A Complete and Efficient Algebraic Compiler for XQuery

Christopher Ré
University of Washington

Jérôme Siméon
IBM T.J. Watson Research Center

Mary Fernández
AT&T Labs - Research

Abstract

As XQuery nears standardization, more sophisticated XQuery applications are emerging, which often exploit the entire language and are applied to non-trivial XML sources. We propose an algebra and optimization techniques that are suitable for building an XQuery compiler that is complete, correct, and efficient. We describe the compilation rules for the complete language into that algebra and present novel optimization techniques that address the needs of complex queries. These techniques include new query unnesting rewritings and specialized join algorithms that account for XQuery's complex predicate semantics. The algebra and optimizations are implemented in the Galax XQuery engine, and yield execution plans that are up to three orders of magnitude faster than earlier versions of Galax.

1. Introduction

As XQuery [21] nears standardization, more sophisticated applications are emerging [2, 16, 17, 20]. From our experience working with developers, new XQuery applications often utilize the entire language (e.g., schema validation; modules), rely on complex aspects of XQuery's semantics (e.g., implicit conversions in predicates; sorting by document order), and are applied to non-trivial XML sources. Consequently, they require XQuery implementations that are at the same time complete, correct, and efficient. In this paper, we describe an algebra and optimization techniques that are suitable for building an XQuery compiler that satisfies all these requirements.

The queries generated by the Clio project [16] provide a good example of the complexity that XQuery implementations must handle. Clio allows users to specify mappings between multiple schemata in a declarative way. Figure 1 shows the Clio user interface with a simple mapping from the schema for the DBLP database to the schema for an authors database[1]. Below that interface is the query generated to transform data instances from one schema to the other. This query uses a large subset of XQuery including nested FLWOR blocks with joins, path navigation, element construction, and function calls. More advanced mappings quickly result in very complex nested queries with multi-way joins for which a naive nested-loop evaluation is unacceptably slow.

[1]Screenshot and Clio queries are courtesy of the Clio team.

A number of query processing and optimization techniques have been developed for subsets of XQuery, including techniques for XPath evaluation [4, 5, 9, 10], XQuery unnesting [7, 14], XML streaming [1, 8, 13], and bulk tree evaluation [6, 9, 15]. However, none of these techniques are integrated into XQuery implementations that are capable of running the Clio queries. Few available XQuery implementations, with the notable exceptions of XQRL [8] and Saxon [11], are complete and provide good performance. XQRL is specifically designed for XML stream processing and does not perform as well on large data sets or on nested queries (See experiments in [8]). Our experimental evaluation shows that on Clio queries, our techniques result in query plans that outperform Saxon by an order of magnitude. The main obstacle to using existing optimization techniques in a complete processor is the absence of an algebra that supports the whole XQuery language and that is amenable to applying these optimization techniques.

Existing algebras for XQuery fall into two camps. On one side, tuple-based algebras [7, 14] facilitate the use of relational optimization techniques. On the other side, tree-based algebras [6] provide more natural support for novel XML-specific optimizations. Our algebra borrows ideas from both camps, extending a large fragment of the tuple-based algebra from [14] with XML-specific operators. Both fragments include new operators (e.g., type operations, positional maps) that are necessary for complete coverage of XQuery 1.0. Our approach also encompasses the work on algebraic compilation of XPath 1.0 presented in [4], extending it to support path expressions in the context of arbitrary XQueries. Our algebra bares some similarity to the unified algebra proposed by Manolescu and Papakonstantinou [12] for classifying XQuery algebras. To the best of our knowledge, our work is the first algebraic treatment of XQuery that formalizes compilation of the complete language. We believe that other XML optimizations from the literature can be integrated into the resulting framework.

In this paper, we focus on the logical algebra and compilation rules, along with query unnesting and join optimizations that are critical for Clio queries.

The paper makes the following technical contributions:

- We propose a complete algebra for XQuery 1.0, and describe how to compile the complete XQuery language into that algebra.
- We define a group-by operation that is specifically designed to support unnesting rewritings for XQuery, along with new logical rewritings that make unnesting optimizations more robust for complex queries.
- We develop new join algorithms that account for XQuery's complex predicate semantics including existential quantification, atomization, implicit casting of untyped values, and type promotion.
- We present an experimental evaluation of the compiler that demonstrates its effectiveness on a wide range of queries including those generated by Clio.

The compiler and optimizations have been implemented in Galax 0.5.0[2]. The compiler passes XQuery regression tests, and its optimized plans are up to two orders of magnitude faster than the naïve evaluation strategy. On Clio queries, the resulting compiler is three orders of magnitude faster than previous versions of Galax and is one order of magnitude faster than Saxon 8.1.1.

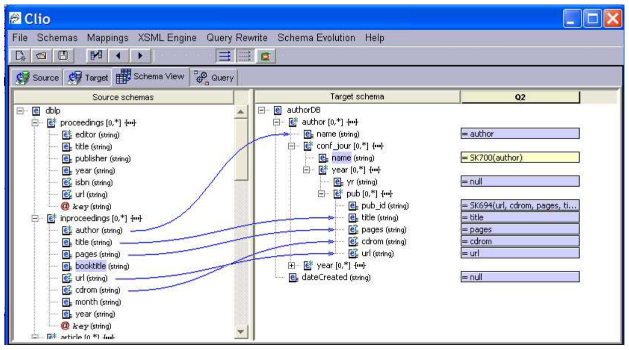
Section 2 illustrates the proposed compilation and optimization techniques on an example and identifies the most important differences with previous work. Section 3 presents the algebra and Section 4 describes the compilation rules for XQuery 1.0. Section 5 presents the logical optimizations and Section 6 describes the join algorithms for XQuery. Section 7 presents an experimental evaluation of these techniques. Section 8 concludes the paper.

2. A Complete Example

We use the following variant of XMark's Query 8 to illustrate our techniques. This query builds one `item` element for each person, containing his name and the number of artifacts he acquired from a `USSeller`. The type test on line 7 and type assertion on line 2 assume the existence of an XML schema and of schema-validated data.

(Q1)
```
1.  for $p in $auction//person
2.  let $a as element(*,Auction)* :=
3.    for $t in $auction//closed_auction
4.    where $t/buyer/@person = $p/@id
5.    return validate { $t }
6.  return <item person="{ $p/name }">
7.           { count($a/element(*,USSeller)) }
8.         </item>
```

[2]`http://www.galaxquery.org/distrib.html`



```
(: Clio Query Generator for XQuery 1.0 :)
let $doc0 := document("DBLP-after98.xml")
return
<authorDB>
 { clio:deep-distinct (
 for $x0 in $doc0/dblp/inproceedings,
     $x1 in $x0/author
 return
  <author>
   <name>{ $x1/text() }</name>
   <conf_jour>
    <name>{  "SK700(", $x1/text(), ")" }</name>
    <year>
     <yr/>
   { clio:deep-distinct (
     for $x0L1 in $doc0/dblp/inproceedings,
         $x1L1 in $x0L1/author
     where $x1/text() = $x1L1/text()
     return
      <pub>
       <pub_id>{ "SK694(",$x0L1/..., ")"}</pub_id>
       <title>{ $x0L1/title/text() }</title>
       <pages>{ $x0L1/pages/text() }</pages>
       <cdrom>{ $x0L1/cdrom/text() }</cdrom>
       <url>{ $x0L1/url/text() }</url>
      </pub> ) }
    </year>
   </conf_jour>
  </author> ) }
 <dateCreated/>
</authorDB>
```

Figure 1. Using XQuery in Clio

Although simpler than the query in Figure 1, this query exhibits most of the features that make optimization of Clio queries difficult. First, it uses a nested FLWOR (lines 3–6) to compute the items purchased in a closed auction. A typical plan for such a query is an outer-join between the persons and auctions followed by a group-by on the persons. However, none of the existing query unnesting techniques [7, 14] can handle this query because of the presence of XML operations (the type assertion on `$a` on line 2, and validation on line 5) in the nested FLWOR block.

