OreintXA:一种有效的 XQuery 查询代数

罗道锋 [1]　蒋瑜 [1]　孟小峰 [1]

[1](中国人民大学信息学院　北京　100872)
(luodaofeng@yahoo.com.cn)

摘　要　现有的 XQuery 处理策略有两种方法，第一种是基于核心语法一次一结点的方法，另一种是基于查询代数一次一集合的方法。本文认为，单独使用这两种方法都不能很好地处理 XQuery 查询：一方面，基于核心语法树的的方法效率低，而且很难优化；而基于代数的方法又不能很好地满足 XQuery 的过程化查询语言的特性。本文试图结合这两种处理策略，使能有效地表达 XQuery 查询的同时，又能获得很高的处理效率。本文讨论了 XQuery 代数设计的思路，现状和问题，在此基础上提出了 OrientXA XQuery 查询代数设计。OrientXA 代数系统使用一次一集合的方法表达查询，但是，在它的结果构造符中，体现了 XQuery 查询语言的过程化编程语言的特点，因此具有很强的表达能力，能够表达 W3C Use Case 和 XMark 数据集的所有查询。

关键词　XQuery 查询;查询代数，核心语法树
中图法分类号　TP311.13

OrientXA: An Effective XQuery Algebra

LUO Dao-Feng[1], JIANG Yu[1], and MENG Xiao-Feng[1]

[1](*Information School, Renmin University of China, Beijing, 100872*)

Abstract　XQuery is the recommended standard for XML Query. XQuery processing strategies can be classified into two categories: core syntax based strategy(Node-Oriented) and algebra based strategy(Set-Oriented). Neither of them can handle XQuery well. This paper focus on the latter, and borrow some idea from the former. After summarizing the current stage and unsolved problem of current works, this papger proposed an effective XQuery algebra system---OrientXA. OrientXA has powerful expressiveness, and is able to express all the queries in W3C use cases and XMark benchmark queries.

Key words　XQuery, Algebra, Core Syntax

1　引　　言

随着 XML 数据越来越被广泛地使用，对 XML 数据的查询提出了越来越高的要求。前期的研究主要集中在 XPath 的查询处理上。XPath 相对比较简单，表达能力有限，比如，不能表示连接操作等。作为 W3C 的推荐标准，XQuery 的表达能力比 XPath 强大很多，同时，XQuery 也比 XPath 复杂很多。

XQuery 兼有结构化查询语言和过程化编程语言的特点。一方面，XQuery 的 FLWR 子句一定程度上类似于 SQL 的 Select-From-Where 子句，是 XQuery 的最重要的表达式；另一方面，XQuery 支持表达式的任意嵌套，支持诸如条件表达式(IF-THEN-ELSE)，循环表达式（FOR）和返回值(RETURN)等，还有变量和谓词的作用域问题，这些，都是一门编程语言的重要特征。

因为 XQuery 兼有这两种特点，因此，XQuery 的处理方法分为两大类：基于核心语法的处理和基于 XML 代数的处理。

基于核心语法的处理（一次一结点）:

现在流行的 XQuery 查询引擎，比如 Galax[5]，IPSI[7]等，都是使用这种方法。这种方法把 XQuery 看做一门编程语言来处理。这种方法通常先把 XQuery 转换成核心语法（Core Syntax），然后以 XML 树的根结点作为输入，依次执行该核心语法树，最后得到结果。

比如对于以下查询：

Q1:

```
FOR $b in (document(“bib.xml”)/bib/book
WHERE $b/price<50
RETURN
<cheap-books>{$b/title}</cheap-books>
```

转换成的核心语法是（由于核心语法表示很复杂，这里用简化的伪码来表示）：

```
1) for $bib in root/bib do
2)     for $book in $bib/book do
3)      for $price in $book/price do
4)       if $price<50 then
```

```
    5)          construct a <cheap-books>
    6)          for $title in $book/title do
    7)             append the content of $title as child of the
element <cheap-books>
    8)          endfor
    9)          break;
    10)         else ()
    11)     endfor
    12)   endfor
    13)  endfor
```

可以看出，这种方法的特点是一次一结点(Node-Oriented)。这种方法比较简单，它的好处是：它天生符合 XQuery 的程序化的特点，能正确地处理各种 XQuery 语句的嵌套，能自动地保证结点的正确的顺序。它最大的劣势在于很难进行优化。关于一次一结点和一次一集合的方式，人们的共识是一次一集合的方式的效率要优于一次一结点，而且利于优化。因此，XQuery 的另一条处理路线是一次一集合的方式。

基于 XML 代数的处理（一次一集合）：

这种方法提出了一套类似于关系代数的代数体系。每一个代数操作符的输入都是一个或者多个 XML 树集合，输出也是一个 XML 树集合。通常，为了生成 XML 树集合，从 XQuery 语句中生成一个或多个模式树（Pattern Tree），模式树表示了该查询感兴趣的（包括谓词结点和目标结点）一组有祖先后代关系的变量绑定，然后，用模式树从输入的 XML 树中抽取实例树。比如，对于查询 1，会先抽取出模式树 bib(book(price, title))，然后用这个模式树对 XML 树进行匹配，得到模式树的实例树集。然后在这个实例树集上进行谓词判断(price<5)，最后输出希望的结果。

可以看出，这种方式是一次一集合的处理方式。它最大的优点是能够对逻辑操作树进行优化，从而获得较高的效率。然而，一次一集合的方式跟 XQuery 的编程语言特点和 XML 数据本身的特点存在许多不相容的地方，从而带来一些新的问题。比如如何保证结点顺序，如何处理谓词的作用域等。

