Implementing XQuery 1.0: The Story of Galax

Mary Fernández
AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932, USA
mff@research.att.com

Jérôme Siméon
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532, USA
simeon@us.ibm.com

Abstract: XQuery 1.0 and its sister language XPath 2.0 have set a fire underneath database vendors and researchers alike. More than thirty commercial and research XQuery implementations are listed on the XML Query working group home page.

Galax [FS] is an open-source, general-purpose XQuery engine, designed to be complete, efficient, *and* extensible. During Galax's development, we have focused on each of these three requirements in turn, while never losing sight of the other two. Our success or failure in satisfying these requirements depends entirely on the design and implementation of Galax's architecture. We describe Galax's architecture in detail and identify several key principles that guide our decisions on Galax's design and implementation.

1 Introduction

For the past four years, we have been actively involved in defining XQuery 1.0 [XQ04], a query language for XML designed to meet the diverse needs of applications that query and exchange XML documents. XQuery is a strongly-typed, compositional, and functional language. XQuery is also a good "XML citizen"—it supports all of XML 1.0 [XML04] and XML namespaces [XNS04], and its type system is based on XML Schema 1.0 [XS04, XSD04]. Several books [Bru04, Kat04] and numerous articles provide excellent introductions to XQuery 1.0.

XQuery and its sister language XPath 2.0 are designed jointly by members of the World-wide Web Consortium's XSLT and XML Query working groups, which includes constituencies such as large database vendors, small middle-ware start-ups, "power" XML-user communities, and industrial research labs. Each constituency has produced several XQuery implementations, which is unprecedented for a language that is still not standardized. A current listing of XQuery implementations is on the XML Query working group home page (http://www.w3.org/XML/Query).

Our implementation of XQuery 1.0, Galax [FS], is an open-source, general-purpose XQuery engine, that has evolved to be complete, efficient, *and* extensible. Galax began as a platform for validating the XQuery formal semantics [XFS04], which required implementing the complete processing models for XML documents, XML Schema documents, and

XQuery programs[1]. For this reason, our first and only requirement for Galax was completeness. Completeness distinguishes Galax from most other research implementations of XQuery, which focus on novel query-evaluation algorithms and document representations [J+03] or that utilize existing database technologies.

Building a complete XQuery implementation has yielded several unexpected benefits. First, we were able to use Galax in applications that had non-trivial query requirements and quickly develop a user base that depended upon a complete XQuery implementation [VFS04]. Interestingly, our early users never demanded that Galax be the fastest implementation, but instead required that queries always yield the right result. Second, Galax is an ideal environment in which other researchers can evaluate their own techniques for query analysis and evaluation, document storage and indexing, etc., without having to build a complete implementation themselves. Third, we have been able to measure, identify, and address actual sources of inefficiency in a working query engine. For example, moving from a fully interpreted evaluation strategy to a semi-compiled strategy improved Galax's performance by more than one order of magnitude [RSF04].

Completeness is the foundation of Galax, but as our actual and potential user base has grown, Galax's requirements have expanded to include efficiency and extensibility. As users become familiar with XQuery, they write increasingly complex queries that may include multi-way joins and re-grouping of document content, which require non-trivial evaluation algorithms for acceptable performance. Current research in XQuery evaluation focuses on applying known and new techniques for evaluating join and group-by queries efficiently [MHM03, MHM04]. Galax incorporates and contributes to this research area.

Users always surprise implementors by using their tools in unexpected ways. Galax's users are no exception. Researchers often want to capture the output of a particular processing phase, for example, a document after schema validation or a query after static typing, and then consume that output in their own tools. For this reason, almost every phase in Galax can consume and produce a particular representation of a document, query, or type, making it possible for the architecture to be used by others in novel ways. Other examples of extensibility include adding new query rewriting rules and providing alternative implementations of Galax's data model.

Previous papers on Galax focus on specific technical issues [VFS04, MS03, FHM+04, RSF04]. In contrast, this paper describes Galax holistically. In particular, we identify the design and implementation principles that make it possible to satisfy simultaneously the completeness, efficiency, and extensibility requirements, how these principles are applied, and how they help satisfy our requirements. These principles include:

- The Galax architecture strictly separates the processing models for documents, schemata, and queries;
- Each processing model is based on one or more formal specifications;
- Each phase within a processing model is strongly typed with respect to the representation of a document, query, or type it consumes and the representation it produces.

[1] The first IPSI XQuery processor [FGO02] was also designed for this purpose.

These principles are good engineering and applying them consistently results in a flexible architecture that is both a practical tool for end users and a viable experimental platform for other researchers.

In the rest of this section, we give usage scenarios that illustrate each of the three requirements and the technical challenges in satisfying them. Sections 2 gives an overview of Galax's architecture and describes the XML and XML Schema processing models in detail. The XQuery processing model is described in Section 3. Throughout, we identify how the architecture meets each of our three requirements. In Section 4, we put all the pieces together and share some of the lessons that we have learned in building a complete XQuery implementation.

