International Journal of Database Theory and Application
Vol.7, No.5 (2014), pp.211-226
http://dx.doi.org/10.14257/ijdta.2014.7.5.16

An Accurate Identification of Extended XML Tree Pattern for XQuery Language

Husheng Liao, Xiaoqing Li and Junpeng Chen

Beijing University of Technology, Beijing, China
liaohs@bjut.edu.cn, {li_xiaoqing, chenjp}@emails.bjut.edu.cn

Abstract

In order to utilize high-performance XML tree pattern query (TPQ) for implementing of XQuery language effectively, it is necessary to analysis the query plan and identify tree pattern from it. In this paper, we extend the functional intermediate language FXQL, which is used to implement XQuery language, with an extended XML generalized tree pattern representation (GTP++). Then, we propose an XML tree pattern identification approach, which is composed of a suit of query expression rewriting rules for extracting tree pattern and a GTP++ construction algorithm. Based on this approach, both explicit and implied propositional logic, various structural constraints and predicates can be extracted across nested query blocks in XQuery FLWOR expressions. The tree pattern identified by this approach is more holistic and precisely than previous methods. The approach expands the application of XML tree pattern query technology in the implementation of XQuery language. Experiments show its effectiveness and practicability.

Keywords: *XML tree pattern, tree pattern identification, XQuery*

1. Introduction

XQuery, as W3C standard, is used to query XML data. In related studies of XQuery, one of the core challenges is how to improve query efficiency. Different from the relational data, XML data is a kind of semi-structural data. Therefore, XML data query often contains a variety of structural joins, which can be represented as a tree-shaped query pattern. As main characteristics of semi-structured data query, the tree-shaped query pattern is considered as the core operation of XML data query, called XML tree pattern query (TPQ), or twig query.

TAX in [1] first introduced the concept of pattern tree, which is derived from XML query request. The nodes in the pattern tree are called query node whose label specifies the XML node label needs to be satisfied. The edges in the pattern specify the type of structural constraints between XML nodes, including ancestor-descendant relationship (AD) and parent-child relationship (PC). In such query, the XML nodes conformed to the structural constraints and name tests in tree pattern will be selected. Then, a number of efficient TPQ algorithms are proposed in the past ten years.

Structural joins in for clauses in FLWOR expression which are the core of XQuery can be represented by basic TPQ. Operations described in return clause will be performed on the TPQ results, such as selection, join and node construction including TPQ in the nested FLWOR expression. In order to take advantage of efficient TPQ algorithms, many studies have extended XML tree pattern with different features. For instance, GTP in [2] adds mandatory/optional relationships on the basis of TAX pattern tree. References [3] and [4] extend tree pattern with AND, OR, XOR, NOT logical operations and existence count respectively. However, the current studies on TPQ identification were insufficient to identify all TPQs from nested XQuery program. To solve this issue, we extend the intermediate language FXQL, which is used to descript query plans for XQuery language, with extended XML generalized tree pattern representation, and develop a holistic and precisely XML tree pattern identification approach. The contributions of the paper are as follows:

(1). In order to express tree pattern query as a physical form in query plan, we extend the functional XML query plan description language FXQL with the extended XML generalized tree pattern representation (GTP++), which extends GTP with AND, OR, NOT and various predicates. GTP++ fully support propositional logic of structural constraints, and is able to express the query in nested FLWOR expressions.

(2). Proposes an identification approach for GTP++, which is composed of a suit of query expression rewriting rules for extracting tree pattern and a GTP++ construction algorithm. Based on this approach, both explicit and implied propositional logic, various structural constraints and predicates can be extracted across nested query blocks in XQuery FLWOR expressions. Compared with existing works, our identification approach is able to extract holistic and precisely tree patterns which contain more XQuery query semantics than previous methods

(3). Experiments on two benchmark dataset (DBLP and XMark) demonstrate the effectiveness and practicability of our approach. The performances of the programs with nested FLWOR expressions are improved by utilizing our algorithm.

The remainder of this paper is organized as follows. Section II reviews related work and Section III describes the motivation of the work. The extended GTP++ along with its language description was introduced in Section IV. In Section V, we present the GTP++ identification algorithm in detail. The experiment results are reported in Section VI and Section VII concludes the paper.

2. Related Work

As the core operation of XML data queries, since the notion of TPQ was introduced in [1], many works have been done to extend the descriptive power of TPQ. Apart from GTP[2], APT [3] (Annotated Pattern Tree) improve the matching precision further through edge annotations. The edges in APT can be annotated with one of the four matching options: "+"(one to many matches), "-" (one match only), "*" (zero to many matches), "?" (zero or one match). In addition, many TPQs have been proposed for particular optimization purposes. For example, logic operators AND, OR, XOR and NOT are introduced in [4]. G-QPT in [5] supports ordering via associating pre-order numbers with TPQs. The TPQ proposed in [6] is designed for XML graphs data. Here, XML documents are considered having a graph structure, due to the ID references. Reference [7] makes a study on extended XML tree pattern which include P-C, A-D relationships, negation functions, wildcards and order restriction.

To utilize tree pattern query for effectively realization of XQuery, it is inevitable to analysis the query plan and extract tree pattern from it by query rewriting. There are several studies on TPQ identification. References [8] and [9] concentrate on XPath, the extracted TPs only have one return node and do not support optional relationship. The algorithms in [2] and [3] can identify TPs from XQuery programs, supporting both multiple return nodes and optional relationship. However, these algorithms cannot work across nested FLOWR expressions. Reference [10] proposes an extraction algorithm which can span over nested XQuery blocks but without the ability of identifying the logical constraints in XQuery programs.

3. Motivation

With XML becoming a ubiquitous language for data exchange in various domains, efficiently querying XML data is a critical issue. Since XQuery is a kind of XML data query language as well as functional programming language. This has lead to the design of algebraic frameworks based on tree-shaped patterns akin to the tree-structured data model of XML[11].