无论是那种方法，都不能很好地解决 XQuery 查许处理的问题。一方面，一次一结点的操作，效率很低，而且很难优化，在关系数据库中，一次一集合的操作被证明效率远高于一次一结点的操作；另一方面，一次一集合的方法又和 XQuery 过程化编程语言的特点不相符，XQuery 是任意嵌套的语言，特别是在结果构造符中，一次一集合的操作有时不可能表达复杂的结果嵌套情况，而必须借用编程语言的线性执行的特点。

因此，本文提出的 OrientXA，试图结合这两种方法，为 XQuery 提供一个有效的代数系统。这种结合主要体现在结果构造符中。

本文的主要贡献有：

a) 总结分析了现有的查询代数的研究现状和问题所在，根据 XQuery 语言的特点，明确提出了包括强绑定/弱绑定，结点绑定/序列绑定等概念。
b) 提出了一套有效的 XML 查询代数系统，引入 Sequence 操作符，有效结合了两种 XQuery 处理策略的特点，能有效地表达 XQuery 查询，这是 OrientXA 区别于现有 XML 代数系统的最大不同之处。
c) 提出了大量 XQuery 中有别于关系代数的逻辑优化的问题。

本文的组织结构是这样的：第二节分析 XML 代数设计的现状；第三节提出 XML 代数设计中新的问题；第四节介绍 OrientXA 查询，并给出三个示例，第五节讨论 XQuery 代数处理中优化的问题。第六节是结论和未来工作。

2 相关工作

XML代数的目的是使用一次一集合的方式来处理查询。逻辑操作符的输入是记录的集合。问题是：什么是记录；XML代数与关系代数相比有什么不同。

目前，人们提出了很多 XML 代数[1,2,3,4,6,8,9,11,13]，著名的包括：TAX[6]，Xtasy[2]，TTX[3]，SAL[1]，XAL[4]等。

2.1 XML 代数处理的对象

XML 代数处理的对象是记录的集合。在 TAX 中，记录是 XML 树。TAX 引入了 pattern tree 和 witness tree 的概念。pattern tree 就是一组相关的变量绑定，witness tree 是 pattern tree 的实例。

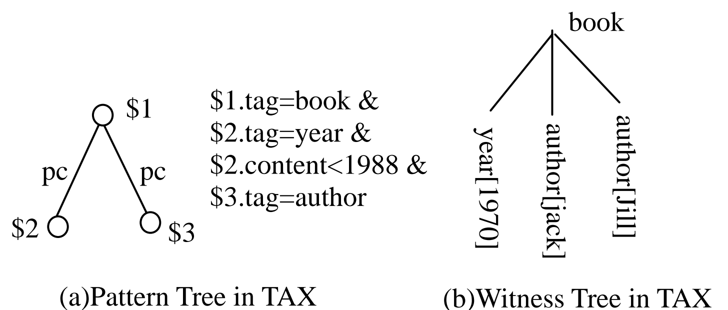


Figure 1: Pattern Tree and Witness Tree in TAX

在Xtasy和TTX中，记录是一种叫做Env的结构。他们首先使用input filter(类似于pattern tree)对输入文档用path操作符进行过滤，过滤的结果就是Env的集合。Input filter的定义是：

F := *(op; var; binder)label[F]*

它的含义是：

从当前上下文结点找到满足 op 关系（/, //, 或者@）的结点名为 *lable* 的结点，并用 binder 方式（in 绑定或者=绑定）绑定到变量 *var*，然后以当

前变量绑定为上下文结点，处理嵌套的 F。

比如，对于XQuery查询：

FOR $b in book,
$a in $b//author,

input filter 是：

$(_;\$b;in)book[(//;\$a;in)author[\phi]]$

表示以当前上下文结点找到 book 结点并"in"绑定到$b，然后以$b 为上下文结点，找到满足//关系的 author 结点并"in"绑定到$a。

FOR $b in book,
LET $a in $b//author,

input filter是：

$(_;\$b;in)book[(//;\$a;=)author[\phi]]$

唯一的区别是对$a的绑定是"="绑定。

绑定的结果是 Env 结构的集合。一个 Env 结构就是一个记录。如下图：

$b:$o_1$	$a:$o_3$
$b:$o_1$	$a:$o_4$
$b:$o_1$	$a:$o_5$
……	……

a) Env Set in Xtasy(in binding)

$b:$o_1$	$a:{$o_1,o_2,o_3$}
……	……

b) Env Set in Xtasy(= binding)

Figure 3 Flat Env in Xtasy

其中，o_i表示Object ID。在W3C的XML data model中，每一个结点都有一个全局唯一的ID。

TTX 中的处理与 Xtasy 很相似，只不过 TTX 把 input filter 变成树状表示，把 Env 结构也表示成树状。

2.2 XML 代数的操作符

XML 代数跟关系代数一样，都需要表达 select, project, product, rename, join 等操作。我们把这类操作符叫做类关系操作符。那么，XML 代数还需要哪些额外的操作符呢？由于 XQuery 语言和 XML 数据本身的特点，XML 代数还应该有两类操作符：Extraction 操作和 Construction 操作。

Extraction 操作符

在关系数据查询处理中，操作符处理的对象是结构化的扁平的元组。但是在 XML 查询处理中，XML 查询只是对 XML 文档上的某些特定结点有兴趣（谓词结点和目标结点），而这些结点往往只是整个文档的一小部分。所以，必须先把感兴趣的结点从输入文档中抽取出来，才能做下一步的处理。

TAX 把数据抽取操作合并在选择操作或者投影操作中，选择或者投影操作符的输入是 pattern tree 和谓词列表 PL。在 Xtasy 和 TTX 中，使用 path 操作符来进行数据抽取，path 操作符的输入就是 input filter。

