XML用木パターン言語XPath解説

田島　敬史

本稿では，XML用の簡単な検索言語とみなせるXPathについて解説する．まず，XPathの言語概要について説明し，次に，XPathに関する近年の研究の中から，XPathの評価の計算量に関する研究，XPathの様々な部分言語の表現力に関する研究，および，XPathの様々な部分言語の集合演算に対する閉包性に関する研究について簡単に解説する．

キーワード：XML，XPath，計算量，部分言語，表現力，閉包性

1. はじめに

本特集中の文献[1]でも触れられていたように，XMLデータ中の一部を指定するための言語としてXPath[2]が広く使われており，これは簡単なXML用検索言語ともみなせる．そこで本稿では，XPathの概要と，関連するいくつかの研究について解説する．なお，ここでの言語の定義の仕方は，文献[2]とは若干異なるが，問合せ式の意味は変わらない．

2. XPath言語概要

文献[1]で述べられていたように，XMLデータは木構造で表現でき，例えば，次に示すXMLデータは，図1に示すような木とみなせる．図中の数字は本文中で各要素を参照するために便宜上ふった物である．

```
<?xml version ="1.0"?>
<book>
  <foreword>.....</foreword>
  <chapter>
    <section>
      ...<figure>...</figure>...
    </section>
    <section>...</section>
  </chapter>
  <chapter>
    <section>...</section>
    <section>...</section>
  </chapter>
</book>
```

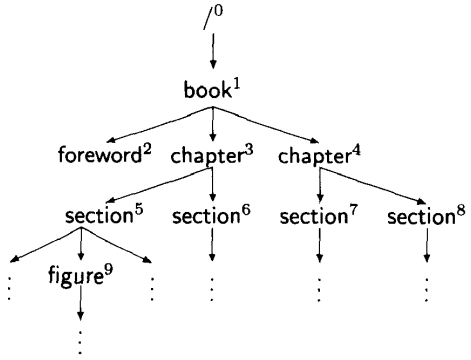


図1　XPathでのデータモデル

このデータに対するXPath問合せの例を次に示す．

/child :: book/child :: chapter[descendant :: figure]

この式のように，XPathによる問合せ式は，「/」で区切られたロケーションステップと呼ばれるものの並びである．そのうち，「/」で始まるものを絶対ロケーションパス，ロケーションステップで始まるものを相対ロケーションパスと呼ぶ．この二つ以外の場合もありうるが，ここでは省略する．各ロケーションステップは，

軸 :: ノードテスト[XPath表現]...[XPath表現]

という形をしている．[...]の部分は述語と呼ばれ，各ステップは，0個以上，任意個の述語を持つ．

ロケーションステップは，ある「文脈ノード」のもとで評価されて，ノードの集合を返す．軸は，文脈ノードから見て木構造中でどのような位置関係にあるノードを選択するかを指定するもので，主なところでは次のものが指定できる．

self, child, parent, descendant, ancestor, descendant-or-self, ancestor-or-self, following-sibling, preceding-sibling