On the one hand, almost all of studies on tree pattern queries focus on queries algorithm, the TPQs are graphical representation as shown in Figure 1. Note that a TPQ that cannot be expressed in a physical form is usually considered useless [11], such TPQs are only used for other purpose but querying, such as containment judging, equivalence judging and so on. In order to expressed tree pattern which contains more XQuery query semantics in a physical form, this paper embeds the GTP++ proposed in [12] into the intermediate language FXQL, which is based on query algebra technology for XQuery and compiler technology, and proposes a framework for XQuery system with XML algebra and tree pattern query. GTP++ extends GTP with AND, OR, AND nodes, wildcard and various predicates to describe richer FLWOR query semantics.

On the other hand, the identification ability of tree pattern extraction algorithm can influence the size and number of the tree pattern, which will affect the query performance and the correctness of the final query results. However, the current studies on TPQ identification were insufficient to identify all TPQs from various structural constraints in FLWOR expressions. According to the algorithm in [2], a number of separated GPTs are identified during processing nested FLWOR expression in XQuery. Consider the sample XQuery program in Figure 1 (a), which contains a return clause with nested FLWOR expression. The identification algorithm in [2] will generate the two tree patterns t_1 and t_2 in Figure 1 (b). Since the extracted TPQs are too scattered and duplicate query nodes matching exists between them, it hampers the efficient implementation of the XQuery program. The more powerful identification algorithm in [10] extracts a tree pattern like t_3 in Figure 1 (c). Whereas, since this algorithm is not able to support the identification of tree pattern which with logical operators, the derived TPQ cannot accurately describe the query semantics for some cases. For example, in Q_1, there is nested FLWOR in *return* clause, which applies on the results of the previous query, but the results of this kind of nested queries can be empty. The nested *for-where* clauses use the previous query results, nodes binding to *$b*, and *author* child and *price* child must exist both at once (mandatory relationship) although they are optional relationship

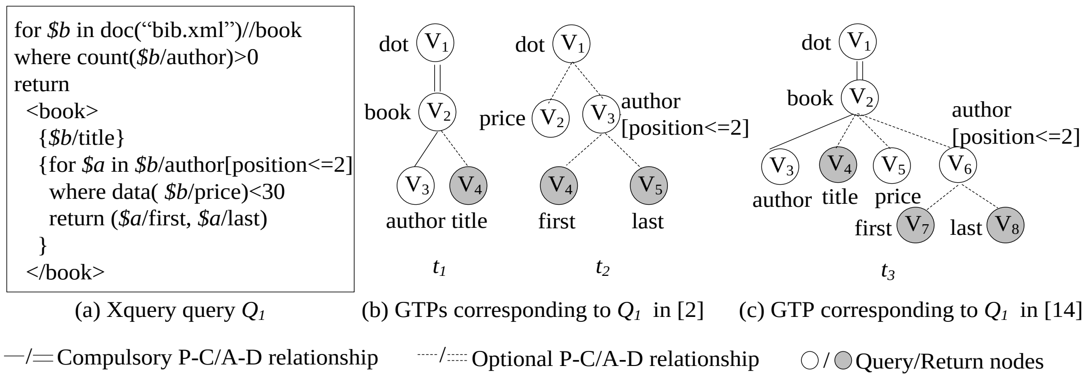

```
for $b in doc("bib.xml")//book
where count($b/author)>0
return
  <book>
    {$b/title}
    {for $a in $b/author[position<=2]
      where data( $b/price)<30
      return ($a/first, $a/last)
    }
  </book>
```

Figure 1. A Sample XQuery Program and Corresponding Pattern Tree

for previous query. Only the *book* elements with both *author* child and *price* child are used in the execution of the *return* clause. The tree pattern t_3 in Figure 1 (c) fails to represent this constraint since the logic operators identification is not supported by algorithm in [10]. There are some other similar situations that up-to-data tree pattern identification method cannot figure out. To solve this kind of issue, this paper develops a tree pattern identification approach, which can identify both displayed and implied propositional logic of structural constraints, to extract holistic and accurate TPQs for XQuery queries.

4. GTP++ With Its Representation

4.1. GTP++ Tree Pattern

GTP++ is an extended generalized tree pattern. It extends GTP with AND, OR, NOT operations, wildcard and various predicates, which is able to express the query request in nested FLWOR expressions. There are four kinds of nodes in GTP++: query node, AND node, OR node and NOT node. Query node can bind to variables, and any query node can be annotated with predicates. There are four kinds of edges in GTP++: mandatory PC relationship, mandatory AD relationship, optional PC relationship and optional AD relationship. Optional relationship indicates that the matching of connected query sub tree is not essential, but the XML nodes matched will also be returned as query results.

Figure 2 shows the GTP++ corresponding to the XQuery program Q_1 in Figure 1 (a). Gray single circle nodes stand for return nodes, such as V_4, V_8 and V_9; double circle nodes stand logic node, such as V_5; solid edges denote the mandatory relationship, and dotted edges denote optional relationship; single edges denote PC relationship, double edges denote AD relationship. For instance, edge <V_2, V_3> is mandatory AD relationship, and <V_2, V_4> is optional PC relationship which means that an *book* element which contain an author element commit to this GTP++ even though it does not contain a *title* element. The nodes in GTP++ can be annotated with predicate constraints, for example, node V_7 is required to satisfy the predicate *position() <= 2*. This shows that GTP++ t_4 is able to precisely represent the query request of Q_1 with the help of AND operation and optional relationship, while the GTP does not have this capability. Similarly, there may be some situations need logic OR and NOT in tree pattern to describe XQuery queries.