实际上，pattern tree 或者 input filter 可以视为 XPath 的树状表示。所以，Extraction 操作实际上是 XPath 处理，而 XPath 是 XQuery 的核心部分。所以，Extraction 操作是 XML 代数中最影响查询效率的操作符之一。目前，处理 Extraction 操作的方法主要有两种：

1) Navigation：对文档树进行遍历，找到满足 pattern tree 的实例树；这种方法比较适合绝对路径的处理；
2) Structure join：利用对 XML 数据的编码和 Tag Index，快速地找到满足祖先后代关系的结点。这种方法比较适合相对路径（//）的处理。

如何混合使用这种两种方法，是目前的一个研究热点之一。

Construction 操作符

在 SQL 中，除了重命名，并没有结果构造的需求。但是，在 XML 查询处理中，往往要求查询结果以一定格式输出。所以，XML 代数中还需要结果构造符来满足这个需要。

在 Xtasy 和 TTX 中，结果构造符是 return 操作符。return 操作符的输入是 Env 结构和一个 output filter，output filter 就是想要输出的结果的框架。output filter 的定义是：

$OF ::= OF_1; : : : ;OF_n$
$\quad | \ label[OF]$
$\quad | \ @label[val]$
$\quad | \ val$
$val ::= v_B/var \ | \ vvar$

*label[OF]*表示元素构造，元素结点名是 label，内容由[OF]指定；*@label[val]*是属性构造符，属性名是 *label*，值是 *val*. *val* 可以是自定义的原子值 v_b，或者拷贝（*vvar*）或者引用（*var*）。*vvar* 是对结点的拷贝，不会保留原来的 oid，从而与源数据不再有联系；*var* 是对结点的引用，会保留原来的 oid，从而与原来的数据仍有联系。所以 *var* 可以用来定义视图。

比如，对于以下查询：

FOR $b in document("bib.xml")//book
WHERE $b/year<2001
RETURN <books>{ $b }</books>

转换成操作树是：

$return_{books[\$b]}(\sigma_{\$y<2001}(path(_ ;\$b; in)book[(/;\$y; in)year[\phi]](db1)))$

在 TAX 中，没有一个明确的结构构造符，它用 *grout-by*, *copy-and-paste*, *rename* 来完成结果构造的工作。

3 问题的提出

OrientXA 借鉴了上面的代数的思想。OrientXA 中，记录是 XML 树，操作符处理的对象是 XML 树的集合。OrientXA 使用 pattern tree 来表示感兴趣的结点，使用选择操作符来做抽取操作，使用构造操作来进行结果构造。在设计并实现 OrientXA 的过程中，我们遇到了一些新的问题，需要新的方法来处理，同时这些问题对查询优化提出了新的要求。

3.1 序列化操作符

XQuery 是可以任意嵌套的查询语言。这给代数式处理带来了额外的困难。[10]对 OO 的嵌套化进行了讨论，但是 XQuery 的嵌套比 OO 的嵌套更复杂。XQuery 更像一个第三代编程语言的代码，而不像 SQL 那样的结构化查询语言。有时候，由于表达式的任意嵌套，一些操作符必须序列化执行。比如：

Q2:

```
<bib>
  {
    FOR $b in doc("http://bstore1.example.com/bib.xml")//book
    WHERE count($b/author) > 0
    RETURN
        <book>
            { $b/title }
            {
                FOR $a in $b/author[position()<=2]
                RETURN $a
            }
            {
                if (count($b/author) > 2)
                then <et-al/>
                else ()
            }
        </book>
  }
</bib>
```

从 Q2 产生的 Pattern Tree 是 book(author, title)。在第一个 RETURN 子句中，嵌套了三个表达式：一个 XPath 表达式，一个 XPath 表达式，一个 FLWR 表达式和一个 IF-THEN-ELSE 表达式。给定一个实例树，这三个表达式应该被序列化地执行。需要注意地是，序列化执行和流水线执行是不同的。给定操作 op_1, op_2, ..op_n，所谓流水线执行，是指 op_{i+1} 的输入是 op_i 的输出；而序列化操作是指：对于一个记录 r，顺序地执行 op_1, op_2, ..op_n。

序列化操作实际上引入了一次一结点的处理思想。现有的 XML 代数和传统的关系代数都没有这样的操作符，因此不能很好地处理任意嵌套化的 XML 查询。OrientXA 引入了序列化操作符，很好地结合了一次一集合和一次一结点的处理策略。

3.2 强绑定和弱绑定

不是所有的变量绑定都要求一定绑定到结点，有些绑定可以绑定到空结点（或空结点集）。我们把前者叫作强绑定，后者叫做弱绑定。pattern tree 是相关变量绑定的树状表示，一个实例树应该满足 pattern tree 的所有的强绑定，而不要求必须满足弱绑定。在 OrientXA 中，对 pattern tree 的变量绑定，都标明了是强绑定还是弱绑定。[12]注意到了这个问题，但是并没有讨论哪些是强绑定。

我们发现，除了 FOR 语句的绑定是强绑定外，其他语句中的绑定（如 LET 语句，WHERE 语句和 RETURN 语句等）都是弱绑定。

比如对于以下查询：

FOR $b in (document(bib.xml)//bib/book
LET $a := $b/author
WHERE $b/year<1998 or $b/title="XML"
RETUN
<authorss isbn="{$b/@isbn}">
{$a}
</authors>

FOR 子句把 book 显式地绑定到$b，同时还包含了对 bib 的隐式绑定；LET 子句把 author 显式地绑定到$a，而 WHERE 子句中$b/year 对 year 的绑定，$/title 对 title 的绑定和 return 子句中$b/@isbn 对 isbn 的绑定则是隐式绑定。