Algebraic compilation of XQuery. The first challenge is to compile the query into a representation that makes traditional join and unnesting optimizations possible. This typically means compiling the iterators in the query (here FLWOR expressions) into a tuple-based algebra. The initial plan after naïve compilation into our algebra is shown below. For readability, we do not expand the plans for XPath expressions or the equality predicate. The reader is not expected to fully understand the details of the plan at this point, but we use it to illustrate the main ideas of our approach. The algebra and the corresponding notations are introduced in the next section.

```
1. MapToItem                                          (P1)
2.   {Element[item]
3.      (Sequence
4.         (Attribute[person]((IN#p)/name/text()),
5.         count(IN#a/element(*,USSeller))))}
6.   (MapConcat
7.      {[a:TypeAssert[element(*,Auction)*]
8.             (MapToItem{Validate(IN#t)}
9                  (Select
10.                    {IN#t/buyer/@person = IN#p/@id}
11.                    (MapConcat{MapFromItem{[t:IN]}
12.                           ($auction//closed_auction)}(IN))))]}
13.     (MapConcat{MapFromItem{[p:IN]}($auction//person)}([])))
```

The plan is a tree of operators and is read from the bottom up. The plan contains operators over both tuples and XML trees. Each operator has a name (e.g., Select), some independent (or input) operators delimited by (), and some dependent sub-operators delimited by {}, whose input is denoted by IN. The path expressions accessing the input data are on lines 12 and 13. The MapConcat is a dependent join equivalent to the D-Join operator in [4]. MapToItem and MapFromItem are maps used at the boundary between the fragments of the plan that process tuples and those that process XML values. Combinations of MapFromItem and MapConcat are used to create the proper tuple stream (lines 13, 11 and 6) for the FLWOR expressions in the query. Most other operators, such as tuple construction (`[p:IN]`), tuple access (#p), tuple selection (Select), or XML type matching (TypeAssert), and validation (Validate) have the obvious semantics. Finally, the new elements composing the result are constructed in the last part of the plan (lines 2-5) using the Element operator.

An important difference with [14] is that we do not convert sequences of items into sequences of singleton tuples. Instead, our algebra treats sequences of items as holistic values, making it easier to apply XML operators. For instance, the TypeAssert operator on line 7 applies directly to the item sequence produced by the MapToItem operator on line 8. This approach eliminates the need for additional nest(unnest) operators to recover(flatten) item sequences, which complicate the plan and make unnesting rewritings harder to detect. We describe the compilation rules from XQuery to our algebra in Section 4.

Detecting opportunities for unnesting. The second challenge is how to detect opportunities for query unnesting. Logical rewritings proposed by May et al [14] are very general, but often rely on complex combinations of operators (usually a map over a certain class of selection predicate). When dealing with complex queries, such combinations of map and selection do not appear *in situ*, but are almost always interleaved with other operators. In our example, MapFromItem with Validate and TypeAssert occur between the MapConcat on line 6 and the Select on line 9. In Section 5, we propose logical rewritings that make the introduction of the group-bys more robust to such variations in the query plan.

A custom XQuery group-by operator. We also propose a group-by operator specifically adapted to XQuery's semantics. Previous work, notably [14], has relied on standard nested-relational group-bys, which typically produce tuple partitions based on a grouping criteria. In the case of XQuery, it makes more sense to apply the type assertion `element(*,Auction)*`, as well as the path navigation and the `count`, directly on each partition inside the group-by. This calls for an XML-specific group-by in which each partition contains sequences of items instead of tuples of individual items. Our final plan, which uses this XML-specific GroupBy operator is:

```
1. MapToItem                                                        (P2)
2.    {Element[item]
3.        (Sequence
4.           (Attribute[person]((IN#p)/name/text()),
5.              count(IN#a/element(*,USSeller))))}
6.    (GroupBy[a,index,null]
7.        {TypeAssert[element(*,Auction)*](IN)}
8.        {Validate(IN#t)}
9.       (LOuterJoin[null]{(IN#t)/buyer/@person = (IN#p)/@id}
10.          (MapIndexStep[index]
11.              (MapFromItem{[p : IN]}($auction//person)),
12.           MapFromItem{[t:IN]}($auction//closed_auction))))
```

This plan features the expected GroupBy (line 6) and LOuterJoin (line 9). An important difference from the general purpose group-by used in [14] is that our GroupBy has two dependent sub-operators: One that is applied to the entire partition (here, applying type matching, selecting the `USSeller`s and the `count`), and one that is applied to each item in the partition (here, applying validation). This GroupBy operator can be implemented efficiently as one physical operator. Another difference is that since we do not use nested tuples, another method must be used to preserve the input sequence order during evaluation. This is the purpose of the MapIndexStep on line 10. The group-by operator and the logical rewritings to obtain **(P2)** are described in detail in Section 5.

Efficient join algorithms for XQuery. The last challenge is to introduce the proper algorithm for the outer-join. This is more complex for XQuery than in the relational context. First, the join operator must preserve order. As a result, standard hash or sort joins cannot be used as is. Instead, we implement variants of standard index-hash and B-tree index joins that preserve the input sequence order. Second, the appropriate hash function or B-tree index for the join must take the semantics of XQuery predicates into account. This semantics is best understood by looking at the normalization of the equality predicate `$t/buyer/@person = $p/@id`:

```
some $x' in fn:data($t/buyer/@person) satisfies
some $y' in fn:data($p/@id) satisfies
  op:equal(fs:convert-operand($x',$y'),
           fs:convert-operand($y',$x'))
```

In this expression, the function `fn:data` denotes atomization; the function `fs:convert-operand` casts untyped data to a target type; and `op:equal` is an overloaded comparison between the various XQuery atomic types that also handles type promotion.

In a traditional hash-join algorithm, each branch of the join must be *independent* of the other. Developing such an algorithm for XQuery, therefore, seems impossible given the above equality test, because the result of `fs:convert-operand` depends on values from both branches of the join. In Section 6, we show that it is possible to develop a hash-join algorithm that takes XQuery's predicate semantics into account.

3. The XQuery Algebra

Data model. The algebra is defined on a logical data model that includes XML values and tables of XML values. At the physical level, our implementation supports tables represented as cursors or as main-memory arrays over tuples, and XML values represented as XML token streams [8], as XML stored in relational indices [19], or as a main-memory tree data structure similar to the DOM. We focus here on the logical algebra. Physical algorithms for joins are described in Section 6.

A value in the logical data model is an XML value, i.e., an ordered sequences of items in the XQuery data model, or a table, i.e., an ordered sequence of tuples containing XML values. $\mathtt{S}(\tau)$ denotes an ordered sequence of type τ. An item is either an atomic value or a node (element, attribute, comment, etc.). Items are denoted by $i_1, \ldots, i_n$. XML values are denoted by $\mathtt{S}(i_1), \ldots, \mathtt{S}(i_n)$. Atomic values and XML nodes are denoted using the formal notation defined in [22]. For example, `1` is an atomic value of type `xs:integer`, and `element name {"John"}` is an element `name` containing the string `"John"`.