各軸の意味は，順に，文脈ノード自身，子供，親，子

たじま　けいし
北陸先端科学技術大学院大学 情報科学研究科
〒923-1292 能美市旭台1-1

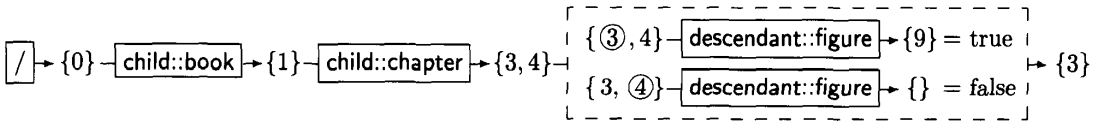


図2　XPathの実行の流れ

孫，祖先，自身または子孫，自身または祖先，後方の兄弟，前方の兄弟である．

また，ノードテストはノードの種類や名前を指定するもので，主なところでは次のものが指定できる．

名前　—その名前を持つノード
*　　—(上記の軸の場合は) 任意の要素ノード
text() —文字列ノード
node()—任意のノード

また，述語は，軸，ノードテスト，および自分より左にある述語によって絞り込まれた中間結果のノード集合をさらに絞り込むためのもので，述語中にはロケーションパス（絶対，相対），定数（数値，文字列），組込み関数，これらを演算子（or, and,=, >等）でつないだもの等が書ける．これらは，中間結果集合中の各ノードを文脈ノードとして評価され，この結果がtrueになる文脈ノードだけからなる集合が返される．この時，文脈ノードに加えて，中間結果集合の大きさ(文脈サイズ)，中間結果集合中で文脈ノードが何番目であるか（文脈ノード位置，詳細は省略）も文脈として参照できる．また，評価結果が真偽値型ではない場合は，

●ノード集合型は，空集合でなければtrue
●数値型は，文脈ノード位置と等しければtrue

等の規則によって，真偽値型に変換される．

ロケーションパスでは，左側のロケーションステップが返したノード集合中の各要素を文脈ノードとして，次のロケーションステップを評価し，これらの各文脈ノードに対する結果の集合すべての和集合を返す．これを，右端のステップまで繰り返していく．

例えば，上述の問合せ式を図1のデータに対して評価した場合の実行の様子は図2のようになる．図2中の番号は，図1中でノードに付けられていた番号，実線四角が軸とノードテストによる集合から集合へのマップ，破線四角が述語によるフィルタ操作である．

まず先頭の「/」は特別な意味を持ち，対象となるXML文書のトップレベル全体を表すノードを返す．次に，このノードを文脈ノードとしてchild:: bookを評価すると，このトップレベルにあるbook要素を元とする集合を返す．この集合中の各ノード（実際にはノード1のみ）を文脈ノードとして続くchild:: chapterを評価すると，ここまでで，根からbook, chapterとたどって得られるすべてのchapter要素の集合が返される．

述語内も同様の手順で評価され，descendant:: figureは，/book/chapterまでで得られる各要素について，その要素の子孫でfigureという要素名を持つものの場合を返す．ただし，述語の評価結果がノード集合となった場合は，空集合であればfalse，そうでなければtrueとすることになっているので，この場合，そのようなfigure要素が存在するchapter要素に対しては述語がtrueとなり，最終結果の集合に含まれることになる．

また，XPathでは，様々な省略記法が用意されている．これらのうち，主なものを次に示す．

●「軸 ::」を省略した場合はchild軸が使用される
●//≡/descendant-or-self :: node()/
●.≡self :: node()/
●..≡parent :: node()/

最後に，省略記法を用いて記述した問合せ式の例をいくつかと，それらの意味を示す．

●/book/chapter[2]/section[3]：根の下のbookノードの子供のchatperノードの中で2番目のものの，その子供のsectionノードの中で3番目のノード．
●/book/chapter[figure[10]][3]: bookの下のchapterノードのうち，その子供にfigureノードが10以上あるものの中で数えて，3番目のもの．
●/book/chapter[3][figure[10]]: bookの下のchapterノードのうちの3番目のものが，その子供にfigureノードを10以上持てば，その要素のみの集合，そうでなければ，空集合．
●//figure[1]: /descendant-or-self :: node()/figure[1]と等価で，任意の深さにあるfigureのうち，兄弟の中で最初のfigureノードであるようなもの（任意の深さにあるfigure要素で，最初のものを返す/descendant ::figure[1]とは意味が異なる）．

3. XPath評価の計算量

次に，本節では，XPathの評価の問題の計算量について概観する．一般に，問合せ処理の計算量を考える場合，問合せ式が固定でデータベースのサイズのみを考える場合（data complexity），データベースが固定で問合せ式のサイズのみを考える場合（query complexity），双方を問題の入力とし，両方のサイズを考える場合（combined complexity）があるが，ここでは最後のcombined complexityについて考える．