在这些绑定中，$b 是强绑定。需要注意的是，如果 pattern tree 上的一个结点是强绑定，那么对 pattern tree 上它的祖先结点也是强绑定。所以，对 bib 的隐式绑定是强绑定，而其他隐式绑定都是弱绑定。如果一个 book 结点没有 author，year 等子结点或 isbn 属性，该 book 结点仍应该被视为符合该 pattern tree 的实例树。

强绑定可以过滤更多的结点，使结果集更小，从而有利于提高查询效率。有些情况下，一些弱绑定结点可以转化为强绑定结点，比如，如果 WHERE 子句变为：

WHERE $b/year<1998

那么我们可以认为对 year 的绑定是强绑定，因为如果 book 结点没有 year 子结点，那么它肯定不满足这个谓词，因此该 book 结点就会被过滤掉。

尽可能地把弱绑定转化为强绑定，是XML查询处理中一个很大的优化点。

3.3 结点绑定和序列绑定

在 XQuery 中，一个变量可以绑定到一个结点，如 FOR 绑定，也可以绑定到一个结点序列(Sequence)，如 LET 绑定，隐式绑定等。我们把前者叫作结点绑定，后者叫做序列绑定。在 XQuery 中，只有 FOR 绑定是结点绑定。结点绑定和序列绑定对结果的构造是有影响的。比如，下列两个相似的查询：

Q3 对 author 的绑定是序列绑定，而 Q4 对 author 的绑定是结点绑定，它们的结果就很不一样。

在 Xtasy 和 TTX 中，用"in"绑定来表示结点绑定，用"="来表示序列绑定。

TAX 中没有显式地标明。在 TAX 中，如果一个 Pattern tree 被用在选择操作符上，则应用结点绑定；如果用在投影操作符上，则应用序列绑定。但是，如果一个 pattern tree 上同时有结点绑定和序列绑定，就不好处理了。

[13]提出了一个 Tree Logical Class 的概念，用来表示序列绑定的概念。它对 Pattern Tree 上的结点做标记，来表明它是哪种绑定类型。

OrientXA 借鉴 Xtasy 和[13]的思路，显式地标明 pattern tree 的结点哪些是结点绑定，哪些是序列绑定。

Q3:

```
<books>
    FOR $b in document("bib.xml")//book
    LET $a:=$b/author
    WHERE $b/year<2001
    RETURN
      <authors>
        <aNum>{count($a)}</aNum>
          { $a }
      </authors>
</books>
```

Example Results:

```
<books>
  <authors>
    <aNum>2</aNum>
    <author>Smith</author>
      <author>Tom</author>
    </authors>
  …
</books>
```

Q4:

```
<books>
    FOR $b in document("bib.xml")//book
        $a:=$b/author
    WHERE $b/year<2001
    RETURN
      <authors>
          <acouont>{count($a)}</acouont>
          { $a }
          </authors>
</books>
```

Example Results:

```
<books>
  <authors>
    <aNum>1</aNum>
    <author>Smith</author>
  </authors>
  <authors>
    <aNum>1</aNum>
    <author>Smith</author>
  </authors>
  …
</books>
```

在 Xtasy 中，进行 path 操作的时候，对于结点绑定，每一个结点应该有一个单独的 instance tree（Env 结构），如 Figure3(a)所示；对于序列绑定，一个序列里的所有结点应该在一个 instance tree 中，如 Figure3(b)所示。

但是，问题似乎不是这么简单。如果对一个结点的绑定是序列绑定，但是对它的子结点的绑定是结点绑定，instance 怎么来表示？比如，对于以下变量绑定：

```
for $b in document("bib.xml")/bib
let $book := $b/book
for $a in $book/author
```

假设某一个 bib 下面有 4 个 book 子结点，由于是序列绑定，按照结点绑定和序列绑定的概念，，instance tree 应该被表示为 bib1 (book1, book2, book3, book4)。接下来，对于 author 是结点绑定，所以，每一个不同的 author 结点都应该有一个 instance tree。于是，就出现了这种奇怪的变量绑定结果：

bib1(book1(author1), book2, book3, book4), bib1(book1(author2),book2,book3,book4).

这里，(book1, book2, book3, book4)在每一个实例中都要重复一遍，是很大的冗余，这对查询处理的效率可不是个好消息，因为它会大大增大中间结果的大小。

OrientXA 正在研究如何在保证语义正确性的情况下减少这种冗余。这是我们未来优化工作的一个研究点之一。

3.4 选择操作和投影操作

在 SQL 中，选择操作和投影操作的区别是很明显的。选择操作选择表中的某些行，投影操作选择表中的某些列。在 XML 查询处理中，记录不是扁平的，它是嵌套的树结构。那么，在 XML 查询处理中，选择操作和投影操作的区别是什么呢？

Xtasy 和 TTX 的回答是：不要投影操作。path 操作用来抽取感兴趣的结点，然后用选择操作在上面作谓词判断。

在 TAX 中，选择操作和投影操作被用来区别结点绑定和序列绑定。

OrientXA 综合了 Xtasy 的思想：不引入投影操作符，也不引入专门的 Extract 操作符，而是直接用选择操作符来完成这个功能。

4 OrientXA 设计

OrientXA 的代数设计充分借鉴了已有的 XML 代数的优点，并且充分考虑了上述新的问题，并针对这些问题作了一些特殊的设计。本节介绍 OrientXA 的具体设计。

4.1 定义

Source Pattern Tree:

一个 Source Pattern Tree 是一棵树(N, E, R)，其中，N 是结点集合，E 是边的集合，R 是树的根。

每个结点有结点名，结点名可以为空，表示可以取任意结点名。结点名在 Pattern Tree 上不唯一。为了唯一地标识一个结点，每个结点分配一个 PID。每一个结点还有若干个修饰符，主要包括：

