Data Model</title>
  <section>
    <title>Syntax For Data Model</title>
  </section>
  <section>
    <title>XML</title>
    <section>
      <title>Basic Syntax</title>
    </section>
    <section>
      <title>XML and Semi-structured Data</title>
    </section>
  </section>
</chapter>
```

Figure D.3: books.xml.

```
<prices>
  <book>
    <title>Advanced Programming in the Unix environment</title>
    <source>www.amazon.com</source>
    <price>65.95</price>
  </book>
  <book>
    <title>Advanced Programming in the Unix environment</title>
    <source>www.bn.com</source>
    <price>65.95</price>
  </book>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.amazon.com</source>
    <price>65.95</price>
  </book>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.bn.com</source>
    <price>65.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.amazon.com</source>
    <price>34.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.bn.com<./source>
    <price>39.95</price>
  </book>
</prices>
```

Figure D.4: prices.xml.

Appendix E

The ToX Document Navigator API

The ToX Document Navigator (developed by Flavio Rizzolo) provides the functionalities to navigate the document and manipulate the nodes of the document.

1. Interfaces

   XAttribute, XDocument, XElement,XNode, XNodeList, XNodeMap

2. Interface Hierarchy

   ◇ interface java.util.Collection
      ◇ interface java.util.List
         ◇ interface xdoc.navigation.XNodeList
   ◇ interface java.util.Map
      ◇ interface xdoc.navigation.XNodeMap
   ◇ interface xdoc.navigation.XDocument
   ◇ interface xdoc.navigation.XNode
      ◇ interface xdoc.navigation.XAttribute
      ◇ interface xdoc.navigation.XElement

3. Interface XDocument

   This interface has the following methods:

- `XAttribute createAttribute(java.lang.String name)`

  Creates an XAttribute node of name `name` in the backend.

- `XElement createElement(java.lang.String name)`

  Creates an XElement node of name `name` in the backend.

- `XNode createNode(java.lang.String Name)`

  Creates an XElement node of name `name` in the backend.

- `XElement getRoot()`

  Returns the XNode root of this XDocument.

- `void setRoot(XElement xRoot)`

  Sets the XElement `xRoot` as root of this XDocument.

4. Interface XNode

   This interface has the following methods:

   - `void appendChild(XNode newChild)`

     Appends the XNode `newChild` to the end of the list of children of this node.

   - `boolean equals(XNode xnode)`

     Compares two XNode nodes and return true if they are equal.

   - `XNodeList getAncestors()`

     Returns an XNodeList that contains all ancestors of this XNode, not necessarily in order.

   - `XNodeList getAncestors(java.lang.String regEx)`

     Returns an XNodeList that contains all ancestors reached by paths from this XNode satisfying the regular expression `regEx`.

   - `XNodeList getChildren()`

     Returns an XNodeList that contains all children of this XNode, not necessarily in order.

   - `XNodeList getChildren(java.lang.String name)`

Returns an XNodeList that contains all children with name `name` of this XNode, not necessarily in order.

- `XNodeList getDescendants()`

  Returns an XNodeList that contains all descendants of this XNode, not necessarily in order.

- `XNodeList getDescendants(java.lang.String regEx)`

  Returns an XNodeList that contains all descendants reached by paths from this XNode satisfying the regular expression `regEx`.

- `XNodeList getFollowingSiblings()`

  Returns an XNodeList that contains all siblings following this XNode.

- `XNodeList getFollowingSiblings(java.lang.String name)`

  Returns an XNodeList that contains all siblings with name `name` following this XNode, not necessarily in order.

- `java.lang.String getName()`

  Returns the name of this XNode.

- `XNode getNextSibling()`

  Returns the XNode sibling immediately following this XNode.

- `XNode getNextSibling(java.lang.String name)`

  Returns the XNode sibling with name `name` immediately following this XNode.

- `XDocument getOwnerDocument()`

  Returns the XDocument object this node is associated with.

- `XNode getParent()`

  Returns the XNode parent of this XNode.

- `XNodeList getPrecedingSiblings()`

  Returns an XNodeList that contains all siblings preceding this XNode, not necessarily in order.

- `XNodeList getPrecedingSiblings(java.lang.String name)`

  Returns an XNodeList that contains all siblings with name `name` preceding this XNode, not necessarily in order.

