# The XQuery Formal Semantics: A Foundation for Implementation and Optimization

Byron Choi, University of Pennsylvania, kkchoi@gradient.cis.upenn.edu
Mary Fernández, AT&T Labs - Research, mff@research.att.com
Jérôme Siméon, Bell Laboratories, simeon@research.bell-labs.com

May 31, 2002

**Abstract**

XQuery is a strongly typed, functional language, which supports the common processing, transformation, and querying tasks of a wide variety of XML applications. Following the tradition of other functional languages, XQuery includes a complete formal semantics. In this paper, we argue that basing an XQuery implementation on the XQuery Formal Semantics not only ensures correctness, but is a good foundation for optimization. We describe an architecture that we have implemented and that is based on the XQuery Formal Semantics and describe several logical and physical optimizations that can be easily integrated in the above architecture.

## 1 Introduction

XML [4] is a flexible exchange format that can represent many classes of data: structured documents with large fragments of marked-up text; homogeneous records with values such as those in relational databases; and heterogeneous records with varied structure and content such as those in object-oriented and hierarchical databases. Many new applications need to deal with all these classes of data simultaneously.

XQuery [29] is a language for XML currently being defined by the World-wide Web Consortium (W3C) to meet the varied needs of XML applications. XQuery is a strongly typed, functional language, which supports the common processing, transformation, and querying tasks of XML applications. It contains XPath [28] expressions for traversing and extracting fragments of XML documents, expressions to join several documents and to construct new XML documents, and a large library of functions on XML documents and values.

Several communities have contributed to the design and definition of XQuery.

- The "document" community contributes their experience in designing languages and tools (e.g., editors and text engines) for processing structured documents. Document-centric applications often require text search that spans markup boundaries and processing that depends on document context and order. Within XPath, XQuery supports operations on document order and *axis* expressions, which are used to navigate in the document and to access the context of particular document fragments.

- The "database" community contributes their experience in designing query languages and processing tools for data-intensive applications. Data-centric applications often require very efficient selection, retrieval, and transformation of small fragments of data stored in massively large databases. XQuery incorporates features from query languages for relational databases (SQL) and object-oriented databases (OQL). Notably, it can express joins between multiple documents and can restructure existing or construct new documents.

- The "programming language" community contributes their experience in designing functional languages, type systems, and specifying formal language semantics. XQuery is a purely functional language with a static type system based on XML Schema [25, 20]. It is fully compositional, supports user defined functions, as well as recursion. Following the tradition of other functional languages [21], XQuery includes a complete formal semantics [30], which is an integral part of its specification.

Despite being a young – and still evolving! – language, XQuery already has numerous implementations. The XML Query working group Web page[1] lists eighteen publicly announced implementations. Not surprisingly, each community favors its own implementation and optimization techniques, and there is active debate over what are the best techniques for implementing XQuery.

In this paper, we argue that basing an XQuery implementation on the XQuery Formal Semantics not only ensures correctness, but is a good foundation for optimization. This claim will not surprise many implementors of programming languages, but is a subject of vigorous debate among implementors from the database and document communities.

Efficient language implementations often rely on the ability to discover alternative execution strategies. Implementations of database query languages typically rely on algebraic query plans, for which the system can perform rewritings and apply some cost comparison. Implementations of functional languages typically rely on optimization techniques that include type-based rewritings, operation elimination, unboxing, and function inlining, among others.

Applying similar optimization techniques to XQuery requires addressing two difficult problems. First, any rewriting technique must preserve the semantics of the original expression. To accommodate the flexibility of structure within XML documents, many XQuery expressions have a complex implicit semantics. For instance, an expression's semantics might depend on the type of its sub-expressions, apply automatic type-conversion rules, or include implicit existential quantification of predicates. Any rewriting must guarantee that these implicit semantics are preserved. A second problem arises from the diversity of physical representations of XML documents. Many techniques already exist for storing XML documents in native XML [18] and in non-XML database systems [3, 8, 12, 23] and for translating XQuery into SQL or other implementation languages [11, 24]. Because efficient evaluation of path expressions is central to any viable XQuery implementation, most optimization techniques focus on path evaluation [5, 1, 15], but only apply to a narrow subset of XQuery.