A tuple is a record with fields containing XML values. Tuples are written $[\mathtt{q}_1\mathtt{:S(}i_1\mathtt{)};\ldots;\mathtt{q}_n\mathtt{:S(}i_n\mathtt{)}]$, where $\mathtt{q}_1, \ldots, \mathtt{q}_n$ are field names. [] denotes the empty tuple. Tuples are denoted by $\tau_1, \ldots, \tau_n$. Tables are denoted by $\mathtt{S}(\tau_1), \ldots, \mathtt{S}(\tau_n)$. Our model allows tuples to contain sequences of items. For example, the following is a tuple with two fields, `size` and `name`, containing, respectively, a sequence of integers and an element node:

```
[size:(1,2); name:element name {"John"}]
```

Supporting the XQuery data model requires sequences of items in tuples as in our model, or nested tuples of items as in [14]. Both models are more expressive than the relational first normal form. As a result, XQuery implementations on top of relational systems, such as System RX [3], take an approach similar to ours.

Notations. Algebraic operators are written as follows:

$$\mathsf{Op}[\mathsf{p}_1, \ldots, \mathsf{p}_i]\{\mathsf{DOp}_1, \ldots, \mathsf{DOp}_h\}(\mathsf{Op}_1, \ldots, \mathsf{Op}_k)$$

where Op is the operator name; p_i's are static parameters of the operator; DOp_i's are dependent sub-operators; and Op_i's are input (or independent) operators. A sub-operator is *dependent* (*independent*) with respect to a given operator Op, if its evaluation does (does not) depend on the evaluation of other sub-operators of Op. For dependent operators, IN denotes the input XML value or tuple. For example, $\mathsf{MapFromItem}\{\mathtt{[x:}\textsc{in}\mathtt{]}\}\mathtt{((0,1))}$ yields the table `([x:0],[x:1])`. The operator $\mathtt{[x:}\textsc{in}\mathtt{]}$ is dependent since its evaluation depends on the independent input, denoted by IN), of MapFromItem.

Each operator can be interpreted as a function on values in the logical data model. The complete XQuery logical algebra and the operators' signatures are in Table 1. For example, Select takes a dependent boolean operator over a tuple of type τ_1, a sequence of tuples of type τ_1, and returns a sequence of tuples of type τ_1.

Operators on XML values. The first part of Table 1 contains operators on XML values. Many of those operators correspond to exactly one expression in the XQuery Core [22], and their semantics is identical to that of the corresponding Core expression.

The first sub-group of XML operators are constructors for items and sequences of items with the obvious semantics of logically creating the corresponding node (or sequence). These are quite different from the Ξ operator proposed in [14], which does not produce actual element nodes, but instead produces a serialized XML form. This operator's semantics is not compositional, which is a requirement to support the whole language. In contrast, our operators are compositional and can be implemented efficiently using token streams, maintaining the benefits of the Ξ operator without its limitations.

The second sub-group contains operators for tree navigation and projection (in the style of [13]). The TreeJoin is a set-at-a-time operator for navigation, which takes a set of nodes in document order and returns a set of nodes in document order after applying the given step. It can be implemented either as a combination of iteration with navigational access on the data model or as an advanced kind of XPath join, such as that proposed in [10].

The third sub-group contains type operators for casting, validation, and type matching. They are used to implement type operations in XQuery.

The fourth sub-group contains operators for functions and conditionals. To be precise, the semantics of logical operators is defined with respect to an implicit *algebra context*, which we omit for conciseness. The algebra context contains notably function parameters and the compiled query plans for user-defined functions. Since XQuery functions

Algebraic operator	Operator's Signature
XML operators	
Constructors	
Sequence	Sequence(S(i_1),S(i_2)) $\rightarrow$ S(i_3)
Empty sequence	Empty() $\rightarrow$ S(i)
Atomic value	Scalar[a]() $\rightarrow$ a
Element	Element[q](S(i)) $\rightarrow$ `element` q {S(i)}
Attribute	Attribute[q](S(a)) $\rightarrow$ `attribute` q {S(a)}
Text node	Text(a) $\rightarrow$ `text` {a}
Comment	Comment(a) $\rightarrow$ `comment` {a}
PI	PI(a) $\rightarrow$ `pi-node` {a}
Navigation, Projection	
Tree Join	TreeJoin[axis,nodetest](S(i_1)) $\rightarrow$ S(i_2)
Tree Projection	TreeProject[paths](i_1) $\rightarrow$ i_2
Type operators	
Casting	Castable[Type](a_0) $\rightarrow$ $boolean$
	Cast[Type](a_0) $\rightarrow$ a_1
Validation	Validate[Type](i_1) $\rightarrow$ i_2
Type matching	TypeMatches[Type](S(i)) $\rightarrow$ $boolean$
	TypeAssert[Type](S(i)) $\rightarrow$ S(i)
Functional operators	
Function parameter	Var[q]() $\rightarrow$ S(i)
Function call	Call[q](S(i_1), ...S(i_n)) $\rightarrow$ S(i_0)
Conditional	Cond{S(i_1), S(i_2)}($boolean$) $\rightarrow$ S(i_0)
I/O operators	
Parsing	Parse(URI) $\rightarrow$ S(i)
Serialization	Serialize(URI, S(i)) $\rightarrow$ ()

Tuple operators	
Constructors	
Tuple construction	[q$_1$,...,q$_n$](S(i_1), . . . , S(i_n)) $\rightarrow$ [q$_1$:S(i_1);. . .;q$_n$:S(i_n)]
Tuple concatenation	++(τ_1, τ_2) $\rightarrow$ τ_1++τ_2
Select, Project, Join	
Tuple access	#q(τ) $\rightarrow$ S(i)
Selection	Select{$\tau_1 \rightarrow boolean$}(S($\tau_1$)) $\rightarrow$ S(τ_1)
Cartesian product	Product(S(τ_1), S(τ_2)) $\rightarrow$ S(τ_1++τ_2)
Join	Join{τ_1++$\tau_2 \rightarrow boolean$}(S($\tau_1$), S($\tau_2$)) $\rightarrow$ S(τ_1++τ_2)
Left-outer join	LOuterJoin[q]{τ_1++$\tau_2 \rightarrow boolean$}(S($\tau_1$), S($\tau_2$)) $\rightarrow$ S([q:$boolean$]++τ_1++τ_2)
Map operators	
Map	Map{$\tau_1 \rightarrow \tau_2$}(S($\tau_1$)) $\rightarrow$ S(τ_2)
Map-null	OMap[q](S(τ_1)) $\rightarrow$ S([q:$boolean$]++τ_1)
Map-concat	MapConcat{$\tau_1 \rightarrow$ S(τ_2)}(S(τ_1)) $\rightarrow$ S(τ_1++τ_2)
Outer-map-concat	OMapConcat[q]{$\tau_1 \rightarrow$ S(τ_2)}(S(τ_1)) $\rightarrow$ S([q:$boolean$]++τ_1++τ_2)
Map-index	MapIndex[q](S(τ)) $\rightarrow$ S([q:$integer$]++τ)
Map-index-step	MapIndexStep[q](S(τ)) $\rightarrow$ S([q:$integer$]++τ)
Grouping, sorting	
Sorting	OrderBy{$\tau, \tau \rightarrow boolean$}(S($\tau$)) $\rightarrow$ S(τ)
Group by	GroupBy[q$_{Agg}$, q$_{Index}$, q$_{Null}$] {S(τ) $\rightarrow i$}{S(i) $\rightarrow$ S(i)}(S(τ)) $\rightarrow$ S([q$_1$: i; ...; q$_{Agg}$: i])

XML/Tuple Operators	
Map from items	MapFromItem{$i \rightarrow \tau$}(S(i)) $\rightarrow$ S(τ)
Map to items	MapToItem{$\tau \rightarrow i$}(S(τ)) $\rightarrow$ S(i)
Some quantifier	MapSome{$\tau \rightarrow boolean$}(S(τ)) $\rightarrow$ $boolean$
Every quantifier	MapEvery{$\tau \rightarrow boolean$}(S(τ)) $\rightarrow$ $boolean$

Table 1. The Complete XQuery Algebra

can be recursive, this makes the algebra Turing complete. Note that a number of XQuery built-in functions are also required for completeness (`fn:data`, etc).

