DEWS2004 7-A-05

文書スキーマに基づくラベル付け手法を用いた XML 文書のための索引構造

清水　敏之† 　吉川　正俊††

† 名古屋大学 情報科学研究科
†† 名古屋大学 情報連携基盤センター
E-mail: †shimizu@dl.itc.nagoya-u.ac.jp, ††yosikawa@itc.nagoya-u.ac.jp

あらまし　WWW の普及とともに XML(Extensible Markup Language) は電子化された情報の標準的な記述フォーマットとなりつつある．XML は単にデータを記述するだけではなく，データの伝達や変換にも優れたものである．本研究ではスキーマを持つ XML 文書を対象として索引付けを行い，高速な検索を実現することを目的とする．XML の各ノードに識別子を付与し，その識別子を元に木構造の索引を構築する．XML 文書の検索は XPath を用いて行うことを想定し，XPath 式を，構築した木構造索引に適合した形に変換して索引のエントリとすることにより，高速な検索を実現する．識別子の付与の方法と索引の構成の仕方を工夫することで，XML 文書の更新や削除などの操作にも強い索引を提案する．

キーワード　XML, スキーマ, 木構造索引, XPath

An Index Structure for XML Documents Using a Schema-based Identifier

Toshiyuki SHIMIZU† and Masatoshi YOSHIKAWA††

† Graduate School of Information Science, Nagoya University
†† Information Technology Center, Nagoya University
E-mail: †shimizu@dl.itc.nagoya-u.ac.jp, ††yosikawa@itc.nagoya-u.ac.jp

Abstract　XML is now one of the standard format for computerized information. XML is superior in not only description of data but also transmission or conversion of them. The aim of this study is to search faster by indexing XML document that has schema. The nodes in XML document are given an identifier, and tree-structured indices are created using them. It's usual to searche XML document by XPath, and the indices can be used by inputting appropriately transformed XPath. The indices is good for operation like updating or deleting XML document because of being thought out techniques of giving identifier and creating indices

Key words　XML, schema, tree-structured index, XPath

1. はじめに

XML [1] は，その単純な構造と，人間にも機械にも理解しやすい特徴から，現在，電子化された情報の標準的な記述フォーマットの一つになっている．XML の普及とともに，XML の変換，連結などの関連技術も開発され，もはや，XML はそれを利用している意識がない場合でも，システム内部のどこかでは利用されている，といっても過言ではない．

このような状況の中で，XML の部分的な構造を高速に取り出す技術は，非常に有用である．本研究では XPath [2] を用いて XML の構造と内容に関する検索を行うことを考え，XML 文書に対して，木構造索引を生成することによって，XPath による問合せの結果を高速に得る事を目的とする．

提案する索引構造では XML 中の各ノードに，スキーマから得られた情報を反映した識別子を付与し，その識別子と，XML 文書中のテキストを組み合わせて索引を構築する．

問合せとなる XPath が入力されると，それを本研究で提案する索引構造にそった形に変換し，索引を用いることによって高速に検索を行うことができる．

XML 文書に対して，木構造の索引を付与するものとしては XR-Tree [7] をあげることができる．XR-Tree では，XML 文書中のノードに対して，(*start,end*) で表現される識別子を付与し，

```
<!ELEMENT document (title, section+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT section (title, section+) | (#PCDATA)>
<!ATTLIST title
        lang CDATA #IMPLIED
        >
```

図 1　XML 文書の DTD

```
<?xml version="1.0" encoding="Shift_JIS"?>
<document>
 <title lang="en">XML index</title>
 <section>
  <title>B-Tree</title>
  <section>use B-Tree</section>
  <section>B-Tree extension for special case</section>
 </section>
 <section>summary</section>
</document>
```

図 2　実際の XML 文書の例

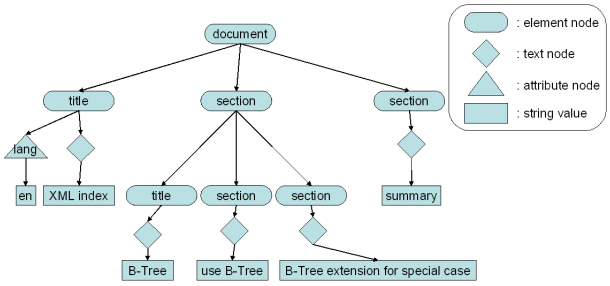

図 3　XML 文書の例

木構造の索引を構築することで，ノードの子孫先祖関係，親子関係を効率的に得ることができるものである．

本研究では，索引構造として B+木を使用することが適切であると考え，索引の構成要素 (キー) として XML の構造と内容を反映した NRP Compact Suffix というものを生成し利用する．この NRP Compact Suffix は広範囲の XPath による検索が可能であり，さらに，B+木のキーとして適切であるようにできる限り短くなるように工夫されている．構造に関する情報を効率的に扱うため，XML 文書中のエレメントノードにはスキーマから得られる構造情報を反映した識別子が付与される．ノードに付与する識別子としては，経路依存の識別子である SPIDER [5] に兄弟番号を加えた SPIDERS を利用する．この識別子と XML 文書中の内容 (テキスト) を組み合わせて NRP Suffix Array(Normal and Reverse Path Suffix Array) [6] の概念を取り入れ，さらに，データ長を短くするための工夫を行うことで，NRP Compact Suffix は生成される．