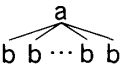
また，XMLデータがどのようなデータ構造で与えられるかも問合せ評価の計算量に影響する．XMLデータの表現形式としては，ポインタによる木構造として与えられる場合や，XMLの記述形式の文字列として与えられる場合，また，関係データベースやネイティブデータベースの中に，関係や専用のデータ構造によって格納されている場合等がある．このうち，データベースに格納されている場合については文献[1]に譲り，ここでは，ポインタによる木構造と文字列を考える．しかし，両者は対数領域で互いに変換可能[3]なので，**L**より上の計算量の処理を考える場合には，区別する必要はない．よって，ここではポインタによる木構造のみを考え，各ノードに対して，最初の子供へのポインタ（子供がない場合はnull）であるfirstChild，次の兄弟へのポインタ（ない場合はnull）であるnextSibling，およびこれらの逆である，$\mathsf{firstChild}^{-1}$，$\mathsf{nextSibling}^{-1}$が与えられるものとする．

3.1 時間計算量

まず，XPathの評価の時間計算量について考える．XPath評価の最も単純で分かりやすいアルゴリズムの一つとして，文脈とロケーションステップを受け取って結果となるノード集合を返す関数を定義し，この関数を根ノードと最初のロケーションステップに適用してノード集合を生成し，その各要素を文脈ノードとして次のロケーションステップを評価する関数を再帰的に呼び出し，以降これを繰り返していくようなアルゴリズムが考えられる．しかし，この方法は，問合せの大きさに対して指数時間の計算量となる．例えば，

/a/b/following-sibling :: b/following-sibling :: b

という問合せを，根の下のaの子供にn個のbがある，次のようなデータに対して実行したとする．



i番目の兄弟のbをb_iと書くことにすると，この場合，まず/a/bの中間解として$\{\mathsf{b}_1, ..., \mathsf{b}_n\}$を求め，この各要素をコンテキストノードとして次のステップfollowing-sibling :: bを実行していくと，まず，b_1に対して$\{\mathsf{b}_2, ..., \mathsf{b}_n\}$が求まり，これらの各要素をコンテキストノードとして次のステップを実行していくと，まず，b_2に対して$\{\mathsf{b}_3, ..., \mathsf{b}_n\}$が求まり，...というように，ロケーションステップを評価する関数への再帰的な呼び出しの回数が，ロケーションステップ数の指数オーダになる．実際，文献[4]によると，当時，存在した主な実装で実験を行ったところ，このような指数オーダの計算時間が観測されたそうである．

しかし，上述のような単純な再起呼出しによる実行には無駄がある．例えば，$\mathsf{a} \to \mathsf{b}_1 \to \mathsf{b}_3 \to \mathsf{b}_4$というパスで$\mathsf{b}_4$が解として求まる計算過程と，$\mathsf{a} \to \mathsf{b}_2 \to \mathsf{b}_3 \to \mathsf{b}_4$というパスで$\mathsf{b}_4$が解として求まる計算過程とでは，$\mathsf{b}_3 \to \mathsf{b}_4$の部分は同じ計算を重複して行っている．

文献[4]では，これに対して，XPathの評価を多項式時間，多項式領域で行うアルゴリズムが示されている．このアルゴリズムの要点は，次の二点である．まず，あるノード集合Sと軸$\mathcal{X}$が与えられた場合，$\mathcal{X}(S)=\{x' | x \in S, x \mathcal{X} x'\}$（ここで，$x \mathcal{X} x'$は$x$から$x'$が軸$\mathcal{X}$で到達できることを表す）は，入力データ木の大きさを$|D|$として，$O(|D|)$で，firstChild，nextSibling，$\mathsf{firstChild}^{-1}$，$\mathsf{nextSibling}^{-1}$から計算できる．