Finally, the last sub-group contains I/O operators Parse for parsing documents and Serialize for serialization.

Tuple Operators. The tuple part of the algebra follows the same principle as [14], using standard NRA operators with a semantics that preserves sequence order. The first two sub-groups of tuple operators in Table 1 contain the standard tuple construction, concatenation and field access, selection, projection and joins. A small difference from [14] is that we do not model nulls with a special value, but instead use the empty sequence. To distinguish the empty sequence from an actual null value, the LOuterJoin adds a new field (with name q) of type boolean that is set to true if the value is null.

The third sub-group contains maps. Map is the general functional map on sequences of tuples. MapConcat is a dependent join, equivalent to the D-Join operator in [4]. The OMap, OMapConcat, MapIndex and MapIndexStep operators are unique to our algebra and are used during query unnesting. OMapConcat (resp. OMap) is the "outer" counterpart of MapConcat (resp. Map), which introduces a null value when its dependent expression (resp. its input) returns the empty sequence. MapIndex is used to compile FLWOR expressions with an index variable (e.g., `at $a`), and can be used notably in path expressions to compute `position()`. It takes a tuple sequence as input and for each tuple in the sequence, adds a new field q that contains the tuple's index within the input tuple sequence, starting with the value `1`. MapIndexStep is a variant of the MapIndex that does not require consecutive integers, which facilitates its use in rewritings.

Finally, OrderBy is similar to its relational counterpart, except that it accounts for XQuery's rules for coercing and promoting values to comparable types. As we mentioned in Section 2, our GroupBy is a new operator specific to XQuery. We describe its semantics and how it is used in Section 5.

Boundary operations. Four operators sit at the boundary between the tuple part of the algebra and the XML part of the algebra. The MapFromItem and MapToItem operations are simple maps. We give them distinct names rather than relying on a polymorphic map to more easily distinguish between parts of a query plan that operate on tuples from parts that operate on XML values. Lastly, the MapSome and MapEvery operators are used to compile XQuery quantified expressions.

4. Algebraic compilation

Notations. The compilation algorithm is described using inference-rule notation [22]. The judgment $Expr \Rightarrow \mathsf{Op}$ holds when the Core expression $Expr$ is compiled to the algebraic plan Op.

Compilation architecture. Galax produces an evaluation plan through successive compilation phases, the first of which is based on normalization as specified in [22]. Normalization results in queries expressed in a subset of XQuery known as the XQuery Core. Defining the compilation rules from the Core to the algebra guarantees complete coverage of XQuery 1.0. Many Core expressions, such as constructors and function calls, correspond directly to an algebraic operator. For instance, the following inference rule describes the compilation for sequence construction, which maps to the algebraic operator Sequence.

$$\frac{Expr_1 \Rightarrow \mathsf{Op}_1 \qquad Expr_2 \Rightarrow \mathsf{Op}_2}{Expr_1, Expr_2 \Rightarrow \mathsf{Sequence}(\mathsf{Op}_1, \mathsf{Op}_2)} \quad (\text{SEQUENCE})$$

In the rest of this section, we focus on fragments of the XQuery Core that require special treatment, in particular FLWOR, path, and type expressions.

Compiling FLWOR expressions. The XQuery specification [21] describes the semantics of FLWOR expressions in terms of *streams of tuples* that correspond to variable bindings. However, normalization [22] breaks FLWOR expressions into single `for` and `let` clauses. This is problematic in two ways: First, the semantics of order-by cannot be described without relying on a notion of tuple stream; second, it makes the introduction of algebraic tuple-based operators unnecessarily difficult. We change normalization to preserve the structure of FLWOR expressions, which permits an easier introduction of tuple operators and enables a proper treatment of order-by. Quantified expressions, which also operate on tuple streams, are compiled similarly.

Figure 2 contains the compilation rules for FLWOR expressions. The notation $Clauses_{|\$Var/\text{IN}\#Var}$ denotes variable substitution, where every occurrence of variable $\$Var$ is replaced by the corresponding tuple field access ($\text{IN}\#Var$) in the given clauses. The trickiest aspect of compiling FLWOR expressions is that generating the proper tuple streams requires reversing top-down data flow in a XQuery Core abstract syntax tree to bottom-up data flow in an algebraic query plan. To do this, we need an auxiliary judgment $[\![Expr]\!]_{(\mathsf{Op}_1)} \Rightarrow \mathsf{Op}_2$ that compiles an expression $Expr$ into an algebraic plan Op_2 *in the context of* an intermediate algebraic plan Op_1. The intermediate plan is the result of compiling previous clauses. The following

$$\frac{\mathsf{Op}_1 = \mathsf{TypeAssert[T]}(\mathsf{Op}_0)}{[\![\texttt{as T}]\!]_{\mathsf{Op}_0} \Rightarrow \mathsf{Op}_1} \tag{AS}$$

$$\frac{\begin{array}{c} Expr_1 \Rightarrow \mathsf{Op}_1 \\ [\![\texttt{as T}]\!]_{\text{IN}} \Rightarrow \mathsf{Op}_2 \\ \mathsf{Op}_3 = \mathsf{MapFromItem}\{\texttt{[x : }\mathsf{Op}_2\texttt{]}\}(\mathsf{Op}_1) \\ \mathsf{Op}_4 = \mathsf{MapConcat}\{\mathsf{Op}_3\}(\mathsf{Op}_0) \\ \mathsf{Op}_5 = \mathsf{MapIndex[i]}(\mathsf{Op}_4) \\ [\![Clauses|\$Var/\text{IN}\#Var, \texttt{\$i}/\text{IN}\#\texttt{i}]\!]_{\mathsf{Op}_5} \Rightarrow \mathsf{Op}_6 \end{array}}{\begin{array}{c}[\![\texttt{for \$}Var\texttt{[as T][at \$i] in } Expr_1\ Clauses]\!]_{\mathsf{Op}_0} \\ \Rightarrow \mathsf{Op}_6\end{array}} \tag{FOR}$$