2. ノードへの識別子の付与法

本研究で提案する索引を構築するには，まず，文書中の各エレメントノードへ識別子を付与することが必要である．以下では図 1 の DTD に基づく図 3 の XML 文書を例として説明を進める．図 3 は図 2 の XML 文書の木構造を図で表したものである。

ノードへの識別子の付与として以下のようなものを提案する．まず，根ノードからの経路を一意に特定するような識別子を各ノードに付ける．この識別子を SPIDER (Schema-based Path IDentifiER) と呼ぶこととする．SPIDER は文献 [5] で提案された手法を用いて付与される．SPIDER は親子，先祖，子孫といった関係を計算によって求めることができる経路依存の識別子である．

StruDTD 表を生成するアルゴリズム

Input: A DTD or XML schema
Output: Table StruDTD and fanout f

save pairs a->b in PAR-CHI
for each pair a->b
  if element content type is 'choice'
    cOrder ← 1;
  else
    if(a->b is the start of a segment)
      cOrder ← 1;
    else *cOrder* ← previous *cOrder* + 1; endif
  endif
endfor

loop
  for each pair a->b
    if ∃ a preceding pair a' ->b &&
    cOrder are equal then
      ⇑ *cOrder* of a->* by 1
    endif
  endfor
  if all segments are fixed then break;
endloop
return PAR, CHI, *cOrder*, *f*←max(*cOrder*)

図 4　StruDTD 表を生成するアルゴリズム

PAR	CHI	cOrder
document	title	1
document	section	2
section	title	2
section	section	3

表 1　StruDTD 表

この手法では，まず，与えられたスキーマからノードの親子関係を抜き出し，それぞれの関係に cOrder と呼ばれる番号を付ける．この cOrder は，親ノード名と子ノード名が分かれば特定でき，かつ，親ノード名と cOrder が分かれば子ノード名が分かり，さらに，子ノード名と cOrder が分かれば親ノード名が分かるようなものである．そして，その cOrder を保持する表を作成する．この表を StruDTD 表と呼ぶこととする．たとえば図 1 の DTD に対する StruDTD 表の例を表 1 に与える．StruDTD 表を作成するアルゴリズムを図 4 に与える．

この cOrder を元にノードに SPIDER を付与する．SPIDER を付与するアルゴリズムは以下のようなものである．ここで，この手法で付けた，根ノードからのパスを一意に特定するようなノード n の識別子 (SPIDER) を n.sid とする．また，入力となる XML を T とし，$parent(n)$ はノード n の親ノードを返す関数，childOrd(tag1, tag2) は StruDTD 表において親のタグ名が tag1 で子のタグ名が tag2 であるような cOrder を返す関数である．

ノードに SPIDER を付与するアルゴリズム

Input: An XML tree T rooted at γ, a fanout f>1
Output: sid of nodes in T

travel T in the preorder
  if n is the root
    n.sid ← 1;
  else p ← *parent*(n);
    corder ← childOrd(p.tag, n.tag);
    n.sid ← f * (p.sid - 1) + 1 + *corder*;
  endif
endtravel

図 5　ノードに SPIDER を付与するアルゴリズム

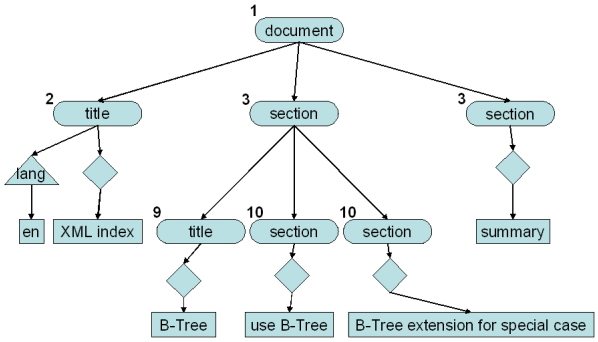

図 6　SPIDER

ノード n のタグ名は n.tag で得られるものとする．f は cOrder の最大値である．

このアルゴリズムでは，まず根ノードの SPIDER を 1 とする．そして，ノード n がノード p の子ノードであるとすると，ノード n の SPIDER は (ノード pの $SPIDER-1) \times f+1+corder$ となる．$corder$ は親ノードが p であり子ノードが n であるときの cOrder である．

この手法でノードに SPIDER を付与した XML の例として図 6 を挙げ，以降，この XML を元に考えていく．ここで，SPIDER の最大値を MAX_ID として覚えておく．この場合は $MAX\_ID = 10$ となる．

さらに同じ SPIDER を持つ兄弟ノードを識別するために，同じ SPIDER を持つような兄弟ノードに兄弟番号を振り，その兄弟ノードを特定する．

このようにして振られた識別子を SPIDERS (Schema-based Path IDentifiER with Sibling enumeration) と呼び，これは図 7 のようなものになる．この識別子付けを元に索引を生成していくことを考える．