第二に，上述のような重複した計算を行わないために，左のステップが返すノード集合の各ノードを文脈として次のステップを順に実行していくのではなく，各ステップについて，context-value tableという物を計算していく．これは，何を文脈としてそのステップを実行したら，何が解になるかを表す，文脈と解の対の集合である．例えば，上述の問合せ中のステップfollowing-sibling :: bについては，次の集合となる．

$$\{(\mathsf{b}_1, \mathsf{b}_2), ..., (\mathsf{b}_1, \mathsf{b}_n), (\mathsf{b}_2, \mathsf{b}_3), ..., (\mathsf{b}_2, \mathsf{b}_n), (\mathsf{b}_3, \mathsf{b}_4), ..., (\mathsf{b}_3, \mathsf{b}_n), ..., (\mathsf{b}_{n-1}, \mathsf{b}_n)\}$$

正確には，文脈は，文脈ノード，文脈サイズ，文脈ノード位置の三つ組なので，context-value tableはこの三つと解の計四つ組の集合である．上では，文脈サイズと文脈ノード位置は解に影響しないので省略した．

このようなcontext-value tableを，問合せ式を木構造で表した際の葉に当たる部分式についてまず求め，これを結合することによって，ボトムアップに上位の部分式についてのcontext-value tableを求めて行き，最終的に根に当たる式，すなわち問合せ式全体に対す

るcontext-value tableを求める．この手法によって，問合せ式の大きさを$|Q|$，データ木の大きさを$|D|$として，$O(|D|^5*|Q|^2)$の時間計算量と$O(|D|^4*|Q|^2)$の領域計算量でXPathを評価することができる．

実際には，このアルゴリズムは多項式計算量ではあるものの，中間結果として不要なcontext-valueの四つ組を大量に生成して非効率的なので，同文献では，同様の計算を問合せ木の根からトップダウンに行う，より無駄の少ないアルゴリズムも示している．また，文献[5]では，上の計算量を$O(|D|^4*|Q|^2)$時間，$O(|D|^2*|Q|^2)$領域に改善したアルゴリズムが示されている．

また，文献[4]では，より小さい時間計算量で解けるXPathの部分言語として，Core XPathが示されている．Core XPathは，XPathから算術演算に関する機能と，文字列演算に関する機能を取り除いて，木構造のパターンマッチの機能のみに制限したものである．このようなCore XPathの評価は，単純な集合演算，前述の各軸に対応する二項関係$\mathcal{R}$，あるラベルlを持つ全ノードの集合を求める$T(l)$から構成される大きさ$|Q|$の代数式に変換することができ，$\mathcal{R}$も$T(l)$も$O(D)$で計算できることから，全体として$O(|Q|*|D|)$，すなわち問合せの大きさに対してもデータ木の大きさに対しても線形時間で計算できる．

3.2 並列化可能性

逐次計算による計算量に関しては，前節のように多項式時間，多項式領域による計算アルゴリズムが示されているが，では，並列計算によって，効率良くXPathの評価を行うことは可能であろうか．これについては，XPathの部分言語である前述のCore XPathでさえ**P**困難であることが文献[6]に示されている．一般に，**P**困難な問題は本質的に逐次的な計算を含んでおり，並列化によって効率良く解くことはできないと考えられているので，Core XPath，あるいはより大きなXPath部分言語の評価は，並列化によって効率良く解くことはできないと考えられる．

しかし，同じ文献[6]および文献[3]では，それぞれ，Core XPathから論理否定演算子を取り除いた部分言語positive Core XPathの評価の問題が，**LOGCFL**に属することを（文献[6]では，さらに**LOGCFL**完全であることも）示し，positive Core XPathの評価が並列化によって効率良く計算可能であることが示されている．**LOGCFL**とは，対数領域（**L**）で文脈自由文法（context free language, CFL）の問題に還元可能な問題のクラスであり，次のことが知られている．

$$\mathbf{NC}^1\subset\mathbf{L}\subset\mathbf{NL}\subset\mathbf{LOGCFL}\subset\mathbf{NC}^2\subset\mathbf{P}$$