- `XNode getPreviousSibling()`

  Returns the XNode sibling immediately preceding this XNode.

- `XNode getPreviousSibling(java.lang.String name)`

  Returns the XNode sibling with name `name` immediately preceding this XNode.

- `short getType()`

  Returns a code representing the type of the underlying XNode.

- `java.lang.String getValue()`

  Returns the value of this XNode.

- `boolean hasChildren()`

  Returns true if this XNode has children and false otherwise.

- `boolean hasNextSibling()`

  Returns true if the XNode has a next sibling and false otherwise.

- `boolean hasParent()`

  Returns true if this XNode has a parent and false otherwise.

- `boolean hasPreviousSibling()`

  Returns true if the XNode has a previous sibling and false otherwise.

- `void removeChild(XNode oldChild)`

  Removes the XNode `oldChild` from the list of children of this node.

- `void setValue(java.lang.String value)`

  Sets the value of this XNode.

5. Interface XAttribute

   This interface has the following method:

   - `XElement getOwnerElement()`

Returns the XElement node that this XAttribute is associated with.

6. Interface XElement

   This interface has the following methods:

   - `XAttribute getAttribute(java.lang.String attrName)`

     Retrieves an XAttribute node of this XElement by name.

   - `XNodeMap getAttributes()`

     Returns an XNodeMap containing the XAttribute nodes of this XElement.

   - `boolean hasAttribute(java.lang.String attrName)`

     Returns true if this XElement has an XAttribute of name `attrName` and false otherwise.

   - `boolean hasAttributes()`

     Returns true if this XElement has XAttributes and false otherwise.

   - `void removeAttribute(XAttribute oldAttribute)`

     Removes the XAttribute `oldAttribute` from the list of attributes of this XElement.

   - `void setAttribute(XAttribute newAttribute)`

     Adds the XAttribute `newAttribute` to the set of attributes of this XElement.

7. Interface XNodeList

   This interface has the following method and methods inherited from the interface java.util.List.

   - `XNodeList sort()`

     Returns this XNodeList sorted according to the preorder traversal of the XML tree.

8. Interface XNodeMap

   The interface provides the abstraction of an unordered collections of XNodes that can be accessed by name. The methods are inherited from java.util.Map.

Appendix F

Translation Rules of IRX

Let E denote an XQuery Core expression, and T(E) denote the translation from E into the IRX. For example, T(5) = <number>5</number> means that the IRX of the number 5 is <number>5</number>.

- Literal

  T(l) = <tag>l</tag> where l is a string, numeric or boolean value, and tag is one of `string`, `number` or `boolean` according to the type of the literal.

- Variable

  T(\$$v$) = <variable>\$vx</variable> where x is a postfix added to distinguish the variable names.

- ERROR

  T($ERROR$) = <error/>

- Function call

  Let f(E) be the function call where f is the name of the function.

  T(f(E)) = <funCall name="f"><arguments>T(E)</arguments></funCall>

- Expression sequence

  T($E1$, $E2$, ...,En) = <sequence> T($E1$) T($E2$) ... T(En) </sequence>

- Empty sequence

  T(()) = <emptySequence/>

- Operator

  Postfix notation is used here.

  T($E1$ *bop* $E2$) = T($E1$) T($E2$) <bop/>

  where *bop* is the name of the binary operator and one of the following: `plus`, `minus`, `multiply`, `divide`, `mod`, `lt`, `gt`, `lteq`, `gteq`, `and`, `or`, `equal`, `node-equal`, `not-equal`, `union`, and `intersect`.

  T($uopE1$) = T($E1$) <uop/>

  where *uop* is the unary operator name and one of `not` and `uminus`.

- IF expression

  T($IF\ (E1)\ THEN\ E2\ ELSE\ E3$) =

  <conditionalExpr>
  <condition>T($E1$)</condition>
  <trueBranch>T($E2$)</trueBranch>
  <falseBranch>T($E3$)</falseBranch>
  </conditionalExpr>

- LET expression

  T($LET\ \$v := E1\ RETURN\ E2$) =

  <letExpr> T($E1$) <varBinding name="$vx"/> T($E2$) </letExpr>

  where v is the name of the variable and x is a postfix added to distinguish variable names.