3. 索引の生成

本研究で提案する索引は B+木上の索引であり，2 節の手法で付与された識別子，SPIDERS を利用して構築される．

3.1 木構造索引の特徴

ここで，本研究で使用する木構造索引の特徴を簡単に述べておく．

木構造索引は古くから関係データベース管理システム

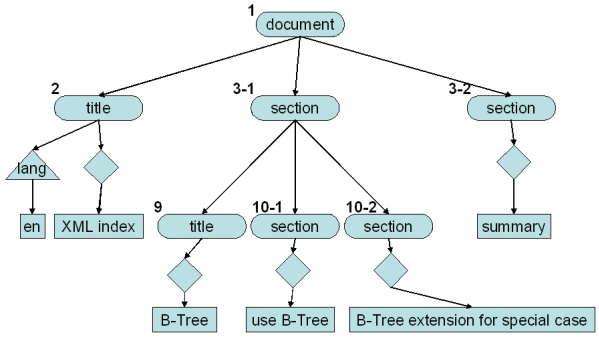

図 7　SPIDERS

(RDBMS) などで用いられてきた，非常に強力な索引手法である．等号検索だけでなく，範囲検索にも強い．また，データの挿入や削除などの操作に強く，索引は動的に構築される．

3.2 索引の構成要素

図 7 のような識別子を想定したとき，索引は以下のように構成される．

索引構造として B+木を考え，そのエントリとして適切であるように，できるだけ短く，かつ一定に近い長さの構成要素を構築していく．

まず，XML の葉ノードに対応するノードに着目し，その各ノードに対して，そのノードに至るまでの経路と，そのノードの内容，さらに，そのノードから根ノードまでの経路を得る．次に，属性ノードに着目し，それぞれの属性ノードに対して，その属性ノードが所属するエレメントノードまでの経路と，属性名，属性値，さらにその経路の逆順を得る．経路は識別子の “,” で区切られた列で表現し，もし葉ノードがテキストノードであれば，テキストノードの識別子として “#text” を与える．また，属性名は “@lang” のように属性名の前に “@” を付けるものとする．この処理を図 3 の全ての葉ノードと属性ノードに適用した結果は以下のようになる．

```
1, 2, @lang, en, @lang, 2 , 1
1, 2, #text, XML index, #text, 2, 1
1, 3-1, 9, #text, B-Tree, #text, 9, 3-1, 1
1, 3-1, 10-1, #text, use B-Tree, #text, 10-1, 3-1, 1
1, 3-1, 10-2, #text, B-Tree extension for special …
1, 3-2, #text, summary, #text, 3-2, 1
```

このようにして得られた経路を [6] にならって NRP(Normal and Reverse Path) と呼ぶこととする．このようにして得られた各 NRP の “,” と “ ”(スペース) で区切った suffix を NRP Suffix(Normal and Reverse Path Suffix) と呼び，そのような全ての NRP Suffix の配列を NRP Suffix Array と呼ぶ．上記の NRP に対しては以下のような NRP Suffix Array が得られる．

```
1, 2, @lang, en, @lang, 2 , 1
2, @lang, en, @lang, 2 , 1
@lang, en, @lang, 2 , 1
en, @lang, 2 , 1
@lang, 2 , 1
2 , 1
1
1, 2, #text, XML index, #text, 2, 1
2, #text, XML index, #text, 2, 1
#text, XML index, #text, 2, 1
XML index, #text, 2, 1
index, #text, 2, 1
#text, 2, 1
2, 1
```

```
1
1, 3-1, 9, #text, B-Tree, #text, 9, 3-1, 1
3-1, 9, #text, B-Tree, #text, 9, 3-1, 1
9, #text, B-Tree, #text, 9, 3-1, 1
#text, B-Tree, #text, 9, 3-1, 1
B-Tree, #text, 9, 3-1, 1
#text, 9, 3-1, 1
9, 3-1, 1
3-1, 1
1
1, 3-1, 10-1, #text, use B-Tree, #text, 10-1, 3-1, 1
3-1, 10-1, #text, use B-Tree, #text, 10-1, 3-1, 1
10-1, #text, use B-Tree, #text, 10-1, 3-1, 1
#text, use B-Tree, #text, 10-1, 3-1, 1
use B-Tree, #text, 10-1, 3-1, 1
B-Tree, #text, 10-1, 3-1, 1
#text, 10-1, 3-1, 1
10-1, 3-1, 1
3-1, 1
1
1, 3-1, 10-2, #text, B-Tree extension for special …
3-1, 10-2, #text, B-Tree extension for special …
10-2, #text, B-Tree extension for special case, …
#text, B-Tree extension for special case, #text, …
B-Tree extension for special case, #text, 10-2, …
extension for special case, #text, 10-2, 3-1, 1
for special case, #text, 10-2, 3-1, 1
special case, #text, 10-2, 3-1, 1
case, #text, 10-2, 3-1, 1
#text, 10-2, 3-1, 1
10-2, 3-1, 1
3-1, 1
1
1, 3-2, #text, summary, #text, 3-2, 1
3-2, #text, summary, #text, 3-2, 1
#text, summary, #text, 3-2, 1
summary, #text, 3-2, 1
#text, 3-2, 1
3-2, 1
1
```