1.1 Completeness

Satisfying the completeness requirement for an expressive and sometimes complex language like XQuery poses many challenges. XQuery's numerous features include, for example, functions, modules, and XML Schema types. Even implementing basic path expressions completely and correctly is a challenge. For example, below is a simple path expression that selects all books in a catalog that have a publication date greater than 2000:

```
$cat/book[@pubdate > 2000]
```

This expression has a deceivingly complex *implicit* semantics. First, whenever an arithmetic or comparison operator is applied to a node, such as the `pubdate` attribute, the node is *atomized*, which extracts its scalar or *atomic* content. Second, comparison operators distinguish between node content that has been validated against an XML Schema type and that which has not. Assuming that the `pubdate` attribute is a validated `xs:date` value, comparing it to an integer value raises a type error, however, if the `pubdate` attribute is unvalidated, then its text content is cast to an integer before comparison. Third, when comparing two numeric values, implicit rules of type promotion and casting are applied. For example, when comparing a decimal and a float, the decimal is promoted to a float. Last, all comparison operators are existentially quantified, so if multiple atomic values are compared, then the predicate is true if any pair of values satisfies the comparison. For example, if `pubdate` contained multiple integer values, then the comparison is true if any one of those values is greater than 2000.

XQuery's implicit semantics makes querying XML documents easier for the user—he need not worry about whether an input document is validated or not, or whether a particular element occurs zero or more times. A complex implicit semantics, however, makes implementation more challenging. XQuery's implicit semantics is handled during query normalization, which is described in Section 3.

1.2 Efficiency

XQuery supports recursive functions and is therefore Turing complete, so identifying efficient evaluation plans for an arbitrary XQuery program is as difficult as optimizing a

```
for $c in distinct-values($auction/site/people/person/profile/interest/@category)
let $people :=
  for $p in $auction/site/people/person
  where $p/profile/interest/@category = $c
  return
    <personne>
      <statistiques>
        <sexe> { $p/profile/gender/text() } </sexe>
        <age> { $p/profile/age/text() } </age>
        <education> { $p/profile/education/text() } </education>
        <revenu> { fn:data($p/profile/@income) } </revenu>
      </statistiques>
    </personne>
return <categorie><id>{ $c }</id>{ $people }</categorie>
```

Figure 1: Example of grouping and element construction in XMark Benchmark Query 10

program in a general-purpose programming language. To date, database researchers have focused on efficient evaluation of the subset of XQuery that excludes functions. This subset, however, is as expressive as SQL and includes multi-way joins and group-bys, possibly over multiple documents.

Unlike in SQL, in which joins and group-by expressions can be identified syntactically, syntactically distinct expressions in XQuery may express semantically equivalent joins or group-bys. For example, the query in Figure 1 is a fragment of Query 10 from the XMark benchmark suite [SWK+02]. The XMark benchmark suite contains queries about auction items, bidders, and bids. The expression in Figure 1 re-groups all bidders in an auction by the categories in which they have bid.

In a naïve evaluation of the query in Figure 1, the input document might be scanned once for *each* category value. Efficient evaluation of this query requires first recognizing that the nested FLWORs express a group-by of people on categories and then producing an un-nested evaluation plan in which each person in the input document is examined once to determine the category groups to which the person belongs. The query compilation and optimization steps that lead to an unnested evaluation plan are described in Section 3.

1.3 Extensibility

One requirement that we did not anticipate but that has become critical to Galax is extensibility. In particular, Galax's abstract tree data model supports both real and virtual XML data sources. Galax provides two built-in implementations of the abstract tree data model: one that supports random access to XML documents stored in main memory and one for documents stored in Galax's secondary storage manager, called Jungle [VFS04]. Both these implementations can be used by applications that process native XML documents.

One custom implementation of the tree data model supports access to non-XML, semi-structured data sources. The data sources are described by PADS [FG03], a declarative data-description language that specifies ad hoc data formats, such as COBOL copy books and variable-width records, among many other formats. From PADS descriptions, the PADS compiler generates libraries and tools for manipulating data in the ad hoc format, including

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /turkey/amnty1.gif HTTP/1.0" 200 -
anx-lkf0044.deltanet.com - - [15/Oct/1997:21:13:59 -0700] "GET / HTTP/1.0" 200 3082
152.163.207.138 - - [15/Oct/1997:19:17:19 -0700] "GET /asa/china/images/world.gif HTTP/1.0" 304 -
```

Figure 2: HTTP Common-Log Format Records

```
<http-clf>
  <host><resolved>207.136.97.49<resolved></host>
  <remoteID>unauthorized</remoteID>
  <auth><unauthorized/></auth>
  <mydate>15/Oct/1997:18:46:51 -0700</mydate>
  <request>
    <meth>GET</meth>
    <req_uri>/turkey/amnty1.gif</req_uri>
    <version>1.0</version>
  </request>
  <response>200</response>
  <contentLength><unavailable/></contentLength>
<http-clf>
```

Figure 3: HTTP Common-Log Record in XML

parsing routines, statistical profiling tools, and an implementation of Galax's tree data model. The PADS compiler also supports a canonical mapping from any PADS specification into XML Schema. This mapping is quite natural, as both PADS and XML are languages for describing ordered, semi-structured data.

PADS, for example, can easily specify the HTTP common log format (CLF), which has both syntactic and structural variability. Figure 2 contains three example HTTP CLF records. From the PADS description of these records, we can derive a virtual XML representation. Figure 3 contains the virtual XML representation of the first record in Figure 2. The PADS description provides meaningful names for missing fields, e.g., "-" may denote an unauthorized user or an unavailable content length, and by mapping virtually to XML, we can examine and query those values. In Figure 3, we see that in the first CLF record, the user was unauthorized and the content length unavailable. A user of the PADS-Galax data model can then pose queries like the following, which selects the resolved hosts of CLF records whose content length is unavailable:

```
$pads/http-clf[contentLength/unavailable]/host[resolved]
```

Galax's extensible data model makes it possible for PADS users to perform simple querying tasks that are tedious to express in an imperative language and to evaluate those queries without materializing their non-XML data sources in XML. Galax's tree data model is described in Section 2.1.

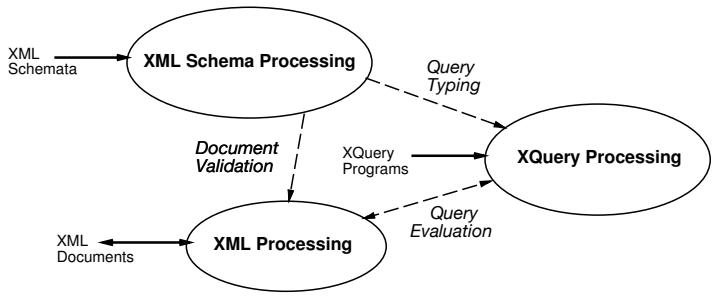

Figure 4: Processing Models

2 Galax's Architecture

Galax's architecture is closely aligned with the XQuery processing models, which are defined in several formal specifications. From the designer's perspective, this alignment guarantees that the Galax architecture implements the entire language, and from the implementer's perspective, this alignment facilitates identifying the modules that require update when the formal specifications change.

Figure 4 depicts the three mutually dependent processing models: XML document processing, XML Schema processing, and XQuery program processing, and their relationships. Processing models produce various representations of documents, schemas, and queries, and their phases relate these representations. For example, XML Schema processing takes XML Schema documents as input and produces a representation of schemata that is used by document validation, which relates schema types to XML values, and one used by static typing, which relates schema types to query expressions.

Satisfying the completeness requirement essentially forced us to design each processing model top down and and to implement each one from scratch. In particular, we did not begin with an existing technology, such as a relational storage system or an object-oriented query engine, and implement bottom-up, centered on that technology. Because we had to implement each processing model completely, we chose the simplest and most expedient implementation strategy and focused on designing the most semantically simple and transparent representations of queries, documents, and types.

The remainder of this section examines each processing model and their phases in detail and describes several of Galax's representations of queries, documents, and types.

2.1 XML Document Processing Model

Figure 5 depicts Galax's XML processing model. This processing model takes native XML documents as input and produces native XML documents as output, and it implements four related specifications: XML 1.0, the XML Infoset [XIn04], the post-schema validated infoset [XS04], and the XSLT 2.0 and XQuery 1.0 Serialization [XSe04]. Its

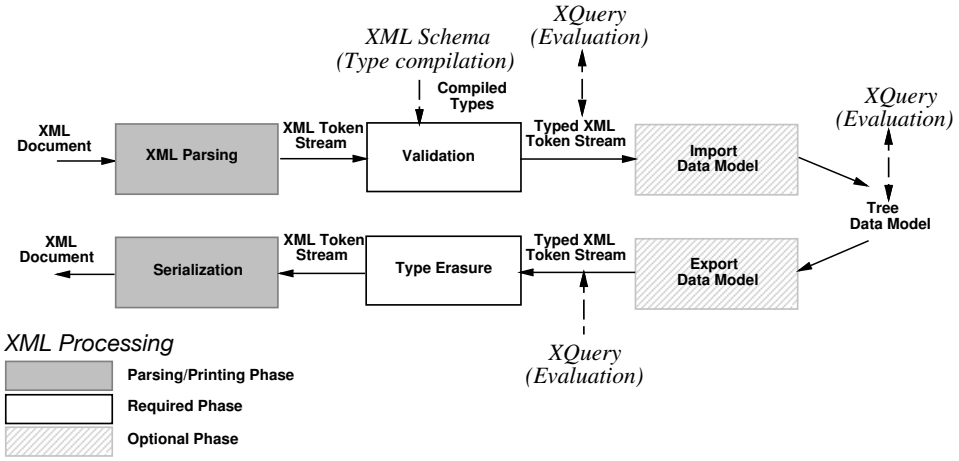

Figure 5: XML document processing architecture

phases consume and produce two representations of documents: an XML token stream, which provides sequential access to a document, and an instance of Galax's abstract tree data model, which provides random access.

The "in-bound" phases include document parsing, validation, and importing a validated document into the tree data model. The "out-bound" phases are the duals of the in-bound phases: exporting a data-model instance, type erasure, and document serialization. Note that document validation is a required phase — for well-formed documents with no associated schema, validation assigns default types. Importing into the tree data model is an optional phase, however, because queries sometimes can be evaluated directly on a typed XML token stream.

XML tokens are similar to SAX events but unlike SAX events, which are typically implemented as call-back functions, XML token streams are consumed by phases that pull XML tokens on demand from previous phases. Figure 6 contains a document fragment and Figure 7 depicts its representation as a typed XML token stream after validation. A `startDoc` token has no arguments, whereas a `startElem` token takes the resolved qualified name (QName) of the element and a set of attributes. For example, the start-element token corresponding to the `ns:book` element contains the resolved QName `ns{http://book.xsd}book` and one `pubdate` attribute. Note that the representation of a resolved QName contains a prefix, a URL, and a local name. This representation is independent of the scoped namespace declarations in the original document, which permits the query processor to treat nodes and QName values as context-independent values.

Validation takes an untyped XML token stream and adds type annotations and typed atomic values. Default types are added for documents that have no associated schema. In Figure 7, the node's type and its typed atomic value are in bold. For example, the `pubdate` attribute has type `xsd:integer` and its typed value is the integer 1994. A typed XML stream is consumed directly by the evaluation phase of the XQuery processing