- Path expression

  A path expression always starts from a node. Let $E0$ be the expression for the starting node; then an path expression can be expressed as:

pathExpression ::= $E0$((/ | //)QName)+((/(ATTRIBUTE AttrName | data()))?

where QName and AttrName are the qualified names for the elements and the attribute respectively, + denotes one or more occurrences and ? denotes optional occurrence. A "path" is the regular expression of the sequence of the QNames and is used as the argument of a proper backend API function according to the type of path expression. If the path expression ends up with an element, such as $a/book, then

T($E0$ $((/\,|\,//)QName)+$) =

<pathExpr> <startNode> T($E0$) </startNode>

<backendAPIcall name="getDescendants">

<arguments><string>path</string></arguments>

</backendAPIcall>

</pathExpr>

If the path expression ends up with an attribute, such as $a/book/attribute year, then

T($E0((/\,|\,//)QName) + \;(/ATTRIBUTE\; AttrName)$) =

<pathExpr> <startNode> T($E0$) </startNode>

<backendAPIcall name="getDescendants">

<arguments><string>path</string></arguments>

</backendAPIcall>

<iteration>

<backendAPIcall name="getAttribute">

<arguments><string>AttrName</string></arguments>

</backendAPIcall>

</iteration>

</pathExpr>

If the path expression ends up with *data()* of an element, then

$\mathrm{T}(E0((/ \mid //)QName) + /data()) =$

<pathExpr> <startNode> T($E0$) </startNode>

<backendAPIcall name="getDescendants">

<arguments><string>path</string></arguments>

</backendAPIcall>

<iteration>

<backendAPIcall name="getValue">

<arguments></arguments>

</backendAPIcall>

</iteration>

</pathExpr>

If the path expression ends up with *data()* of an attribute, then

$\mathrm{T}(E0((/ \mid //)QName) + /ATTRIBUTE\ AttrName/data()) =$

<pathExpr> <startNode> T($E0$) </startNode>

<backendAPIcall name="getDescendants">

<arguments><string>path</string></arguments>

</backendAPIcall>

<iteration>

<backendAPIcall name="getAttribute">

<arguments><string>AttrName</string></arguments>

</backendAPIcall>

<backendAPIcall name="getValue">

<arguments></arguments>

</backendAPIcall>

</iteration>

</pathExpr>

- FOR expression

  T($FOR\ \$p\ IN\ E1\ RETURN\ E2$) =

  <forExpr> T($E1$)

  <iteration><varBinding name="$px"/>T($E2$)</iteration>

  </forExpr>

  where x is the postfix added to distinguish variable names.

- Element constructor

  T($< QName > \{E\} < /QName >$) =

  <elementConstructor>

  <backendAPIcall name="createElement">

  <arguments><string>QName</string></arguments>

  </backendAPIcall>

  T(E)

  <iteration>

  <backendAPIcall name="appendChild">

  <arguments></arguments>

  </backendAPIcall>

  </iteration>

  </elementConstructor>

- Attribute constructor

  T(ATTRIBUTE QName (E)) =

  <attributeConstructor>

  <backendAPIcall name="createAttribute">

  <arguments><string>QName</string></arguments>

  </backendAPIcall>

<backendAPIcall name="setValue">

<arguments>T(E)</arguments>

</backendAPIcall>

</attributeConstructor>

- Sort expression

  Let *ord* be either ASCENDING or DESCENDING,

  T($E1$ $SORTBY$ $E2$ ord) =

  <sortExpr order="ord"> T($E1$) T($E2$) </sortExpr>

- Typeswitch expression

  T($TYPESWITCH$ ($E0$) AS $\$v$

  $CASE$ $Type1$ $RETURN$ $E1$

  $CASE$ $Type2$ $RETURN$ $E2$

  ...

  $CASE$ $Typem$ $RETURN$ Em

  $DEFAULT$ $RETURN$ $E00$)

  = <typeswitchExpr>

  T($E0$)<varBinding name="$vx"/>

  <case type="Type1">T(E1)</case>

  <case type="Type2">T(E2)</case>

  ...

  <default>T($E00$)</default>

  </typeswitchExpr>

Bibliography

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] Software AG. QuiP, software AG's XQuery Prototype, 2001. http://www.softwareag/com/developer/quip.