これに対して，重複を取り除き，ソートすると，以下のようになる．

```
#text, 2, 1
#text, 3-2, 1
#text, 9, 3-1, 1
#text, 10-1, 3-1, 1
#text, 10-2, 3-1, 1
#text, B-Tree, #text, 9, 3-1, 1
#text, B-Tree extension for special case, #text, 10-2, …

#text, summary, #text, 3-2, 1
#text, use B-Tree, #text, 10-1, 3-1, 1
#text, XML index, #text, 2, 1
@lang, 2 , 1
@lang, en, @lang, 2 , 1
1
1, 2, #text, XML index, #text, 2, 1
1, 2, @lang, en, @lang, 2 , 1
1, 3-1, 9, #text, B-Tree, #text, 9, 3-1, 1
1, 3-1, 10-1, #text, use B-Tree, #text, 10-1, 3-1, 1
1, 3-1, 10-2, #text, B-Tree extension for special …
1, 3-2, #text, summary, #text, 3-2, 1
2, #text, XML index, #text, 2, 1
2, @lang, en, @lang, 2 , 1
2, 1
3-1, 1
3-1, 9, #text, B-Tree, #text, 9, 3-1, 1
3-1, 10-1, #text, use B-Tree, #text, 10-1, 3-1, 1
3-1, 10-2, #text, B-Tree extension for special case, …
3-2, #text, summary, #text, 3-2, 1
3-2, 1
9, 3-1, 1
9, #text, B-Tree, #text, 9, 3-1, 1
10-1, #text, use B-Tree, #text, 10-1, 3-1, 1
10-1, 3-1, 1
10-2, #text, B-Tree extension for special case, #text, …

10-2, 3-1, 1
B-Tree, #text, 9, 3-1, 1
B-Tree, #text, 10-1, 3-1, 1
B-Tree extension for special case, #text, 10-2, 3-1, 1
case, #text, 10-2, 3-1, 1
en, @lang, 2 , 1
extension for special case, #text, 10-2, 3-1, 1
for special case, #text, 10-2, 3-1, 1
index, #text, 2, 1
special case, #text, 10-2, 3-1, 1
summary, #text, 3-2, 1
use B-Tree, #text, 10-1, 3-1, 1
XML index, #text, 2, 1
```

ここで得られた NRP Suffix は，長さにばらつきが大きく，このまま B+木のエントリとするには適さない．そこで，まず，この NRP Suffix から冗長な情報を取り除くことを考える．たとえば，

```
3-1, 9, #text, B-Tree, #text, 9, 3-1, 1
```

という NRP Suffix に対しては，その先頭部分の “3-1, 9, #text” と逆順経路部分の “#text, 9, 3-1” が単純に順番を入れ替えただけのものであることがわかる．よって，“#text, 9, 3-1” の部分を取り除いても，もとの情報は保存されるため，

```
3-1, 9, #text, B-Tree, 1
```

とすることができる．SPIDER では，必ず子供の識別子は親の識別子より数字が大きいようになっているため，このような省略が可能である．

そして，識別子から始まるような NRP Suffix に対して，B+木のエントリとしての適切性を考え，なるべく短くするためにそのテキスト部分を取り除く．テキストの情報はテキストから始まるような NRP Suffix のみに保持される．

たとえば，

```
3-1, 9, #text, B-Tree, 1
```

というものに対しては，

```
3-1, 9, #text, 1
```

とする．

最後に，テキストから始まるような NRP Suffix に対して，ある閾値 L 以上の長さのテキスト部分は削除する．これは NRP Suffix の長さを，できるだけ短く，かつできるだけ一定の長さにするための作業であり，B+木のエントリとしての適切性を考慮したものである．

