An XQuery Canonical Form and its Translation to Extended Relational Algebra

Hui Zhang and Frank Tompa
School of Computer Science, University of Waterloo, Canada
{h8zhang,fwtompa}@uwaterloo.ca
Tel: (519)8884567 ext. 3230

Abstract

As XML becomes more widespread as a standard representation for data, XML-based query languages and their evaluations are increasingly important. For this purpose, several XML based query languages have been proposed, including W3C's XQuery. In this paper, we define a query canonical form which provides a conceptually uniform vision of path expressions, element constructors and FLWR expressions in XQuery. The power of this canonical form is shown by identifying an important subset of XQuery that can be translated to this canonical form. Moreover, this canonical form nicely separates different aspects of an XML query, i.e., structure, navigation, and condition. This property makes it easy to be extended, and a possible extension of the canonical form is presented. Having this canonical form, we present an algorithm to translate from it into an extended relational algebra that includes operators defined for the structured text datatype, and we prove its correctness. This algorithm can be used as the basis of a sound translation from XQuery to SQL, and the starting point for query optimization, which is required for XML to be supported by relational database technology.

1 Introduction

As XML [2] is becoming a standard data exchange form, querying XML data draws increasing attention. For this purpose, several XML based query languages have been proposed, including W3C's XQuery [6] which is becoming predominant. However, query processors for XML data are currently in a very initial phase.

To query XML data using such a query language requires a clear understanding of the language. While some effort is being put into developing XML query processors, there is little work being done outside W3C on providing a better understanding of XQuery and thus providing a common base for implementation. In order to learn the language more easily and study implementation tradeoffs, we focus on a core part of XQuery in this paper. More precisely, we define a concise query canonical form which provides the basis for a clearer semantics. This canonical form provides a conceptually uniform vision of path expressions, element constructors and FLWR expressions in XQuery, and thus it provides a simple way to understand these important features of XQuery. Although this canonical form seems small, its expressive power is not overly limited. To show this point, an important subset of XQuery which can be translated to this canonical form is identified together with a set of transformation rules. Therefore, as developers and users of relational technology always think about `select... from... where...` structures, now we can start from the canonical form instead of worrying about different expressions semantically equivalent to the canonical form. Certainly this simplifies development of correct implementations of XML query processors.

It is desirable to be able to separate different aspects of an XML query language, such as structure, navigation and condition of a query, so that variant XML query languages can be defined flexibly and modified or extended easily. The canonical form defined in this paper has this convenient property.

Certainly there are many alternative ways to process XML queries. In common with other researchers [32, 23], we wish to capitalize on the extensive work invested in relational database technology. In this paper, we present an algorithm to translate from the defined canonical form to an extended relational algebra with support for text[13]. In this algebra, functions on a text datatype, including a tree pattern matching sub-language, make full-text search queries or XPath-like queries [4] easily supported.

Our approach does not require specific mapping schemas between XML documents and relational data. It constructs relational views of XML data on the fly, and keeps the original XML text untouched while providing access to various components. Our approach is also useful for any XML-enabled DBMS, such as the DB2 XML Extender which stores some significant structural units as BLOB or VARCHAR in a relational table. Our research should be of immediate benefit to RDBMS vendors, as our approach provides an easy way to integrate text and relational technology to process XML queries.

The rest of this paper is organized as follows: Section 2 defines the canonical form for XQuery. Section 3 develops a set of transformation rules, identifies a subset of XQuery that can be translated to the defined canonical form. Section 4 briefly introduces the structured text ADT and Section 5 describes our algebra. Section 6 presents a translation from the defined canonical form to our algebra, along with a proof of correctness. Section 7 gives an example to illustrate our research, i.e., steps of canonicalization, translation and possible optimization. Section 8 reviews related work. Section 9 concludes the paper and gives a possible extension of the canonical form.

2 An XQuery Canonical Form

Definition 2.1 (XQuery Canonical Form (CF)) An XQuery canonical form is limited to the grammar shown in Figure 1.

Figure 1a gives the definition of path expressions allowed in the canonical form. Production rules for such simple path expressions follow the form of the XQuery grammar [6] and are numbered exactly as their counterparts in that grammar. However, some production rules are modified. In particular, these modified rules are:

- Axis (production rule 35′) has only four kinds: *child*, *descendent*, *self* and *attribute*.
- NameTest (production rule 37′) does not allow wildcard '*', with or without namespace.
- KindTest (production rule 39′) is limited to text test, i.e., text(), and does not allow comment test, processing instruction test and any node test.
- StepQualifier (production rule 46′) only allows predicates, not arbitrary expressions. It also does not allow a reference operation.
- PrimaryExpr (production rule 47′) does not allow Literal, Function call, Wildcard, KindTest and ParenthesizedExpr.

```
[25`] PathExpr := AbsolutePathExpr | RelativePathExpr |
`document` `(` StringLiteral `)` (`/` | `//`)? RelativePathExpr?
[31`] AbsolutePathExpr := (`/` RelativePathExpr ? ) | (`//`
RelativePathExpr)
[32`] RelativePathExpr := StepExpr ( ( `/` | `//` ) StepExpr )*
[33`] StepExpr := AxisStep | GeneralStep
[34`] AxisStep := Axis NodeTest StepQualifiers
[35`] Axis := (`child` | `descendent` | `self` | attribute) `::` | `/`
              | `//` | `.` | `@`
[36`] NodeTest := NameTest | KindTest
[37`] NameTest := QName
[39`] KindTest := TextTest
[42`] TextTest := `text` `(` `)`
[44`] GeneralStep := PrimaryExpr StepQualifiers
[46`] StepQualifers := (`[` PredicateExpr `]` )*
[47`] PrimaryExpr := `.` | NodeTest | Variable
```

(a) Path expression in the Canonical Form

```
[1 ]CF := FLWR | `<`tagid`>` FLWR `</` `tagid`>` | SortCF
[2] FLWR := (FCL | LCL)+ WCL? RCL
[3] FCL := `for` `$` Variable `in` `distinct`? PathExpr
[4] LCL :=`let` `$` Variable `:=` `distinct`? (PathExpr | CF)
[5] WCL :=`where` BooleanExpr
[6] RCL := `return` ( TaggedReturnItem | `{` ReturnItem `}` )
           (`,` (TaggedReturnItem | `{` ReturnItem `}` ) )*
[7] ReturnItem := Variable |AGG`(`Variable`)` | CF
                  | TaggedReturnItem
[8] AGG:=min |max | sum | avg | count
[9] SortCF:= CF `sortby` `(` SortSpecList `)`
[10]SortSpecList := (Qname | Variable | AGG(Variable)
          (`ascending`| `descending` )? (`,` SortSpecList)?
[11]AttributeList:=(S(Qname`=`AttributeValue AttributeList)?)?
[12]AttributeValue := StringListeral | [``] Variable[``]
[13]TaggedReturnItem := `<`Qname AttributeList (`/>` | (`>`
                  `{` ReturnItem * `}` `</` `Qname S?)?`>` )`
```

(c) XQuery Canonical Form(CF)

```
PredicateExpr := BooleanExpr | RangeExpr
RangeExpr := ArithmeticExpr | ArithmeticExpr `to` ArithmeticExpr
BooleanExpr := RelationalExpr | `not`? `empty` (Variable) | LogicExpr
LogicExpr := `not` BooleanExpr | BooleanExpr `and` BooleanExpr
                | BooleanExpr `or` BooleanExpr
RelationalExpr := ArithmeticExpr (`eq` | `ne` | `lt` | `le` | `gt` | `ge`
                                  | `==` | `!==` ) ArithmeticExpr
ArithmeticExpr := AtomicExpr | AdditiveExpr
                  | MultiplicativeExpr | UnaryExpr
AdditiveExpr := ArithmeticExpr (`+` | `-` ) ArithmeticExpr
MultiplicativeExpr := ArithmeticExpr (`*` | `div` | `mod` ) ArithmeticExpr
UnaryExpr := (`+` | `-` )ArithmeticExpr
AtomicExpr := Literal | Variable | AGG`(`Variable`)`
Literal := NumericLiteral | StringLiteral
```

(b) Predicate expression in the Canonical Form

```
Expr := CF | CF ( `intersect` | `except`) Expr
```

(d) XCF: a language built on the CF

Note: 1. In grammar b), `empty` only tests a Let variable
2. In grammar c):
(i) variables returned must be defined in FOR/LET clauses
(ii) variables used in aggregate functions must be defined in LET clauses
3. All other non-terminals without production rules here are defined in references [2, 3, 6]

Figure 1: XQuery Canonical Form

Figure 1b gives the definition of predicate expressions and boolean expressions in the canonical form. The grammar in Figure 1b is "loose" in the sense that it does allow operators to have operands that do not make sense. We assume similar type checking as in conventional programming languages. Note that a boolean term "not empty (Variable)" is allowed in the grammar of boolean expressions, the reason for including it can be seen in the next section.

Figure 1c shows the actual "form" of the canonical form. It has an optional tag at the top which resembles element constructors defined in XQuery. The structure inside the top tag resembles the structure of FLWR expressions, but with restrictions as indicated. Specifically, it allows only simple path expressions to appear in the FOR clause, while it allows CF in the LET clause. Only boolean conditions can appear in the WHERE clause, and the RETURN clause can only return variables (optionally surrounded with tags) defined in the FOR/LET clauses, six traditional aggregate functions on the variables defined in the LET clauses, and nested canonical form as well. The returned contents can be sorted as required. For ease of presentation, here we ignore the possibility of returning arithmetic expressions on returned components mentioned above. However, this can be added very easily as shown in Section 9. A possible language built on this canonical form, called XCF, is shown in Figure 1d.

It is easy to see that this canonical form is simple, has a clear semantics (that could be formally defined using a subset of the XQuery Formal Semantics [8]) and is easy to understand. Moreover, note that the grammar for path expressions and predicate expressions does not allow canonical forms to appear in them, so they will not affect the structure of a query. Figure 1a, 1b and 1c address different aspects of the canonical form separately, i.e. navigation, condition and structure. This separation makes the canonical form easy to extend because modifying one component will not affect other components. New navigation or condition rules can be defined by simply plugging in a new set of production rules for navigation and condition. Therefore, the language represented by this canonical form is very flexible, but still with a uniform structure.

3 A translatable subset of XQuery

Although the defined canonical form seems small, its power is not overly limited. In this section, we identify a subset of XQuery, which can be translated to XCF, and we will see that this translatable subset is an important subset of XQuery. We first give a set of transformation rules and then give a grammar for this subset. The correctness of these transformation rules follows from XQuery semantics.