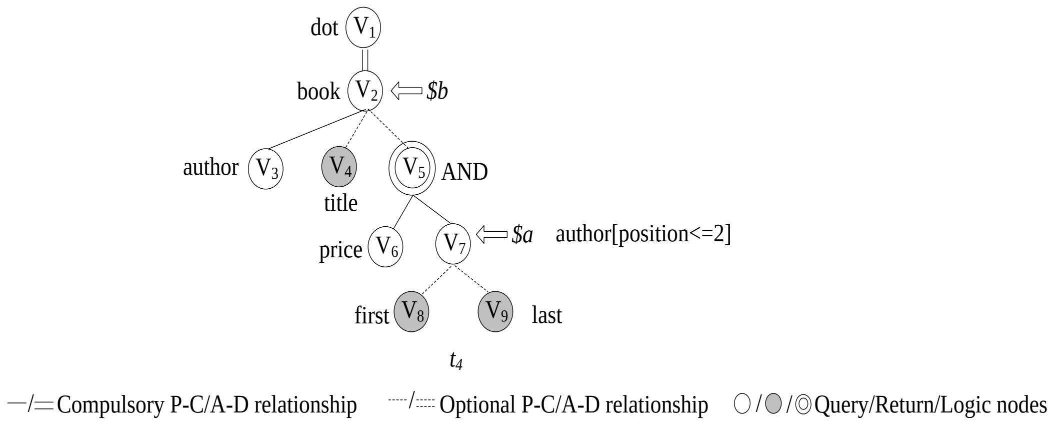


Figure 2. GTP++ Corresponding to Q_1

4.2. Query Plan Description Language FXQL

FXQL is a concise functional intermediate language to describe XML query plan for XQuery which contains query algebraic and TPQ. Its syntax is shown in Table 1. Among these production rules, *Prog* is start symbol and denotes an XQuery program; *Func* stands for a function definition; *Exp* is the core language structure which stands for the FXQL expression; *Arg* stands for actual argument, the definition shows that actual argument can be any expression or anonymous function (*Farg*).

The rules of translating XQuery to FXQL are shown in [13]. Figure 3 shows the FXQL program corresponding to the XQuery program Q_1 in Figure 1 (a). The *foreach*, *filter* in

Table 1. Syntax of FXQL

NO.	Production	Instruction
(1)	Prog ::= Exp (“where” Func+)?	Query body
(2)	Func ::= Idn “(” Idn* “)” “(” Exp “}”	Function definition
(3)	Exp ::= Const | Idn	Constant, variable name
(4)	Exp ::= Axis “(” Exp, Test (“[”Arg“]”)* “)”	Axis operation
(5)	Exp ::= Exp “[” Farg “]”	Filter expression
(6)	Exp ::= “if” Exp “then” Exp (“else” Exp)?	Selection expression
(7)	Exp ::= Idn “(”Arg* “)”	Function Call
(8)	Exp ::= Exp “where” (Id “:=” Exp)+	Expression with local definition
(9)	Arg ::= Exp | Farg	Actual argument
(10)	Farg ::= “fun” “(” Idn* “)” Exp	Actual anonymous function

Table 2. Extended Syntax of FXQL

NO.	Production	Instruction
(11)	Exp ::= Idn “.” Idn	Get tree pattern result according to branch variable
(12)	Exp ::= Exp “with” (Id “:=” TBind)+	With expression (with several tree pattern)
(13)	TBind ::= Exp “{“ TNode* “}”	Root of tree pattern
(14)	TNode ::= TStep “?”? “{“ TNode* “}”	Query node without binding
(15)	TNode ::= Idn “=” TStep “?”? “{“ TNode* “}”	Query node with binding
(16)	TNode ::= and “?”? (“[“ Exp“]”)? “{“ TNode* “}”	Logic AND node
(17)	TNode ::= or “?”? (“[“ Exp “]”)? “{“ TNode* “}”	Logic OR node
(18)	TNode ::= not “?”? (“[“ Exp “]”)? “{“ TNode “}”	Logic NOT node
(19)	TStep ::= (“/”|”//”)? Test (“[“ Farg “]”)*	Element query step (node test + predicates)
(20)	TStep ::= (“/@”|”//@”)? Test (“[“ Farg “]”)*	Attribute query step (node test + predicates)

FXQL
are query algebra operators for projection and selection operation respectively, *child* is axis operation, other functions are built-in functions. In FXQL, all standard functions in XQuery and query algebra operators are implemented. Anonymous functions in the form “*fun(x) e*” can be used as the actual argument of function call.

4.3. The Language Representation of GTP++

GTP++ has the ability to express the query request of nested FLWOR expressions, but the query results still need to be processed based on the processing logic in FLWOR expressions. Therefore, TPQ should be integrated into the query plan as a special operator. Since FXQL is a kind of functional language, each function has only one return value, while GTP++ may has several return nodes. In order to describe GTP++ and the references to different return nodes, this paper extends FXQL with *with* clause and *branch variable reference*. The extension syntax for GTP++ to FXQL is shown in Table 2.

Production (11) denotes that branch variable reference is composed of two variables which are divided by symbol ".". The former is the bind variable in *with* clause for a GTP++, while the latter is name of a tree pattern branch variable. It's used to get its sub query results from return nodes which bind to these branch variable names. Production (12) describes that the structure of *with* clause is composed of several *TBind* structures and each of them binds to a specified variable. Each *TBind* structure stands for a GTP++, which indicates that the TPQ will be applied on computation result of the given expression. Result of the TPQ can be referred with the specified variable. The nodes in GTP++ are represented in *TNode*, including query node, AND node, OR node and NOT node. Production (15) denotes return node, where the binding variable can be used to access the matched XML nodes. Such variables called *tree*

```
flat(foreach(
      filter(child(doc("bib.xml"), book), fun($b) gt(count(child($b, author)), 0)),
      fun($b)
        newElement(
          expanded-Qname(" ", "book"),
          concat(
            child($b, title),
            flat(foreach(
                    filter(child($b, author)[fun(_dot, _ps, _sz)lt(data(child($b, price), 30))],
                    fun($a)concat(child($a, first), child($a, last) ))))))))
```

Figure 3. FXQL Program Without Tree Pattern

pattern branch variable. Among various *TNode* representations, "?" indicates that the structural constraint is optional, while the match option of the other nodes are mandatory. *TStep* in *TNode* shows the representation of node tests and predicates. Recursive definition of *TNode* structure is used to describe the hierarchical relationships of the nodes in GTP++. Any expression or anonymous function can be used to descript predicates.