たとえば，L=4 とすると，

```
B-Tree extension for special case, #text, 10-2, 3-1, 1
```

という NRP Suffix は，

```
B-Tree extension for, #text, 10-2, 3-1, 1
```

とする．

これらの作業を NRP Suffix に適用し，ソートした上で重複したものを取り除くと以下のようになる．この経路情報を NRP Compact Suffix (Normal and Reverse Path Compact Suffix) と呼ぶこととする．

```
#text, 2, 1
#text, 3-2, 1
#text, 9, 3-1, 1
#text, 10-1, 3-1, 1
#text, 10-2, 3-1, 1
@lang, 2 , 1
1
1, 2, #text
1, 2, @lang
1, 3-1, 9, #text
1, 3-1, 10-1, #text
1, 3-1, 10-2, #text
1, 3-2, #text
2, #text, 1
```

```
2, @lang , 1
2, 1
3-1, 1
3-1, 9, #text, 1
3-1, 10-1, #text, 1
3-1, 10-2, #text, 1
3-2, #text, 1
3-2, 1
9, #text, 3-1, 1
9, 3-1, 1
10-1, #text, 3-1, 1
10-1, 3-1, 1
10-2, #text, 3-1, 1
10-2, 3-1, 1
B-Tree, #text, 9, 3-1, 1
B-Tree, #text, 10-1, 3-1, 1
B-Tree extension for, #text, 10-2, 3-1, 1
case, #text, 10-2, 3-1, 1
en, @lang, 2 ,1
extension for special, #text, 10-2, 3-1, 1
for special case, #text, 10-2, 3-1, 1
index, #text, 2, 1
special case, #text, 10-2, 3-1, 1
summary, #text, 3-2, 1
use B-Tree, #text, 10-1, 3-1, 1
XML index, #text, 2, 1
```

このようにして得られた NRP Compact Suffix は一つ一つが「ある条件を満たすあるノード」を表している，と見ることができる．たとえば，識別子から始まるような，

```
3-1, 9, #text, 1
```

という NRP Compact Suffix は,「根ノードである document ノードの子要素である 1 番目の section ノードのうち，テキストノードを持つような title ノードを子ノードに持ってるもの」を表している．また，テキストから始まるような，

```
case, #text, 10-2, 3-1, 1
```

は,「根ノードである document ノードの子要素である 1 番目の section ノードの子要素の 2 番目の section ノードの子要素のテキストノードに “case” という文字列を含むような，document ノード，ならびに 1 番目の section ノード，ならびに 2 番目の section ノード，テキストノードの 4 つのノード」を表している，とみなす．

それぞれの NRP Compact Suffix には，対応するその経路が示すノードの記憶装置上の位置へのポインタが付随する．たとえば，識別子から始まるような，

```
3-1, 9, #text, 1
```

に対しては，図 8 の②にあたるノードへのポインタが付随し，テキストから始まるような，

```
case, #text, 10-2, 3-1, 1
```

に対しては，その 4 種類の識別子部分に対応するような，図 8 の①，②，③，④ にあたる各ノードへの 4 つのポインタが付随する．

そして，この NRP Compact Suffix を索引の構成要素として索引を構築する．

3.3 索引の構築

前節の手法で求めた NRP Compact Suffix をもとにして，索引を構築する．まず NRP Compact Suffix を識別子から始まるようなものと，テキストから始まるようなものの 2 種類に分類する．

これらの 2 種類の NRP Compact Suffix はそれぞれ特性が異なるため，別々に木構造の索引を構築する．NRP Compact Suffix を B+木のキーとして B+木を構築する．図 9 に識別子から始まるような NRP Compact Suffix に対して構築された B+木の例を示す．また，図 10 にテキストから始まるような NRP Compact Suffix に対して構築された B+木の例を示す．

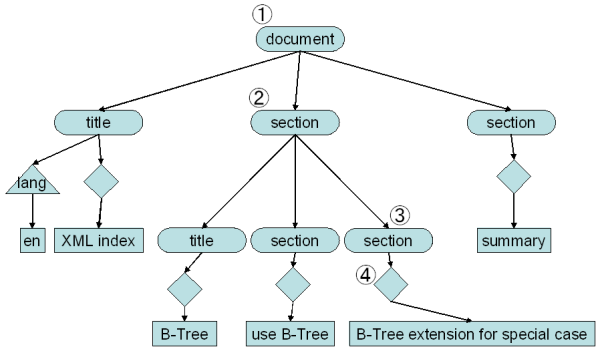

図 8　NRP Compact Suffix が示すノード

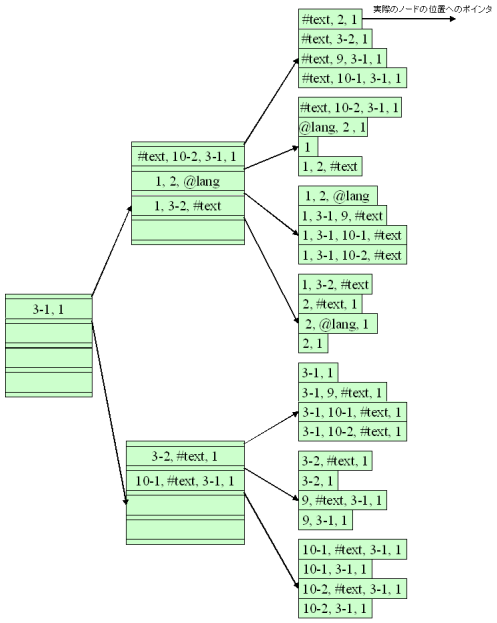

図 9　識別子から始まるような NRP Compact Suffix に対して構築された B+木

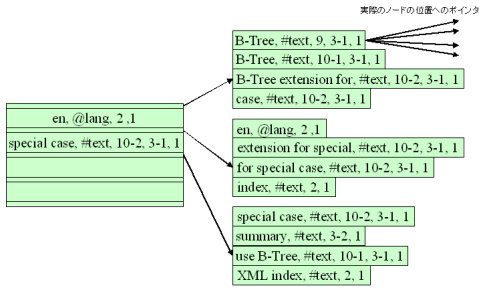

図 10　テキストから始まるような NRP Compact Suffix に対して構築された B+木

3.2 節で NRP Suffix から NRP Compact Suffix に変換する際に行った作業をしておくことで，ここで索引を構築する際に B+木の葉のエントリ一つ一つの大きさを小さくすることができる．