As we will discuss, one important property of the XQuery Formal Semantics is that it makes all of the implicit semantics of XQuery expressions explicit by normalizing them into expressions in the *XQuery Core*. Since the XQuery Core makes all operations atomic and explicit, it provides a foundation to support many traditional and XQuery-specific optimizations. This paper makes the following contributions.

- We describe how an XQuery implementation can be derived directly from the XQuery Formal Semantics. We describe the architecture of such a system and explain the role of the XQuery Core in this architecture.
- We show that many useful logical and physical optimizations are easily expressed as rewritings over the XQuery Core and can be easily integrated in the above architecture. We present examples of traditional and XQuery-specific optimizations.
- We identify some of the challenges of supporting physical-level optimizations in such an architecture.

To the best of our knowledge, this paper is the first to propose an approach that can support a comprehensive set of optimization techniques while preserving XQuery's complex semantics. This paper is based largely on our experience with Galax [13], one of the first implementations of XQuery and the first complete implementation of the XQuery type system. Galax implements the proposed architecture. An interactive demo of Galax is available on the Galax web site.

[1] `http://www.w3.org/XML/Query`

```
<bibliography>
  <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="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>
  </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>
</bibliography>
```

Figure 1: Example XML document

The rest of the paper is organized as follows. In Section 2, we introduce the family of XQuery specifications, give an overview of the XQuery Formal Semantics and give an introduction to the XQuery Core. In Section 3, we explain how one can build a "naive" XQuery implementation based on the XQuery Formal Semantics. In Section 4, we show how many optimization techniques can be adapted to such an implementation.

# 2 XQuery and the XQuery Formal Semantics

## 2.1 XQuery Overview

To illustrate XQuery, we use the example XML document in Figure 1. The document contains one `bibliography` element that contains three `book` elements. This document is "well-formed" because every open element tag has a corresponding closing element tag and the elements are properly nested. One strength of XML is its support for variability in document content and structure. In this example, all the books contain a `year` attribute, a `title` element, and a `publisher` element, but only two books contain a `price` element, two books contain one or more `author` elements, and only one contains an `editor`.

XQuery is designed to support variance in XML data. For example, this expression:

```
input()//book[author/last = "Stevens"]
```

returns `book` elements at any level in the input document that contains one `author` element with one `last` element with content equal to "Stevens". The expression is implicitly existentially quantified: A book element is returned if it has *at least* one author with *at least* one last name with the given value.

A common querying task is to transform the structure and content of an XML document, which requires construction of XML values. For example, this expression illustrates element construction and the

```
document { element bibliography }
element bibliography {
  element book*
}
element book {
  attribute year { xs:integer },
  element title,
  (element author+ | element editor+),
  element publisher,
  element price?
}
element title { xs:string }
element author {
  element last { xs:string },
  element first { xs:string }
}
element publisher { xs:string }
element price { xs:float }
```

Figure 2: An example XML Schema in XQuery type notation

`for-let-where-return` (pronounced "flower") expression:

```
<books> {
  for $b in input()//book[publisher = "Addison-Wesley"]
  let $authorct := count($b/author)
  where $authorct > 3
  return <awbook> { $b/title, $b/price } </awbook>
} </books>
```

It constructs one `books` element, which contains one `awbook` for each `book` (in the input document) that has at least three authors and whose publisher is "Addison-Wesley". The `awbook` contains the `title` and `price` of the selected book.

Although XQuery supports well-formed XML documents, it is specifically designed to support XML documents that have associated XML Schemas [25, 20]. At its core, an XML Schema is a tree grammar that specifies the permissible element and attribute names, types of atomic terminal values, and vertical and horizontal structure of a collection of XML documents. An XML document is *valid* with respect to an XML Schema if it is in the collection of documents specified by the XML Schema. The XQuery type system is based on XML Schema. XML Schemas can be imported in the XQuery type system and used for static type analysis [30]. We will not discuss static typing further in this paper, but we will make use of static type information in some optimizations. We refer the reader to the XQuery Formal Semantics for more details [30].