```
<ns:catalog
  xmlns:ns="http://book.xsd">
  <ns:book pubdate="1994">
    <title>TCP/IP Illustrated</title>
    <author>Stevens</author>
  </ns:book>
  ...
</ns:catalog>
```

Figure 6: Example XML document fragment

```
startDoc
  startElem ns{http://book.xsd}:catalog  {} element(ns:catalog) ()
    startElem ns{http:...}:book {({}:pubdate,"1994",xsd:integer,xsd:integer(1994))}
                  element(ns:book) ()
      startElem empty{}:title {} element(empty:title) xsd:string("TCP/IP Illustrated")
        text ("TCP/IP Illustrated")
      endElem
      startElem empty{}:author {} element(empty:author) xsd:string("Stevens")
        text ("Stevens")
      endElem
    endElem
    ...
  endElem
endDoc
```

Figure 7: Representation of fragment as typed XML token stream

model and/or by a data-model import phase, which produces an instance of the abstract tree data model.

Figure 2.1 gives the functional interface for the abstract tree data model. Each node in the tree data model (document, element, attribute, and text) have accessors that return its name, base URI, type, typed value, unique node identifier, and global document order. The axis accessors return cursors of nodes in document order, e.g., the `children` accessor returns those nodes directly accessible from a node. Like XML token streams, node cursors are materialized on demand when a phase consumes a node from the cursor. This permits an implementation of the abstract data model to materialize nodes lazily. The `parent`, `children`, and `attributes` accessors are virtual, meaning that an implementation must provide them. The other four axes have default implementations that may be overridden.

The abstract tree data model is the minimum interface necessary to support XQuery, and this minimality helped us easily meet the completeness and extensibility requirements. Other implementations of XQuery have focussed on more complex data-model interfaces by providing, for example, special-purpose axis indices. Special-purpose indices can improve performance, but also propagate complexity throughout the engine, because the query engine must be customized to take advantage of them. Moveover, complex data-model interfaces make it more difficult for others to provide their own implementations, which limits extensibility.

The XQuery processing model may consume either or both of the XML token stream or tree representations during evaluation. The representation chosen depends in part on the

```
virtual node : object inherit item
  (* Infoset accessors *)
  method virtual name        : unit -> atomicQName option
  method virtual base_uri    : unit -> atomicString option

  (* PSVI accessors *)
  method virtual type        : unit -> atomicQName
  method virtual typed_value : unit -> atomicValue cursor

  (* Node identity *)
  method virtual nodeid    : unit -> Nodeid.nodeid
  method virtual docorder  : unit -> Nodeid.docorder

  (* Axes *)
  method virtual parent       : unit -> node option
  method virtual children     : unit -> node cursor
  method virtual attributes   : unit -> attribute cursor
  method descendant_or_self   : unit -> node cursor
  method descendant           : unit -> node cursor
  method ancestor_or_self     : unit -> node cursor
  method ancestor             : unit -> node cursor
```

Figure 8: Abstract tree data model

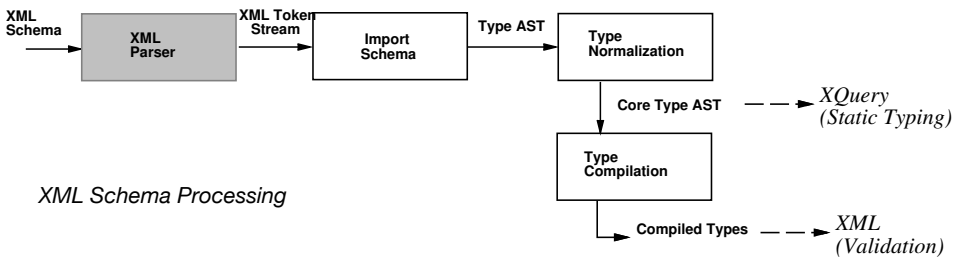

Figure 9: XML Schema processing architecture

query optimizer's ability to determine whether a query expression can be evaluated efficiently on an XML token stream or whether the tree data model is necessary. Regardless of the query evaluated, large XML documents ($>$ 100MB) are typically imported into the tree data model off-line and stored in Jungle. Smaller documents are typically imported into the tree data model during evaluation if query evaluation requires full random-access to the document.

2.2 XML Schema Processing Model

Figure 9 depicts Galax's XML Schema processing model. This processing model takes XML Schema documents as input and produces two representations of the schemata, which are used by the XML document and XQuery processing modules. Although this processing model has the fewest phases, it implements some of XQuery's most complex semantics, in particular, the formalization of XML Schema defined by Siméon and Wadler [SW03]. The static type system defined in the XQuery 1.0 Formal Semantics [XFS04]

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://book.xsd"
            xmlns="http://book.xsd">
  <xsd:element name="catalog" type="CatalogType"/>
  <xsd:element name="book" type="BookType"/>
  <xsd:complexType name="CatalogType">
    <xsd:sequence>
      <xsd:element ref="book" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:group name="BookGroup">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="author" type="xsd:string"/>
    </xsd:sequence>
  </xsd:group>
  <xsd:complexType name="BookType">
    <xsd:sequence>
      <xsd:group ref="BookGroup"/>
    </xsd:sequence>
    <xsd:attribute name="pubdate" type="xsd:date"/>
    <xsd:attribute name="isbn" type="ISBNType"/>
  </xsd:complexType>
  <xsd:simpleType name="ISBNType">
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
</xsd:schema>
```

Figure 10: Example XML Schema

is based on this formalism and is implemented in the static typing phase of the XQuery processing model.