是否序列绑定：用双圆圈表示序列绑定的结点，单圆圈表示结点绑定的结点；

是否强绑定：用实线圆圈表示强绑定，虚线圆圈表示弱绑定；

是否连带绑定所有子孙结点：用 p 表示需要连带绑定所有子孙结点。

边用来表达结点间的关系。Pattern Tree 上有三种边：

父子边，用单实线表示；

祖先后代边，用双实线表示；

元素属性边，用单虚线表示。

Constructor Pattern Tree:

为了表示结果构造操作，引入 Constructor Pattern Tree。注意，Constructor Pattern Tree 上的边都是父子边或者元素属性边。

结点定义如下：

N := n(TagName) [= val]
| c (PID)
val := atom | PID

其中，n(TagName) = val 表示新建一个结点，结点名为 TagName，结点值为 val；c(PID)表示拷贝 input pattern tree 上特定 PID 结点，包括结点名，结点值，以及所有子孙结点；

val 可以取原子值 atom，也可以取某个 PID 结点的文本值。

Constructor Pattern Tree 的结点可以有以下修饰符：

是否序列构造：用双圆圈表示序列构造。序列构造只用于拷贝构造结点，表示输入集所有拷贝的结点挂载在同一个父结点下面。

为了方便描述，给出以下定义：

- 如果 X 是一个 XML 树集合，x 是该集合中一棵树，记做 $x \in X$
- 如果是 s 是 x 的一棵子树，记做 $s \in x, s \prec X$
- 如果 P 是一个 Pattern Tree, x 是 P 的实例树，记做 P(x)
- 如果 PE 是一个值谓词表达式，实例树 x 上的相应结点满足该谓词，记做 PE(x)。

4.2 OrientXA 操作符

每个操作符实际上都有一个 Pi 表示了输入数据的格式，有一个 Po 表示了输出数据的格式。我们把 Pi 叫做输入 Pattern，把 Po 叫做输出 Pattern。这里，有几个规则：

1）选择操作符的 Pi 可以为空，其它操作的 Pi 均不能为空；

2）结构构造操作符的 Po 是 Constructor Pattern Tree，其它均为 Source Pattern Tree。

3）每一个操作的 Po 是下一个操作的 Pi

选择操作符

$$\sigma_{P_i,P_o,PE}(X) = \{x \mid x \prec X, P_o(x), PE(x)\}$$

输入：

Pi：输入 pattern tree，是 Source Pattern Tree，可以为空；

Po：输出 pattern tree，是 Source Pattern Tree，不能为空；

PE：谓词链表，可以为空；

X：XML 树集合

输出：Po 的实例树集合

输出 Pattern：Po

结果构造操作符

$$\chi_{P_i,P_o}(X)$$

输入：

Pi：输入 pattern tree，是 Source Pattern Tree，不能为空；

Po：输出 pattern tree，是 Constructor Pattern Tree，不能为空；

X：XML 树集合

输出：按照 Po 构造的实例树集合

输出 Pattern：Po

序列操作符

$$\xi_{Pi,(op_1,op_2,\ldots op_n)}(X) = \{s(op_1(x), op_2(x), \ldots op_n(x)) \mid x \in X\}$$

输入：

Pi：输入 Pattern，是 Source Pattern Tree，不能为空。

op_i：一元操作符；

X：XML 树集合

输出：

$s(op_1(x), op_2(x), \ldots op_n(x))$，其中，s 是虚根结点。

输出 Pattern：

$s(Po_1, Po_2, \ldots Po_n)$，其中，$Po_i$ 是 op_i 的输出 Pattern。

连接操作符：

$$\rhd\lhd_{p_1,p_2,C}(X_1, X_2) = \{j(x_1, x_2) \mid x_1 \in X_1, x_2 \in X_2, C(x_1, x_2)\}$$

$$\otimes_{p_1,p_2,C}(X_1, X_2) = \{j(x_1, (x_{21}, x_{22}, \ldots x_{2n})) \mid x_1 \in X_1, x_{2i} \in X_2, C(x_1, x_{2i})\}$$

输入：

Pi：输入 pattern tree，是 Source Pattern Tree，不能为空；

Po：输出 pattern tree，是 Source Pattern Tree，不能为空；

C：连接谓词

X：XML 树集合

输出：

满足连接谓词的 j (x_1, x_2)集合，其中，j 是虚根结点。

输出 Pattern：j (P1, P2)

还可以在此基础上定义左连接。注意，没有右连接，因为由于 XQuery 的特性，不会出现右连接的情况：

$$L \rhd\lhd_{p_1,p_2,C}(X_1, X_2)$$

$$L \otimes_{p_1,p_2,C}(X_1, X_2)$$

分组操作符：

$$\gamma_{P,f}(X) = \{g(v, r(x_1, x_2, x_n)) \mid xi \in X, v = f(x_i)\}$$

输入：

P：输入 pattern tree，是 Source Pattern Tree，不能为空；

f：分组函数。

X：XML 树集合

输出：

$g(v, r(x_1, x_2, \ldots x_n))$，其中，$x_1, x_2, \ldots x_n$ 具有相同

的分组函数值 v；g 是虚根

输出 Pattern：g(v, r (P))

聚集操作符：

$$A_{p_i,f}(X)=\{a(v,x)\mid x\in X, v=f(X)\}$$

输入：

P：输入 pattern tree，是 Source Pattern Tree，不能为空；

f：聚集函数，如 sum, count, average 等。

X：XML 树集合

输出：a(v,x)，v 是聚集函数值；

输出 Pattern：a(v, P)