The example schema in Figure 2 uses XQuery's type notation. A `bibliography` element contains zero or more `book` elements. A `book` element contains a `year` attribute and one `title` element followed by a choice of one or more `author` elements or one or more `editor` elements, followed by one `publisher`, followed by an optional `price` element. The sequence operator (`,`) combines types in a sequence; the repetition operator `*` (`+`) denotes zero (one) or more instances of a type; and the optionality operator (`?`) denotes zero or one instance of a type. An `author` contains one `last` and one `first` element, each with string content. The `title` and `publisher` elements contain a single string and the `price` element contains a floating point number. The document in Figure 1 is an instance of the type in Figure 2. We return to this example when we discuss type-based optimizations.

XQuery is strongly typed: The operand, argument, and return types of all built-in operators and functions are specified and the signature of a user-defined function must specify the types of its arguments and its return value. XQuery's type semantics requires that an operand or argument value be of the corresponding required type. Type checking may be implemented dynamically or statically. Given the types for input documents, XQuery's static type semantics infers the static type for every expression in a query and checks that the inferred type of an expression is subsumed by the required type for the expression. For example,

assume that the variable `$book` is bound to the first book in the document in Figure 1 and has the type `element book` defined in Figure 2, then the following expression is well-typed:

```
data($book/price) * 0.07
```

because the inferred type of `data($book/price)` is `xs:float` and multiplication is defined on `xs:float`. The following expression, however is ill-typed:

```
data($book/publisher) * 0.07
```

because multiplication is not defined on `xs:string`.

**The XQuery Family of Specification** Implementing XQuery is a non-trivial task, in part, because its specification spans several documents. The definition of XQuery comprises the following W3C working drafts:

- the XQuery and XPath 2.0 data model [9], which defines the information in an XML document that is available to a query processor;
- a language specification [29], which includes XQuery's user-level grammar and an English-language description of XQuery's semantics;
- a formal semantics [30], which includes a formal-language definition of the static and dynamic semantics of each XQuery expression; and
- a library of built-in functions and operators [27].

In addition, these working drafts depend on other W3C specifications, in particular, XML [4], and XML Schema [25, 20].

## 2.2 XQuery Formal Semantics

The purpose of the XQuery formal semantics is to provide implementors with a processing model and a complete description description of the language's static and dynamic semantics. This can be used as a detailed "road map" for a complete (albeit naive) XQuery implementation.

The formal semantics processing model is composed of four phases: parsing, normalization, static type analysis, and dynamic evaluation. Parsing checks that an input query is an instance of the language defined by the XQuery grammar – we do not discuss this phase further.

**The XQuery Core and Normalization** The XQuery language provides many features that make queries simpler to write and use, but are also redundant. For instance, complex `for-let-where-return` expressions can be rewritten as the composition of individual `for`, `let`, and `if-then-else` expressions. The formal semantics defines a proper subset of the XQuery language, called the XQuery Core language, and gives rules that rewrite or *normalize* every XQuery expression as a XQuery Core expression. The static type and dynamic value semantics of XQuery are defined on this core language. The core grammar is in Appendix A.

Many normalization rules rewrite expressions with complex implicit semantics into Core expressions with simpler, but more verbose, explicit semantics. For example, the expression:

```
input()//book[publisher = "Addison-Wesley"]
```

is normalized into the (surprisingly verbose) Core expression:

```
 1.  for $book in
 2.    (for $dot in input() return descendant-or-self::book)
 3.  return
 4.  let $bool :=
 5.      (some $pub in (for $dot in $book return child::publisher)
 6.       satisfies
 7.         (let $pubval :=
 8.             typeswitch $pub
 9.             case atomic value return $pub
10.             case node return
11.               (typeswitch data($pub) as $val
12.                case atomic value return $val
13.                default return xf:error())
14.             default return xf:error()
15.           return $pubval eq "Addison-Wesley"))
16.   return if ($bool) then $book else ()
```

The two `for` expressions on line 2 and line 5 implement, respectively, the path expressions `input()//book` and `book/publisher`. They each bind the *context node* (represented by the special variable `$dot`); the axis expressions `descendant-or-self::book` and `child::publisher` are defined in terms of this context node. The existentially quantified `some` expression on lines 5–15 expresses the selection predicate `[publisher = "Addison-Wesley"]`. Its body is evaluated once for each `publisher` node bound to `$pub`. and contains two `typeswitch` expressions, which handle the potential variability of values bound to `$pub`. The outer typeswitch (lines 8–14) raises an error if `$pub` is not a single atomic value or node. If it is an atomic value, its value is returned; otherwise, the node's content value is extracted; The inner typeswitch (lines 11–13) raises an error if the node's value is not a single atomic value. It is often possible to simplify Core expressions given the type associated with the input document. We discuss type-based optimizations in Section 4.

