Implementing XQuery 1.0: The Galax Experience

Mary Fernández
AT&T Labs

Jérôme Siméon
Bell Laboratories

Byron Choi
University of Pennsylvania

Amélie Marian
Columbia University

Gargi Sur
University of Florida

Abstract

Galax is a light-weight, portable, open-source implementation of XQuery 1.0. Started in December 2000 as a small prototype designed to test the XQuery static type system, Galax has now become a solid implementation, aiming at full conformance with the family of XQuery 1.0 specifications. Because of its completeness and open architecture, Galax also turns out to be a very convenient platform for researchers interested in experimenting with XQuery optimization.

We demonstrate the Galax system as well as its most advanced features, including support for XPath 2.0, XML Schema and static typechecking. We also present some of our first experiments with optimization. Notably, we demonstrate query rewriting capabilities in the Galax compiler, and the ability to run queries on documents up to a Gigabyte without the need for preindexing. Although early versions of Galax have been shown in industrial conferences over the last two years, this is the first time it is demonstrated in the database community.

1 Introduction

XQuery 1.0 [9] is the XML Query language promoted by the World Wide Web Consortium. After several years of development, XQuery starts being adopted, implemented and used [6, 1, 8, 2, 5]. Being based on quickly evolving working drafts, few implementations support XQuery's latest and most complex features. Galax is a light-weight, portable, open-source implementation of XQuery that aims at full conformance with the family of XQuery 1.0 specifications. Galax is keeping track very closely of the latest XQuery working drafts, with a new release often following each publication by the XML Query working group. Our first goal with this demo is to present Galax, describe its architecture, and demonstrate its most advanced features notably support for XPath 2.0 semantics (support for document order, overloaded arithmetic and comparison operators, etc.), and support for XML Schema (XML Schema import, validation, dynamic operations on types such as `typeswitch`, static typing checking, etc.).

Because of its completeness and open architecture, Galax is also well suited as a platform for experimentation. Users who need support for specific datatypes and operations can easily extend the Galax interpretor with their own libraries. Researchers interested in optimization for XQuery can implement their work in a context where the details of XQuery cannot be overlooked. Our second goal with this demo is to present some of our first experiments with query optimization in Galax. More precisely, we demonstrate features related to query rewriting, and to memory management for large documents.

The development of Galax started in December 2000, and it was first released to the public in April 2002. Although some early prototypes have been presented in a few industrial venues (at XML DevCon 2001 and at XML'2001), this is the first time it is demonstrated to the database community.

2 Galax

2.1 Overview

Galax is an implementation of the family of XQuery 1.0 specifications, with completeness, strict conformance to the specifications, and semantic integrity as first goals. Galax's implementation is based directly on the XQuery 1.0 Formal Semantics [10], as it is the surest way to achieve these goals.

Galax tracks down the XML Query working drafts very closely. At the time we write, the current version of Galax (0.3.0) was released on January 20th 2003, and implements the latest W3C working drafts published on November 15

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

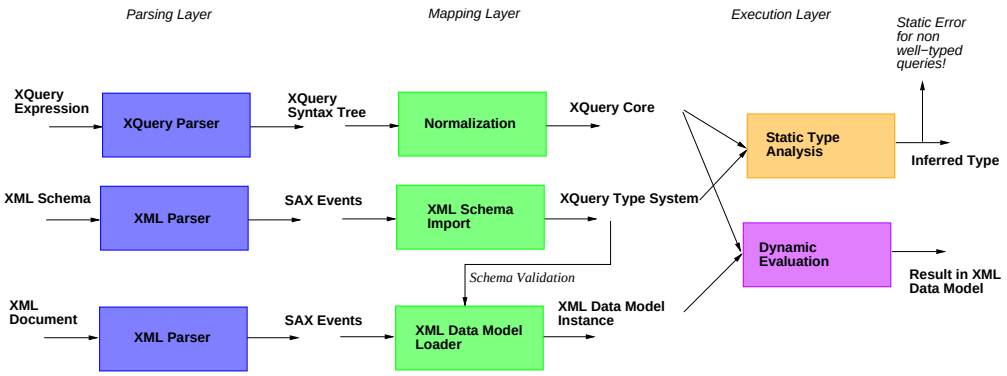

Figure 1: Galax's Architecture

2002. In some cases, Galax even implements decisions of the XML Query working group which have not yet been made public. For instance, some aspects of the semantics of function calls which were decided in at the XML Query face to face meeting in December 2002 are already implemented in the most recent version of Galax.