Figure 4 shows the FXQL program with *with* clause, *i.e.*, tree pattern query, which corresponds to the XQuery program Q_1 in Figure 1 (a). Line 11-15 describes the extracted GTP++. The query result of GTP++ is bound to variable *$0*. Line 12 corresponds to the node V_2 in t_4, which denotes that the *book* node has to satisfy AD relationship with its parent. In line 2, the argument "*$0.$b*" of *filter* function is a branch variable reference for sub query results which bind to branch variable *$b* from the query result *$0*. The final result contains the matched *book* elements and associated sub query result. These results will be delivered to anonymous functions through variable *$b*, so that the sub query results can be obtained through *$b.$1*, that is, *author* elements in the query results. The built-in function *node* is used to take out XML nodes from various query branch variables, for example, *node($b.$1)* can be used to get *author* nodes bind to branch variable *$1* in query result specified by *$b*.

```
flat(foreach(
        filter($0.$b,fun($b)gt(count(node($b.$1)),0)),
        fun($b)
          newElement(
            expanded-Qname(“”,“book”),
            concat(node($b,$2),
                  concat(
                     flat(foreach(
                           filter($b.$3,lt(data(node($b.$4),30)),
                           fun($a)concat(node($a.$5),node($a.$6))))))))
with $0 = doc(“bib.xml”){
           $b=/book{$1=/author,$2=?/title,
                    and?(
                    $3=/author[fun(_dot,_pos,_sz)le(_pos,2)]{$5=?/first,$6=?/last}
                    $4=/price) }}
```

Figure 4. FXQL Program With Tree Pattern

5. Identification Approach of GTP++

The identification approach in this paper is extract GTP++s from the equivalent FXQL program for XQuery query. In order to identify various TPQs which implicitly exist in query plan, we need analysis the program structure of FXQL to find out the various structural joins and transform them to GTP++ represented with *with* clause.

The structural joins within XQuery program exist in XPath and FLWOR expressions, which are represented as combined axis operation expressions. In expressions such as FLWOR, there may are logical computation relationships among various XPath expressions, which will be represented as AND operators between query operators, like *foreach*, *filter*, *cross* and so on. Thus, basic TPQs can be obtained from the combination of query operators and XPath expressions, which contain AD relationship and PC relationship only. Besides, the scope of TPQ is related to the program module such as function body, *if* branch and nested FLWOR body. Between TPQs in different blocks, there may be data dependency. With the help of optional relationship in GTP++, it is capable of merging TPs with such data dependence to a single GTP++, so that the number of TPQ is decreased.

The GTP++ identification approach is composed of two steps: First, extract the basic TPQs from FXQL program and construct the FXQL program with basic TPQs in *with* clause; After that, rewriting FXQL program with GTP++ patterns by using optional relationship mechanism to merge the TPQs in the FXQL program.

5.1. Extraction of Basic Tree Pattern

The basic tree pattern extraction is the process to analyze and rewrite expressions, and each expression may be transformed into corresponding one with *with* clause. The query results of return nodes must be accessed via the branch variables of TPQs. During the extracting, all AD and PC axis operations will be replaced with TPQs. Thus, for an axis operation, if the source expression is a branch variable of a TPQ, this current axis operation should be extended into it; otherwise a new TPQ needs to be constructed. In the same way, the variables defined by axis expression in *where* clause also may be represented as branch variables, and the binding should be delivered to the position where refers it, so that current TPQ can contains as much axis operations as possible. With respect to this consideration, there are two possible results of each expression transforming: (1) rewritten expression; (2) a TPQ branch variable. On the other hand, the processing of any expression needs to take their context information into account. The context information contains free variables and variable *_dot* which represent the current item. During the rewriting of expression, the free variables may be replaced with TPQ branch variables, and the current item binding to *_dot* variable is often the source of axis expression, and is also treated as TPQ branch variable.

Based on the above approach, the rewriting rule *ExtraExp* of FXQL expression for extraction are declared as follows:

ExtraExp: Exp → ExEnv* → Exp ×TBind, where

ExEnv: Idn → Var ×TBind

Exp and *TBind* in this rule stand for FXQL expressions and the TPQ representation in *with* clause respectively. *ExEnv* is the context environment which is used to store binding relationships between variables and tree pattern branch. The second parameter in rule *ExtraExp* is an environment list. Whenever processing a function call, branches of an *if-then-else* expression or a *let* clause, a new environment should be created as the head of the list. For instance, *(e,b)= ExtraExp[exp]w* stands for rewriting *exp* and extracting tree pattern as *with* clause in context environment list *w*. If the extracting result *b* is nil, *e* is rewritten expression, else the result *e* and *b* stand for the variable and its tree pattern branch in environment respectively.

During the identifying of TPQ, if a variable is used to store the result of AD or PC axis operation, it will be bound with the tree pattern branch variable representing the result in the context environment *ExEnv*. Local variables which do not bind to a tree pattern branch variable have nothing to do with the tree pattern extraction, so they will not be stored in *ExEnv*. The TPQs bound in *ExEnv* are called *external tree patterns*. Whenever handling AD and PC axis operations, query nodes are need to be constructed to extend tree pattern. At this moment, current TPQ which bind to the variable *_dot* or a tree pattern branch variable which bind to a variable may be extended.

Another core issue of tree pattern extraction is to identify all the structural joins within conjunctive relationships. In FXQL, such relationships occur among the continuous axis operations as well as the axis operations in predicates. For example, the first argument of the *foreach* operator is an anonymous functional, which is applied to each element of the data list given by the second argument. At this moment, it is likely to extract structural constraints from the query operators in the second actual argument and this anonymous functional body, which will be returned as tree pattern branch variables. Therefore, the scope of TPQ extraction is determined by the conjunctive relationships. However, in FXQL expressions, each branch of conditional expression and the structural constraints contained in each actual argument of the functions except the query operators are independent; they do not belong to a same TPQ. In order to divide the scope of tree pattern extraction clearly, the expression rewriting rules make use of a environment list. Whenever processing a function call, branches of an *if-then-else* expression or a *let* clause, a new environment should be created as the head of the list.