このように索引を B+木を用いて作成することによって，ノードの挿入や削除といった操作を B+木のアルゴリズムを用いて

行うことができる．このことは，効率的にノードを検索できるだけでなく，索引の管理を容易にすることができる．

4. XPath の変換

XML を検索する際には XPath [2] を用いるのが一般的である．3 節のような手法で索引を構築するが，XPath による検索要求があると，その XPath を索引の入力として適切な形に変換しなければならない．

4.1 XPath 変換の例

図 3 の XML に対する XPath をいくつか挙げ，その変換手順を示す．

（1） `/document/section/title`

この問合せは，根ノードから，“document” という名前のノードをたどり，“section” という名前の子ノードをたどり，そして，その子ノードの中で “title” という名前のノードを得る，という問合せである．まず，

```
/document
```

の部分を “1” という識別子に変換し，さらに，

```
/document/section
```

の部分を “3-?” という識別子にする．ここで “3-?” の “?” は任意の数字に適合することを示す．同様に，

```
/document/section/title
```

の部分から “9” という識別子を得る．そして，これらの数字を大きい順に並べて “9, 3-?, 1” とし，変換を終了する．

（2） `//title`

この問合せは，根ノード以下の子孫ノードのうち，“title” という名前のノードを得る問合せである．この場合は，まず，表 1 から “title” という名前を持つノードがどのような経路にあるかを計算する．実際に計算してみると，

```
/document/title
/document/section/title
/document/section/section/title
:
:
```

というパターンであることが分かる．これらのうち，

```
/document/section/section/title
```

以降のものは，識別子へ変換する際に，識別子の最大値 (MAX_ID) を超えてしまうため，不適合な経路として削除する．結局，

```
/document/title
/document/section/title
```

の 2 通りの XPath を変換する問題となり，先ほどと同様の計算により，“2, 1”，“9, 3-?, 1” という変換結果が得られる．

（3） `/document/section[title]`

この問合せは，根ノードから，“document” という名前のノードをたどり，“section” という名前の子ノードのうち，その子ノードに “title” という名前のノードを持つものを得る，という問合せである．この場合は，同様の手順により，“1”，“3-?” を得て，さらに，

```
/document/section[title]
```

から “9” という識別子を得る．そして，それらの識別子から “3-?, 9, ~, 1” と変換する．ここで，“~” は任意の識別子列に適合することを示す．

（4） `//section[contains(., 'B-Tree')]`

この問合せは，根ノード以下の子孫ノードのうち，子ノードにテキストとして’B-Tree’ という文字列を持つような “section” という名前のノードを得る問合せである．この場合は，まず，

```
/document/section
/document/section/section
```

という経路を計算によって得て，“1”，“3-?”，“10-?” という識別子を得る．そして，それらの情報を組み合わせ，“B-Tree*, #text, 3-?, 1”，“B-Tree*, #text, 10-?, 3-?, 1” という変換結果を得る．ここで，“*” は任意の文字列に適合することを示す．

（5） `/document/section[contains(title, 'B-Tree')]`

この問合せは，根ノードから，“document” という名前のノードをたどり，“section” という名前の子ノードのうち，その子ノードに’B-Tree’ という文字列を持つような “title” という名前のノードを持つものを得る，という問合せである．この場合は，まず，計算によって “1”，“3-?”，“9” という識別子を得る．そして，それらの情報を組み合わせ，“B-Tree*, #text, 9, 3-?, 1” という変換結果を得る．

このような手順で変換された問合せを索引の入力として，検索を行う．本研究で提案する NRP Compact Suffix は，スキーマ情報を反映した識別子である SPIDERS を用いているため，親や子、兄弟の識別子を計算によって求めることができ，そのことを利用して，高速に問合せを処理できる．また，XML の構造とテキストを組み合わせた表現であるため，構造とテキストの両方で絞り込むような問合せに強いことが期待される．

4.2 XPath 変換の限界

本手法では索引の構成要素として NRP Compact Suffix を用いているため，NRP Compact Suffix で表現しきれないような XPath は変換できない．その場合は，一度，索引を用いて検索し，その結果の情報も用いて XPath を変換する，などの工夫が必要である．たとえば，

```
//title[@lang="en"]/text()
```

という問合せに対しては，まず，lang 属性が “en” であるような title ノードを求める．これは “en, @lang, 2, 1”，“en, @lang, 9, 3-?, 1” という問合せに変換される．この変換結果を用いて索引をたどると，B+木の葉には “en, @lang, 2, 1” というパターンのエントリのみが発見される．