$$\frac{\begin{array}{c} Expr_1 \Rightarrow \mathsf{Op}_1 \\ [\![\texttt{as T}]\!]_{\mathsf{Op}_1} \Rightarrow \mathsf{Op}_2 \\ \mathsf{Op}_3 = \mathsf{MapConcat}\{\texttt{[}Var\texttt{:}\mathsf{Op}_2\texttt{]}\}(\mathsf{Op}_0) \\ [\![Clauses|\$Var/\text{IN}\#Var]\!]_{\mathsf{Op}_3} \Rightarrow \mathsf{Op}_4 \end{array}}{[\![\texttt{let \$}Var\texttt{ [as T] in } Expr_1\ Clauses]\!]_{\mathsf{Op}_0} \Rightarrow \mathsf{Op}_4} \tag{LET}$$

$$\frac{\begin{array}{c} Expr_1 \Rightarrow \mathsf{Op}_1 \\ \mathsf{Op}_2 = \mathsf{Select}\{\mathsf{Op}_1\}(\mathsf{Op}_0) \\ [\![Clauses]\!]_{\mathsf{Op}_2} \Rightarrow \mathsf{Op}_4 \end{array}}{[\![\texttt{where } Expr_1\ Clauses]\!]_{\mathsf{Op}_0} \Rightarrow \mathsf{Op}_4} \tag{WHERE}$$

$$\frac{\begin{array}{c} Expr_1 \Rightarrow \mathsf{Op}_1 \\ \mathsf{Op}_2 = \mathsf{OrderBy}\{\mathsf{Op}_1\}(\mathsf{Op}_3) \\ [\![Clauses]\!]_{\mathsf{Op}_2} \Rightarrow \mathsf{Op}_4 \end{array}}{[\![\texttt{order by } Expr_1\ Clauses]\!]_{\mathsf{Op}_3} \Rightarrow \mathsf{Op}_4} \tag{ORDERBY}$$

$$\frac{\begin{array}{c} Expr_1 \Rightarrow \mathsf{Op}_1 \\ \mathsf{Op}_2 = \mathsf{MapToItem}\{\mathsf{Op}_1\}(\mathsf{Op}_3) \end{array}}{[\![\texttt{return } Expr_1]\!]_{\mathsf{Op}_3} \Rightarrow \mathsf{Op}_2} \tag{RETURN}$$

Figure 2. Compilation rules for FLWORs

compilation rule triggers the auxiliary judgment over the input tuple currently in scope.

$$\frac{[\![FLWORExpr]\!]_{(\text{IN})} \Rightarrow \mathsf{Op}_0}{FLWORExpr \Rightarrow \mathsf{Op}_0}$$

We illustrate the FLWOR rules on the expression:

```
1.  for $p in $auction//person/@id
2.  let $a := count($auction//@person[. = $p])
3.  where $p = "person103"
4.  return $a
```

Compilation applies the rules on each clause in the FLWOR, passing the resulting plan as a parameter to the rule that compiles the next clause. The first rule compiles the (FOR) clause on line 1, producing a composition of a MapFromItem operator, which builds a sequence of tuples with a single field p, followed by a MapConcat operator, which adds those tuples to the input tuple sequence.

```
Opfor = [[for $p in $auction//person/@id]]IN
                    ⇒
MapConcat{MapFromItem{[p:IN]}(IN#auction//person/@id)}(IN)
```

The MapConcat accounts for the case where a `for` clause occurs in the middle of a FLWOR expression and is passed a previously computed input sequence of tuples. Also, the rule replaces all subsequent accesses to the variable `$p` by the appropriate tuple-field access. That plan is then passed as an argument to the compilation rule for the `let` on line 2. Note that this rule introduces a single MapConcat which adds the `a` field containing the sequence of items corresponding to the `let` variable:

```
[[let $a := count((IN#auction)//@person[.=(IN#p)])]]Opfor
                          ⇒
MapConcat
  {[a:count((IN#auction)//@person[.=(IN#p)])]}(Opfor)
```

Finally, the rules for the `where` and `return` clauses on lines 3 and 4 introduce selection and a final map that returns the resulting XML value.

```
MapToItem{IN#a}(Select{IN#p = "person103"}
    (MapConcat
      {[a:count(IN#auction//@person[.=(IN#p)])]}
        (MapConcat{MapFromItem{[p:IN]}(IN#auction//@person)}
          (IN))))
```

Compiling path expressions. Interestingly, most of the semantics of path expressions can be expressed through normalization into FLWOR, step, and conditional expressions. As a result, most of the compilation approach presented in [4] can be obtained by compiling the normalized representation of path expressions. However, a few fixes to normalization as specified in [22] are necessary.

Consider the compilation of the simple path expression `$d/descendant::person[position()=1]`. The first problem is to make sure that each step is compiled into a complete FLWOR block, as opposed to a combination of `for` loops and conditionals. This results in the following normalized XQuery expression. Note the use of a single FLWOR block, with an `at` clause and a `where` clause for the predicate instead of an if-then-else.

```
for $fs:dot in $d return
  for $fs:dot at $fs:position in $fs:dot/descendant::person
  where $fs:position = 1
  return $fs:dot
```

From the above expression, an axis step is compiled into one TreeJoin operator, and the FLWOR expression is compiled using the rules given previously. Note also that at the implementation level, the context item, denoted by the variable `$fs:dot`, has to be passed as input to the TreeJoin. This results in the following algebraic plan in which the MapIndex operator computes the context position:

```
MapToItem{MapToItem{IN#dot}
    (MapConcat{Select{(IN#position) = 1}
      (MapIndex[position]
         (MapFromItem{[fs:dot:IN]}
            (TreeJoin[descendant::person](IN#fs:dot))))}
      (IN))}
   (MapFromItem{[fs:dot:IN]}(IN#d))
```

$$\frac{Expr_1 \Rightarrow \mathsf{Op}_1}{[\![\texttt{default return } Expr_1]\!]_{\texttt{\$x}} \Rightarrow \mathsf{Op}_1(\textsc{in})} \quad (\textsc{default})$$

$$\frac{Expr_1 \Rightarrow \mathsf{Op}_1 \qquad [\![Clauses]\!]_{\texttt{\$x}} \Rightarrow \mathsf{Op}_2}{[\![\texttt{case T1 return } Expr_1\ Clauses]\!]_{\texttt{\$x}} \Rightarrow \mathsf{Cond}\{\mathsf{Op}_1(\textsc{in}),\mathsf{Op}_2\}(\mathsf{TypeMatches[T1]}(\textsc{in}\#\mathsf{x}))} \quad (\textsc{case})$$

$$\frac{Expr_1 \Rightarrow \mathsf{Op}_1 \qquad [\![Clauses]\!]_{\texttt{\$x}} \Rightarrow \mathsf{Op}_2}{\texttt{typeswitch \$x:=(}Expr_1\texttt{)}\ Clauses \Rightarrow \mathsf{MapToItem}\{\mathsf{Op}_2\}(\texttt{[x:}\mathsf{Op}_1\texttt{] ++ }\textsc{in})} \quad (\textsc{typeswitch})$$

Figure 3. Compilation rules for typeswitch

Obviously, additional work is needed to integrate other XPath evaluation techniques [5, 9, 10]. However, the compilation approach presented here has the following two important properties: (1) it supports path expressions arbitrarily interleaved with other XQuery expressions, and (2) it enables query de-correlation on queries that perform joins through a combination of path and FLWOR expressions. Concretely, it means our optimization techniques can unnest the following variant of query **Q1**, which uses a nested path expression instead of a nested FLWOR expression:

```
for $p in $auction//person
let $a := $auction//closed_auction[.//@person = $p/@id]
return count($a/element(*,USSeller))
```