3.1 Transformation rules

We use similar notation as in the work of Manolescu et al. [28]: lower case letters such as *x, y, z* represent individual XQuery variables; capital letters such as *E, PE, EC* denote XQuery expressions, path expressions and element constructors respectively; a single For clause "for $\vec{v}$ in $\vec{FE}$" represents "for v_1 in FE_1,v_2 in FE_2, $\cdots$, v_n in FE_n"; a single Let clause "let $\vec{u} := \vec{LE}$" represents "let $u_1 := LE_1$,$u_2 := LE_2$, $\cdots$, $u_m := LE_m$". In these cases, $\vec{FE}$ is a vector of expressions with arity n, and $\vec{v}$ are consecutively bound to each binding value combinations that result from $\vec{FE}$'s evaluation; similar meanings are extended for $\vec{LE}$ and $\vec{u}$.

Simple path expressions in the RETURN clause. One motivation for defining the above canonical form is to deal with path expressions appearing in RETURN clauses. Due to the implicit

list unnesting of XPath semantics [4], returning a path expression is essentially equivalent to introducing a nested FLWR expression which defines a For-variable iterating on the values of the path expression and returns those values in the RETURN clause. That is, "v_i/PE" is equivalent to "for t in v_i/PE return t". Returning an aggregate function on a path expression is essentially equivalent to introducing a nested FLWR expression which defines a Let-variable bound to the values of the path expression and returns the aggregate function applied on this Let-variable in the RETURN clause. That is, "$agg(u_j/PE)$" is equivalent to "let $t := u_j/PE$ return $agg(t)$". Therefore, path expressions and FLWR expressions can be viewed in a uniform way in the canonical form, which contributes to the simplicity and elegance of the canonical form. The following rule pair 1a and 1b illustrate such transformation:

Rule 1a: $PE(\vec{v}, \vec{u})$	$\Rightarrow$	for t in $PE(\vec{v}, \vec{u})$ return t

Rule 1b: $agg(PE(\vec{v}, \vec{u}))$	$\Rightarrow$	let $t := PE(\vec{v}, \vec{u})$ return $agg(t)$

Note that $PE(\vec{v}, \vec{u})$ is a valid simple path expressions (i.e., comply with the production rule of path expressions) of variables in $\vec{v} \cup \vec{u}$ which are defined in the FOR/LET clauses of a query in the canonical form, and $agg(PE(\vec{v}, \vec{u}))$ is an aggregate function on valid simple path expressions of variables in $\vec{v} \cup \vec{u}$.

Simple path expressions in the WHERE clause. In XQuery, simple path expressions or aggregate functions on them can appear in the conditions of a WHERE clause. These conditions can be translated to the conditions allowed in the canonical form. Denote these conditions as $WC(WC_1, WC_2, \cdots, WC_n)$, each of WC_i is a boolean function $f_i(opd_1, opd_2)$ in which a general comparison operator f_i operates on operands involving simple path expressions. Depending on the specific form of each operand in WC_i, we have the following transformation rules. Again we transform path expressions as above, i.e., for each path expression without an aggregate function applied on it, we introduce a For-variable bound to it; otherwise we introduce a Let-variable bound to it.

1. Only one operand involves a simple path expression but without an aggregate function on it, and the other operand, opd_2, is a constant, a For-variable, or an aggregate function on a variable.

Rule 2a: $f_i(PE, opd_2)$	$\Rightarrow$	let t_i := (for w_1 in PE where $f_i'(w_1, opd_2)$ return w_1) *not empty*(t_i)

   Here f_i'=$[f_i]_{ValueOp}$, which is a normalization of the general comparison operator given in W3C XQuery Formal Semantics [8].

2. Both operands involve simple path expressions but without aggregate functions on them.

Rule 2b:

$$f_i(PE_1, PE_2) \Rightarrow \begin{array}{l}\text{let } t_i := (\text{for } w_1 \text{ in } PE_1 \\ \qquad\qquad \text{for } w_2 \text{ in } PE_2 \\ \qquad\qquad \text{where } f_i'(w_1, w_2) \\ \qquad\qquad \text{return } w_1\ ) \\ \textit{not empty}(t_i)\end{array}$$

3. One operand involves a simple path expression, the other one involves an aggregate function on a simple path expression.

Rule 2c:

$$f_i(PE_1, agg(PE_2)) \Rightarrow \begin{array}{l}\text{let } t_i := (\text{for } w_1 \text{ in } PE_1 \\ \qquad\qquad \text{let } w_2 := PE_2 \\ \qquad\qquad \text{where } f_i'(w_1, agg(w_2)) \\ \qquad\qquad \text{return } w_1\ ) \\ \textit{not empty}(t_i)\end{array}$$

4. Only one operand involves an aggregate function on a simple path expression, and the other operand, opd_2, is a constant, a For-variable, or an aggregate function on a variable.

Rule 2d:

$$f_i(agg(PE), opd_2) \Rightarrow \begin{array}{l}\text{let } t_i := \text{PE} \\ f_i'(agg(t_i), opd_2)\end{array}$$

5. Each operand involves an aggregate function on a simple path expression.

Rule 2e:

$$f_i(agg(PE_1), agg(PE_2)) \Rightarrow \begin{array}{l}\text{let } t_{i_1} := PE_1 \\ \text{let } t_{i_2} := PE_2 \\ f_i(agg(t_{i_1}), agg(t_{i_2}))\end{array}$$

Existential/universal quantifiers in the WHERE clause. Existential and universal quantifiers can be eliminated by using rules 2f and 2g. Note that WC in these rules may require further transformation as necessary. Therefore, we could handle a where condition of arbitrary quantifier nesting.

Rule 2f:

$$\textit{some } z \text{ in } FE \textit{ satisfy } WC \Rightarrow \begin{array}{l}\text{let } t_i := (\text{for } z \text{ in } FE \\ \qquad\qquad \text{where } WC \\ \qquad\qquad \text{return } z\ ) \\ \textit{not empty}(t_i)\end{array}$$

Rule 2g:

$$\textit{every } z \text{ in } FE \text{ satisfy } WC \Rightarrow \begin{array}{l}\text{let } t_i := (\text{for } z \text{ in } FE \\ \qquad\qquad \text{where } \textit{not } WC \\ \qquad\qquad \text{return } z\ ) \\ \textit{empty}(t_i)\end{array}$$

Transforming WHERE clause. Now given a where condition $WC(WC_1, WC_2, \cdots, WC_n)$ in a FLWR expression and the transformation rules for each kind of WC_i as shown in rules 2a-2g, rule 3 shows the transformation rule for the whole WC.

Rule 3:

$$\begin{array}{l}\text{for } \vec{v} \text{ in } \vec{FE} \\ \text{let } \vec{u} := \vec{LE} \\ \text{where } WC(WC_1, WC_2, \cdots, WCn) \\ \text{return } RE\end{array} \Rightarrow \begin{array}{l}\text{for } \vec{v} \text{ in } \vec{FE} \\ \text{let } \vec{u} := \vec{LE} \\ \text{let } \vec{t} := \vec{LE'} \\ \text{where } WC(WC_1', WC_2', \cdots, WCn') \\ \text{return } RE\end{array}$$

The arity of $\vec{t}$ depends on how many and which rules of 2a-2g are used, and LE' is a vector of expressions bound to t_i in those rules. The translated condition WC_i' is determined as follows:

1. if WC_i involves no simple path expressions, then WC_i'=WC_i;
2. if WC_i is one of the left sides of rules 2a-2c and 2f, then WC_i'= *not empty* (t_i); if WC_i is the left side of the rule 2g, then WC_i'= *empty* (t_i);
3. if WC_i=$f_i(agg(PE), opd_2)$, i.e. the left side of rule 2d, then WC_i'=$f_i'(agg(t_i), opd_2)$;
4. if WC_i=$f_i(agg(PE_1), agg(PE_2))$, i.e. the left side of rule 2e, then WC_i'=$f_i'(agg(t_{i_1}), agg(t_{i_2}))$.

Predicates in simple path expressions. In XQuery, path expressions can have predicates in each navigation step. Rule 4 shows a transformation from predicates in path expressions to the WHERE condition of the canonical form. For predicates which are allowable conditions in the canonical form, it is easy to transform them to the WHERE condition. If these predicates only appear in the last step of a path expression, pushing them into the WHERE condition is straightforward. If these predicates appear in the middle of a path expression, then one possible way to push them is to introduce a nested CF in the place where they occur. Predicates that themselves involve path expressions can be translated to the WHERE condition of the canonical form using rules 2a-2g.

Rule 4:

for x in $PE_1[C]/PE_2$ ⇒

for x in (for y in PE_1
where $C(y)$
return y
)$/PE_2$ ⇒

for x in (for y in PE_1
where $C(y)$
return y/PE_2
)

The inner FLWR query in the FOR clause on the rightmost side of the above rule can be further translated by using rule 1, and the whole FOR clause is rewritten as a FOR clause having an inner FLWR query which itself is a nested CF of level 2. This can be further transformed by using rule 6a below. Note that the step indicated by the second ⇒ is similar to the rewriting rule NR_2 given by Manolescu, et al. [28].

Simple path expressions connected by set operators. In XQuery, all the collection operators remove duplicates based on node identity or value depending on whether the element of a collection is a node or simple value. In the result, nodes appear in document order, and the order of simple values is implementation dependent. Because of this order issue, we cannot simply transform an original query having a union operator as the union of two canonical queries. To avoid the ambiguous semantics of union operator, we choose to ignore the union operator here.

1. Intersect. The semantics of intersect is defined using existential quantification [8]. Note that existential quantification in the WHERE condition can be translated by using rule 2f.

Rule 5a:

for x in FE_1 *intersect* FE_2 ⇒

for x in (for y in FE_1
where *some* z in FE_2 *satisfy* $z = y$
return y)

2. Except. Its semantics is also based on existential quantification [8].

Rule 5b:

for x in FE_1 *except* FE_2 $\Rightarrow$ for x in (for y in FE_1
where *not some* z in FE_2 *satisfy* $z = y$
return y)

The right side of the rule 5b can be further transformed by using rule 2f.

FLWR expressions nested in the FOR/WHERE clause. A FLWR expression nested in the FOR/WHERE clause can be pulled out and bound to a LET variable. The new LET clause is put in the immediately surrounding CF and right before the original place where it occurs. This is shown in rule 6a and 6b.