消重操作符：

$$\delta_{P,f}(X)=\{x\mid x\in X,\forall x_i\in\delta(X), if\ x_i\neq x\Rightarrow f(x_i)\neq f(x)\}$$

输入：

P：输入 pattern tree，是 Source Pattern Tree，不能为空；

f：消重函数，对于每一个输入的 XML 树，返回一个可比较的值。

X：XML 树集合

输出：消除重复值之后的 XML 树集合。

输出 Pattern：P

排序操作符

$$\tau_{P,f}(X)$$

输入：

P：输入 pattern tree，是 Source Pattern Tree，不能为空；

f：排序函数，对于每一个输入的 XML 树，返回一个可排序的值。

X：XML 树集合

输出：排序后的 XML 树集合

输出 Pattern：P

4.3 OrientXA 示例

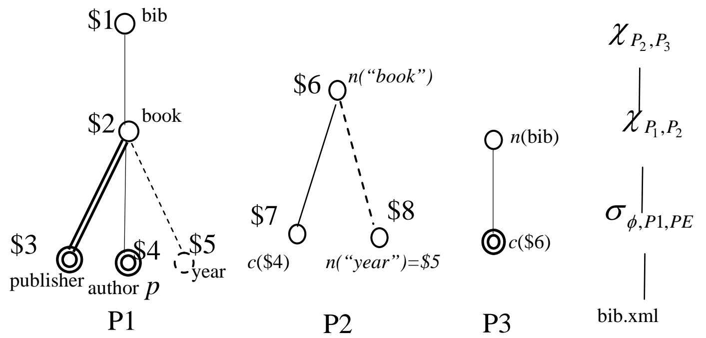
图例：

线型/符号	含义
———	父子边
═══	祖先后代边
- - - - -	元素属性边
p:	绑定所有后代结点
n:	新建结点
c:	拷贝结点
○	强结点绑定
虚线圆	弱结点绑定
◎	序列绑定或序列构造

示例 1：W3C Use Case 1.1.9.1:

```
<bib>
 {
  for $b in doc("bib.xml")/bib/book
  let $a := $b/author
   where $b//publisher/text() = "Addison-Wesley" and $b/@year > 1991
  return
     <book year="{ $b/@year }">
       { $b/author}
     </book>
 }
</bib>
```