$\mathbf{NC}^i$は，$O(n^i)$台の計算機によって$O(\log^i n)$の計算時間によって解ける問題のクラスである．つまり，**LOGCFL**は，**NL**（対数領域で非決定的に解ける問題のクラス）と**P**の間にあるクラスで，また，$\mathbf{NC}^2$に含まれるので，$O(n^2)$台の計算機を使った並列計算で$O(\log^2 n)$の計算時間で解けることになる．なお，

$$\mathbf{LOGCFL}\subset\mathbf{DSPACE}(\log^2)$$

となることも知られている（$\mathbf{DSPACE}(\log^2)$は$\log^2$領域で決定的に解ける問題のクラス）ため，この部分言語は，メモリ量が限られる環境においても解くことができるということも分かる[3]．

文献[6]では，positive Core XPathより大きい部分言語で**LOGCFL**に属する物として，pWFが示されている．pWFは，positive Core XPathに，文脈ノード位置と文脈サイズを返す関数position()，last()と，基本的な算術演算子，等号・不等号演算，数値リテラルを加え，逆に次の制限を加えた物である．

- $p[e_1]...[e_n]$のような複数の述語の並びを含むステップは許さない
- 算術演算の入れ子は許さない

これらの制限は，core XPathでは意味がないことに注意されたい．後者についてはcore XPathは算術演算を含まないことから明らかであり，前者については，core XPathにおいて次が成り立つことによる．

$$p[e_1]...[e_n]\equiv p[e_1 \text{ and } ... \text{ and } e_n]$$

上の式が成り立たないのは，e_i中にposition()やlast()が現れる場合のみである．以上から，pWFはpositive Core XPathを含む言語となる．

このpWFまで拡張しても**LOGCFL**内にとどまる理由は，上のような制限を加えた場合，評価の中間結果となるノード集合を明示的にに求める必要がないためである．逆に，上の制限がないと，例えば

/a/b[position()=...][position()=...]...

のような問合せでは，順にノード集合を求めていかないと，述語内の条件の計算ができない．

文献[6]では，完全なXPathに対しても，pWFと同様の制限に加えて，一部の組込み関数を禁じるなどして，**LOGCFL**内に押えたpXPathを定義しているが，ここでは詳細は省略する．

3.3 領域計算量

次に，領域計算量については，文献[3]に，その評

価問題がL（対数領域で決定的に解ける問題のクラス）に属する二つの部分言語Core XPath[1]とCore XPath^{+*}が示されている．Core XPath[1]は，軸を距離1のノードにしか進まないもの，すなわち，self, child, parent, previous-sibling, next-siblingのみに制限したもので，XPath^{+*}は，軸を前方の任意の距離のノードに進むもの，すなわち，self, descendant, descendant-or-selfのみに制限したものである．

文献[3]には，これらの部分言語の対数領域での評価アルゴリズムが示されている．このアルゴリズムは，問合せ木とデータ木を並行して深さ優先探索していき，問合せ木中の次のステップに進む度に，データ木中でも，現在見ているノードから，そのステップにマッチするパスで到達できるノードへと進んで行く．そして，そのようなノードがない場合は，バックトラックを行う．通常のXPathの評価をこのようなアルゴリズムで行う場合は，このバックトラックを可能にするために，これまでのステップにマッチしたノードをスタックに積んで覚えておく必要がある．

しかし，軸をXPath[1]のように制限した場合，現在見ているノードから，一つ前のステップにマッチしていたノードへは，対応するステップの軸を逆にたどればいつでも戻れるので，各ステップにマッチしたノードをすべて覚えていく必要はない．同様に，XPath^{+*}では，データ木中の各パス上でのノードの並び順だけが重要である．よって，例えば，//a//b//cという問合せを評価する時に，現在，あるaノードの下の，あるbノードにいて，その子孫にcノードが見つからなかったためバックトラックする場合，今見ているbノードの位置さえ記憶しておけば，aノードの位置は記憶しておかなくても，bノードから遡って最も根に近いaノードを探し，そのaノードの子孫で，位置を記憶している現在のbノードよりも先に現れる次のbノードへと進めば良い．よって，これらの部分言語では，これらの定数個のポインタを格納するため（およびその他の計算を行うため）に必要な対数領域のみで評価が可能となる．