XML Schema has an XML syntax, so like any other XML document, the schema is parsed yielding an XML token stream. The schema-import phase takes the XML token stream, checks for syntactic and semantic correctness, and if the schema is valid, produces the abstract-syntax tree of the XQuery types that correspond to the schema.

Like expressions in the XQuery language, type expressions in XML Schema have an implicit semantics. Type normalization takes an XML Schema type and makes the implicit semantics explicit in the simpler, orthogonal XQuery Core Type language. The orthogonality of the Core Type language simplifies static typing, because unique types can be computed for each expression.

To illustrate type normalization, Figure 10 contains a simple XML Schema document, and Figure 11 contains the corresponding Core type after normalization. The highlighted reference to the group `BookGroup` in the complex type `BookType` is expanded in the normalized type, because groups are purely syntactic constructs. The distinction between complex and simple types (e.g., `ns:CatalogType` and `ns:ISBNType`) disappears in the Core Type language: All types have a name, a derivation from another named type, and a content type expression. The restrictions placed by XML Schema on type content also disappear in the Core Type language. The content type expression is a regular tree expression over node types, which are combined with the sequence (`,`), choice (`|`), repetition (`*`), and interleaving (`&`) operators.

During static typing, expressions are annotated with a Core type, because it is a useful

```
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
declare namespace ns = "http://book.xsd";
declare namespace empty = "";

declare element ns:book of type ns:BookType;
declare element ns:catalog of type ns:CatalogType;

declare type ns:CatalogType restricts xs:anyType {
  element ns:book*
};
declare type ns:BookType restricts xs:anyType {
  attribute empty:pubdate of type xs:date ? &
  attribute empty:isbn    of type ns:ISBNType ?;
  element   empty:title   of type xs:string,
  element   empty:author  of type xs:string
};
declare type ns:ISBNType restricts xs:string;
```

Figure 11: Representation of XML Schema in Core type language

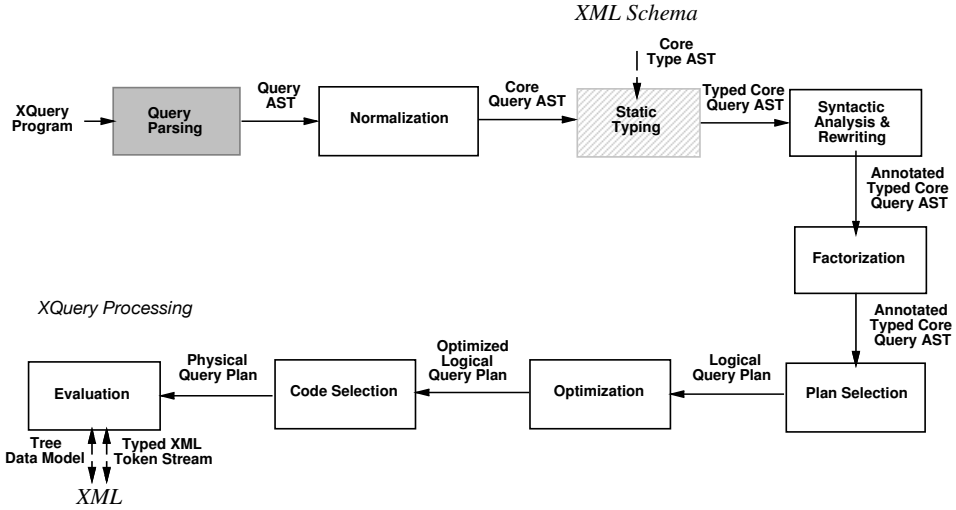

Figure 12: XQuery processing architecture

representation for reporting errors and easily permits others to consume the output of the static typing phase. The Core type ast is also compiled into a compact representation used during validation to annotate XML token streams. The compiled type representation is space efficient and permits fast comparison of type values at runtime.

3 XQuery Processing Model

The XQuery processing model is the heart of Galax engine. Figure 12 depicts this model, which takes as input an XQuery program consisting of one main module and zero or more

```
fs:distinct-docorder(for $_c in $cat return
  for $_b in fs:distinct-docorder($_c/child::ns:book) return
    where
      some $v1 in fn:data($_b/attribute::pubdate) satisfies
      some $v2 in fn:data(2000) satisfies
        let $u1 := fs:promote-operand($v1,$v2) return
        let $u2 := fs:promote-operand($v2,$v1) return
        op:ge($u1, $u2)
      return $_b
  )
```

Figure 13: Core AST of `$cat/ns:book[@pubdate >= 2000]` after normalization

library modules and produces one output value as an instance of the XQuery data model. A library module contains a prolog (e.g., schema import and function declarations). A main module contains a prolog and one expression, whose value is the result.