After normalization, the semantics of an expression is obtained by applying static type and dynamic value inference rules to its normalized Core expression. This is done during the next two phases.

**Static type analysis** Static type analysis checks that an expression is type correct, and if so, determines its static type. Static type analysis is defined only on Core expressions and proceeds by applying type inference rules to the abstract syntax tree of a Core expression, starting with the types of literals and valid input documents and proceeding up the tree. We refer the reader to the formal semantics [30] for examples of type inference rules.

Static type analysis can result in a static error, if the expression is not type correct. For instance, a comparison between an integer value and a date value is an error that can be detected during static type analysis. If static type analysis succeeds, it results in an abstract syntax tree where the top-level Core expression and each sub-expression is annotated with its static type.

**Dynamic evaluation** In this phase, the value of an expression is computed. The dynamic semantics is defined only on Core expressions. Evaluation proceeds by applying value inference rules to the abstract syntax tree of a Core expression, starting with the top-level query expression and conditionally applying inference rules to sub-expressions top down and synthesizing values bottom up. The dynamic semantics guarantees that every Core expression can be unambiguously reduced to a value. We refer the reader to the formal semantics [30] for examples of value inference rules.

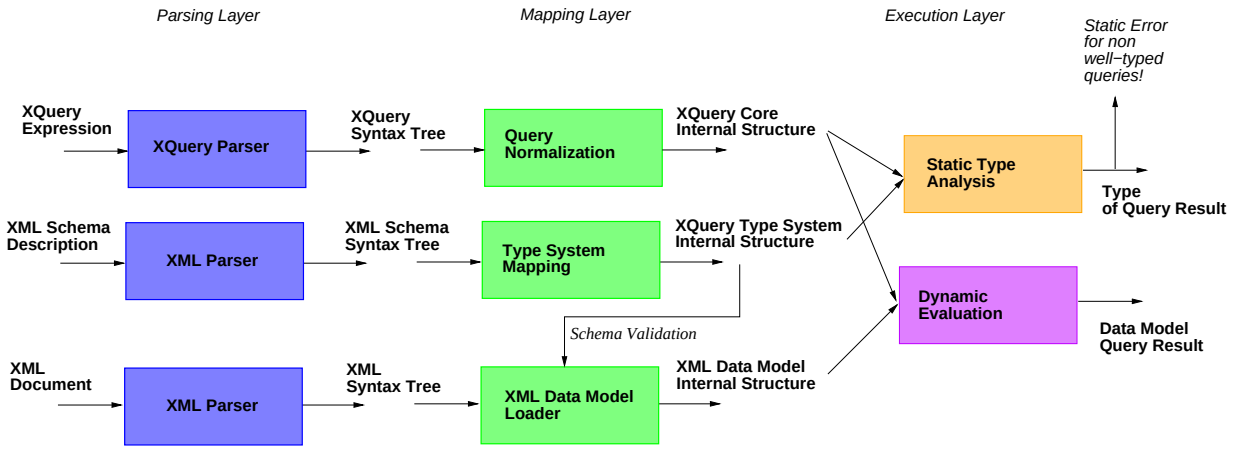


Figure 3: Galax's Architecture

# 3 Implementing the XQuery Formal Semantics

We give an overview of the architecture of Galax, an implementation of XQuery 1.0 based on the XQuery Formal Semantics. We note that another XQuery prototype developed by GMD-IPSI [10] is also based on the XQuery Formal Semantics. Our goal is for Galax to be a reference implementation of XQuery and therefore, completeness, strict conformance to the specifications, and semantic integrity are the goals of our current prototype. Because XQuery is a complex language, we believe that basing Galax's architecture and implementation directly on the XQuery 1.0 Formal Semantics is the surest way to achieve these goals. This approach also permits us to evaluate the soundness of the XQuery specification and to identify semantic problems during early stages of specification.