Rule 6a:

for x in CF $\Rightarrow$ let t:= CF
for x in t

Rule 6b:

where WC(CF) $\Rightarrow$ let t:= CF
where WC(t)

Element constructors. An element constructor could have various expressions as its content, including constants, variables, path expressions, and FLWR expressions, among others. If these four expressions are returned as attributes, then we can transform expressions more complex than simple string constants and variables by first binding them to temporary Let variables and then returning the string values of those variables as attributes. This is covered in the canonical form. If these expressions are returned as children of the element being constructed, all of these four expressions can be rewritten as CFs. For case 1, we could use a LET clause to bind the constant to a Let variable and then return it. For case 3, path expressions can be rewritten to CFs as shown above. Case 2 and case 4 are covered in the canonical form. Hence the content of an element constructor can be CFs or be rewritten as CFs.

An element constructor could also be nested anywhere in path expressions, predicates, the FOR clause, the WHERE clause, the LET clause, and the RETURN clause. In the first four cases, we could use the same translation as rule 6, i.e. use a LET clause to bind the result of an element constructor to a temporary Let variable. The fourth and the fifth cases are already covered in the canonical form. Rule 7 shows the case when an element constructor appears in the FOR clause. Rules for other cases can be given similarly.

Rule 7:

for x in (<tag>E</tag>) $\Rightarrow$ let t := (<tag>E</tag>)
for x in t

Therefore, due to the structure of the canonical form, an element constructor itself is a CF. It can be a nested CF in another CF, or it itself has nested CFs. Thus the expressivity of the defined canonical form provides a uniform way to view element constructors, FLWR expressions and path expressions.

Conditional expressions. The normalization rule $NR_6(b)$ given by Manolescu, et al. [28] eliminates conditional expressions directly nested within a FOR/WHERE/RETURN clause. However their normalization rules for FOR/RETURN cases use union operations which modify the order of the result. In order to preserve the order during transformation, we develop different rules to eliminate conditional expressions without using union operations. Rule 8b uses the exclusivity of x and $\neg x$ to ensure that exactly one value is returned from a clause that appears to have two results.

In addition, we give a rule, numbered rule 8d, for the case that conditional expressions appear in a LET clause. Both the right side of rules 8c and 8d can be further transformed by using rule 8b.

1. Conditional expressions appearing in the WHERE clause:

Rule 8a:

for $\vec{v}$ in $\vec{FE}$
let $\vec{u}$ in $\vec{LE}$
where (if $WC_1(\vec{v},\vec{u})$ then $E_1(\vec{v},\vec{u})$
else $E_2(\vec{v},\vec{u})$)
return $RE(\vec{v},\vec{u})$

$\Rightarrow$

for $\vec{v}$ in $\vec{FE}$
let $\vec{u}$ in $\vec{LE}$
where ($WC_1(\vec{v},\vec{u})$ and $E_1(\vec{v},\vec{u})$) or
($\neg WC_1(\vec{v},\vec{u})$ and $E_2(\vec{v},\vec{u})$)
return $RE(\vec{v},\vec{u})$

2. Conditional expressions appearing in the RETURN clause:

Rule 8b:

for $\vec{v}$ in $\vec{FE}$
let $\vec{u}$ in $\vec{LE}$
where $WC_1(\vec{v},\vec{u})$
return
if $WC_2(\vec{v},\vec{u})$ then $E_1(\vec{v},\vec{u})$
else $E_2(\vec{v},\vec{u})$

$\Rightarrow$

for $\vec{v}$ in $\vec{FE}$
let $\vec{u}$ in $\vec{LE}$
where $WC_1(\vec{v},\vec{u})$
return
let $\vec{v'}$:= $\vec{v}$
where $WC_2(\vec{v'},\vec{u})$
return $E_1(\vec{v'},\vec{u})$
let $\vec{v'}$:= $\vec{v}$
where $\neg WC_2(\vec{v'},\vec{u})$
return $E_2(\vec{v'},\vec{u})$

3. Conditional expressions appearing in the FOR clause:

Rule 8c:

for $\vec{v}$ in $\vec{FE}$
w in (if $WC_1(\vec{v})$ then $E_1(\vec{v})$
else $E_2(\vec{v})$)
let $\vec{u}$:= $\vec{LE}$
where $WC_2(\vec{v},w,\vec{u})$
return $RE(\vec{v},w,\vec{u})$

$\Rightarrow$

for $\vec{v}$ in $\vec{FE}$
let w':=(let $\vec{v'}$:= $\vec{v}$
return
if $WC_1(\vec{v'})$ then $E_1(\vec{v'})$
else $E_2(\vec{v'})$)
for w in w'
let $\vec{u}$:= $\vec{LE}$
where $WC_2(\vec{v},w,\vec{u})$
return $RE(\vec{v},w,\vec{u})$

4. Conditional expressions appearing in the LET clause:

Rule 8d:

for $\vec{v}$ in $\vec{FE}$
let w := (if $WC_1(\vec{v})$ then $E_1(\vec{v})$
else $E_2(\vec{v})$)
let $\vec{u}$:= $\vec{LE}$
where $WC_2(\vec{v},w,\vec{u})$
return $RE(\vec{v},w,\vec{u})$

$\Rightarrow$

for $\vec{v}$ in $\vec{FE}$
let w:=(let $\vec{v'}$:= $\vec{v}$
return
if $WC_1(\vec{v'})$ then $E_1(\vec{v'})$
else $E_2(\vec{v'})$)
let $\vec{u}$:= $\vec{LE}$
where $WC_2(\vec{v},w,\vec{u})$
return $RE(\vec{v},w,\vec{u})$

Manolescu, et al. [28] also gives a normalization rule $NR_6(a)$ (listed below) which deals with the case that conditional expressions appear in path expression steps and element constructors, among

others. This rule is a rewriting rule which does not eliminate conditional expressions. However, since path expressions and element constructors can be translated to CFs, we do not really need rule $NR_6(a)$ but use our rules to eliminate conditional expressions eventually. Again, this shows the elegance of a uniform view of three different XQuery expressions in the canonical form.

NR_6 **(a):**	
E(if $C(v)$ then $E_1(v)$ else $E_2(v)$)	$\Rightarrow$ if $C(v)$ then E($E_1(v)$) else E($E_2(v)$)

Enclosed/parenthesized expressions. For these kinds of expressions, we could use the same transformation strategy as before, i.e., binding them to Let variables. We omit the details for the sake of space.

3.2 A grammar of the translatable subset of XQuery

Figure 2 shows a grammar for the translatable subset of XQuery. In this grammar, the production rules for path expressions and boolean expressions are similar as those in Figure 1a and 1b, so we omit the same productions and only list the modified or new ones. The new production rules are numbered exactly as their counterparts, if any, in the XQuery grammar. Note that this grammar is "loose" in the sense that some queries generated by this grammar do not make sense. Again, we assume the use of a type checking mechanism.

```
[4`] Expr := SortExpr | FLWRExpr | IfExpr | PathExpr | ElementConstructor | CollectionExpr
[5`] SortExpr := Expr `sortby` `(` SortSpecList `)`
[8`] FLWRExpr := (ForClause | LetClause)+ WhereClause? `return` Expr
[11`] IfExpr := `if` BooleanExpr `then` Expr `else` Expr
[26`] SortSpecList := (Qname | Variable | AGG (Variable) ) ( `ascending`| `descending` )? (`,` SortSpecList)?
[27`] ForClause := `for` Variable `in` `distinct`? Expr
[28`] LetClause := `let` Variable `:=` `distinct`? Expr
[29`] WhereClause := `where` BooleanExpr

CollectionExpr := Expr (`union` | `intersect` | `except`) Expr

// Similar to Figure 1b, with modification to the following production rules and including new rules as indicated:
BooleanExpr := RelationalExpr | QuantifiedExpr | LogicExpr | `not`? `empty``(` PathExpr `)`

[9`]QuantifiedExpr := (`some` | `every`) Variable `in` Expr `satisfies` BooleanExpr

AtomicExpr := Literal | Variable | AGG`(`Variable`)` | PathExpr | AGG`(`PathExpr`)`

//Similar to Figure 1a, with modification to the following production rules and including new production rules:
[3`] ExprSequence := Expr (`,` Expr)*

[47`] PrimaryExpr := `.` | NodeTest | Variable | ParenthesizedExpr | ElementConstructor
[50`] ParenthesizedExpr := `(` ExprSequence? `)`

[57`] ElementConstructor := `<` Qname AttributeList (`/>` | ( `>` ElementContent* `</` (Qname S?)? `>` )
[60`] CdataSection := `<![CDATA[` Char* `]]>`
[63`] ElementContent := Char | ElementConstructor | EnclosedExpr | CdataSection
[64`] AttributeList := ( S (Qname `=` (AttributeValue| EnclosedExpr) AttributeList )? )?
[65`] AttributeValue := [`"`]AttributeValueContent*[`"`] | [`']AttributeValueContent*[`']
[66`] AttributeValueContent := Char | EnclosedExpr
[67`] EnclosedExpr := `{` ExprSequence `}`
(Note that Char and S are defined in reference [2], Qname and NCName are defined in reference [3])
```

Figure 2: A translatable subset of XQuery

There are several changes from the original XQuery grammar:

- Production rules 57′ and 64′ only allow QName as tag or attribute names.
- Production rule 63′ does not allow CharRef and PredefinedEntityRef as element contents.
- Production rule 66′ does not allow CharRef and PredefinedEntityRef as attribute value contents.

Clearly, this subset covers important features of XQuery. The subset covers 31 of the 51 queries in the first five of W3C's XML Query Use Cases [5]. The uncovered queries include those that have built-in function calls (e.g., name(), after(), etc.), user-defined functions (e.g., those used in the 'PARTS' case), and those with '*' in path navigation steps. We discuss how to extend the canonical form to cover them in Section 9.

It can be shown that a meaningful query generated by this grammar can be translated to the canonical form or XCF by using those transformation rules. Due to the space limits, we omit the details of proof.

4 Processing XQuery Using T/RDBMS

In common with other researchers [18, 23, 32, 14, 21, 31, 28], we wish to capitalize on the extensive work invested in relational database technology, and study how to evaluate XQuery queries over XML data by using relational database systems. However, we do not limit ourselves to standard commercial relational database systems, but build instead on T/RDBMS, a system that enhances SQL to include support for a structured text ADT [13].

The Text and Relational Database Management System (T/RDBMS) was developed between 1994 and 1997 at the Center for the New Oxford English Dictionary and Text Research, University of Waterloo, in conjunction with other members of the Canadian Strategic Software Consortium (CSSC) [11, 17]. This software provides a federation of text and relational database engines, under a Hybrid Query Processor (HQP), by defining a minimal extension of SQL-92 that includes a *text ADT* [13]. A prototype implementation has run for several years with high efficiency for several classes of application (http://db.uwaterloo.ca/trdbms/).