Due to space limitations, this paper only introduces the main expression rewriting rules. As listed in Table 3 (1), the constant *const* doesn't need to be extracted, it will be simply rewritten as the return pair<*const,nil*> . According to rule (2), the variable *idn* without binding in the context environment have no need to be extracted too; if the variable comes from the head of environment list, then this variable should be replaced with a TPQ branch variable; otherwise, it must come from the tail of the list, then should call *genTBind* to construct a new TPQ. Rule (3) describes the process of *if-then-else* expression, a new empty environment should be created for the extraction of each branches. Rule (4) deals with most function calls, including the built-in functions except comparison operation, user-defined function and query operator function without functional arguments. Rule (5) processes the common comparison operations in predicates. In order to facilitate description, some auxiliary functions and data structure are used in expression extraction rules. Function *genExp* represent generating FXQL expression based on given template in which its arguments are specified with symbol '<' and '>'; *genTBind* generates tree pattern with representation of *with* clause; *newVar* generate a new variable name.

The expression extraction rule (6) in Table 4 are the core part of tree pattern extraction, namely the rewriting rule of the axis operation expression. Auxiliary function *newTNode* generate query node in tree pattern; *addTNode* add sub query node along with its binding variable to the given query node; *addPred* add predicates to the given query node; *genPred* construct a representation of predicates in *with* clause. *var* is used to get tree pattern branch variable; *node* gets XML nodes in root of the query results instance. For forward axis, which

Table 3. Rewriting Rules of Simple Expressions

NO.	Expression Rewriting Rule
(1)	ExtraExp[const] w = <const, nil>
(2)	ExtraExp[idn]w = if w(idn) = Ø if idn exists in ancestor environment of w then <v, genTBind[v=<idn>.<v'>] where v = newVar() else <idn, nil> else w(idn)
(3)	ExtraExp[**if** exp$_1$ **then** exp$_2$ **else** exp$_3$]w = if $b_i \neq$ nil then add b_i to w for i=2,3 <genExp[**if** <e_1> **then** <e_2> **else** <e_3>], nil> where <a_1, b_1> = ExtraExp[exp$_1$] w w = u ++ { Ø } <a_i, b_i> = ExtraExp[exp$_i$] w for i=2,3 e_1 = if $b_1 \neq$nil then genExp[getNode(<var(b_1)>.<a_1>)] else a_1 e_i = if $b_i \neq$nil then a_i else genExp[<a_i> with <b_i>] for i=2,3
(4)	ExtraExp[idn(arg$_1$, … , arg$_n$)]w = if $b_i \neq$ nil then add bi to w for i = 1,…,n <genExp[idn(a_1, … , a_n)], nil> where w = υ ++ {Ø} <a_i, b_i> = ExtraExp[arg$_i$] w for i=1,…,n
(5)	ExtraExp[cmp(arg$_1$, arg$_2$)] w= <genExp[cmp(<e_1>, <e_2>)], nil> where <a_1, b_1> = ExtraExp[arg$_1$] w <a_2, b_2> = ExtraExp[arg$_2$] w e_1 = if $b_1 \neq$nil then genExp[getNode(<var(b_1)>.<a_1>)] else a_1 e_2 = if $b_2 \neq$nil then genExp[getNode(<var(b_2)>.<a_2>)] else a_2

is used for the axis operations supported by basic tree pattern, such as PC, AD, property and so on, the source expression *exp* of axis operation is processed first. If the result is a tree pattern branch variable (b'≠nil), then extend this tree pattern with new query nodes using current axis operation; otherwise, a new *TBind* instance will be constructed and extended with a new query node *n* using current axis operation. Subsequently, all predicates are processed to extend this query node. Other predicates p_i except exist predicate are added to this query node. The new constructed tree pattern will be rewritten with *with* clause, otherwise, return a new constructed tree pattern branch variable <v, b#>.

In XQuery, most tree patterns are from FLWOR expression, especially the case which exists multiple *for* clause and *let* clause. In FXQL, such query often represents as the combination of several *foreach*, *cross*, *filter* query operators and *where* expression. The rule (7) describe the rewriting rule for extracting tree pattern from projection operation *foreach* function which are translated by *for* clause in FLOWR expression. The rewriting rule for *where* expression, which translated from *let* clause, is shown in rule (8). For all the variables v_i defined in *where* expression, corresponding expression a_i will be constructed after rewriting the definition expression exp_i of each variable. However, this *where* expression may occur in the combination of computation among query operators, as well as the condition branches or

Table 4. Expression Rewriting Rules of Other Expressions

NO.	Expression Extraction Rule

(6)	ExtraExp[axis(exp, test[fun(_dot,_pos,_sz)exp_1]… [fun(_dot,_pos,_sz)exp_n])] w =
	  if axis∈{child, desendent-or-self, attribute} then
	   if b’≠ nil then <v, $b^\#$> else <genExp[<v> with <b#>], nil>
	   where <e_0, b’> = ExtraExp[exp] <x, b> w
	      b”= if b’≠nil then b’ else genTBind[<newVar( )>=<e_0>]
	      v = newVar()
	      <e_i, b_i> = ExtraExp[exp_i] <v, b”> w for i=1,…,n
	      n = newTNode(v, axis, test)
	      p_i = if b_i≠nil then Ø else genPred[e_i] for i=1,…,n
	      n’ = addPred(n, p_1…p_n)
	      $b^\#$ = addTNode(b”, v, n’)
	  else <a,nil >
	   where <e_0, b’>= ExtraExp[exp]w
	      e’= if b’≠nil then genExp[<var(b).<e_0>] else e_0
	      v = newVar( )
	      b_0= newTBind(v, e’)
	      <e_i, b_i> =ExtraExp[expi]υfor i=1,…,n
	      pi=if bi≠nilthen Ø else genPred[fun(_dot,_pos,_sz)ei]
	      a = genExp[axis(<v>, <test><p_1>…<p_n>)]