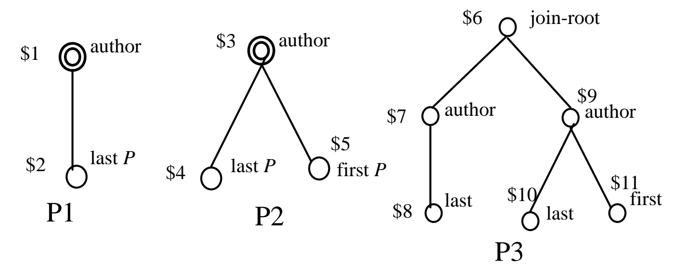
PE：$3=”Addison-Wesley” and $5>1991

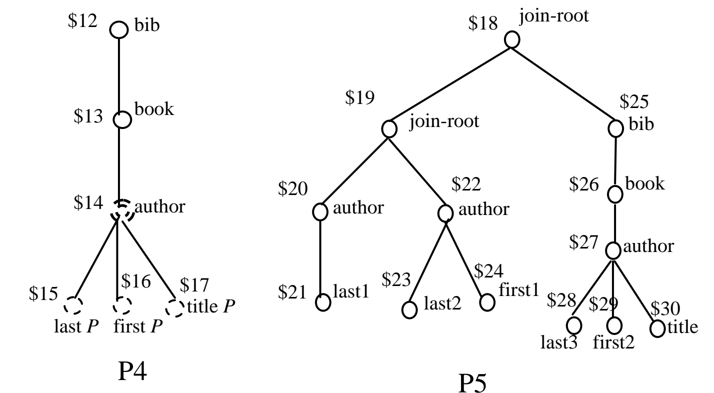


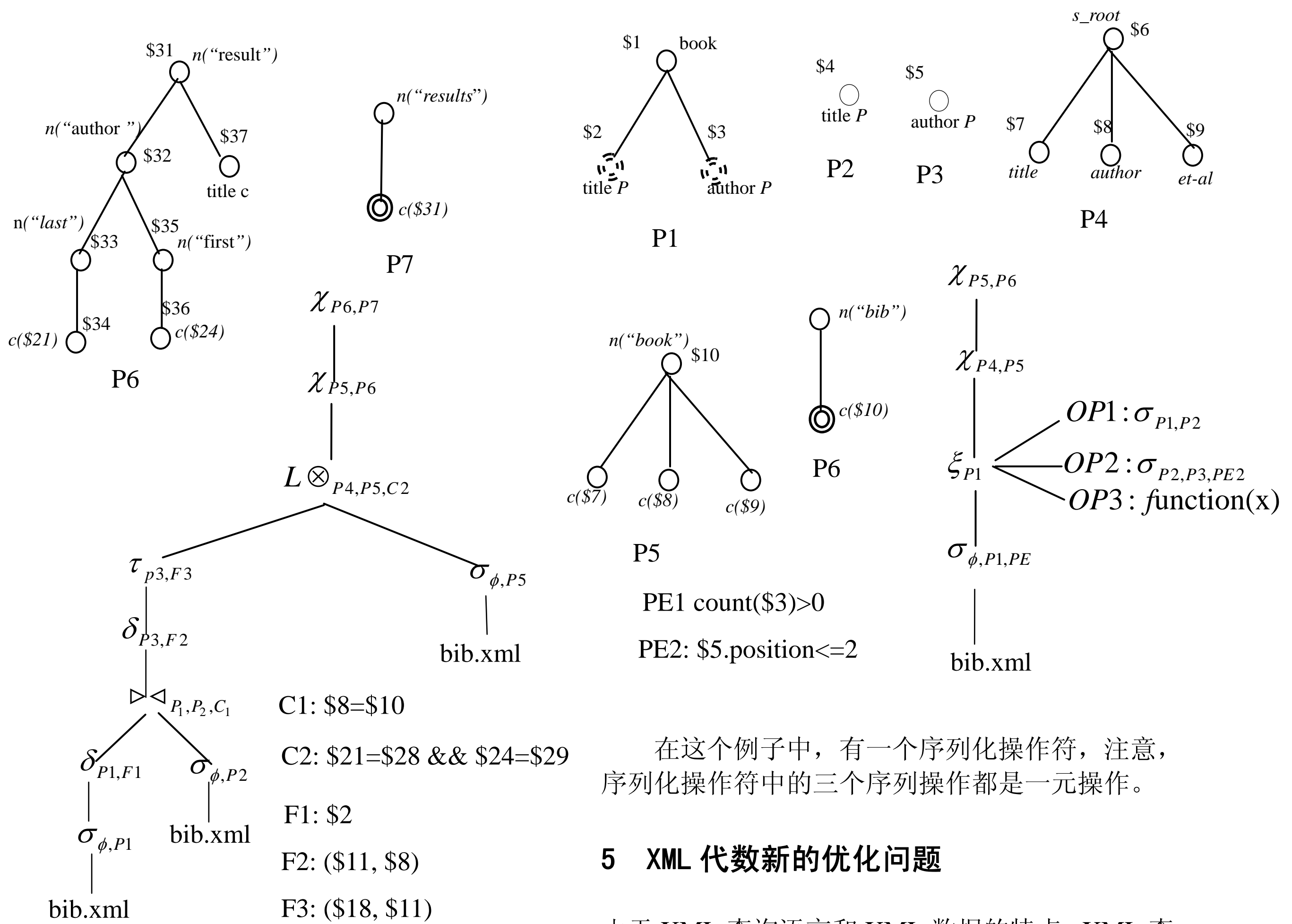
这个例子很简单，先用 P1 和 PE 抽取出感兴趣的数据，然后进行结果构造。

示例 2：W3C Use Case 1.1.9.4:

```
<results>
{
  LET $a := doc("http://bstore1.example.com/bib/bib.xml")//author
  FOR $last in distinct-values($a/last),
      $first in distinct-values($a[last=$last]/first)
  order by $last, $first
  RETURN
   <result>
      <author>
         <last>{ $last }</last>
         <first>{ $first }</first>
         </author>
         {
          FOR $b in doc("http://bstore1.example.com/bib.xml")/bib/book
          WHERE some $ba in $b/author
                 satisfies ($ba/last = $last and $ba/first=$first)
          RETURN $b/title
          }
  </result>
}
</results>
```







在这个例子中，有一个序列化操作符，注意，序列化操作符中的三个序列操作都是一元操作。

示例 2 比较复杂。先用 P1 抽取出数据，做一个消重操作；然后用 P2 抽取出数据，做一个连接操作，并对连接的结果消重并排序；然后用 P4 抽取出数据，跟上次的连接结果做一个左连接，最后进行结果构造。

示例 3：W3C Use Case 1.1.9.6

```
<bib>
  {
    FOR $b in doc("http://bstore1.example.com/bib.xml")//book
    WHERE count($b/author) > 0
    RETURN
        <book>
            { $b/title }
            {
                FOR $a in $b/author[position()<=2]
                RETURN $a
            }
            {
                if (count($b/author) > 2)
                 then <et-al/>
                 else ()
            }
        </book>
  }
</bib>
```

5 XML 代数新的优化问题

由于 XML 查询语言和 XML 数据的特点，XML 查询处理中出现了一些新的需要研究的优化问题。主要包括：

5.1 谓词的作用域

在 XQuery 查询中，由于表达式可以任意嵌套，谓词可以出现在任意地方。谓词是有作用域的。同一个谓词在不同的地方，会产生不同的查询结果。这同时意味着原来关系代数优化中普遍使用的选择下推规则，在 XML 查询处理中可能会导致不正确的结果。比如这个查询：

Q5:

```
<bib>
{
 for $b in doc("http://bstore1.example.com/bib.xml")//book,
 where $b/publisher="Adison"
 return
 <book-reviews>
 {
   {$b/title}
   for $v in doc("http://bstore2.example.com/reviews.xml")//review
   where $b/title=$v/title
   return
     <reviews>
       {$v/content}
       {$b/author[first="Smith"]}
     </reviews>
 }
 </book-reviews>
}
</bib>
```

在这个查询中，对于 bib.xml 上的结点，一共出现了两个选择谓词：\$b/publisher="Adison"和\$b/author/first="Smith" 和一个连接谓词\$b/title=\$v/title。在这里，两个选择谓词的作用域是不同的，谓词\$b/author/first="Smith"不能在连接之前做，否则结果会不正确。

在另一些情况下，谓词是可以被提到外层子句的，比如下面这个查询：

```
<bib>
{
  for $b in doc("http://bstore1.example.com/bib.xml")//book,
  where $b/publisher="Adison"
  return
   <book-reviews>
    {
    for $v in doc("http://bstore2.example.com/reviews.xml")//review
    where $b/title=$v/title and $b/author/first="Smith"
    return
       <reviews>{$v/content}</reviews>
    }
   </book-reviews>
}
</bib>
```

这时候，谓词\$b/author/first="Smith"就可以先于连接谓词进行判断，以使参与连接的集合变小。一般来说，如果一个内层谓词提到外层之后，该谓词所约束的子句是一样的，那么该谓词就可以提到外层，以提高查询效率。