4.1 Structured Text ADT

A structured text is represented conceptually by T/RDBMS as an ordered tree. Each node in this tree corresponds to a component in the text and carries two string labels: one identifying the component type and name, and the other representing the data content subsumed by this structure. Thus, in contrast to the XML Query Data Model [7], there is only one node type in the tree, and, for example, XML attributes are distinguished from XML elements through the node name (e.g., ":year" labels a node corresponding to an XML attribute node named year and "<book>" labels a node corresponding to an XML element node named book). In addition to the label and content strings, each node in the text ADT model also includes a boolean value representing whether or not that node is "marked" in the text. These marks are set and reset by various operators defined for the text ADT, and they are used to identify subtexts *in context*.

4.1.1 Operations defined for text ADT

Since the text ADT is intended to be used within a database environment, the principal functionality to be supported within the ADT is to locate and extract desired subtexts to form collections that

can be further manipulated by the database system. The text operators include 17 functions. The first two functions convert from strings (usually VARCHARs or CLOBs) to text and from text to strings; six functions are used for manipulating marks in texts; two functions query the existence of or the number of marks in a text; two functions extract subtexts from a text; the last five functions manipulate the DTD corresponding to an XML text. Details of these operators are provided elsewhere [13].

The function **extract_subtexts()** is crucial for simulating XQuery expressions. It becomes the basis of our *extraction* operator which will be described shortly in the algebra section. For now it is worth pointing out that all columns of a sub-relation generated by **extract_subtexts()** are of type *text*, each of which can be implemented as an unmaterialized view over a stored XML-encoded character string. As a result of this property, many of the sample applications implemented in T/RDBMS store a single document (such as the 540 megabyte *Oxford English Dictionary*) as a one-row, one-column relation on which one or more views are defined using **extract_subtexts()**. For example, if the complete works of Shakespeare are stored as a one-row, one-column relation, **extract_subtexts()** followed by a simple projection can be used to define a view in which each play is represented in a multi-row, one-column relation. Alternatively, for example, all speeches from Shakespeare's plays can be accessed through a 5-column "extracted" view representing the name of the play, act number, scene number, speaker, and content, or instead through a 3-column view representing the speech itself, the speech in the context of a complete scene, and the speech in the context of a complete play. Of course, any of these views could be materialized if desired.

4.1.2 Text matching patterns

Several of the text ADT operators match structured text against a tree pattern, which corresponds to XQuery's use of an XPath expression [4]. In the tree pattern matching sub-language, '%' matches zero or more consecutive characters within a text label, `A[B]` represents an ancestor-descendant relationship, `A[^B]` represents a parent-child relationship, `[C&D]` represents two descendants from some node, and hash marks `#` flag which nodes are to be marked upon a successful match. Thus, a tree pattern differs from an XPath expression by identifying several nodes in a tree that correspond to a single match rather than identifying only the last node in some path. For example, whereas the XPath expression "`/a[b]//c`" returns bindings to nodes of type `c`, a similarly constructed tree pattern "`^a#[^b#&c#]`" containing three hash marks returns *triples* of bindings to nodes of type `a`, `b`, and `c`, respectively.

4.2 Why this approach?

Due to the design of the Text/Relational DBMS, it is possible to interpret many XQuery queries using that system. On one hand, relational queries involving join, grouping, aggregation, set operations, and sorting are supported directly in SQL; on the other hand, full-text search queries or XPath-like queries are naturally supported by using text ADT operators and the tree pattern matching sub-language. Moreover, since T/RDBMS seamlessly integrates a text ADT with SQL, queries involving both traditional SQL and structured text manipulation can be easily accommodated. Furthermore, we claim that T/RDBMS is particularly suitable to implement an XQuery processor for the following reasons:

- Storing whole documents in a column of text rather than chopping them into pieces to be mapped to relational tables and columns preserves the original XML data. Hence the problem of reconstructing XML data from relational tables, which is tedious and inefficient for

alternative approaches, does not exist. Meanwhile, there is no undesired duplication of XML data needed to keep the document as a whole while providing access to various components. However, this approach is only feasible if appropriate structured text operators are supported.

- Path expressions in an XQuery query are easily translated into tree pattern matching operators, which provide simultaneous access to related components. We anticipate extensive opportunities for query optimization through appropriate choices of tree patterns that cover multiple XPath expressions spanning one or more XQuery queries.

- Text ADT operations such as **extract_subtexts()** isolate desired text fragments but also remember the context within which they occur. This provides a means for defining views over the text and facilitates subsequent queries over those views. This same approach allows intermediate results to be constructed dynamically. If we think of XQuery expressions as defining views on the XML data, it is not surprising that interpreting XQuery using a T/RDBMS is straightforward.

- Because it is built on SQL-92, T/RDBMS supports for a wide variety of datatypes that are therefore readily available to be included in an XQuery processor. If the underlying SQL is upgraded to SQL-99 or beyond, extra features are immediately also available for XQuery support.

5 Our algebra

Our algebra is an extended relational algebra based on tables rather than relations [29] with support for text functions [13]. Thus collections of rows are ordered and permit duplicates, and data values may include structured text as well as integer, string, date, etc. More specifically, selection and join conditions may include text-related conditions, and the projection list may include text functions as well. An important concept in the text ADT is to preserve the context of selected text as well as extracting the subtext. These contexts are useful in our evaluation of XQuery queries since they indicate where these extracted subtexts come from, which in turn provide a possible mechanism to simulate 'node identity'. Hence, translation issues related to keeping document order and node identity can be solved by utilizing these contexts.

It is observed that these contexts can be 'virtual' so that users or applications do not see them, but only the DBMSs manipulate them for the purpose of bookkeeping or enforcing correctness. Therefore, we choose to distinguish between *visible* and *hidden* columns. Visible columns are those visible to users or applications, while hidden columns are used solely by the DBMS. Similar consideration has appeared in the database literature, where a typical virtual attribute is a tuple identifier which is used to represent each tuple uniquely.

Since traditional operators are well understood, we omit them here and only focus on the non-standard operators, as shown in table 1.

Extraction. An operator crucial for simulating XQuery expressions is the *extraction* operator, $\chi_{A,s}(R)$, adapted from the *extract_subtexts*() function in [13]. The *extraction* operator takes a relation as input and two parameters, A and s, where A is a column of table R of type *text*; s is a tree pattern to match against each text in the given column A. The tree pattern matching sub-language is a variant of XPath that describes tree patterns instead of path patterns.

The result of $\chi_{A,s}(R)$ is computed by considering each row of R in turn, as illustrated in Figure 3. Each application of a tree pattern (having k flagged node labels) against a text tree produces a

Operator	Definition
$\chi_{A,s}(R)$	Extract components matching pattern s from column A in table R
$\gamma_A(R)$	Partition table R on grouping columns A
$\tau_A(R)$	Sort table R based on sorting columns A
$\mu_{A(R)}$	Aggregate construction on column A of table R
$\nu_{A^{C_1},A^{C_2},tag(R)}$	Element construction on columns A^{C_1}, A^{C_2} of a table R

Table 1: List of operators

$2k$-column table with one row for every distinct tuple of bindings and one column for each of the extracted subtexts that matches a flagged node label plus one column indicating *where* in the input text this subtext come from, referred to as the *mark column*. Thus given a table R with n rows and a column A of type *text*, an application of extraction χ to A produces n $2k$-ary tables, one per row in R. The resulting tables are then "attached" to R as if by a join that correlates each row in R with all rows produced from the A-value in that row. Hence the result of applying this operator is a "flat" table. The new column names by default are the same as the root names of the extracted texts, with suitable renaming as necessary.

Figure 3 shows the extraction of 'book' and 'author' from a table with one row and one column containing a bib text, i.e., $\chi_{bib,\text{`}<bib>[\hat{}<book>\#[\hat{}<author>\#]]'}(R)$. The first column is the original bib text, the third and the fifth column are the extracted book text and author text, while the second and the fourth columns are the associated mark columns (with marks represented by italics) for book and author. These marks can be used to simulate "node identity", and need not be visible to users, i.e., they are "hidden" columns. A possible implementation of these marks is using (*doc id* + *position* in a document).

Sorting. A sorting operator τ is used to sort a table according to some sorting criteria. $\tau_A(R)$ takes a table R as input and a sorting columns list A as a parameter. The result of this operation is the table R, but with the rows of R sorted in the order indicated by A. The parameter may indicate the sorting is to be done in ascending or descending order, or based on document order rather than values of each column in A. That is, the actual sorting may be performed on respective hidden "mark columns" associated with each column in A.

Groupby. In order to facilitate optimization involving grouping operations and aggregate constructions, we separate the grouping operation from the computation of aggregate functions, similar to [25]. Our *groupby* γ operator partitions a table into a grouped table that contains one row per partition, and performs a *grouped table projection* π' immediately. In this way, a grouped query in SQL can be written in term of these two consecutive operators: γ followed by π'. As our algebra supports *text* ADT, the groupby operator not only partitions a table based on values, but also on "node identity". For a grouping column *col* of simple types, γ partitions a table based on the value of *col*; otherwise, it partitions the table based on the value of the (hidden) node identifier of *col*.

Construction operators. To facilitate construction of XQuery results, we define two construction operators in our algebra:

<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<bib> <*book* year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<book year="1994"> <title> TCP/IP Illustrated </title> <author> W. Stevens </author> </book>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <*author*>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<author> W. Stevens </author>
<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <*book* year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<book year="2001"> <title> Computer Networking </title> <author>J. Kurose </author> <author>K. Rosse </author> </book>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <*author*>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<author> J. Kurose </author>
<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <*book* year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <author>K. Rosse</author> </book> </bib>	<book year="2001"> <title> Computer Networking </title> <author>J. Kurose </author> <author>K. Rosse </author> </book>	<bib> <book year="1994"> <title>TCP/IP illustrated</title> <author>W. Stevens</author> </book> <book year="2001"> <title>Computer Networking</title> <author>J. Kurose</author> <*author*>K. Rosse</author> </book> </bib>	<author> K. Rosse </author>

Figure 3: Results of extraction with a tree pattern that flags book and author nodes

- **Aggregate constructor** (μ_A) is used mainly for representing a set-valued column as a single tree. It is a form of aggregate function applied on column A. Instead of computing an aggregated scalar value for each group, it constructs a tree from the values over column A in that group, appropriately handling `null` values.