Galax implements all of the standard XQuery operations, such as path expressions, FLWOR expressions, user defined functions, etc. We do not recall those operations here and refer the reader to other XQuery material [9] and to the Galax online demo[1] for more details. More advanced XQuery features, such as operations on types, overloaded built-in functions, and XML Schema import and validation are illustrated in Section 3.

Galax is written in Objective Caml [3], a modern, statically typed, functional language developed at INRIA. Galax is reasonably light weight (its footprint on Linux is about 1.2 MB) and very portable (OCaml targets include Win32, Macintosh, and virtually all Unix platforms). Although most implementations of XQuery will not be implemented in functional languages, we found that OCaml was 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.

2.2 Architecture

Figure 1 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 XQuery parser takes and XQuery expression and builds an abstract syntax tree (AST) of the query. The XML parser is used for both the XML documents and the schemas. In those cases, an abstract syntax tree is never materialized. Instead, a stream of SAX events is produced and consumed by the data model loader (resp. the XML Schema import module) to create an instance of XML query data model (resp. an instance of the XQuery type system). Data model instances tend to be very large in memory and are often the bottleneck of query processing. In Section 4.2, we illustrate some Galax optimizations that can be used to evaluate queries on documents up to a gigabyte.

Mapping Layer. The mapping layer 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. Normalization often results in large and complex expressions, containing many unnecessary operations. In Section 4.1, we illustrate query rewriting techniques used in the Galax compiler to address that problem.

Execution Layer. The execution layer implements the static type analysis and dynamic evaluation phases of the XQuery 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.

3 Advanced Features

In the first part of the demonstration, we show some of the more advanced features of XQuery, notably support

[1] `http://db.bell-labs.com/galax/demo/`

for XML Schema and support for the new XPath 2.0 semantics. The demo will use examples from the various XML Query usecases, queries from the XMark benchmark demo, and some queries on live data, including the EDICT English-Japanese dictionary[2] (about 28Mb), the XML version of DBLP[3] (about 145Mb).

3.1 XML Schema import and validation

XQuery gives the ability to work on typed data. In XML, typed data can be obtained from document validated against a schema. For instance, consider the following XML Schema description:

```
<element name="person" type="Person"/>
<complexType name="Person">
 <sequence>
  <element name="name" type="xs:string"/>
  <element name="age" type="xs:integer"
           minOccurs="0"/>
  <element name="address" type="xs:integer"
           maxOccurs="unbounded"/>
 </sequence>
</complexType>
```

This schema can be imported and used for validation directly in the text of the query, as follows:

```
import schema "person.xsd"

let $bilbo :=
  validate {
     <person>
       <name>Bilbo Baggins</name>
       <age>111</age>
       <address>Bag End</address>
     </person>
  }
```

After validation, elements are annotated with the name of the type against which they have been validated [7]. In the example above, the `person` element is annotated with the type `Person`. Type annotations are then used when matching a value against a type. For instance, the following function call would be rejected since the person element passed to the `get_name` function has not been validated and therefore does not have the right type annotation:

```
define function get_name(
  $p as element of type Person
) as xs:string
   $p/name

get_name(<person/>)
```

3.2 XPath 2.0 semantics

XPath 2.0 has been designed to preserve backward compatibility with XPath 1.0 as much as possible. However, the semantics of XPath 2.0 differs from XPath 1.0 on a few important operations. Notably, arithmetic operations are defined on the hierarchy of XML Schema numeric types (`xs:integer`, `xs:decimal`, `xs:float` and `xs:double`). To deal with this hierarchy XQuery arithmetic operations are overloaded, that is they behave differently depending on the type of their operands. As an example, here are the results for a few simple arithmetic operations applied on the previous example:

```
$bilbo/age + 1         ==> 112
$bilbo/age + 1.0       ==> 112.0
$bilbo/age + <a>1</a> ==> 1.12E2
```

In all cases, the typed-value of the `age` element is extracted, returning the integer `111`. In the first case, integer addition is applied resulting in the integer `112`. In the second case, the age is cast to the type of the other operand, here a decimal value. Finally, in the last case, the age is added to the (untyped) content of the `a` element, both of which are cast to a double.