Galax is implemented in Objective Caml [19] and is open source, making it a useful reference for other XQuery implementors. Galax is reasonably light weight (its native-code footprint under Linux is about 1.2 MB) and very portable (O'Caml targets include Win32, Macintosh, and virtually all Unix platforms). Although most implementations of XQuery will not be implemented in functional languages, we found that O'Caml is ideal for implementing Galax. Its algebraic types and higher-order functions simplify the symbolic manipulation that is central to the query transformation, analysis, and optimization that we need to perform. A detailed description of Galax can be found on the Galax home page at: `http://db.bell-labs.com/galax/`.

Figure 3 depicts Galax's architecture and relates Galax's modules with XQuery's processing model and formal semantics. Inputs to the Galax engine comprise one or more input XML documents, one or more XML Schema associated with input documents, and one or more queries that process the input documents. The system is decomposed into three layers:

**Parsing Layer** The parsing layer implements the parsing phase of the XQuery processing model. It takes all of the inputs, parses them, and builds abstract syntax trees (AST) corresponding to the inputs. XML documents and XML Schema documents are parsed according to the grammar rules in the XML 1.0 specification [4]. Queries are parsed according to the grammar rules in the XQuery 1.0 document [29]. For input documents, an abstract syntax tree is never materialized. Instead, Galax's SAX parser instantiates an XML document directly in its memory-resident implementation of the XQuery data model.

**Mapping Layer** The mapping layer implements the normalization phase of the processing model and transforms the input ASTs into their corresponding internal representations. XQuery expressions are

normalized into XQuery Core expressions and XML Schema documents are mapped into XQuery's internal type values. The XQuery normalization rules are implemented (almost) literally in Galax making it possible to correlate easily the definition of an expression with its implementation. Input documents that have associated XML Schemas are validated while the documents are parsed and instantiated in the XQuery data model.

**Execution Layer** The execution layer implements the static type analysis and dynamic evaluation phases of the processing model. First, static type analysis is applied to the Core expressions and input types. The type inference rules are implemented (almost) literally in Galax making it possible to correlate easily each typing rule with its implementation. If static typing fails, the system raises an error and halts.

If static typing succeeds, the evaluation module is applied to the core expressions and to the data model representation of the input documents. The value inference rules are implemented (almost) literally in Galax making it possible to correlate easily each evaluation rule with its implementation. Evaluation can either raise a run-time error (for errors that static analysis cannot detect) or return an XML value as the result.

Clearly, an evaluation strategy based on literal interpretation of Core expressions does not scale, but it is adequate for testing semantic correctness and is a sound basis for other implementation strategies, which we discuss next.

# 4 Optimizing XQuery

Although XQuery is a new language, it can benefit from the many optimization techniques that exist for functional programming languages and for other database query languages. Optimizations for both functional programming languages and database query languages broadly fall into two categories: logical and physical. Logical optimizations typically transform the intermediate representation of a program or query and make improvements that are independent of a particular target architecture or query engine. Physical optimizations typically transform the executable representation of a program or query (e.g., machine instructions or a physical data operations in a query execution plan) and make improvements that depend on features of the target architecture or query engine.

In this section, we illustrate how to use the XQuery Core as a basis for optimization. We illustrate how existing and new techniques can be applied directly to the Core representation.

## 4.1 Type-based optimization

As shown in Section 2.2, the dynamic semantics of an XQuery expression often depends on the dynamic types of sub-expressions and on the cardinality of the sequences computed by sub-expressions. In XQuery Core expressions, the `typeswitch`, explicit quantifier `some`, and casting expressions implement the semantics that handles this potential variability of the XML input.

For queries applied to input documents with known schemas, however, static type information can be used to simplify the queries. These type-based optimizations remove unnecessary operations, which is often a prerequisite to more advanced optimizations.

**The `typeswitch` expression** The `typeswitch` expression is used extensively in the normalized semantics of XQuery. For example, the normalized query in Section 2.2 contains two nested typeswitch expression (lines 8 to 14):

```
...
  typeswitch $pub
    case atomic value return $pub
```

```
    case node return
      (typeswitch data($pub) as $val
       case atomic value return $val
       default return xf:error())
    default return xf:error()
...
```