[3] D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, and P. Rodriguez-Gianolli. ToX - The Toronto XML Engine. *International Workshop on Information Integration on the Web*, 2001.

[4] Catriel Beeri and Yariv Tzaban. SAL: An algebra for semistructured data and XML. In *WebDB (Informal Proceedings)*, pages 37–42, 1999. http://citeseer.nj.nec.com/beeri99sal.html.

[5] D. Chamberlin, J. Robie, and D. Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. 2000. http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html.

[6] S. Cluet, S. Jacqmin, and J. Simeon. The New YATL: Design and Specifications, 1999.

[7] World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification. *W3C Recommendation*, October 1998. http://www.w3.org/TR/REC-DOM-Level-1.

[8] World Wide Web Consortium. Namespaces in XML. *W3C Recommendation*, January 1999. http://www.w3.org/TR/REC-xml-names.

[9] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. *W3C Recommendation*, November 1999. http://www.w3.org/TR/xpath.

[10] World Wide Web Consortium. XML Query Requirements. *W3C Working Draft*, February 2001. http://www.w3.org/TR/xmlquery-req.

[11] World Wide Web Consortium. XML Query Use Cases. *W3C Working Draft*, June 2001. http://www.w3.org/TR/xmlquery-use-cases.

[12] World Wide Web Consortium. XML Schema: Formal Description. *W3C Working Draft*, March 2001. http://www.w3.org/TR/xmlschema-formal.

[13] World Wide Web Consortium. XML Schema Part 2: Datatypes. *W3C Working Draft*, May 2001. http://www.w3.org/TR/xmlschema-2.

[14] World Wide Web Consortium. XML Syntax for XQuery 1.0 (XQueryX). *W3C working Draft*, June 2001. http://www.w3.org/TR/xqueryx.

[15] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model. *W3C working Draft*, June 2001. http://www.w3.org/TR/query-datamodel.

[16] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators. *W3C Working Draft*, December 2001. http://www.w3.org/TR/xquery-operators/.

[17] World Wide Web Consortium. XQuery 1.0 Formal Semantics. *W3C working Draft*, June 2001. http://www.w3.org/TR/WD-query-semantics-20010607.

[18] X-Hive Corporation. http://www.x-hive.com/xquery.

[19] Wayne Demery. XML for RDBMS systems. http://www.cs.colorodo.edu/˜getrich/Classes/csci5817/Term_Papers/demery.

[20] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. December 1998. http://www.w3.org/TR/1998/NOTE-xmlql-19980809.

[21] Karima Echihabi. Indexing for XML Types. Master's thesis, University of Toronto, 2002.

[22] Rick Cattell et al. *The Object Database Standard: ODMG-93, Release 1.2.* . Morgan Kaufmann Publishers, San Francisco, 1996.

[23] WebGain Inc. JavaCC. http://www.webgain.com/products/.

[24] H. V. Jagadish and D. Srivastava. http://www.eecs.umich.edu/db/timber.

[25] Kweelt. http://db.cis.upenn.edu/Kweelt/.

[26] J. lapp, J. Robie, and D. Schach. XML Query Language. 1998. http://www.w3c.org/TandS/QL/QL98pp/xql.html.

[27] M. Lay. DBLP database web site. http://www.infomatik.unitrier.de/˜ley/db.

[28] B. McLaughlin. *Java and XML.* OReilly and Associates, Sebastopol, 2000.

[29] Microsoft. XQuery Demo. http://131.107.228.20/xquerydemo/demo.aspx.

[30] F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. *Fourth International Workshop on the Web and Databases*, 2001.

[31] Jonathan Robie, Joe Lapp, and David Schach. Extensible Stylesheet Language (XSL) Version 1.0. *W3C Recommendation*, October 2001. http://www.w3.org/TR/xsl/.

[32] SAX. http://www.saxproject.org.

[33] Fatdog Software. http://www.fatdog.com.

[34] Lucent Technologies. http://www-db.research.bell-labs.com/galax.

[35] Edward Xia. Efficient Implementation of An XML Query Algebra. Master's thesis, University of Toronto, 2002.

[36] XML Query Working Group. XQuery implementations. http://www.w3.org/XML/Query#implementations.

[37] The xml.apache.org project. Apache Cocoon. http://xml.apache.org/cocoon.

[38] The xml.apache.org project. Xerces Java Perser. http://xml.apache.org/xerces-j/.