	ExtraExp[foreach(exp_1,fun(v) exp_0)]w =
	  if b_1≠nil then
	   if b_0≠nil then <e_0,b_0> else <genExp[foreach(<var(b_1)>.<e_1>, fun(v) e_0)], nil>
	   where u = w ++ <v, <e_1,b_1>>, <e_0, b_0> = ExtraExp[exp_0] <x, b> u
	  else
	   if b_0≠nil then <e_0, b’> else <genExp[foreach(<e_1>, fun(v) e_0) with b’], nil>
	   where v’ = newVar( )
	      b’ = genTBind[<v’>=<e_1>]
	      u = w ++ <v, <v’,b’>>
	      <e_0, b_0> = ExtraExp[exp_0] <x, b> u
	  where <e_1, b_1> = ExtraExp[exp_1] w

	ExtraExp[exp_0 where v_1 = exp_1, …, v_n = exp_n]w =
	  if x=’#’ then
	   if b_0≠nil then <genExp[<var(b_0).<e_0> with <b_0>], nil> else <genExp[<e_0> where <defs>], nil>
	   where vs = getVar(e_0, v_1…v_n), defs = genDefs(vs, v_1…v_n, a_1..a_n)
	  else
	   if b_0≠nil then
	     if inOutside(b_0, b, υ) then <e_0, b_0> else <genExp[<var(b_0)>,<e_0> with <b_0>], nil>
	   else <genExp[<e_0> where <defs>], nil>
	  where <e_i, b_i> = ExtraExp[exp_i]<x,b> υ for i=1,…,n
	     a_i = if b_i=nil then e_i
	       else if inOutside(b_i, b, υ) then genExp[<var(b_i)>.<e_i>] else genExp[<var(b_i)>.<e_i> with b_i]
	     bd_i = if b_i≠nil then <v_i,<a_i,b_i>> else Ø for i=1,…,n
	     u = w ++ bd_1 ++ … ++ bd_n
	     <e_0, b_0> = ExtraExp[exp_0] <x, b> u

actual arguments of functions. For the former case, axis operations in *where* expression should be extended to tree patterns as much as possible. In the latter case, there may be internal tree patterns after rewriting of *where* expression. If *x='#'*, it can be sure that external tree patterns will not involve the internal axis operations, a new *where* clause should be constructed (using *genDefs*). Internal tree patterns will be occurred in the definition of the new expression. Otherwise, for the *where* expression used in the combination of query operators, the tree pattern within the main body expression after rewriting should be determined is an external tree pattern or internal tree pattern (using *inOutside*). It returns directly if it is a branch variable of an external tree pattern, so as to guarantee that the axis operation in *let* clause will be merged into tree patterns.

The previous rules have shown the approach for extracting tree pattern which cover FXQL core expressions. The similar rules can be applied to process other FXQL expressions.

5.2. Construction of GTP++

After the rewriting of basic tree pattern extraction, all extracted basic tree patterns are store in the environment list. Let us support that *t* and *t'* are two tree patterns extracted by previous expression rewriting rules. If the root node of *t* binds to a tree pattern branch variable within *t'*, we say that *t* is depended on *t'*. GTP++ can be constructed by connecting the directly related tree patterns using optional relationship, according to the dependent relationships between different tree patterns.

The algorithm *TPQMerge* described in Figure 5 merges the basic tree patterns in the environment list into GTP++. Line1-5 merge the tree patterns within each layer of environment list, for the tree patterns in the same layer, if the root of these tree patterns refer to a same variable, then merge them via AND node. The procedure *logical_merge* is used to merge the TPs which belong to a same layer of environment; Line 7-9 merge the tree patterns which belong to different layers, if these are dependency among these tree patterns, then connect these related tree patterns with optional relationship by procedure *merge*.

Figure 6 illustrates the example of merging several basic tree patterns to the single GTP++ which is corresponds to Q_1. Fig. 6 (a) shows the initial state, which contains an environment list, L_0, L_1, L_2, L_3 and L_4. Firstly, it process the tree patterns in the same layers of environment.

```
Algorithm TPQMerge
Input: tHe context environment list u
Output:  the context environment list  after being merged

1.  logical_merge(u); // merge current environment by AND  node
2.  p = u;
3.  while p is not the top level environment do
4.      p = next( u ) ; // get the parent environment of u
5.      logical_merge(p);
6.  p = u;
7.  while p is not the top level environment do
8.      p = merge( p ); // merge the TPQs in p to its parent environment by optional relationship
9.  return p;

Procedure logical_merge(u)
10. divide the TPQs in u into groups $g_1 \dots g_n$ based on the root of TPQ, so that the root of every TPQ in one group bind to a same variable;
11. for each g in ($g_i \dots g_n$)
12.     merge the TPQs in g with logical and node;
13. return u;

Procedure merge(u)
14. p = next( u )
15. for each tree pattern $t_u$ in u do
16.     src = root($t_u$);
17.     if src is variable and p(src) != null
18.         extend p(src) with $t_u$ via optional relationship;
19. return p;
```