  We adopt the *catenate* operator originally defined in [24] to manipulate vectors. A *vector* is an ordered collection of trees organized in a larger tree structure, with the root labeled *vector* and the roots of the trees in the collection being the children. The catenate operator takes two vectors and returns a single vector including all subtrees of the arguments. Hence, the root of the generated tree of μ_A is labeled with "vector". This root is a "virtual" root because when a vector V_1 is concatenated with another tree to form a new tree T, all the children of V_1 becomes the children of T directly. In the presence of `null`, catenating a vector T with `null` returns T itself, and catenating two `null`s returns `null`. Therefore, if the group being aggregated is the empty set, then the aggregate result is `null`. Again by default, the column name in the resulting table corresponding to the aggregated value of applying μ is the same as before, i.e., A.

- **Element constructor** (ν) is another construction operator. It takes three parameters, A^{C_1}, A^{C_2}, and *tag*, where (1)A^{C_1} is a list of columns which becomes the attributes of the resulting element being constructed, here denoted *elet*. The order that columns occur in the list A^{C_1} does not matter. (2)A^{C_2} is a list of columns which becomes the sub-elements of *elet*. The order that columns occur in the A^{C_2} list is also the order that these sub-elements occur in the resulting element. (3)*tag* is the tag name of *elet*. By default, the column corresponding to *elet* in the resulting table is named *tag*, but if *tag* conflicts with an existing column name,

some renaming is necessary.

Element constructor is applied to each tuple and the result is computed as: concatenate the value T_i of each column i appearing in A^{C_1} and A^{C_2} to construct a tree T with all T_i as children, and root labeled *tag* instead of *vector*. Since the order of each column appearing in A^{C_2} is important, we apply the *catenate* operator in a way such that each child is in the same order as it is in A^{C_2}. Hence, while *aggregate constructor* is a vertical concatenation, *element constructor* is a horizontal concatenation. Figure 4 shows the application of aggregate constructors on the table resulting from grouping on columns G and A, followed by the application of an element constructor on column A, B, and C. For simplicity, we assume that tuples are in the right order before applying the aggregate constructor.

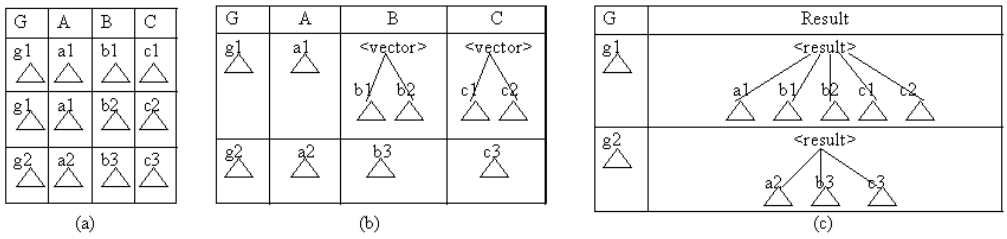


Figure 4: Aggregate constructor on column B, C, followed by element constructor <result>@A, B, C</result>

6 Translating XQuery FLWR expressions

In this section, we discuss the translation of XQuery FLWR expressions to our algebra. We focus on the subset of XQuery that can be transformed to our canonical form by using transformation rules given in Section 3. Note that in the canonical form we defined, there are two places where nested canonical forms can appear: one is in LET clauses, and the other is in RETURN clauses.

We start with the following definitions:

Definition 6.1 (Query Tree.) We represent an XQuery in canonical form by a tree, called a *query tree*. Each node of the tree is one of four kinds: `V node` for For-variables, `U node` for Let-variables, `aggU node` for aggregate function applied on Let-variables, and `CF node` for whole FLWR expressions. V node, U node and aggU node correspond to one of three kinds of returned items in the RETURN clause of an XQuery canonical form, and a CF node either corresponds to the fourth kind of returned item in the RETURN clause or corresponds to a CF bound to a Let-variable. In the latter case, the incoming edge to this CF node is labeled with 'LET' and this CF node is referred to as a `LET-CF node`. V node, U node and aggU node are leaf nodes, and a CF node is an inner node, with a child for each Let-variable bound to another CF and each item in the RETURN clause of the CF in the order of their appearance in the LET and RETURN clauses.

Each CF node has attributes: FOR/Basic-LET/WHERE corresponding to the FOR/Basic-LET(LET clause without nested CF)/WHERE clause in the CF it represents, and two tags *result-tag* (the one surrounding the CF node, possibly absent) and *tag* (the one immediately following the RETURN clause of the CF node, possibly absent). Each leaf node has as its attribute its respective tag in RETURN clause. Figure 5b shows the query tree for the canonical query in Figure 5a, where

leaf nodes are represented with rectangles and inner nodes are represented with circles. For clarity, attributes of the nodes are not shown.

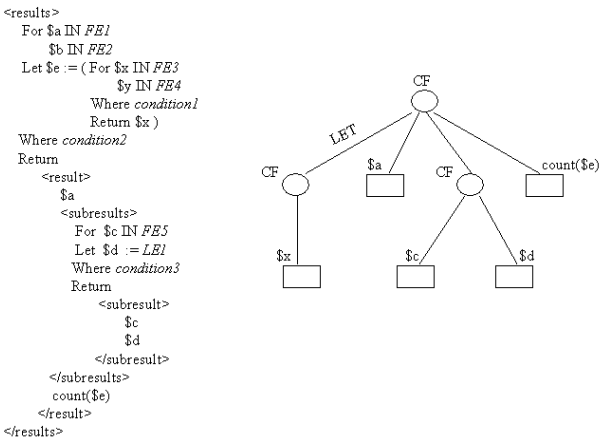


Figure 5: (a) An example of XQuery canonical form and (b) its corresponding query tree

Definition 6.2 (Result Generation Tree.) We represent the result generation of a canonical query CF against specific documents by a tree, called the *result generation tree*. Its root is a node with the tag specified by *result-tag* as its attribute. For each tuple binding extracted/evaluated from the FOR/LET clause that satisfies the WHERE condition, the root has a subtree that represents the result of the RETURN clause when evaluated with that tuple binding. The root of each subtree has the tag specified by *tag* as its attribute, and has a child for each item in the RETURN clause of the CF, in the order of the appearance of the items in the RETURN clause. Those children fall into two categories: (1) leaf children, the value of each non-CF item in the RETURN clause; (2) subtree children t_k, where t_k is the result generation tree for each nested canonical query CF_k in the RETURN clause. All of these children are evaluated with the variable binding of this node.

There exists an obvious *mapping* from a CF query tree (without branches for LET-CF nodes) to a result generation tree: each level of a CF query tree is mapped to two levels in the result generation tree, one level for iteration over variable bindings, and one level for the return structure, which is the CF query tree evaluated with a particular binding. We label the edges in the iteration level with the tuple binding values, the edges in the structure level with the items in the RETURN clause.

Note that since the result of evaluating a CF is a single tree, similar to the result of assigning a path expression to a LET variable, a Let-variable bound to a CF is essentially the same as a normal Let-variable bound to a path expression. Therefore, the LET-CF nodes in a CF query tree can be treated as separate queries and the query evaluation results, i.e., bindings, together with other For- and Let-variable bindings, appear in its associated iteration level.

Figure 6 shows the result generation tree for the canonical query in Figure 5. In Figure 6, we assume the Let-variable $\$e$ bound to a CF has two bindings e_1 and e_2, and assume there are four binding value combinations for $\$a, \$b, \$e$ satisfying the WHERE condition (i.e., $a_1b_1e_1, a_1b_2e_1, a_2b_1e_2, a_2b_2e_2$), and in the case of $a_1b_1e_1$ binding, there are three binding value combinations for $\$c, \$d$ (i.e., c_1d_1, c_2d_2, c_3d_3) satisfying the WHERE condition.

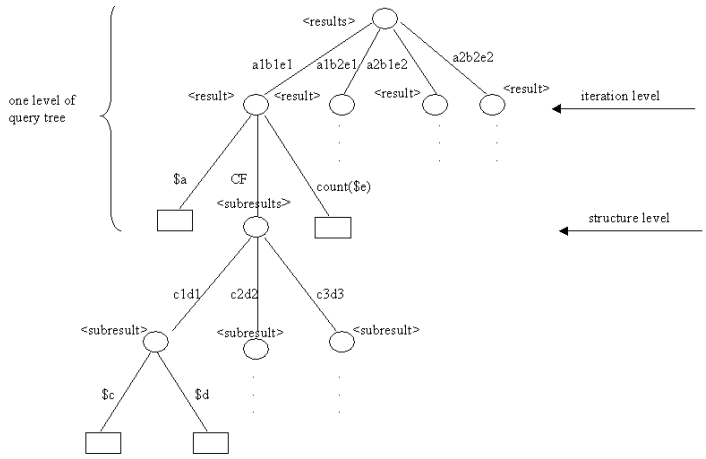


Figure 6: Result generation tree for query in Figure 5

Lemma 6.1 Given a canonical query Q and its mapped result generation tree RT, the depth-first traversal of RT as follows will generate the correct result for Q: output the tag (if any) of an inner node (or root) as start tag when first visiting the node, and output the tag (if any) as end tag when returning to the node; output the value of a leaf node, wrapped with the leaf node tag (if any) when the leaf is visited.

Proof: The definition of the result generation tree matches the semantics of XQuery as defined by W3C.

We now show how to translate an XQuery in canonical form into a corresponding relational algebra expression, whose execution will result in the correct result generation tree.

Definition 6.3 (Basic CF) A basic XQuery canonical form is an XQuery CF without nesting.

Translation 6.1 (Basic XQuery Canonical Form Translation.) Let Q be a query in basic XQuery canonical form. Suppose there is a two-column table R^0 (which could be a view), with one column for document name and one for document text, serving as a directory to support XQuery's `document()` function. This table provides the starting point of our translation, denoted as `Trans`, which is performed as follows by translating each clause of Q subsequently.

1. Translate FOR-clause: extract binding values of each For-variable $\$v_i$ using the extraction operator χ.

   1.1. If $\$v_i$ starts with the `document()` function (i.e., no other variable appears in $\$v_i$'s binding expression FE_i), `Trans` starts from the corresponding row and column of the initial table R^0 and forms a new one-row, one-column table R^1 with the document text *doc* stored in it, then `Trans` extracts $\$v_i$ from R^1 resulting in a new column together with a hidden mark column on R^1.