Because XQuery is both a functional language and a query language, Galax's architecture incorporates phases from functional-language compilers [App98] and query-language engines [GMUW02]. The top half of Figure 12 is analogous to the front end of a functional-language compiler, which annotates and transforms AST representations of a program (query) before code generation and instruction selection (plan and code selection). The bottom half of Figure 12 corresponds to the back end of a query-language engine, which takes a Core AST and transforms it into an executable evaluation plan. The remainder of this section describes each phase.

3.1 Parsing and Normalization

Parsing takes an XQuery program and produces an abstract syntax tree (AST) of the XQuery language and is defined by XQuery's grammar rules [XQ04]. Normalization takes a query AST and rewrites it into a semantically equivalent AST in the smaller Core XQuery language. As described in Section 1.1, normalization makes the implicit semantics of the surface syntax explict in the Core language. Galax's normalization phase is an almost literal interpretation of the formal normalization rules.

Figure 13 contains the (slightly sanitized) normalized Core expression corresponding to the example `$cat/ns:book[@pubdate >= 2000]`. The Core expressions in bold capture some of the expression's implicit semantics. The `fn:data` function applies atomization to a sequence of nodes; the `some-satisfies` expression is existential quantification; the `fs:promote-operand` function applies the appropriate type promotion and casting rules; the overloaded, polymorphic comparison operator `op:ge` is applied, because no type information is available during normalization; and, finally, the `fs:distinct-docorder` function guarantees that the each step of the path expression and the final result is in document order with no duplicates.

If this expression were interpreted literally, it would be quite inefficient, nonetheless it captures the expression's complete semantics. Subsequent phases prepare the Core expression for compilation into an efficient query plan.

```
fs:distinct-docorder(for $_c [element(ns:catalog)] in $cat [element(ns:catalog)] return
  for $_b [element(ns:book)] in $_c/child::ns:book [element(ns:book)*] return
    where
      some $v1 in (fn:data($_b/attribute::pubdate [attribute(pubdate)]) [xs:integer]) satisfies
      some $v2 in fn:data(2000) [xs:integer] satisfies
        let $u1 [xs:integer] := fs:promote-operand($v1,$v2) return
        let $u2 [xs:integer] := fs:promote-operand($v2,$v1) return
        op:ge($u1, $u2) [xs:boolean]
      return $_b [element(ns:book)?]
  [element(ns:book)*]
)[element(ns:book)*]
```

Figure 14: Typed Core AST of `$cat/ns:book[@pubdate >= 2000]` after static typing

3.2 Static Typing

Static Typing takes a normalized Core AST, infers the static type of each expression in the AST, and annotates each expression with its static type. Static typing is defined in the XQuery Formal Semantics, and Galax's implementation of each typing rule, like each normalization rule, is a near-literal interpretation of the formal definition.

Figure 14 contains the typed Core expression of our example annotated with types from the normalized schema in Figure 11. [2] Assuming that the $cat variable is bound to one `ns:catalog` element, which may be inferred from a query prolog or provided by the context of the programming environment, the types of other expressions are inferred by applying XQuery's static typing rules.

Given XQuery's complex implicit semantics, static typing is especially important, because it enables expression simplification and generation of more efficient evaluation plans in later phases. Even the default types assigned to expressions when no input schema is available can be useful during simplification. For example, the default annotation for the variable $_b is a single `element`, because the child axis always yields a sequence of elements and a for-bound variable is always bound to a single item, or element in this case. Even without the knowledge that $_b is a book element, knowing that it is always a single element can be useful.

3.3 Syntactic Analysis and Rewriting

The syntactic analysis and rewriting phase applies many simplification rules standard in compilers for functional languages, such are removing unused variable definitions, inlining of non-recursive functions, and applying type-aware rewritings. This phase also applies analyses and rewritings that are unique to XQuery. One analysis detects when intermediate steps in path expressions always yield nodes in document order and with no duplicates, and the corresponding rewrite rule eliminates redundant `fs:distinct-docorder` operations [FHM+04]. Other analyses identifies the data source(s) to which each expression

[2] This is not valid XQuery syntax, but is just meant to illustrate the type annotations associated with expressions.

```
for $_b [element(ns:book)] in $cat/child::ns:book [element(ns:book)*] return
  where (op:integer-ge(fn:data($_b/attribute::pubdate), 2000) [xs:boolean])
    return $_b [element(ns:book)?]
[element(ns:book)*]
```

Figure 15: Simplified Core AST of `$cat/ns:book[@pubdate >= 2000]` after syntactic analysis and rewriting

is applied and the paths applied to each data source [MS03]. The overall goal is to produce the simplest Core expression that is semantically complete and to provide annotations that can facilitate plan selection and optimization in subsequent phases.

Figure 15 contains the typed Core expression of our example after syntactic rewriting. The type-aware rewrite rules make several simplifications. For example, the first `for` expression is eliminated, because its type `ns:catalog` is a singleton type and therefore iteration is unnecessary. The two existential quantifications are eliminated because the constant 2000 and the contents of a `pubdate` attribute are both singleton integers, and the overloaded comparison operator `op:ge` is replaced by an integer-comparison operator. The document-order analysis determines that the path expression `$cat/ns:book` is always in document order with no duplicates, so the `fs:distinct-docorder` operations are also eliminated.