Figure 5. Example of GTP++ Construction

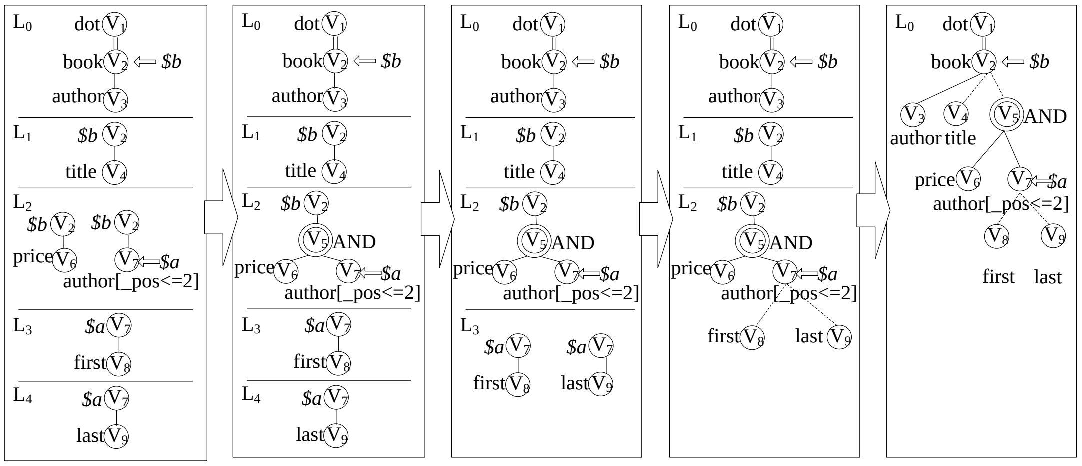


Figure 6. The Example of Merging TPQs to GTP++

The tree patterns in L_2 whose roots refer to a same variable *$b* are merged to a single GTP++ by AND node, the result is shown in (b). Secondly, it will merge the tree patterns among different layers of environment. Because tree pattern in L_4 does not depend on any tree patterns in L_3, it is simply merged into L_3, though the roots of tree patterns both refer to *$a*. Figure 6 (c) shows the result after merging L_4 to L_3. Then, the roots of tree patterns which refers to *$a* in L_3 are depended on tree pattern in L_2, so merge them into tree pattern in L_2 by optional relationships, as shown in (d). Now tree patterns in L_2 and L_1 will be merged similarly. Figure 6 (e) shows the final result. Eventually, the six initial basic tree patterns are merged into a single GTP++ via logical node and optional relationship mechanism.

6. Experiments

We have implemented an XQuery engine with FXQL, in which an extend tree pattern matching algorithm is used to evaluate the GTP++ in an FXQL interpreter. This section presents experimental study using DBLP (size of 127MB) and XMark (sizeof 111MB) as a benchmark, which are carried out on a Windows 7 PC with Intel Core i5-2300 2.67Ghz CPU, 2G RAM and the JRE of version 1.6. The XQuery programs used in experiments are shown in Table 5. We compared our approach with pattern extract approaches [2] and [10] and they are denoted as *myExt*, *Ext2* and *Ext14* in the experiment respectively. Each sample XQuery programs is translated into FXQL and TPQs are extracted by three approaches. Every FXQL program with different TPQs is running on the same engine which is developed by our group.

The experiments are divided into three groups: (1) Compare the execution time of each program with the TPQs which are extracted by three approaches. (2) Compare the size of the query results of the TPQs which are extracted from each XQuery program in Appendix except DQ1 and XM1 by *myExt* and *Ext14*. (3) Compare the execution time of each program with the TPQs which are extracted by three approaches in the case of the same query and different amount of data.

Table 5. XQuery Programs Used in Experiments

NO.	XQuery Program

DQ1	for $b in doc(“dblp.xml”)//book return <book>{$b//title,
	for $a in $b//author return <first>{$a/first}</first>}</book>

DQ2	for $b in doc(“dblp.xml”)//book return <res>{$b//title,
	for $e in $b//editor where $b//url return $e}</res>

DQ3	for $b in doc(“dblp.xml”)//article where $b//ee or $b//author
	return <res>{$b//title, for $e in $b//author return $e}</res>

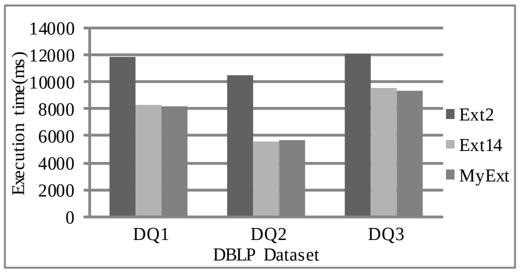
DQ4	for $b in doc(“dblp.xml”)//book where not($b//editor) return $b

XQ1	for $x in doc("XMark.xml")//item[//mail] return <res> { $x/name/text(),
	for $y in $x//listitem return <key> { $y//keyword } </key> }</res>

XQ2	for $x in doc("XMark.xml")//open_auction
	return <res> { $x//current, for $y in $x//listitem where $x//privacy return $y//keyword }</res>

XQ3	for $x in doc("XMark.xml")//open_auction
	where $x//privacy or $x//reserve
	return <res>{$x//current,for $y in $x//listitem return $y//keyword }</res>

The results of the first group are shown in Figure 7. It illustrates that the running speed of programs with TPQs extracted by *Ext2* is obviously slower than the other two, since smaller TPQs are extracted by [2] so that it has to perform more evaluation of TPQs. Since the number of TPQs extracted by the other two approaches is same, the execution times for them are almost same.



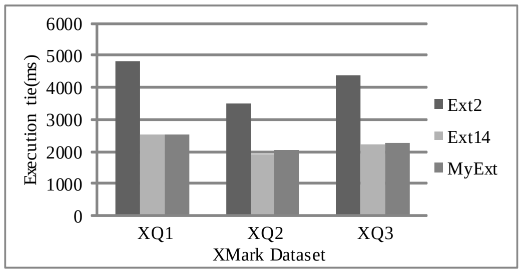


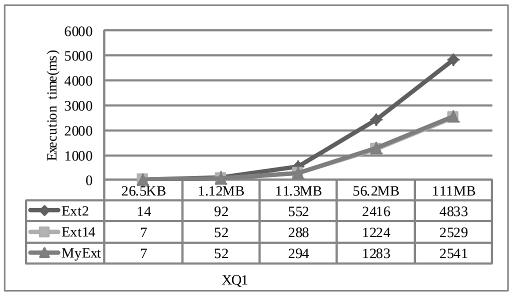
Figure 7. The Execution Time of TPQs

On the other hand, the result size of the TPQs extracted by our approach is smaller than [10], as shown in Table 6. The reason is that our GTP++ contains more logical constraints like AND, OR and NOT which may filter more useless nodes. Besides, DQ4 is a special case using of logical operation NOT, which is only supported by our approach.