Compiling type expressions. Compiling type operations is mostly straightforward due to the ability of the algebra to combine XML operators with tuple operators. We have already seen examples illustrating the compilation of type assertions. Typeswitch is the only expression requiring special care. Consider the following example:

```
  typeswitch ($auction//closed_auction)
    case $u as element(*,USAuction) return $u/@state
    case $e as element(*,EUAuction) return $e/@country
    default $o return $o/@country_code
```

One difficulty with typeswitch is that it uses an heterogeneous set of variables in each branch. This makes compilation into tuples less natural. As a first step, we change normalization of typeswitch to use the same variable in every branch. This variable can then be used among the various tuple operators involved in the query plan. For convenience, we write the corresponding typeswitch in the XQuery Core as follows: `typeswitch $x := (`$Expr$`)` $CaseClauses$. Figure 3 shows the compilation rules from this new Core `typeswitch` expression. The input is in one tuple field whose name is the common variable. Each branch of the typeswitch is compiled into a conditional expression, using type matching as the condition. For the above example, this results in the following plan:

```
MapToItem{Cond{(IN#x)/@state,
              Cond{(IN#x)/@country,
                   (IN#x)/@country_code}
                   (TypeMatches[element(*,EUAuction)](IN#x))}
              (TypeMatches[element(*,USAuction)](IN#x))}
([x : $auction//closed_auction] ++ IN)
```

5. XQuery GroupBy and Unnesting

The GroupBy operator takes three sets of field names and three input operators:

$\mathsf{GroupBy}[\mathsf{q}_{Agg},\mathsf{q}_{Indices},\mathsf{q}_{Nulls}]\{\mathsf{Op}_2\}\{\mathsf{Op}_1\}(\mathsf{Op}_0)$

Op_0 yields a table of tuples. These tuples must contain boolean-valued fields with names in q_{Nulls} and integer-valued fields with names in $\mathsf{q}_{Indices}$. The pre-grouping operator Op_1 is applied to each input tuple that has all q_{Nulls} fields equal to false. The resulting tuples are grouped by the values in the $\mathsf{q}_{Indices}$ fields into partitions. The post-grouping operator Op_2 is applied to each partition and the field denoted by q_{Agg} is bound to the result of Op_2. The tuples in each partition are sorted in a stable, ascending order by the $\mathsf{q}_{Indices}$ fields. For each partition, the GroupBy yields a tuple containing the fields in the input table and the corresponding q_{Agg} field.

We use the following simple XQuery expression to illustrate the GroupBy operator's semantics.

```
for $x in (1,1,3)
let $a := avg(for $y in (1,2) where $x<=$y return $y*10)
return ($x, $a)
```

Compilation and rewriting of this query yields the following plan with a GroupBy:

```
1. MapToItem{Sequence(IN#x,IN#a)}
2.   (GroupBy[a, index, null]{avg(IN)}{IN#y * 10}
3.     (LOuterJoin[null]{IN#x <= IN#y}
4.       (MapIndexStep[index]
5.         (MapFromItem{[x:IN]}(1,1,3)),
6.       MapFromItem{[y:IN]}(1,2))))
```

We first describe how this plan works before explaining the rewrite rules necessary to obtain it. The MapIndexStep operator on line 4 introduces an integer field `index` containing the input sequence order and is used as the index field for the GroupBy on line 2. The null field is introduced by the LOuterJoin on line 3, identifying when no right-hand side tuple matches a given left-hand side tuple. Figure 4 shows the sequence of input and output tuples and the corresponding partitions for the GroupBy. Note that the `index` field distinguishes between the first and second occurrence of the value `1` in the input sequence, and that the pre-grouping operator (IN#`y*10`) is not applied when the input tuple's `null` flag is true.

Query unnesting rewritings. We return to the naïve plan **P1** from Section 2 to explain the rewrite rules. The complete set of rewritings used to optimize **P1** are given in Figure 5.

Standard rewritings

MapConcat{Op$_1$}([])	$\rightarrow$	Op$_1$	(remove map)
MapConcat{Op$_1$}(Op$_2$)	$\rightarrow$	Product(Op$_2$,Op$_1$), *when* Op$_1$ *independent of* IN	(insert product)
Select{Op$_1$}(Product(Op$_2$,Op$_3$))	$\rightarrow$	Join{Op$_1$}(Op$_2$,Op$_3$)	(insert join)

New rewritings

[x: Op$_1$(MapFromItem{Op$_2$}(Op$_3$))]	$\rightarrow$	GroupBy[x,[],[null]]{Op$_1$(IN)}{Op$_2$}(OMap[null](Op$_3$))	(insert group-by)
MapConcat{GroupBy[x,inds,nulls]{Op$_1$}{Op$_2$}(Op$_3$)}(Op$_4$)	$\rightarrow$	GroupBy[x,inds+ind$_1$,nulls+null$_1$] {Op$_1$}{Op$_2$} (OMapConcat[null$_1$]{Op$_3$}(MapIndex[ind$_1$](Op$_4$)))	(map through group-by)
OMapConcat[null1]{OMap[null2](Op$_1$)}(Op$_2$)	$\rightarrow$	OMapConcat[null1]{Op$_1$}(Op$_2$)	(remove duplicate null)
OMapConcat[null]{Join{Op$_3$}(IN,Op$_1$)}(Op$_2$)	$\rightarrow$	LOuterJoin[null]{Op$_3$}(Op$_2$,Op$_1$)	(insert outer-join)

Figure 5. Algebraic rewritings

Input				Output	
x	y	index	null	x	a
1	1	1	false	1	15
1	2	1	false		
1	1	2	false	1	15
1	2	2	false		
3	()	3	true	3	()

Figure 4. Input and output of GroupBy

The first set contains traditional product and join rewritings. The second contains new rewritings unique to our algebra.

The most important rewrite rule is (insert group-by), which replaces a unary tuple-constructor over an item operator by a trivial GroupBy. The key observation is that a unary tuple constructor is equivalent to a GroupBy with no grouping criteria and corresponds to a group-by in which each partition contains one tuple. Applying this rewriting to the tuple constructor on line 7 of **P1** results in the following query plan, in which the new operators are underlined:

```
1.  MapToItem                                          (P1')
2.    {Element[item]
3.      (Sequence
4.        (Attribute[person]((IN#p)/name/text()),
5.         count(IN#a/element(*,USSeller))))}
6.    (MapConcat
7.      {GroupBy[a,[],[null]]
8.        {TypeAssert[element(*,Auction)*](IN)}
9.        {Validate(IN#t)}
10.       (OMap[null]
11.         (Select{IN#t/buyer/@person = IN#p/@id}
12.           (MapConcat{MapFromItem{[t:IN]}
13.               ($auction//closed_auction)}(IN))))}
14.     (MapFromItem{[p:IN]}($auction//person)))
```

Once the GroupBy operator is introduced, the rest of query unnesting consists of pushing operations through the GroupBy. The (map through group-by) rule pushes the MapConcat on line 6 in **P1'** through the GroupBy. This results in plan **P1"** below, in which the new operators are underlined.

```
1.  MapToItem                                          (P1'')
2.    {Element[item]
3.      (Sequence
4.        (Attribute[person]((IN#p)/name/text()),
5.         count(IN#a/element(*,USSeller))))}
6.    (GroupBy[a,[index],[null,null1]]
7.      {TypeAssert[element(*,Auction)*](IN)}
8.      {Validate(IN#t)}
9.      (OMapConcat[null1]
10.       {OMap[null]
11.       (Select{IN#t/buyer/@person = IN#p/@id}
12.         (MapConcat{MapFromItem{[t:IN]}
13.             ($auction//closed_auction)}(IN)))}
14.       (MapIndexStep[index]
15.         (MapFromItem{[p:IN]}($auction//person)))))
```