如何快速地判断谓词的作用域并且判断谓词什么情况下可以提到外层（即选择下推），是 OrientXA 进行优化的一个方面之一。

5.2 更多的分组操作

XQuery 查询中，除了显式的分组操作，还有很多隐式的分组操作。这是由于 XQuery 的表达式可以任意嵌套。当存在相关嵌套子查询的时候，一般要转换为连接操作。这时候，为了构造期望的结果，就产生了分组的需要。比如以下查询：

```
<bib>
{
  for $b in doc("bib.xml")//book,
  where $b/publisher="Adison"
  return
     <book-reviews>
     {$b/title}
     {
        for $v in doc("reviews.xml")//review
        where $b/title=$v/title
        return
            <reviews> {$v/content} </reviews>
     }
     </book-reviews>
}
</bib>
```

一般来说，连接操作的形式是：

$$\triangleright\triangleleft_C (X_1, X_2) = \{r(x_1, x_2) \mid x_1 \in X_1, x_2 \in X_2, C(x_1, x_2)\}$$

其中，X1 和 X2 是参与连接的两个 XML 树的集合，C 是连接谓词。r 是虚的根结点，为了把连接结果以树的方式输出。

产生连接结果 r(X1,X2)之后，XQuery 的输出要求 X2 的项嵌套在 X1 的项中输出。这时候，就需要根据 X1.OID 进行一个分组操作，以构造期望的结果。

一个可能的避免这种分组操作的方法是引入一种新的连接操作：

$$\otimes_C (X_1, X_2) = \{r(x_1, (x_{21,} x_{22,} ... x_{2n})) \mid x_1 \in X_1, x_{2i} \in X_2, C(x_1, x_{2i})\}$$

这种连接的意思是说：给定 x1，把满足连接条件的所有 x2 挂载在同一个虚根结点下。

在处理两层嵌套的查询中，这种连接确实可以避免上面的分组操作。但是，在三层或者以上的嵌套中，这种连接操作就不管用了。

事实上，TAX 没有专门的结果构造符，它用 group-by, copy-and-paste 等操作符来完成结果构造的任务，这就反映出，在结果构造过程中，往往需要隐式的分组操作。

分组操作往往需要一次排序，是比较费时费力的操作。使用上述的新的连接操作，可以避免两层嵌套的查询。**如何避免或者减少由于结果构造带来的隐含的分组操作，也是 OrientXA 进行查询优化研究的一个方面。**

5.3 更多的排序操作

XML 数据的一个特点是有序性。除非查询语句特别指定，否则应该保持结点在源文档中的结点顺序。这就带来一个重大的问题：原来一些处理连接的算法，比如排序连接，Hash 连接，会打乱结点顺序，在连接完成后，需要对连接结果做一个额外的根据结点顺序的排序操作。而如果用 nest-loop 连接算法，则可以省去这趟额外的排序。在进行代价估计的过程中，这个额外的排序操作要被考虑在内。这就给进行查询优化的过程带来新的考虑因素。

一个可能的解决方法是在所有操作中，忽略结点的有序性，在最后构造结果的时候，再对结点按照文档顺序排序。

究竟是使用 nest-loop，还是最后增加额外的排序的方法，这是 OrientXA 的查询优化的一个研究点。

6 结论和未来工作

本文研究 XQuery 的处理策略，提出了一套 OrientXA 查询代数系统，它主要借鉴了关系代数和现有的 XML 查询代数的思想，同时，又在结果构造符上，体现了 XQuery 程序化语言的特点，因此能够有效地表达各种 XQuery 查询语言。本文还探讨了 OrientXA 代数优化的一些研究点，这些，都是我们未来的工作之一。

参 考 文 献

1 C. Beeri, Y. Tzaban: SAL: An Algebra for Semistructured Data and XML. WebDB 1999, Pennsylvania, USA. P37-42.

2 C. Sartiani, A. Albano. Yet Another Query Algebra For XML Data.

IDEAS 2002, Edmonton, Canada. P106-115.
3 D. Colazzo, C. Sartiani etc. A Typed Text Retrieval Query Language for XML Documents. JASIST 2002, Volume 53. P467-488.
4 F. Frasincar, G. Houben, C. Pau. XAL: An Algebra for XML Query Optimization. ADC 2002, Victoria, Australia. P49-56.
5 Galax XQuery engine. Availble at http://db.bell-labs.com/galax.
6 H. V. Jagadish, Laks V. S. Lakshmanan etc. Tax: A tree algebra for xml. DBPL 2001, Frascati, Rome. P149-164.
7 IPSI-XQ XQuery engine. Availble at http://www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/index_e.html.
8 M. Fern'andez, J. Simeon, P. Wadler. A Data Model and Algebra for XML Query. Technical Report. 2000. Available at http://homepages.inf.ed.ac.uk/wadler/papers/xquery-algebra/xquery-algebra.html.
9 M. Fernandez, J. Simeon, P. Wadler. An Algebra for XML Query. FST TCS 2000, New Delhi, India. P11-46.
10 S. Cluet, G. Moerkotte. Classification and Optimization of Nested Queries in Object Bases. Technical Report. 1994. Available at http://citeseer.ist.psu.edu/cluet94classification.html.
11 V. Christophides, S. Cluet, J. Siméon. On Wrapping Query Languages and Efficient XML Integration. SIGMOD 2000, Texas, USA. P141-152.
12 Z. Chen, H. V. Jagadish, etc. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. VLDB 2003, Berlin, Germany. P237-248.
13 S. Paparizos, Y. Wu, L. V. S. Lakshmanan, H. V. Jagadish: Tree Logical Classes for Efficient Evaluation of XQuery. SIGMOD 2004, Paris, France. To be published.