   1.2. If $\$v_i$ references another variable, then extraction starts from the column corresponding to that variable and forms a new column together with a hidden mark column on an existing table. The expression FE_i is translated to a pattern matching string s, and supplied as the second parameter to χ. Whenever `distinct` is present, duplicate elimination is performed based on value or node identity as desired, depending on the specification of `distinct`.

After this step, each variable has a corresponding column from which the variable takes its binding values, and a hidden mark column indicating where these binding values come from. Since these hidden mark columns are used only by the underlying DBMS, we use the term 'column' to mean 'visible column' unless specified explicitly, and we also ignore the *doc* column in the remainder of this paper.

2. Translate LET-clause: similar to step 1 except that after each extraction, perform a sorting operation on all the remaining columns (except Let-columns) based on document order, then perform a partition on all the remaining columns (including other Let-columns) except the newly extracted one, followed by an aggregate constructor on the newly extracted column. Thus each Let-variable's value is a text tree representing a collection as a vector. The order of elements in the grouping column list and the sorting column list is the same as in the query.

   The reason to include Let-variables in the grouping column list is to comply with SQL GROUP-BY constraints. However, the real groupby operation can be performed without this inclusion.

3. Form a single table: compute the cross product of multiple tables, if any, obtained from the previous steps. Let the resulting table be $R(a_1, a_2, ..., a_n)$.

4. Translate WHERE-clause:

   4.1. Rewrite the WHERE condition such that variable appearances are replaced by their corresponding column names in R. Denote the rewritten condition as WC.

   4.2. Include a selection operator σ with condition WC.

5. Translate RETURN-clause:

   5.1. Sort the table according to document order or as query requires. In the case of sorting on document order, the sorting column list is the For-variable list with each For-variable in the order of its appearance in the FOR clause.

   5.2. Project on columns corresponding to For-variables, Let-variables appearing in the RETURN clause, plus those aggregate functions applied on Let-columns. Note that if a returned variable has a tag around it, then the projection list includes an element constructor applied to that column rather than the column itself.

   5.3. Apply an element constructor using the columns of the previous step. Its parameters are supplied as indicated by the RETURN clause, with the columns in the second parameter (i.e., sub-elements list) in the order of their corresponding variable appearances in the RETURN clause and the third parameter as *tag* if present or *vector*. Let the resulting column be named a_t.

6. Generate the result: Project on column a_t and apply an aggregate construction operator on a_t treating all rows as a single group, followed by an element constructor adding the tag *result-tag* or *vector* on the aggregated value of a_t. Hence, the result of this step is a one-row, one-column table containing a constructed text tree T.

Note that SQL cannot directly evaluate $agg(x)$ when x is a Let-variable binding, since `Trans` generates a tree for x to encapsulate a group of values. Thus $agg(x)$ can be evaluated by first decomposing this tree, then applying an appropriate partition operation so that for each original tuple before decomposition, there is a group corresponding to it; and then applying *agg* on each

group in the usual way. Optimization applied to the resulting query plan can eliminate unnecessary tree creations and decompositions.

Lemma 6.2 Step 4 in the Translation 6.1 keeps a row in table R if and only if those rows are supposed to be kept according to XQuery semantics.

Proof: Due to the translation of FOR/LET clauses, it is guaranteed that all the variables appearing in the WHERE condition can be replaced by their corresponding column names. Hence the original WHERE condition can be resolved to a condition accepted by the relational selection operator, and the lemma follows by the semantics of the selection operator.

Theorem 6.1 `Trans` generates a semantically correct relational query corresponding to any basic XQuery canonical query Q.

Proof: Let Q be a query in basic XQuery canonical form. Therefore, its query tree QT consists of only one CF node *cf*. Let RT be the mapped result generation tree of QT on given documents. Now we need to prove that executing the translated algebra expression generates the correct result generation tree RT.

After step 1 and 2, the above translation `Trans` has extracted columns corresponding to variables appearing in the FOR/LET clauses and Step 3 formed a tuple for each binding combination of these variables in table R. It is easy to see that table R has the exact number of tuple bindings as XQuery semantics indicates, and it does not have extra tuple bindings. Step 4 eliminates rows that do not satisfy the WHERE condition by Lemma 6.2. Step 5 ensures constructing correct results with correct tags and subtrees in the correct order according to XQuery semantics and Step 6 groups all the subtrees together as a tree. Specifically,

- All the nodes at the structure level for *cf* are generated correctly. Since there is no CF node child, all variables in the structure level are only variables defined in the FOR/LET clauses of *cf*. Obviously, these nodes are correctly generated.
- All the nodes at the iteration level for *cf* are generated because we have all the required variable bindings and the construction step 5.3 is applied on all rows in the table R after correct sorting performed by Step 5.2. Moreover, a node at the iteration level is not generated multiple times because the construction is executed only once for each row in the table.
- The root of subtree for *cf* is generated correctly because of Step 6.

Obviously, this tree is the result generation tree for the query (without labels on the edges). By Lemma 6.1, we know that the correct result is therefore generated.

Now we extend the translation of basic XQuery canonical forms to canonical forms with nesting in RETURN and LET clauses.

Translation 6.2 (XQuery Canonical Form Translation.) Let the table R in the translation of basic XQuery CF be denoted as the *working table*. In the translation extended for general canonical form below, the working table is global to all the recursive calls. The translation of all the CF nodes in a query tree could modify or extend the working table.

Let Q be an XQuery in canonical form, QT be the query tree for Q and RT be its mapped result generation tree on given documents. Our translation `GTrans` is a depth-first traversal on the query tree QT. For each CF node in the query tree, our translation `GTrans` is an extension of the basic translation `Trans` with the following modifications:

- Step 0. Remember the current working table R and denote it as S.
- Step 2′. Translate LET-clause. If the LET-clause is a simple variable binding without nested CF, its translation is the same as that in basic translation. Otherwise, call `GTrans` on the nested CF node to compute the set of values which the LET variable is bound to.
- Step 5′. Denote the extracted For-, Let-columns in this current CF node as CC, all other extracted For/Let columns before this CF node as RC, and all other aggregated columns (including both scalar aggregated columns and aggregated construction columns) before this CF node as AC.

  5.0a. If there are nested CFs in the RETURN clause, call `GTrans` on the nested CFs.

  5.0b. Left outer join S with R if the current nesting level is not the top level. This step is to ensure that empty substructures will be generated, if needed, precisely where they are required by XQuery semantics. The result of the left outer join is assigned to the working table R.

  5.1′. Sort the table with the sorting column list including all columns in RC ∪ CC (except all Let-columns) with each For-variable in the order of their appearance in Q, or sort the table as query requires.

  5.2′. Similar to Step 5.2 but with the projection list enlarged to include all columns in RC ∪ AC along with those required in the construction of this CF.

  5.3′. Same as step 5.3.
- Step 6′. Partition on all columns in RC ∪ AC, construct the result of this CF node in the same way as Step 6, and put the result into a new column in table R.

Note that the translation of basic CF is covered by this general translation because in the basic CF translation, both RC and AC are empty.

To demonstrate how the translation code processes a CF node, consider the nested CF node in the query in Figure 5. Because of the bindings for variables $\$a$, $\$b$, and $\$e$ from the root, we start with a table such as shown in Figure 7a. This is also the table S in the translation `GTrans`. Executing the translation of the nested FOR and LET clauses results in a table such as shown in Figure 7b, from which the translated WHERE clause will remove several rows as indicated. Note that the application of WHERE condition eliminates all the tuples with $(\$a, \$b, \$e) = (a_2, b_2, e_2)$. The nested RETURN clause corresponds to code that aggregates *subresult* as in Figure 7c. The left outer join of S and R generates the table shown in Figure 7d, and finally the code that aggregates *subresults* causes the element construction shown in Figure 7e.

Now we prove that our translation is equivalent to Q in terms of generating the query result that XQuery semantics requires.

Lemma 6.3 For any CF node in the query tree QT, before `GTrans` is invoked on this CF node, the working table R contains: (1)the columns for the variables bound in all ancestors of this current CF node; (2)the columns for the constructed nodes. There is one for each aggU node and CF node sibling located to the left of this current CF node and all its ancestor nodes. After `GTrans` is invoked on this CF node, there is one more column for the constructed result of this current CF node.

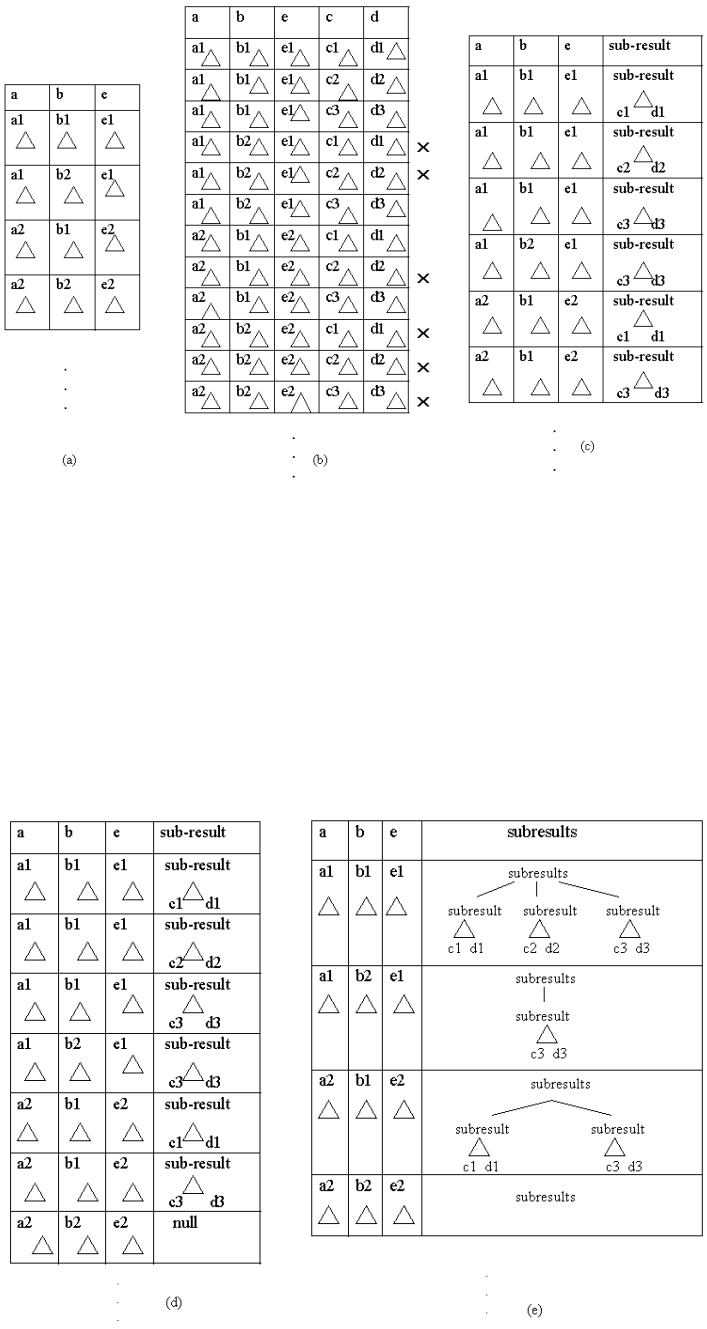


Figure 7: Translating the nested CF node of query in Figure 5

Proof. Note that `GTrans` follows a depth first traversal of the query tree. Let the CF node currently being evaluated be the kth visited CF-node, then this lemma can be easily proved by induction on k.