その情報を元に，もとの XPath 式を “#text, 2, 1” と変換することができる．

また，B+木を用いて索引を構築することを考え，NRP Compact Suffix には閾値 L 以上の長さのテキストは入っていない．そのため，その長さ以上のテキストで検索したい場合は，索引のみでは結果を決定できない．

その場合は，まず，検索したいテキストの始めの L-1 単語を用いて索引を用いて検索し，その検索結果により結果の候補を得る．そして，その後，実際のデータを見て，検索要求に合致しているか否かを判定する必要がある．

これと同様の問題に単語の途中から始まる文字列で検索できないことがあげられる．これは NRP Suffix を構築する際に単語

spider	sibling	st	ed
#text.2.1.	1.1.1.	71	79
#text.3.1.	1.2.1.	218	224
#text.9.3.1.	1.1.1.1.	104	109
#text.10.3.1.	1.1.1.1.	127	136
#text.10.3.1.	1.2.1.1.	156	188
@lang.2.1.	0.1.1.	55	55
1.	1.	44	245
1.2.#text.	1.1.1.	44	245
1.2.@lang.	1.1.0.	44	245
1.3.9.#text.	1.1.1.1.	44	245

図 11　NRP Compact Suffix のうち識別子から始まるものについて生成された関係表の一部

text	spider	sibling	odr	st	ed
B-Tree	#text.9.3.1.	1.1.1.1.	1	104	109
B-Tree	#text.9.3.1.	1.1.1.1.	2	97	117
B-Tree	#text.9.3.1.	1.1.1.1.	3	88	208
B-Tree	#text.9.3.1.	1.1.1.1.	4	44	245
B-Tree	#text.10.3.1.	1.1.1.1.	1	127	136
B-Tree	#text.10.3.1.	1.1.1.1.	2	118	146
B-Tree	#text.10.3.1.	1.1.1.1.	3	88	208
B-Tree	#text.10.3.1.	1.1.1.1.	4	44	245
B-Tree extension for	#text.10.3.1.	1.2.1.1.	1	156	188
B-Tree extension for	#text.10.3.1.	1.2.1.1.	2	147	198
B-Tree extension for	#text.10.3.1.	1.2.1.1.	3	88	208
B-Tree extension for	#text.10.3.1.	1.2.1.1.	4	44	245

図 12　NRP Compact Suffix のうちテキストから始まるものについて生成された関係表の一部

で区切って構築したからである．しかし，実際問題として単語の途中から始まる文字列で検索したい，という状況は考えづらいし，そのような検索要求の可能性がある場合は，NRP Suffix を構築する際に一文字ずつ区切って構築することで，容易に対応できる．

5. 予 備 実 験

本研究で提案した手法の有効性を確認するために，簡単な予備実験を行った．予備実験では関係データベースの木構造索引機能を利用し，NRP Compact Suffix を適切に分解して関係データベースに格納することで行った．

NRP Compact Suffix のうちの，識別子から始まるものとテキストから始まるものは別の関係表に格納する．識別子から始まるものについては，その兄弟番号部分を分離して別属性に格納する．テキストから始まるものについては，テキスト部分，兄弟番号部分をそれぞれ分離して別属性に格納する．そして，それぞれの NRP Compact Suffix に対して，対応するノードの開始タグと終了タグのバイト位置を格納する．テキストから始まるような NRP Compact Suffix に対しては，対応するノードが複数あるため，その数だけ NRP Compact Suffix を複製して格納する．例で示した XML に対しては図 11，図 12 のような関係表が得られる．図 11，図 12 において，spider は識別子部分を，sibling は分離した兄弟番号部分を，text は分離したテキスト部分を，odr はテキストから始まるような NRP Compact Suffix に対して何番目のノードに対応したバイト位置が入っているか，st は開始タグの位置，ed は終了タグの位置を格納する．

このようにテキスト部分や兄弟番号部分を分離して格納することで，SQL による検索時に，できるだけ無駄な文字列照合を行わないようにした．そして，text，spider，sibling に対して木構造索引を生成し，実験を行った．

実験においては XMark [8] によって生成された XML 文書を使用した．使用した XML 文書は 1 ファイルで，容量は 1.12MB，エレメントノード数 17132 個，属性ノード数 3919 個，テキストノード数 31089 個である．そして，図 11，図 12 のような関係表を生成し，検索にかかる時間を計測した．検索においては XPath を SQL に変換する時間は含めず，SQL を処理する時間を検索にかかる時間とした．実験環境は CPU: Intel Pentium III Processor 1.26GHz-S (デュアル), メモリ: 2GB, OS: RedHat Linux 7.1 である．DBMS としては Oracle 9i Release 1 を使用した．

	XPath 式
Q1	/site/open auctions/open auction
Q2	/site/regions//description
Q3	/site/regions//description/text[contains(.,'distinguish')]
Q4	/site/people/person[homepage]
Q5	/site/people/person[homepage and address]

表 2　評価のために使用した XPath 式

	本手法 (秒)	XRel(秒)
Q1	0.291	0.296
Q2	0.289	0.300
Q3	0.511	0.492
Q4	0.289	1.421
Q5	1.770	2.555