Static type analysis infers that the `$pub` variable is always a node and therefore, the second branch of the first `typeswitch` is always executed. In addition, when the input document has the type `document { element bibliography }`, static type analysis infers that the variable `$pub` has type `element publisher` and therefore always contains an atomic value of type string. Again, only the first case in the second typeswitch is always executed. The original `typeswitch` expression can then be replaced by `data($pub)`. This results in the following simplified Core expression:

```
for $book in
  (for $dot in input() return descendant-or-self::book)
return
  let $bool :=
    (some $pub in (for $dot in $book return child::publisher)
     satisfies
       (let $pubval := data($pub)
        return $pubval eq "Addison-Wesley")
  return if ($bool) then $book else ()
```

The `typeswitch` expression is unique to XQuery, but it is similar to pattern-match expressions in ML languages and XDuce [16]. This shows that techniques for simplifying pattern-match expressions [17], or for converting dynamic dispatch into static dispatch [7] are certainly relevant to XQuery.

**Existential quantification** Continuing with the example above, static type analysis infers that the variable `$book` has type `element book`. Since each book has one publisher, the body of the `for` expression in the existential quantification is a single `publisher` element, therefore the existential quantification can be eliminated. This results in the following simplified expression:

```
for $book in
  (for $dot in input() return descendant-or-self::book)
return
  let $bool :=
    (let $pub := (for $dot in $book return child::publisher)
     returns
       (let $pubval := data($pub)
        return $pubval eq "Addison-Wesley")
  return if ($bool) then $book else ()
```

It is important to note that a number of optimizations are enabled by the presence of a schema and by the ability to do static type inference. Indeed, without XQuery being statically typed, many such rewritings are infeasible and will make further optimizations more difficult to apply.

## 4.2 Logical optimizations

The XQuery Core expressions: `let`, `for`, and `if-then-else` are analogs of list-comprehension operations [26], therefore, standard rewritings of list comprehensions, which correspond to some typical logical optimizations for database query languages, can be applied.

**The `let` expression** A `let`-bound variable can be replaced by the expression to which it is bound. Applying this rule to the variables `$pub` and `$pubval` in the example above results in the simplified expression:

```
for $book in
  (for $dot in input() return descendant-or-self::book)
return
  if (data(for $dot in $book return child::publisher)
        eq "Addison-Wesley")
    then $book
    else ()
```

**The `$dot` variable** XPath axis expressions (e.g., `child::publisher`) implicitly refer to the context node, which is bound to the `$dot` variable. As a result, the `$dot` variable cannot always be replaced, because it preserves the context in which XPath axis expressions will be evaluated.

Some simplification, however, is possible. In the example above, static type analysis infers that the `input()` function evaluates to a unique node and that the `$book` variable is bound to one `element book`. This means that two `for` expressions can be replaced by two `let` expressions that bind `$dot`. This results in the simplified expression:

```
for $book in
  (let $dot := input() return descendant-or-self::book)
return
  if (data(let $dot := $book return child::publisher)
        eq "Addison-Wesley")
    then $book
    else ()
```

As the `$book` variable in the outer `for` expression and the second `$dot` variable denote the same value, we can bind the `$dot` variable directly in the outer `for` expression:

```
for $dot in
  (let $dot := input() return descendant-or-self::book)
return
  if (data(child::publisher)
        eq "Addison-Wesley")
    then $dot
    else ()
```

The resulting expression is much simpler than the original Core expression: It contains only one `for` expression, two XPath axis expressions and one conditional expression.

Clearly, any implementation strategy that transforms a core expression into a physical execution plan will be more effective on the simplified expression above than on the original expression.

**The `for` expression** Other traditional optimizations involve `for` expressions. An important query optimization is "pushing selections", that is, moving selection predicates earlier in an execution plan. (In the programming-language literature, this is known as "hoisting" loop-invariant expressions.) For example, consider this expression:

```
for $book in //book
return
  for $title in $book/title return
  where $book/publisher = ‘‘Addison-Wesley’’
  return $title
```

The `where` selection condition is invariant for all bindings of `$title` for a particular binding of `$book`, therefore the selection condition can be "pushed" ("hoisted") before the `for` expression that binds `$title`. This results in the following expression:

```
for $book in //book
where $book/publisher = ''Addison-Wesley''
return
  for $title in $book/title return
  return $title
```

Most of those optimizations have been applied to database query languages [2] and programming languages [26].