3.3 Static type checking

Galax has been designed from the ground up for static type checking. Static typing can be used to detect errors at compile time rather than at run time. Static typing is a conservative analysis, which prevents *any* type errors to occur at run-time. For instance, the following query raises a static error, since some execution of the `get_name` function might raise an error, depending on the value of the variable `$cond`.

```
let $person :=
  if ($cond) then $bilbo else <person/>
return
  get_name($person)
```

The type inferred for the variable person is either the type of the variable `$bilbo` (which is an element `person` with the type annotation `Person`), or an element `person` with the type annotation `xs:anyType` (since it has not been validated). This type is written as follows in the XQuery type system.

```
  element person of type Person
| element person of type xs:anyType
```

Then a subtyping check is perform to see if that type is a valid type for the parameter of the function. Here this subtyping check fails and raises a static type error.

4 Optimization Features

In the second part of the demonstration, we show some of the optimization features that were recently implemented in the Galax engine. Those optimizations are targeted to address the very first bottlenecks we encountered during the development and use of Galax.

4.1 Query rewriting

In order to deal with XPath 2.0 semantics, Galax first performs some *normalization* of the original query. Query expressions as normalized by removing the syntactic sugar and by always using explicit operations instead of implicit

[2] `http://www.csse.monash.edu.au/~jwb/j_jmdict.html`
[3] `http://dblp.uni-trier.de/xml`

XPath 2.0 operations (automatic casting, existential quantification, sorting by document order, etc.). Normalization being an automatic process, it often results in redundant, complex expressions. For instance, consider the following XMark query:

```
for $b in /site/people/person[@id="person0"]
return $b/name
```

The resulting normalized query in the XQuery core is similar to the following query:

```
for $dot at $position in / return
 for $dot at $position in child::site return
  for $dot at $position in child::people return
   for $dot at $position in child::person return
    if ((some $id in (attribute::id) satisfies
         typeswitch ($id)
           case $n as node return data($n)
           default $d return $d) = "person0")
    then child::name
    else ()
```

This expression makes each operation explicit, for instance it binds the context item ('`.`' in XPath) as a variable (`$dot`), the context position ('`position()`' in XPath) as another (`$position`). Path navigation is normalized as an explicit iteration with a `for` expression. Finally, the XPath predicate is normalized to a conditional expression with an explicit existential quantification and a typeswitch to implement the implicit extraction of values from a node.

The Galax compiler can rewrite that expression to a more concise and more efficient one as follows: (i) `$position` variables are bound but never used and can be removed, (iii) the attribute `id` being always a node, the first case clause in the typeswitch is always executed, (iii) assuming the schema indicates there is exactly one `id` attribute for each person, the existential quantification can be removed. This results in the following simplified expression:

```
for $dot in / return
 for $dot in child::site return
  for $dot in child::people return
   for $dot in child::person return
    if (data(attribute::id) = "person0")
    then child::name
    else ()
```

4.2 Running queries on large documents

The second important bottleneck is due to the memory overhead imposed by XML data models. As explained in [4], current main-memory XQuery implementations break for even small document sizes as they build large data models in memory before even starting query processing. In the case of Galax, the biggest document we could run on an IBM laptop with 256Mb memory was about 30 Mb.

In [4], we proposed a technique based on *document projection* to drastically reduce the size of the data model representation before query processing starts. The idea behind document projection is to analyze the query to identify the paths required to evaluate that query. In most cases, only specific paths in the documents are used, and a smaller *projected* document can be built in memory. For instance, in the case of the previous XMark query, only the nodes reachable through the two following paths are necessary:

```
/site/people/person/@id
/site/people/person/name
```

In the case of XMark, the nodes reachable with those paths represents only 2% of the original size of the document. Using that technique, we will demonstrate Galax queries running on documents of up to a Gigabyte on a simple IBM laptop with 256Mb memory.

Acknowledgments. We would like to address special thanks to the people who contributed to Galax's development: Philip Wadler, Volker Renneberg, Lori Resnick, Cindy Chen, Trevor Jim, Volker Stoltz, and M. Radhakrishnan.

References

[1] Enosys software. `http://www.enosys.com/`.

[2] 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.

[3] 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.

[4] A. Marian and J. Siméon. Projecting XML documents. Technical report, Columbia University, Computer Science Department, February 2003.

[5] Quip. `developer.softwareag.com/tamino/quip`.

[6] M. Rys. State-of-the-art XML support in RDBMS: Microsoft SQL Server's XML features. *Bulletin of the Technical Committee on Data Engineering*, 24(2):3–11, June 2001.

[7] Jérôme Siméon and Philip Wadler. The essence of XML. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, LA, January 2003.

[8] XQRL, Inc. `http://www.xqrl.com/`.

[9] XQuery 1.0: An XML query language. W3C Working Draft, November 2002.

[10] XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, November 2002.