Finally, the last step is to recognize the join formed by the combination of the selection and outer-map. First, we apply the (remove duplicate null) rule to remove the OMap that is redundant after the introduction of the OMapConcat. The LOuterJoin is introduced by consecutive applications of the (insert product), (insert join) and (insert outer-join) rules. This results in the expected final plan **P2**:

```
1.  MapToItem                                          (P2)
2.    {Element[item]
3.      (Sequence
4.        (Attribute[person]((IN#p)/name/text()),
5.         count(IN#a/element(*,USSeller))))}
6.    (GroupBy[a,index,null]
7.      {TypeAssert[element(*,Auction)*](IN)}
8.      {Validate(IN#t)}
9.      (LOuterJoin[null]]
10.       { IN#t/buyer/@person = IN#p/@id }
11.       (MapIndexStep[index]
12.         (MapFromItem{[p : IN]}($auction//person)),
13.       MapFromItem{[t:IN]}($auction//closed_auction))))
```

6. XQuery Join Algorithms

So far, we have focused on the logical semantics of our algebra. Here, we address the physical implementation of the join operator, which is complicated by the semantics of XQuery's comparison operators. We show how to implement a hash-join algorithm that accounts for this semantics. We focus on the hash-join algorithm, but the same approach can be used to implement a sort join.

Relational join algorithms typically rely on the transitivity of (in)equality operators and on the ordering properties of inputs for efficiency. XQuery's (in)equality op-

Type of first operand	Type of second operand	Convert first operand to:
`xdt:untyped` or `xs:string`	`xdt:untyped` or `xs:string`	`xs:string`
`xdt:untyped`	numeric	`xs:double`
`xdt:untyped`	Any other $Type$	$Type$
Any other $Type$	Must be $Type$	N/A

Table 2. Semantics of `fs:convert-operand`

erators, however, do not have the standard properties required by traditional join algorithms. Recall from Section 2 that the semantics of predicates in XQuery equality includes existential quantification, atomization (`fn:data`), casting of untyped values to the type of the other operand (`fs:convert-operand`), and overloaded predicate semantics (`op:equal`). For example, the normalization of `$x = $y` is:

```
some $x' in fn:data($x),
     $y' in fn:data($y)
  op:equal(fs:convert-operand($x',$y'),
           fs:convert-operand($y',$x'))
```

One problem is handling the `fs:convert-operand` function. Join algorithms typically require that one input be materialized independently of the other input, but the `fs:convert-operand` function naïvely depends on the value of each of its operands. Fortunately, the formal semantics of `fs:convert-operand` [22] states that the value of the first operand is promoted to the *type* of the second operand, therefore, the function depends only on the second operand's type, not its value. To support this semantics, our algorithm enumerates all the types to which a join key value can be promoted, which is no more than nineteen (the number of primitive XML Schema datatypes). If both operands of the join are untyped, then this number is reduced to two: `xs:string` and `xs:double`. Table 2 summarizes the semantics of `fs:convert-operand`.

Given this observation, we can construct a hash table in which each entry is keyed on a (value, type) pair (after type conversion) and contains the original value and type (before type conversion), the corresponding tuple value, and the ordinal position of the tuple in the input sequence. If the input value is not untyped and is non-numeric, then we store only one entry keyed on the original value and type. If the value is numeric, then we store one entry for each numeric value to which it can be promoted. Clearly, static type analysis can improve our algorithm by reducing the number of entries that must be stored. If, for instance, we can infer statically that both operands are integers, we can build a key directly on the integer value and specialize our join algorithm to use integer-comparison operators.

Figure 6 contains the pseudo code for the hash-join algorithm. Lines 33–49 contains `equalityJoin`, a higher-order function that takes the inner and outer input tables and

```
1.  // Materialize input : hash table keyed on (val,type)
2.  HashTab materialize(TupCursor in,Exp keyExp) {
3.    HashTab ht = new(HashTab);
4.    int order = 1;
5.    while (not(in.empty())) {
6.      Tup tup = in.next();
7.      Val keyVals = fn:data(keyExp.eval(tup));
8.      for-each key in keyVals {
9.        Array valTypePairs = promoteToSimpleTypes(key);
10.       for-each Pair (val,type) in valTypePairs
11.         ht.put((val,type),(key,typeof(key),tup,order));
12.    }
13.    order++;
14.   }
15.   return ht;
16. }

17. // Fetch all tuples matching an input tuple
18. List allMatches(HashTab innerTab,Tup tup,Exp keyExp) {
19.   Val keyVals = fn:data(keyExp.eval(tup));
20.   AssocList allMatches = new(AssocList);
21.   for-each key in keyVals {
22.     Array valTypePairs = promoteToSimpleTypes(key);
23.     for-each Pair (val,type) in valTypePairs
24.       (val1,type1,tup,order) = innerTab.get(val,type);
25.       if (type1,typeof(key)) in (Table 2) then
26.         allMatches.add(order,tup);
27.   }
28.   // Sort matches on original sequence order and
29.   List sortedMatches = allMatches.sortOnOrderField();
30.   // Remove duplicate tuples -- maintains sorted order
31.   return sortedMatches.removeDuplicates();
32. }

33. // Equality join. For both the inner/outer inputs,
34. // takes: the tuple cursor and expression
35. // that accesses join field.
36. TupCursor equalityJoin(TupCursor inner,
37.                        Exp innerKeyExp,
38.                        TupCursor outer,
39.                        Exp outerKeyExp) {
40.   HashTab innerTab = materialize(inner,innerKeyExp);
41.   TupCursor result = new(TupCursor);
42.   while (not(outer.empty())) {
43.     Tup tup = outer.next();
44.     List ms = allMatches(innerTab,tup,outerKeyExp);
45.     for-each match in ms
46.       result.add(tup.copy().concat(match));
47.   }
48.   return result;
49. }
```

Figure 6. Hash-join algorithm

two expressions for computing the inner-key and outer-key values. On lines 40–41, the function materializes the inner input table and creates a result cursor. For each tuple in the outer input table (lines 42–47), it probes the inner table for all tuples that match the outer tuple, and for each matching tuple, concatenates the outer and matching tuples, and adds this new tuple to the result cursor.

Lines 1–16 contains the `materialize` function, which constructs a hash table from the inner input. For each tuple in the inner input (lines 5–12), it populates the hash table with the (value, type) pairs to which the tuple's key value can be converted (lines 10–11). Note that the *order* of tuples in the inner input is significant. To recover the original sequence order during probing, we store a sequence-order counter with each entry in the hash table.

Implementation	Total time
No algebra	3m33.0s
Algebra +	
No optim	50.0s
Optim + nested-loop joins	5.1s
Optim + XQuery joins	1.7s

Table 3. XMark 1-20 on 1MB document

The `allMatches` function on Lines 17–32 takes a tuple from the outer input and probes the inner-input's hash table for all possible matches. The trickiest part of this code is on Line 25, which checks whether the original types of the inner and outer values are in the `fs:convert-operand` table in Table 2, before adding the match to the list of matches. After accumulating all matches, they are sorted by original sequence order and any duplicate tuples are removed, maintaining the sorted order (lines 28–31). We remove duplicates to preserve the existential quantification implied by the original predicate.