表 3　処 理 時 間

比較対照として，XRel [9], [10] を利用した．XRel は XML 文書を関係データベースを利用して格納する XML データベースである．

問合せとして使用する XPath 式としては表 2 を使った．それぞれの問合せに対してかかった時間を表 3 に示す．問合せはそれぞれ 10 回行い，その平均時間を結果として得た．

この予備実験において，Q1〜Q3 については比較対照の XRel とほとんど同じ処理時間で結果を得ることができた．構造とテキストの両方で絞り込む Q3 のような問合せに関しては，そのような問合せに対して特別な構造を持っていない XRel よりも高速に結果が得られることが期待されたが，この予備実験ではほとんど差がなかった．これは，本予備実験において，テキストを複製して格納したため，冗長な情報が関係表の中に含まれていることと，構造の情報からも絞込みができるため，テキストの情報があまり有効に使われなかったことが原因だと思われる．Q4 のような構造的な条件によって絞込みを行う問い合わせに関しては XRel よりも高速に結果を得ることができた．これは本手法では結合操作を行うことなく結果を得ることができるためだと思われる．Q5 のような問い合わせに関しては，本

手法においても結合操作を必要とするが，その回数が少なくてすむため，XRel よりも処理時間が短くなったと思われる．今後，本索引を GiST [3] などで実装することにより，さらに高速に結果を得ることができるようになると思われる．

6. おわりに

6.1 まとめ

本研究では，XML 文書の XPath による検索の高速化を目的とし，木構造の索引を構築する手法を提案した．その際，XML 文書中の各エレメントノードにはスキーマを反映した識別子である SPIDERS を付与し，それと，XML 文書中のテキストを組み合わせた NRP Compact Suffix を構成要素とする B+木上の索引を構築した．NRP Compact Suffix を得る際には，B+木のエントリとしての適切性を考え，できるだけ短く，かつできるだけ一定の長さにするための工夫を行った．また，その索引を使用する際の XPath の変換例を挙げた．そして予備実験を行い，本研究で提案する索引が構造的な条件によって絞込みを行うような問い合わせにおいて特に有効であることを確認した．

6.2 今後の課題

本研究の現段階では XML に対する索引の構成手法を述べ，簡単な予備実験をするにとどまったが，今後，実際に本手法で構築された索引を GiST [3] などの手段により厳密に実装し，その性能を検証しなければならない．

また，本手法には，様々な変種が考えられる．たとえば，NRP Suffix を構築する際に単語単位で区切るか，文字単位で区切るか，というものや，NRP Compact Suffix に対するテキストの長さの閾値 L としてどれくらいの数字をとるか，といったことが考えられる．それぞれについて，検証を行う必要がある．NRP Suffix を構築する際に一文字ずつ区切ったり，閾値 L を大きくしたりすると，検索精度が向上するかわりに索引サイズが大きくなると思われる．

XPath の変換に関しても，その正確なアルゴリズムを提案し，複数回索引をたどる必要のある XPath や，絞り込むことはできるが正確な結果を得ることができない XPath のパターンがどのようなものかを明確にする必要がある．

文　献

[1] W3C. Extensible Markup Language (XML). http://www.w3.org/XML/. 1996-2003.

[2] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath. 1999.

[3] Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer. Generalized Search Trees for Database Systems. VLDB Conf., pp. 562-573 1995.

[4] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. Proc of the 1984 ACM SIGMOD Int'l Conf on Mgmt of Data, 45-57

[5] Dao Dinh Kha, Masatoshi Yoshikawa, Shunsuke Uemura: "Virtual Joins for XML Data", Information Science Technical Report NAIST-IS-TR2003012, Graduate School of Information Science, Nara Institute of Science and Technology, ISSN 0919-9527, November 2003.

[6] Yohei Yamamoto, Masatoshi Yoshikawa and Shunsuke Umeura: "On Indices for XML Documents with Namespaces", Conference Proceedings of Markup Technologies '99, GCA, pp. 127-135, GCA, Philadelphia, U.S.A., December 7-9, 1999.

[7] Haifeng Jiang and Hongjun Lu and Wei Wang and Beng Chin Ooi: "XR-Tree: Indexing XML Data for Efficient Structural Joins", The 19th International Conference on Data Engineering (ICDE 2003), pages 253-264, Bangalore, India, March 5-8, 2003.

[8] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, R. Busse: "XMark: A Benchmark for XML Data Management", In Proceedings of the International Conference on Very Large Data Bases (VLDB), pp 974-985, Hong Kong, China, August 2002.

[9] 吉川正俊, 志村壮是, 植村俊亮: オブジェクト関係データベースを用いた XML 文書の格納と検索, 情報処理学会論文誌, Vol.40, No.SIG 6(TOD 3), pp.115-131 (1999).

[10] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, Shunsuke Uemura: XRel: APath-Base Approach to Strage and Retrieval of XML Documents Using Relational Database, ACM Transactions on Internet Technology, Vol.1, No.1, pp.110-141 (2001).