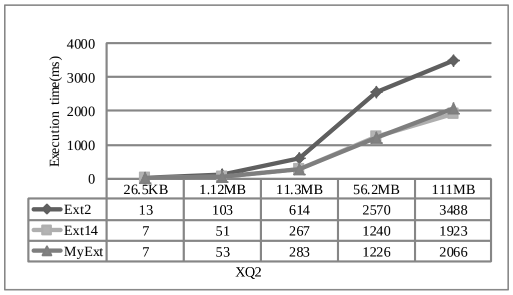
Table 6. Size of Sesults

Query	Dataset	Ext14	MyExt
DQ2	DBLP	1113	973
DQ3	DBLP	446683	444565
DQ4	DBLP	N/A	780
XM2	XMark	75519	34181
XM3	XMark	34106	28609

The third group makes an account on the execution time of XQ1 and XQ2 in the case of different size of XMark benchmark, Figure 8 shows the trend. The trend indicates that the distinction between *Ext2* and *myExt* becomes more obvious with the increase of data sets, while *Ext14* and *myExt* are very close, since the amount of TP extracted by these two approach are same, the result is consistent with group 2.



	26.5KB	1.12MB	11.3MB	56.2MB	111MB
Ext2	14	92	552	2416	4833
Ext14	7	52	288	1224	2529
MyExt	7	52	294	1283	2541



	26.5KB	1.12MB	11.3MB	56.2MB	111MB
Ext2	13	103	614	2570	3488
Ext14	7	51	267	1240	1923
MyExt	7	53	283	1226	2066

Figure 8. The Trend of Execution Time in Different Size of Dataset

7. Conclusion

In this paper, we extend the functional intermediate language FXQL, which is used to implement XQuery language, with an extended XML generalized tree pattern representation (GTP++). Then, we propose an XML tree pattern identification approach, which is composed of a suit of query expression rewriting rules for extracting tree pattern and a GTP++ construction algorithm. It can identify larger tree patterns than previous works, even in the case of patterns across nested query blocks. Based on this approach, both displayed and implied propositional logic, various structural constraints and predicates can be extracted across nested query blocks in XQuery FLWOR expressions. The tree pattern identified by this approach is more holistic and precisely than previous methods. The approach expands the application of XML tree pattern query technology in the implementation of XQuery language. Experiments show its effectiveness and practicability.

Acknowledgements

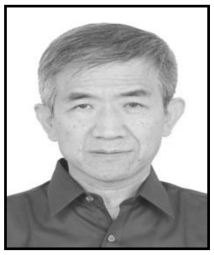
This work was both supported in part by the Beijing Nature Science Foundation under Grant 4122011and the National Science Foundation for Young Scientists of China under Grant 61202074.

References

[1] G. Ghelli and G. Grahne, "Editors, TAX: A Tree Algebra for XML", Proceedings of the 8h International Workshop Database Programming Languages, (**2001**) September 8-10, Frascati, Italy.
[2] Z. Chen, H. V. Jagadish, L. V. S. Laksh-manan and S. Paparizos, "From tree patterns to generalized tree patterns: on efficient evaluation of XQuery", Proceedings of the 29th international conference on Very large data bases, **(2003)** September 9-12, Berlin, Germany.
[3] S. Paparizos, Y. Wu, L. V. S. Lakshmanan and H. V. Jagadish, "Tree logical classes for efficient evaluation of XQuery", Proceedings of the 2004 ACM SIGMOD international conference on Management of data, **(2004)** June 13-18, Paris, France.
[4] S. K. Izadi, T. Harder and M. S. Haghjoo, "Data Knowl", Eng. vol. 68, **(2009)**, pp. 126.
[5] Y. Chen, "Adv. Knowl. Discovery Data Min.", vol. 3056, **(2004)**, pp. 559.
[6] Q. Zeng, X. Jiang and Z. Hai, "Adding Logical Operators to Tree Pattern Queries on Graph Structured Data", Proceedings of the 38th international conference on Very large data bases, **(2012)** August 27-31, Istanbul, Turkey.
[7] J. H. Lu, T. W. Ling, Z. F. Bao and C. Wang, "IEEE Trans. Knowl.", Data Eng., vol. 23, **(2011)**, pp. 402.

[8] P. Michiels, G. A. Mihaila and J. Simeon, "Put a Tree Pattern in Your Algebra", Proceedings of the 23rd International Conference on Data Engineering, **(2007)** April 15-20, Istanbul, Turkey.
[9] K. Beyer, F. Ozcan, S. Saiprasad and B. V. Linden, "DB2/XML: designing for evolution", Proceedings of the 2005 ACM SIGMOD international conference on Management of data, **(2005)** June 13-16, Baltimore, Maryland, USA.
[10] H. L. Larsen, G. Pasi, D. O. Arroyo, T. Andreasen and H. Christiansen, "Algebra-based identification of tree patterns in XQuery", Proceedings of the 7th International Conference on Flexible Query Answering Systems", **(2006)** June 7-10, Milan, Italy.
[11] M. Hachicha and J. Darmont, "IEEE Trans. Knowl. Data Eng.", vol. 25, **(2013)**, pp. 29.
[12] H. S. Liao and X. Q. Li, "Front. Comput. Sci. and Technol", vol. 7, **(2013)**, pp. 431, (in Chinese).
[13] X. B. Zhang and H. S. Liao, "Front. Comput. Sci. and Technol.", vol. 4, **(2010)**, pp. 996, (in Chinese).

Authors



Husheng Liao, was born in Changchun in 1954. He is a professor and doctoral supervisor at Beijing University of Technology in P.R.China. His research interests include software automation methods and data integration technology, etc.

Xiaoqing Li, was born in Tangshan in 1983. She is a Ph.D. candidate at Beijing University of Technology in P.R.China. Her research interest is XML database technology.

Junpeng Chen, was born in Wenzhou in 1988. He is a M.S. candidate at Beijing Universityof Technology in P.R.China. his research interest is XML database technology.