Lemma 6.4 Before `GTrans` is invoked on any CF node in the query tree QT, the working table R contains at least one row that represents each *valid* binding of the variables bound before this CF node. A *valid* binding is defined as a binding that satisfies the WHERE conditions before this CF node.

Proof. Let the CF node currently being evaluated be the kth visited CF node, then this lemma can be easily proved by induction on k.

Theorem 6.2 For any CF node *cf* in the query tree QT, and each valid binding for variables that are defined before *cf*, the invocation of `GTrans` on *cf* generates one and only one semantically correct result generation tree corresponding to *cf* for that binding.

Proof: We define the height of a CF node in a query tree to be the height of the subtree rooted at this CF node, then we prove this theorem by induction on the height h of the CF node *cf*.

- Base case: when h=1, i.e., the CF node *cf* has no nested CF nodes as children.

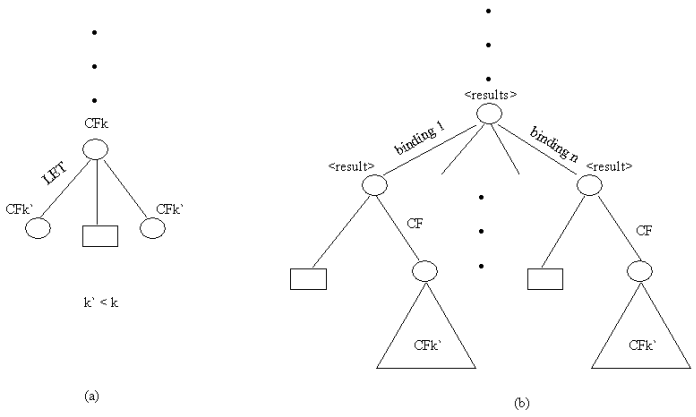
  The similar argument as the proof of Theorem 6.1 proves that the translated algebra expression for this CF node by `GTrans` will generate a semantically correct result generation tree corresponding to this CF node for each valid binding of previous bound variables.

  Note that Lemma 6.4 and the left outer join in Step 5.0b ensure that the `GTrans` will be invoked for all valid bindings, and Lemma 6.3 ensures that all the content required in construction are available in the working table R and the constructed result will be stored in R at least once. Step 5.1′ ensures the sorting is semantically correct by including previous bound variables in the head of the sorting list, and Step 5.2′ ensures that previous bound variables and previous constructed results are still kept in table R. Finally, by partition on all columns in RC ∪ AC, Step 6′ ensures that for each valid previous binding, we only keep one copy of the constructed result of this CF node in table R. Therefore, one and only one result generation tree will be generated for each valid binding of previous bound variables, which is semantically correct.

- Assumption: suppose the claim is correct for any CF node with height h<k.

- Induction: consider a CF node CF_k with height h=k.

  The proof is very similar to that of base case. However, now we could have CFs nested in LET or RETURN clauses (as shown in Figure 6a). In either case, the height of the children of CF_k will be less than k. Therefore, by the induction assumption, after the recursive call of `GTrans` on the nested LET-CF node, the Let variable for the LET-CF has correct set of values, and can be treated the same way as other Let variables. Again by the induction assumption, after the recursive call of `GTrans` on the CF node nested in the RETURN clause, a semantically correct subtree for the nested CF node $CF_{k'}$ ($k' < k$) has been generated. With the similar argument as the proof of Theorem 6.1, we can prove that the semantically correct result generation tree corresponding to this CF node CF_k is generated for each valid binding of previous bound variables, as shown in Figure 6b. Also as we prove in the base case, we can prove that one and only one result generation tree will be generated for each valid binding of previous bound variables. Therefore, we have proved the theorem.



Theorem 6.3 `GTrans` generates the semantically correct result for an XQuery canonical query Q.

Proof: This theorem is a special case of Theorem 6.2 where the CF being considered is the topmost CF, i.e., there is no previous defined variables. Therefore, by Theorem 6.2, we know `GTrans` generates the result generation tree of query Q, and by Lemma 6.1 we know this corresponds to the semantically correct result for Q.

7 An example

In this section, we give an example to illustrate the steps of canonicalization, translation and possible optimization.

Figure 8a gives an XQuery which is not in the canonical form we defined. It has a nested FLWR expression in the FOR clause. By using transformation rule 6 developed in Section 3, we transform the query to the one shown in Figure 8b which is in our canonical form.

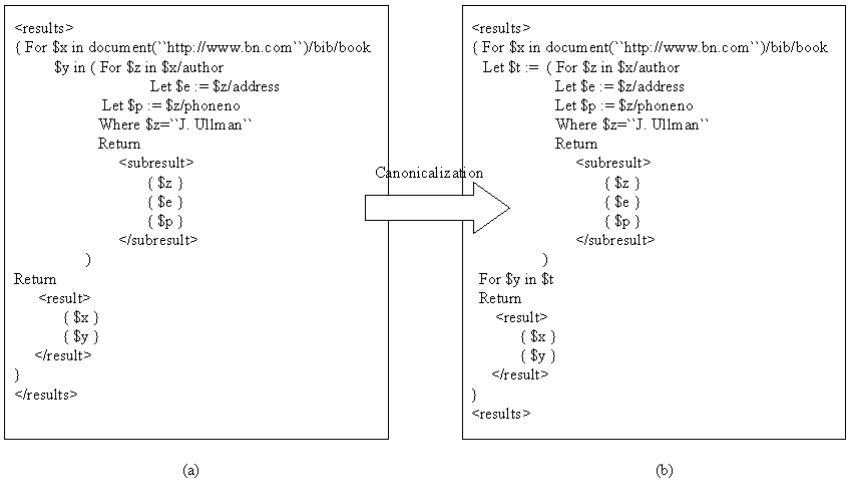


Figure 8: An XQuery and its canonicalized version

The left part of Figure 9 gives the algebraic expression generated by the translation algorithm `GTrans` developed in Section 6. However, this translation is very naive. There are several possible optimization opportunities here. For example, by recognizing unnecessary construction and de-

construction together with the groupby operation, we can eliminate these operations as the circle shows. This example illustrates that seemingly inefficiencies introduced by canonicalization can be removed during algebraic rewriting.

Due to the sibling relationship between 'address' and 'phoneno', the order of the extraction of 'phoneno' and the construction on 'address' can be swapped (as the arrows show). This swap pushes down the extraction of 'phoneno' to occur just after the extraction of 'address'. These two extractions can now be swapped or the sequence of extractions can be combined into one single extraction. Furthermore, the former swap brings together two aggregate constructions (on 'address' and on 'phoneno') which can also be swapped (not shown in the figure). Therefore, this simple transformation could alter the extraction order and the construction order from the original query, and thus provide some optimization opportunities.

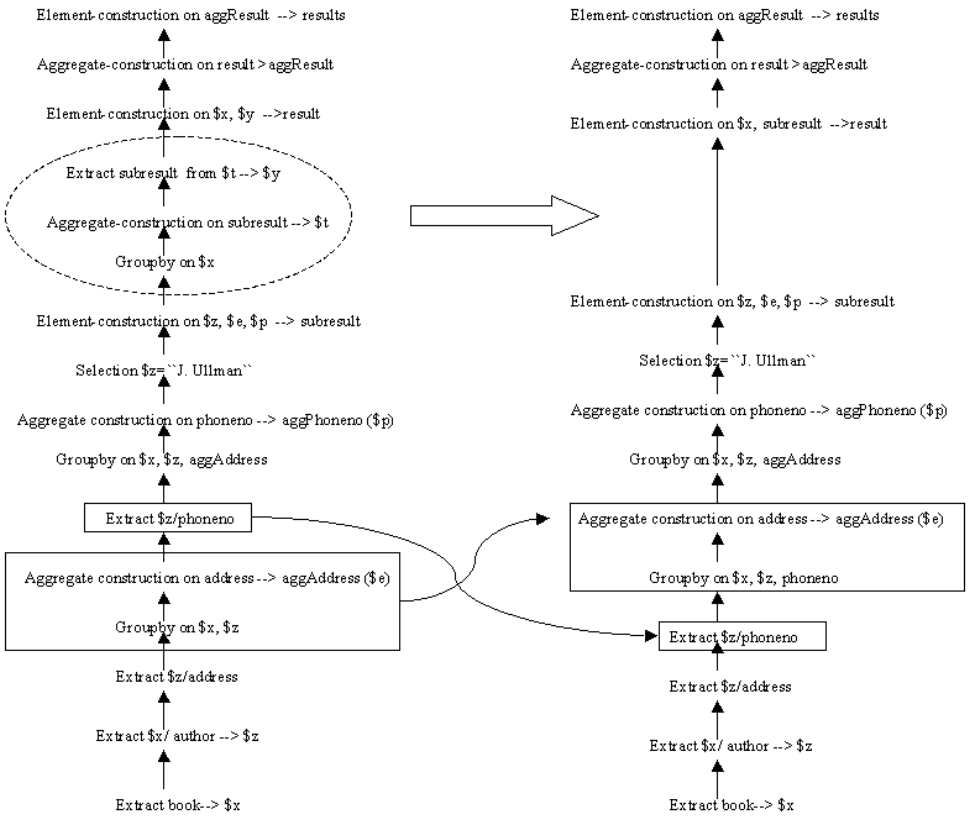


Figure 9: Algebraic expression and optimization for the canonical query in Figure 8b

8 Related work

The W3C working group provides the semantics of XQuery in a working draft, called the XQuery 1.0 Formal Semantics [8]. In that report, the semantics of XQuery is defined with respect to a core syntax of XQuery, and a mapping from the complete syntax of XQuery to the core syntax is given. While their work is essential in understanding XQuery, ours tries to start from a clear and controllable XML query canonical form. It is worth clarifying that our work does not in any sense replace or modify their work. On the contrary, our work is based on the XQuery semantics defined

by W3C.