4. 様々な部分言語とその性質

前節では，XPathの様々な部分言語が登場した．そこで，本節では，XPathのいくつかの部分言語について，それらの関係と性質について調べた文献[7]の内容の一部を簡単に紹介する．この文献では，前述のpositive Core XPathの軸をchild, parent, self-or-descendant, self-or-ancestorに制限し，一方で和演算∪を加えた言語[1]と，その七つの部分言語の計8個について調べている．これらの八つの部分言語は，

- 後方軸，すなわちparentとself-or-ancestorを含むかどうか．（↑）
- 任意長に対応する軸，すなわちself-or-descendantとself-or-ancestorを含むかどうか．（r）
- 述語を含むかどうか．（[]）

の3点の組合せの2*2*2=8通りである．これら各々を上に示した記号を使って表すことにして，八つの部分言語を例えば$\mathcal{X}_r^{\uparrow}$=「rと↑を含み[]を含まない言語」のように書くことにする．

4.1 表現力の部分言語間の関係

まず，これら八つの部分言語間の関係について，次が示せる[2]．

$$\mathcal{X} \subsetneq \mathcal{X}^{\uparrow}=\mathcal{X}_{[]}=\mathcal{X}_{[]}^{\uparrow} \subsetneq \mathcal{X}_r^{\uparrow} \subsetneq \mathcal{X}_{r[]}=\mathcal{X}_{r[]}^{\uparrow}$$

$$\mathcal{X} \subsetneq \mathcal{X}_r \subsetneq \mathcal{X}_r^{\uparrow} \subsetneq \mathcal{X}_{r[]}=\mathcal{X}_{r[]}^{\uparrow}$$

ここでは，要点についてのみ簡単に例を使って説明する．まず，$\mathcal{X}^{\uparrow}=\mathcal{X}_{[]}$については，直感的には，両者の間で，次のような書き換えが可能なことによる．

/a[b[c/d][e]]/f=/a/b/c/d/../../e/../../f

$\mathcal{X}_{[]}^{\uparrow}$から$\mathcal{X}_{[]}$へも同様の書き換えで↑を取り除くことができる．また，$\mathcal{X}_{r[]}=\mathcal{X}_{r[]}^{\uparrow}$については，文献[7]では，これらのいずれもが，ある木パターン言語に等しい事を使って証明しているが，具体的な書き換えとしては，例えば次のような書き換えを行えばよい．

//a//b/ancestor:: d=//d[.//a//b]∪//a//d[.//b]

文献[8]では，より大きなXPath言語について，与えられたXPath式から後方軸を除去する書換え方法が示されている．また，$\mathcal{X}_r^{\uparrow} \subsetneq \mathcal{X}_{r[]}$が等号でないという点については，次のような問合せを考えればよい．

//a//a[.//b]

これを，/a/a[b]=/a/a/b/..と書き換えられたことの類推から//a//a//b/ancestor:: aと書き換えても，等価にならない．//a//a//b/ancestor:: a/ancestor:: a//aも，やはり等価でない（各自で確認されたい）．

---

[1] 正確には，parentやancestorに対するノードテストを述語の形でしか書けないところが異なる．

[2] 文献[7]では，言語表現力に関して，二種類の等価性を考えているが，ここでは文献[7]で言うところのroot equivalenceについてのみ考える．また，前述のように，ここで考える言語は，文献[7]の言語とはノードテストの扱いが少し異なるので，ここで示している関係も，文献[7]で示されている物とは多少異なる．