3.4 Factorization and Plan Selection

The factorization phase prepares an AST for plan selection, which takes an AST and produces a naïve evaluation plan in Galax's algebra [RSF04]. Factorization reduces syntactic variability so that plan selection can recognize common idioms that express, for example, joins and group-bys. The factorized representation of our simple example is equivalent to its simplified representation.

Galax's algebra extends an existing algebra [MHM04] to support all of XQuery and includes, among others, operations on XML types and support for user-defined functions. The algebra is described elsewhere [RSF04], but we note that the it includes operators on tuples (e.g., MapConcat, Select), operators on XML items (e.g., Step, TreeProject) and operators that convert between the two (e.g., MapToItem, MapFromItem).

Figure 16 contains the naïve plan for our example expression. The plan is read "inside-out", beginning with the inner most operator. Given that syntactic analysis inferred the input document to be "catalog.xml" and that it is validated against element `ns:catalog`, the two inner-most operators parse and validate the document. Assuming that path projection was applied during analysis, plan selection can introduce the tree-project operator, which takes a typed XML stream and a tree of paths and yields a typed XML stream that only contains nodes on the specified paths. In this case, all book elements and their descendants are projected. The two subsequent tuple constructors and concatenation operator (++) create a tuple containing the catalog element paired with each book element. The select operator applies its predicate to the input tuple, and the last operator maps the selected tuples back to items.

```
MapToItem
  {Input -> (Input#glx:b_3)}
  (Select
     {Call{op:integer-ge}(
        Call{fn:data}(
          Step{attribute::pubdate}(Input#glx:b_3)),
        Scalar{2000}())}
     (MapConcat
        {MapFromItem
          {glx:comp2 -> Tuple[glx:b_3 : $glx:comp2]}
          (Step{child::ns:book}(Input#glx:c_1)) ++ Input}
        (MapFromItem
           {glx:comp0 -> Tuple[glx:c_1 : $glx:comp0]}
           (TreeProject
              {./ns:book/*}
              (Validate
                {element(ns:catalog)}
                (Parse{"catalog.xml"}()))))))
```

Figure 16: Algebraic plan of `$cat/ns:book[@pubdate >= 2000]`

3.5 Optimization and Code Selection

Query optimization follows plan selection and attempts to improve the default evaluation plan. Common optimizations include query unnesting, which identifies group-bys expressed by nested for-expressions (MapConcat operators) and produces a plan with unnested group-by operators. Pushing selections early in a plan is another example. After optimization, the Select operator in Figure 16 can be pushed inside the first MapFromItem operator.

The last phase before evaluation is code selection, which takes a particular logical operator, such as a Join, and selects a particular implementation for that operator, such as sort merge or hash join. The result is a physical query plan in which every operator and function has been realized by a concrete implementation.

3.6 Evaluation

Finally, evaluation takes a physical plan, evaluates the plan, and produces either a typed XML stream or instance of the tree data model. The result value can be accessed and navigated using Galax's API to its tree data model from a C, Java, or O'Caml program, or the result can be serialized into an XML document.

4 Lessons Learned

We conclude with several lessons that we have learned building Galax.

Our tour of Galax's three processing models shows that most of its architecture is devoted to *transforming* representations of documents, schemas, and queries. Of the 110,000 lines of Galax's source code written in O'Caml [OCa], only 9755 lines (11%) is devoted to

query evaluation. We found that implementing Galax in O'Caml was crucial to Galax's successful development. O'Caml is a member of the ML family of programming languages; ML means *meta-language*, that is, a language for defining and transforming other languages, which is Galax's central task. In particular, we use O'Caml's polymorphic algebraic types extensively—they implement every query representation from the original AST to the physical query plan and the XML token representations of documents. We also extensively use O'Caml's higher-order functions and lazy evaluation of functions to support delayed consumption of XML token streams and node cursors.

We found that a formal model is a good foundation for an initial architectural design. The close alignment between XQuery's formal specifications and Galax's processing models guaranteed that Galax satisfied the completeness requirement, even though early versions were not particularly efficient or extensible. The close alignment also simplified keeping Galax up to date as XQuery's design changed continuously.

Focussing on completeness helped us attract early adopters of XQuery, which in turn, required us to focus on efficiency. Early adopters tend to be demanding, because existing technologies are not addressing their problems satisfactorily. Our users write complex queries that stress Galax in every way and help us identify real inefficiencies. These observations have lead to several interesting research problems on XML storage [VFS04], document projection [MS03] and more recently query optimization [FHM+04, RSF04], and resulted in lasting improvements to Galax's architecture.

Satisfying the extensibility requirement has resulted, not surprisingly, in a more modular architecture. A common mistake when building a large compiler is to coalesce phases that are logically separate, in an attempt to reduce the cost of manipulating many representations of one program. For example, query parsing and normalization or document parsing and validation could easily be coalesced with some reduction in code size. Coalescing phases, however, results in a less modular and therefore less extensible architecture. A strict separation of phases makes it possible for other researchers to co-opt parts of Galax's architecture, which increases its value to the research community.