**Join reordering** Although XML documents are inherently ordered, many applications of XML do not depend on the relative order of elements within a document. When order is insignificant, it is possible to apply "join reordering" techniques. XQuery includes an `unordered` keyword, which indicates that the semantics of the query can disregard order. For example, this expression:

```
unordered
  for $book in //book,
  for $review in //reviews
  where $review/title = $book/title and $review/score > 14
  return ($book/title, $book/author)
```

can be rewritten into the (potentially) more efficient expression:

```
unordered
  for $review in //reviews
  where $review/score > 14 return
    for $book in //book,
    where $review/title = $book/title
    return ($book/title, $book/author)
```

The latter expression first selects reviews with a score greater than fourteen before joining those reviews with books. Join rewriting techniques are the subject of a large body of database research. We refer the reader to the appropriate literature[14, 6, 22] for more detail.

**Sorting by document order** Another complexity of XPath's (and XQuery's) semantics is document order. Most XPath expressions require their result to be sorted in document order and sorting is typically expensive. It is sometimes possible to remove sorting by document order, but some query analysis is necessary to do so. For example, if the original sequence is already sorted, such operation can be removed. For example, the following expression:

```
input()/book/author
```

is normalized into the following core expression, which explicitly sorts the `author` elements by document order:

```
sort-docorder(
  for $dot in
    sort-docorder(for $dot in input() return child::book)
  return
    child::author
)
```

However, in this query, since the result of the input() function is a single node and the `child::` axis returns the children in document order, the result of the expression is already in document order and does not need to be sorted. Since the expression preserves the original order of its input sequence, the same analysis can be performed for the outer `sort-docorder` function. As a result, both sort functions can be removed, which yields the following expression:

```
for $dot in
  (for $dot in input() return child::book)
return
  child::author
```

### 4.3 Physical optimizations

The document-processing and database user communities are already investigating many techniques for optimizing XPath and XQuery. The techniques are typically tuned for a particular application domain and to take advantage of features provided by particular XML storage systems. Document-processing queries, for example, often include full-text operators and therefore can take advantage of full-text indexes.

Database applications shred XML documents into relational tables and rewrite XQuery queries into SQL expressions that can be optimized by the relational query optimizer [3, 8, 12, 23]. Native XML databases also propose specific algorithms and indices specifically tuned to evaluate XML path expressions [18].

It is out of the scope of this paper to provide a complete review of the growing number of algorithms which have been proposed. Still, we would like to illustrate some of the possibilities on a simple example. Assume that our document has been stored, e.g., in a relational database, that supports efficient access to all the elements with a given name. We will call the corresponding physical operation `name-index`. This operation takes the name of the element as input. Then our previous query can be rewritten as:

```
for $book in name-index(book)
return
  let $bool :=
     data(let $dot := $book return child::publisher) eq "Addison-Wesley"
  return if ($bool) then $book else ()
```

This might already perform better than the naive evaluation that would navigate from the root of the document. Assume also that the query engine implements a *twig-join* algorithms [5]. Such an algorithm can be used to evaluate efficiently the parent-child relationship used in XPath navigation. We call `parent-join` the corresponding join operation. Then the expression can be further rewritten as:

```
parent-join (name-index(book),
             for $pub in name-index(publisher)
             where data($pub) eq "Addison-Wesley")
```

Here the query plan can use the `name-index` physical access method for `book` and `publisher` elements, and the `parent-join` operation checks that indeed each `publisher` is a child of a `book`.

**The role of a cost model** To conclude, we would like to point out that many physical optimizations and the ability to detect an efficient rewriting depend on the physical storage model for XML and the presence of a cost model to detect "good" query plans. Because of XML's flexibility and expressiveness, we expect there will be many physical storage models for XML, adapted for particular applications' needs, and that a variety of cost models will be developed that depend on the features of the particular physical storage system. Regardless of the physical features available to a query engine, preserving XQuery's semantics during optimization and translation will be difficult, but starting with the XQuery Core should help simplify this problem.

## References


[1] Sihem Amer-Yahia, S. Cho, Laks Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2001.

[2] Catriel Beeri and Yoram Kornatzky. Algebraic optimization of object-oriented query languages. *Theoretical Computer Science*, 116(1&2):59–94, August 1993.

[3] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2002.

[4] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. `http://www.w3.org/TR/REC-xml/`.

[5] N. Bruno, Divesh Srivastava, and N. Koudas. Holistic twig joins: Optimal xml pattern matching. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2002.