その他の，$\subsetneq$ の箇所については，容易に確認できるものと思う．

4.2 部分言語の集合演算に関する閉包性

次に，各部分言語が，集合演算（和，積，補集合）の各々について閉じているかどうかを考える．ここで考える各部分言語は明示的に∪演算を持っているので，和について閉じていることは明らかである．積については，文献[7]で，$\mathcal{X}_r^{\downarrow}$ 以外が閉じていることが示されている．まず，$\mathcal{X}_{r[]}$ が（よって $\mathcal{X}_{r[]}^{\downarrow}$ も）が積に閉じていることについては，文献[7]では，この言語が，∃，∨，∧と二項関係 *child*, *descendant*, からなるある種の一階論理と等しいことを使って証明しているが，直感的には，例として次のような物を考えればよい．

```
//a[b]//d∩//a[c]//*[e]
  =//a[b]//a[c]//d[e]∪
    //a[c]//a[b]//d[e]∪
    //a[b][c]//d[e]
```

$\mathcal{X}_{[]}$ について（よって，$\mathcal{X}^{\downarrow}$ および $\mathcal{X}_{[]}^{\downarrow}$ についても）や，$\mathcal{X}_r, \mathcal{X}$ についても，同様に考えられる．しかし，後方軸を除去できない $\mathcal{X}_r^{\downarrow}$ の場合は，積について閉じていないことは，次のような例を考えればよい．

```
//a//a∩//b/ancestor :: a
```

この積は，//a//a[.//b]で表せるが，前述のように，これと等価な式は $\mathcal{X}_r^{\downarrow}$ では表現できない．

また，同論文では，これらの部分言語はすべて，補集合演算について閉じていないことが示されている．

5. 結び

本稿では，XPathの言語概要，その評価の計算量，様々な部分言語の表現力や，集合演算に対する閉包性について簡単に解説した．XPathに関するその他の研究としては，次のような問題に関するものがある．

XMLストリームデータに対する評価[9]：ネットワーク上を流れるXMLストリームデータに対してXPathを評価する場合に，データを先頭から一度だけ走査しながら評価を行うアルゴリズムに関する問題．

包含関係や等価性の判定[10]：与えられた二つのXPath式の解の間に，常に包含関係あるいは等価関係があるかを判定する問題．XPath式の簡単化等に応用できる．

最小ビュー問題[11]：XPath式の集合が与えられた時に，別のXPath式集合で，当初の各XPath式の解をそれらの解から計算可能で，かつ，それらの解の総サイズが最小な物を求める問題．ネットワーク越しに問合せを行う場合の通信量の最適化等に応用できる．

参考文献

[1] 天笠俊之，吉川正俊：「XMLデータベース技術概説」，オペレーションズ・リサーチ，第50巻，第6号，pp. 365-372, (2005).

[2] J. Clark, S. DeRose (eds): "*XML Path Language (XPath) Version 1.0. W3C Recommendation*," (1999).

[3] L. Segoufin: "Typing and querying XML documents :Some complexity bounds," In *Proc. of ACM PODS*, pp. 167-178, (2003).

[4] G. Gottlob, C. Koch, R. Pichler: "Efficient algorithms for processing xpath queries, " In *Proc. of VLDB*, pp. 95-106, (2002).

[5] G. Gottlob, C. Koch, R. Pichler: "XPath query evaluation: Improving time and space efficiency, "In *Proc. of IEEE ICDE*, pp. 379-390, (2003).

[6] G. Gottlob, C. Koch, R. Pichler: "The complexity of XPath query evaluation," In *Proc. of ACM PODS*, pp. 179-190, (2003).

[7] M. Benedikt, W. Fan, G. M. Kuper: "Structural properties of XPath fragments," In *Proc. of ICDT*, pp. 79-95, (2003).

[8] D. Olteanu, H. Meuss, T. Furche, F. Bry: "XPath: looking forward," In *Workshop on XML-Based Data Management*, pp. 109-127, (2002).

[9] A. K. Gupta, D. Suciu: "Stream processing of XPath queries with predicates," In *Proc. of ACM SIGMOD*, pp. 419-430, (2003).

[10] G. Miklau, D. Suciu: "Containment and equivalence for an XPath fragment, " In Proc. of ACM PODS, pp. 65-76, (2002).

[11] K. Tajima, Y. Fukui: "Answering XPath queries over networks by sending minimal views," In *Proc.of VLDB*, pp. 48-59, (2004).