Manolescu, Florescu and Kossmann give a set of XQuery normalization rules in the context of translating XQuery to SQL [28]. However, their normalization does not have a formal and clear target. Because they have different goals, many of their normalization rules differ from ours. Their normalization rules are essentially about query optimization at the language level, while ours are transformation rules from a subset of XQuery to the canonical form. Their rules can be useful in our future work, which is to develop query rewrite rules at an algebraic level.

In developing the TAX algebra [26], the authors define a canonical XQuery statement which is a FLWR expression of XQuery with some restrictions. However, their canonical statement is more like a notation rather than a designed form with some design goal in mind. This limits its potential usefulness. For example, their canonical statement does not provide a uniform vision of different expressions in XQuery. Therefore, given a FLWR expression returning a path expression, they translate the path expression to some tree pattern accepted by their algebra directly, while we normalize the original query to our canonical form by transforming the path expressions to a nested FLWR expression, then we translate the resulting nested FLWR expression. Clearly this facilitates the subsequent implementation since fewer cases need to be considered.

Many researchers have investigated using relational technology to process XML queries, with a focus on either defining mappings between XML documents and relational tables [18, 23, 32, 14, 21, 12], or answering XML queries using relational technology [28, 19, 31]. However, none of them considers the problem at an algebraic level and gives translation accordingly. This limits the potential for query optimization using standard techniques. There are also industrial efforts on providing XML support [15, 9, 30] and building query engine to support XML queries, such as Microsoft and Software AG's XQuery implementation prototypes, among others [1]. Being prototypes, they only handle a subset of XQuery and do not scale well. Moreover, many problems related to query processing and optimization are not addressed.

In addition to our algebra for XML, several alternative proposals exist. An early proposed algebra [20], which has become the basis of W3C's XQuery Algebra, does not seem to be a good choice for efficient implementation purposes. SAL [10] is another algebra proposed for XML documents modeled as a graph, whose basic unit for manipulation is a node. However, there is no formal translation from an XML query to their algebra.

The YATL algebra [16] is an extension to relational algebra which introduces a *Bind* operator to create tuples of variable bindings and a *Tree* operator to construct the XML result according to a given tree structured construction specification. The functionalities of these two operators are quite similar to that of our extraction and construction operators. The major differences to our approach are that the *Bind* operator does not preserve context when forming variable bindings and the *Tree* operator is quite complex which makes it difficult to optimize at the algebra rewriting level. On the contrary, our construction operators are easy to understand and optimize.

TAX [26] is a tree algebra whose operators manipulate collections of ordered labeled trees instead of relations. However, the authors do not give a formal translation and their translation illustrated by example is different from ours. Whereas the authors are working on the efficient implementation of their algebra, our algebra is more easily integrated into existing RDBMSs.

Fiebig and Moerkotte [22] have studied the XML construction and its optimization in Natix [27] from the algebraic point of view. However, their target language is YATL [16] rather than XQuery, and they only consider the construction part. Moreover, their work is focused on physical algebra instead of logical algebra. In contrast, our work is aiming at a general algebraic framework for XML query processing and optimization which includes both the query part and the construction part. However their work is complementary to ours in the sense that their construction plans can

be an underlying physical implementation of our logical construction operators.

9 Conclusion and future work

In this paper, we define a canonical form for XQuery. This canonical form provides a mechanism to define a clearer semantics since fewer cases need to be defined, and it provides a conceptually uniform vision of path expressions, element constructors and FLWR expressions defined in XQuery. It has been shown that a significant subset of XQuery can be translated to this canonical form.

A nice property of this canonical form is that it separates different aspects of an XML query, i.e., structure, navigation and condition. Hence, it can be easily extended. Moreover, the defined canonical form is not limited to XQuery. Since most XML query languages have query components for navigation, condition checking and structuring, the basic idea behind our canonical form is applicable to them as well.

Based on the defined canonical form, we give a translation algorithm from it to an extended relational algebra. The correctness of the translation has been proven. An advantage of our approach is that it does not require any mapping schema between XML documents and relational tables. Instead, it constructs relational views of XML data dynamically.

We believe that our canonical form provides a better understanding of XQuery and this study provides a solid foundation for implementing an XQuery processor and optimizer. Moreover, taking an algebraic approach makes our research fit well into the traditional query processing and optimization framework, and thus facilitate subsequent implementations.

Possible straightforward extensions of the defined canonical form would be to allow: a) arithmetic expressions to be returned in the RETURN clause; b) '*' in path navigation steps; c) function calls to appear in the canonical form; and d) namespaces [3]. To allow these extensions, we only need to modify corresponding production rules and add new ones to cover additional features. For example, to allow function calls, we only need to modify the relevant production rules where function calls are allowed and include new productions for function calls, with constraints that expressions appearing as function arguments should match function signatures. To allow arithmetic expressions to be returned, the only change required is to modify production rule 7 in Figure 1c, with ArithmeticExpr defined in Figure 1b:

ReturnItem := ArithmeticExpr | CF | '<'tagid'>' ReturnItem* '<'/tagid'>'

From the implementation point of view, support for returning arithmetic expressions and namespaces are straightforward. In fact, support for namespace is an orthogonal issue to the canonical form. Built-in functions can also be easily supported if they are implemented in the underlying query engine (e.g., first() is supported by our text ADT implementation [13]). To support user-defined functions, due to their possible recursive nature, we may need to adapt techniques used in SQL for stored procedures. To realize '*' in a path expression, i.e., to allow the extraction of arbitrary nodes from an XML document, we could use text functions developed in [13].

Using these extensions, the subset of XQuery that can be translated to canonical form will cover almost all W3C's use cases. In the future, we are planning to develop query optimization techniques that start with the query plan generated from the canonical form and produce semantically equivalent plans that will execute more efficiently. We expect that conventional optimization for SQL will be directly applicable to much of this task.

References

[1] XQuery Implementation. In *http://lists.w3.org/Archives/Public/www-ql/2001OctDec/.*

[2] Extensible Markup Language (XML) 1.0. In *http://www.w3.org/XML/*, Feb. 10 1998.

[3] Namespaces in XML. W3C Recommendation. In *http://www.w3.org/TR/REC-xml-names*, Jan. 14 1999.

[4] XML Path Language (XPath) 1.0. In *http://www.w3.org/TR/xpath.html*, Nov. 16 1999.

[5] XML Query Use Cases. In *http://www.w3.org/TR/xmlquery-use-cases*, Dec. 20 2001.

[6] XQuery 1.0: An XML Query Language. In *http://www.w3.org/TR/xquery*, Dec. 20 2001.

[7] XQuery 1.0 and XPath 2.0 Data Model. In *http://www.w3.org/TR/query-datamodel*, April 30 2002.

[8] XQuery 1.0 Formal Semantics. In *http://www.w3.org/TR/query-semantics*, March 26 2002.

[9] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - The XML enabled database management system. In *Proc. of the 17th Int. Conf. on Data Engineering (ICDE)*, pages 561–568, San Diego, CA, March 2000.

[10] C. Beeri and Y. Tzaban. SAL: An algebra for semi-structured data and XML. In *Proc. of the 2nd Int. Workshop on WebDB*, pages 37–42, Philadelphia, June 1999.

[11] G. E. Blake, M. P. Consens, P. Kilpeläinen, P. A. Larson, T. Snider, and F. W. Tompa. Text/Relational Database Management Systems: Harmonizing SQL and SGML. In *Proc. of Applications of Databases(ADB-94)*, pages 267–280, Vadstena, Sweden, 1994.

[12] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of the 19th Int. Conf. on Data Engineering (ICDE)*, San Jose, Feb. 2002.

[13] L. J. Brown, M. P. Consens, I. J. Davis, C. R. Palmer, and F. W. Tompa. A structured text ADT for object-relational databases. *Theory and Practice of Object Systems (TAPOS)*, 4(4):227–244, 1998.

[14] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmungasundaram, D. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *Proc. of the 3rd Int. Workshop on WebDB*, Dallas, Texas, May 2000.

[15] J. Cheng and J. Xu. XML and DB2. In *Proc. of the 17th Int. Conf. on Data Engineering (ICDE)*, pages 569–573, San Diego, CA, March 2000.

[16] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 141–152, Dallas, Texas, May 2000.

[17] I. J. Davis. *Adding Structured Text to SQL/MM Part 2: Full Text, A Change Proposal.* ISO/IEC JTC1/SC21/WG3 CAC N334R3, April 1996.

[18] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of ACM SIGMOD*, pages 431–442, Philadelphia, June 1999.

[19] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 103–114, Santa Barbara, CA, May 2001.

[20] M. Fernandez, J. Simeon, and P. Wadler. An algebra for XML query. In *Proc. of the 12th Conf. on the Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, Delhi, Dec. 2000.

[21] M. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: Trading between relational and XML. In *Proc. of WWW9*, Amsterdam, Netherland, May 2000.

[22] T. Fiebig and G. Moerkotte. Algebraic xml construction and its optimization in natix. *WISE'01 Special Issue of World Wide Web Journal: Internet and Web Information Systems*, 4(3):167–187, 2001.

[23] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, Sept. 1999.

[24] G. H. Gonnet and F. W. Tompa. Mind your grammar: A new approach to modeling text. In *Proc. of the 13th Int. Conf. on VLDB*, pages 339–346, Brighton, England, Sept. 1987.

[25] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. of the 21st Int. Conf. on VLDB*, pages 358–369, Zurich, Switzerland, Sept. 1995.

[26] H. V. Jagadish, Lakes V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. of the 8th Int. Workshop on DBPL*, Rome, Sept. 2001.

[27] C. C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proc. of the 17th Int. Conf. on Data Engineering (ICDE)*, page 198, San Diego, CA, March 2000.

[28] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *Proc. of the 27th Int. Conf. on VLDB*, Rome, Sept. 2001.

[29] G. N. Paulley. *Exploiting functional dependence in query optimization.* PhD thesis, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Canada, April 2000.

[30] M. Rys. Bringing the internet to your database: Using SQL server 2000 and XML to build loosely-coupled systems. In *Proc. of the 17th Int. Conf. on Data Engineering*, Heidelberg, Germany, April 2001.

[31] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proc. of the 27th Int. Conf. on VLDB*, Rome, Sept. 2001.

[32] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the 25th Int. Conf. on VLDB*, pages 302–314, Edinburgh, Scotland, 1999.