We found that Galax's completeness and extensibility requirements facilitated our ability to work on novel research problems, such as designing an update language for XQuery [SHS04], designing a Web-services programming language [OS04, FOS04] based on XQuery, and implementing XQuery's full-text search operators [AYBCF04]. Both the Web-services programming language and the full-text extensions depended on XQuery's schema import and library import features, which are not widely implemented. We could not have worked on these problems without access to a complete and extensible implementation.

Our most difficult lesson has been realizing and accepting that only 15-20% of our time spent working on Galax involves genuinely novel research. Most of our time is spent on project management, e.g., fixing bugs, writing documentation, porting to various architectures, and on long-term systems engineering, i.e, re-designing, re-implementing, and refining the existing architecture as we encounter new problems and applications. Despite the high "research cost" of working on Galax, we have found that the research problems we do encounter are often original and their solutions have a practical impact, because we are working with complex queries in real applications. Ultimately, access to real users and

their XML applications motivates us to continue working on Galax.

Acknowledgements. We thank the Galax team, past and present: Cindy Chen, Byron Choi, Vladimir Gapeyev, Jan Hidders, Amélie Marian, Philippe Michiels, Nicola Onose, Douglas Petkanics, M. Radhakrishnan, Christopher Re, Lori Resnick, Michael Stark, Gargi Sur, Roel Vercammen, Avinash Vyas, and Philip Wadler.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in C/Java/ML*. Cambridge University Press, 1998.

[AYBCF04] S. Amer-Yahia, P. Brown, E. Curtmola, and M. Fernández. GalaTex: A Conformance Implementation of the XQuery Full-Text Language, 2004. Submitted for publication.

[Bru04] Michael Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

[FG03] Kathleen Fisher and Robert Gruber. PADS : Processing Arbitrary Data Streams. In *Workshop on Management and Processing of Data Streams*, June 2003.

[FGO02] Peter Fankhauser, Tobias Groh, and Sven Overhage. XQuery by the Book: The IPSI XQuery Demonstrator. In *Proceedings of the International Conference on Extending Database Technology*, pages 742–744, Prague, Czech Republic, March 2002.

[FHM+04] Mary Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Automata for Avoiding Unnecessary Ordering Operations in XPath Implementations. Technical Report 2004-02, University of Antwerp and AT&T Research and IBM Watson, 2004. http://www.adrem.ua.ac.be/pub/vldb04-techrep.pdf.

[FOS04] M. Fernández, N. Onose, and J. Siméon. Yoo-Hoo! Building a Presence Service with XQuery and WSDL. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 911–912, 2004. Demonstration program.

[FS] Mary Fernández and Jérôme Siméon. Galax : The XQuery Implementation for Discriminating Hackers. http://www.galaxquery.org.

[GMUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2002.

[J+03] H.V. Jagdish et al. Timber: A Native XML Database. *Proceedings of ACM Conference on Management of Data (SIGMOD)*, page 672, June 2003.

[Kat04] Howard Katz, editor. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2004.

[MHM03] Norman May, Sven Helmer, and Guido Moerkotte. Three Cases for Query Decorrelation in XQuery. In *XSym 2003*, pages 70–84, Berlin, Germany, September 2003.

[MHM04] Norman May, Sven Helmer, and Guido Moerkotte. Nested Queries and Quantifiers in an Ordered Context. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 239–250, Boston, MA, March 2004.

[MS03] Amélie Marian and Jérôme Siméon. Projecting XML Documents. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 213–224, Berlin, Germany, September 2003.

[OCa] The O'Caml Programming Language. http://www.ocaml.org/.

[OS04] Nicola Onose and Jérôme Siméon. XQuery at Your Web Service. In *Proceedings of International World Wide Web Conference*, New York, New York, September 2004.

[RSF04] Christopher Re, Jérôme Siméon, and Mary Fernández. A Complete and Efficient Algebraic Compiler for XQuery, November 2004. Submitted for publication.

[SHS04] Gargi Sur, Joachim Hammer, and Jérôme Siméon. UpdateX - An XQuery-Based Language for Processing Updates in XML. In *PLAN-X: Programming Language Technologies for XML*, Venice, Italy, January 2004.

[SW03] Jérôme Siméon and Philip Wadler. The Essence of XML. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, January 2003.

[SWK+02] A. Schmidt, F. Waas, M. Kersten, M. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 974–985, Hong Kong, China, August 2002. http://monetdb.cwi.nl/xml/.

[VFS04] Avinash Vyas, Mary F. Fernández, and Jérôme Siméon. The Simplest XML Storage Manager Ever. In *XIME-P 2004*, pages 37–42, Paris, France, June 2004.

[XFS04] XQuery 1.0 and XPath 2.0 Formal Semantics, W3C Working Draft, Aug 2004. http://www.w3.org/TR/query-semantics.

[XIn04] XML Information Set. W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-infoset-20040204/.

[XML04] Extensible markup language (XML) 1.0. W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

[XNS04] Namespaces in XML 1.1. W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-names11-20040204/.

[XQ04] XQuery 1.0: An XML Query Language. W3C Working Draft, October 2004. http://www.w3.org/TR/2004/WD-xquery-20041029/.

[XS04] XML Schema Part 1: Structures. W3C Recommendation, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[XSD04] XML Schema Part 2: Datatypes. W3C Recommendation, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[XSe04] XSLT 2.0 and XQuery 1.0 Serialization. W3C Working Draft, October 2004. http://www.w3.org/TR/2004/WD-xslt-xquery-serialization-20041029/.