Some obvious implementation improvements have been omitted to simplify exposition. For example, we can store references rather than actual values. Also, this algorithm handles one key predicate in a join, but can be extended to multiple predicates.

7. Experiments

Our algebra and optimizations are implemented in Galax. The compiler passes a regression suite of over 1000 tests that includes the XQuery Use Cases [23] and the XMark benchmark suite [18]. In this section, we present experimental results obtained using the Galax compiler.

To evaluate the algebra and optimizations, we conducted several experiments, the goals of which are to show: (1) the overall efficiency of the algebraic compiler; (2) the benefits of the optimization techniques presented in Section 5; and (3) the effectiveness of the new compiler on complex queries generated by Clio [16].

Most experiments were run on an Intel Pentium M (1.7GHz) with 500MB main memory running Fedora. The Clio experiments were executed on an Intel Pentium M (2.4GHz) with 1GB main memory running Red Hat 9.0.

Table 3 summarizes the impact of the successive improvements to the original Galax compiler. The table contains the total execution time of all twenty XMark queries on a 1MB document – each measurement includes the time to load the input document once, evaluate all twenty queries, and serialize all the results.

In the original Galax processor, query evaluation was performed directly on the query abstract syntax tree after normalization. Migration to the algebra yielded a four-fold speedup, largely due to the replacement of dynamic lookups in the dynamic context by direct compiled memory access and to pipelined evaluation of many algebraic operators. Query unnesting alone with simple nested-loop join yields another ten-fold speedup. Finally, the more efficient join algorithms described in Section 6 yield another three-fold speedup, with an overall improvement of 125 times faster than the original implementation. Note that in the optimized plans, the evaluation time is dominated by parsing, and to a lesser extent, by the execution of the descendant axis in XMark queries Q6, Q7, Q14 and Q19.

Query	Size	NL Join	Hash Join
Q8	10MB	66.17s	0.14s
	20MB	5m0.9s	0.37s
	50MB	1h54m6.45s	2.70s
Q9	10MB	95.41s	0.24s
	20MB	6m19.7s	0.70s
	50MB	2h31m41.1s	2.31s
Q10	10MB	11.66s	1.68s
	20MB	44.34s	5.72s
	50MB	19m4.46s	17.90s
Q12	10MB	41.1s	0.94s
	20MB	3m4.27s	6.14s
	50MB	3h35m11.9s	11m4.66s
Q20	10MB	0.35s	0.36s
	20MB	0.86s	0.82s
	50MB	2.21s	2.78s

Table 4. Scalability of selected XMark Queries

For comparison, we also ran Galax and Saxon [11] 8.1.1 on XMark Queries 1–20 on a 10MB document. The total time was 20s for Galax and 86s for Saxon.

Table 4 shows that the proposed optimizations scale well with larger documents. The table contains the query evaluation times for the XMark Queries 8, 9, 10, which contain 2-way and 3-way joins, and Query 20, which does not. The measurements exclude the times to load the input document into main memory and to serialize the result as well as all other phases. The document loading times are stable: approximately 6s for 10MB, 13.0s for 20MB, and 41s for 50MB, and dominate all other compilation phases (normalization, rewriting, compilation, etc.), whose times are negligible.

The second column contains the evaluation time for plans that use nested-loop joins and the third column for hash or sort joins. As expected, the quadratic complexity of nested-loop joins is evident with increasing document size, whereas the hash and sort joins grow linearly.

Finally, we compiled Clio queries generated from mappings of increasing complexity applied to a 250K document. Table 5 gives the evaluation time for those queries with and without the optimizations from Section 5. Query **N2** is sim-

	Galax			Saxon 8.1.1
Query	**No optim**	**NL Join**	**Hash Join**	
N2	1m6.1	53.4s	1.5s	15.9s
N3	> 1h	2m28.9s	6.4s	58.3s
N4	> 1h	14m2s	21.7s	2m3.5s

Table 5. Clio queries

ilar to the one in Figure 1, with a doubly nested FLWOR expression and a single join. **N3** is a triple-nested FLWOR with a 3-way join, and **N4** is a quadruple-nested FLWOR with a 6-way join. The results show the dramatic improvements obtained from query unnesting and join optimization for those queries. For comparison, we also give the execution time for the same queries using Saxon [11]. Although our best query plan runs about 6 to 10 times faster, Saxon's execution time does not blow up even for the 6-way join, which seems to indicate that it supports some form of join optimization.

8. Conclusion

XML query optimization has recently been the focus of intense research. Numerous techniques have been proposed for XPath evaluation [5, 10, 9], optimizations based on tree-patterns [6, 15], XQuery unnesting [7, 14], and streaming evaluation [1, 8, 13]. In this paper, we presented an algebraic framework that supports complete compilation of XQuery into which those techniques can be integrated. The compiler is completely implemented and results in a stable and efficient system that can process the complex queries that arise in real XQuery applications. There are further opportunities for improving our compiler, notably adding more advanced optimizations for XPath, integrating techniques for streaming evaluation, and providing better support for recursive functions.

References

[1] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *ICDE*, pages 455–466, 2003.

[2] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated update management for XML integrity constraints. In *PLAN-X: Programming Language Technologies for XML*, Pittsburgh, PA, Oct. 2002.

[3] K. Beyer, R. J. Cochrane, V. Josifovski, et al. System RX: one part relational, one part XML. In *SIGMOD*, pages 347–358, 2005.

[4] M. Branter, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledge algebraic XPath processing in Natix. In *ICDE*, pages 705–716, Boston, MA, 2005.

[5] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, pages 310–321, 2002.

[6] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, pages 237–248, Berlin, Germany, Sept. 2003.

[7] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, pages 168–179, Toronto, Canada, Aug. 2004.

[8] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *VLDB*, pages 997–1008, Berlin, Germany, Sept. 2003.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[10] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its axis steps. In *VLDB*, pages 524–535, Berlin, Germany, Sept. 2003.

[11] M. Kay. SAXON 8.0. SAXONICA.com. `http://www.saxonica.com/`.

[12] I. Manolescu and Y. Papakonstantinou. XQuery midflight: Emerging database-oriented paradigms and a classification of research advances. In *ICDE*, page 1143, 2005.

[13] A. Marian and J. Simeon. Projecting XML documents. In *VLDB*, pages 213–224, Berlin, Germany, Sept. 2003.

[14] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *ICDE*, pages 239–250, Boston, MA, Mar. 2004.

[15] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, pages 71–82, Paris, France, June 2004.

[16] L. Popa, Y. Velegrakis, R. J. Miller, and R. F. Mauricio A. Hernandez. Translating Web data. In *VLDB*, pages 598–609, 2002.

[17] L. Quinn. An XML-based Web site using XML query and SVG. In *XML Conference*, Philadelphia, PA, Dec. 2003.

[18] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, Hong Kong, China, Aug. 2002.

[19] A. Vyas, M. F. Fernandez, and J. Simeon. The simplest XML storage manager ever. In *XIME-P 2004*, pages 37–42, Paris, France, June 2004.

[20] The XBrain project: Xquerying the brain mapping database. `http://quad.biostr.washington.edu:8080/xbrain`.

[21] XQuery 1.0: An XML query language. Candidate Recommendation, Nov. 2005.

[22] XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft. W3C Working Draft, Candidate Recommendation, Nov. 2005.

[23] XML query use cases. W3C Working Draft, Sept. 2005.