[6] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In *Proceedings of International Workshop on Database Programming Languages*, pages 226–242, New York City, USA, August 1993. `http://cosmos.inria.fr:8080/cgi-bin/publisverso?what=abstract&query=064.`

[7] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *OOPSLA Proceedings*, 1996.

[8] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 431–442, Philadelphia, Pennsylvania, June 1999.

[9] XQuery 1.0 and XPath 2.0 data model. W3C Working Draft, April 2002.

[10] Peter Fankhauser, T. Groh, and S. Overhage. Xquery by the book: The ipsi xquery demonstrator. In *Proceedings of the International Conference on Extending Database Technology*, 2002.

[11] Mary Fernandez, Yana Kadiyska, Atsuyuki Morishima, Dan Suciu, and Wang-Chiew Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems*, 2002.

[12] Daniela Florescu and Donald Kossman. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. *IEEE Data Engineering Bulletin 1999*, 1999.

[13] Galax: An implementation of xquery. `http://db.bell-labs.com/galax/`.

[14] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[15] T. Grust. Accelerating xpath location steps. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2002.

[16] Haruo Hosoya and Benjamin C. Pierce. XDuce: an XML processing language. In *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.

[17] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, January 2001.

[18] Carl Christian Kanne and Guido Moerkotte. Efficient storage of XML data. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2000.

[19] Xavier Leroy. *The Objective Caml system, release 3.04, Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, December 2001. http://caml.inria.fr/.

[20] Murray Maloney and Ashok Malhotra. XML schema part 2: Datatypes. W3C Recommendation, May 2001.

[21] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[22] Guido Moerkotte. Building query compilers. `http://pi3.informatik.uni-mannheim.de/staff/mitarbeiter/moer/querycompiler.ps`, May 1999.

[23] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Edinburgh, Scotland, September 1999.

[24] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2002.

[25] Henri S. Thompson, David Beech, Murray Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Recommendation, May 2001.

[26] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[27] Xquery 1.0 and xpath 2.0 functions and operators version 1.0. W3C Working Draft, April 2002.

[28] XPath 2.0. W3C Working Draft, April 2002.

[29] XQuery 1.0: An XML query language. W3C Working Draft, April 2002.

[30] XQuery 1.0 formal semantics. W3C Working Draft, March 2002.

# A Core grammar

The following grammar descrines the XQuery Core. See [30] for a complete description.

```
-- The (almost) full XQuery core grammar --

ExprSequence   ::=    Expr ("," Expr)*
Expr           ::=    "for" Variable "in" Expr "return"  Expr
                |     "let" Variable  ":=" Expr "return" Expr
                |     "typeswitch" "(" Expr ")" "as" Variable TypeswitchClauses
                |     "if" "(" Expr ")" "then" Expr "else"  Expr
                |     ("cast" "as" | "assert" "as") Datatype "(" Expr ")"
                |     ("element" | "attribute" )
                                (QName |  EnclosedExpr) "" ExprSequence? ""
                |     Axis NodeTest
                |     Variable
                |     Literal
                |     "text"  String
```

```
                |     QName "(" (Expr ("," Expr)*)? ")"
                |     "/"
                |     "(" ExprSequence? ")"

TypeswitchClauses ::= CaseClause+ "default" "return" Expr
CaseClause     ::=    "case" Datatype "return" Expr

EnclosedExpr   ::=    "" ExprSequence ""

-- XPath Axis --

Axis           ::=    "child" "::"
                |     "descendant" "::"
                |     "attribute" "::"
                |     "self" "::"

-- Node tests --

NodeTest       ::=    NameTest |  KindTest
NameTest       ::=    QName |  Wildcard
Wildcard       ::=    "*" | NCName ":*" |  "*:" NCName
KindTest       ::=    ProcessingInstructionTest
                |     CommentTest
                |     TextTest
                |     AnyKindTest
ProcessingInstructionTest    ::=    ProcessingInstruction "(" StringLiteral? ")"
CommentTest    ::=    Comment "(" ")"
TextTest       ::=    Text "(" ")"
AnyKindTest    ::=    Node "(" ")"

-- Tokens (only partial) --

Variable       ::=   "$" QName
QName          ::=   ":"? NCName (":" NCName)?
```

Note that there is no '.' expression in the core, which is represented as a standard variable with name $dot.