XML 文書の内容および構造検索のための B+木索引

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　XML 文書に対する問合せを高速に処理する技術は非常に有用である．XML 問合せ処理に関する過去の研究では主に XML 文書の構造に焦点をあてた検索を取り扱っているが，本研究では索引を生成し利用することによって構造検索と全文検索の両方を高速化することを目的とする．実用性の観点から，提案する索引は，広く普及している B+木を用いることを設計指針とする．XML 文書の構造を保持し，問合せを効率的に処理するために，XML 文書中の各ノードには根ノードからの経路情報の符号を含む識別子を付与する．提案する索引構造では，ノード識別子をエントリとする B+木と，テキストと識別子の組合せをエントリとする B+木を構築する．索引は GiST を用いて実装し，実験において索引容量と問合せ処理時間を示す．

キーワード　XML，全文検索，B+木，ノード識別子

An XML Index on B^+-Tree for Content and Structural Search

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 query processing is one of the most active database research areas. Although the main focus of the past research has been the processing of structural XML queries, there are growing demands for full-text search for XML documents. In this paper, we propose new indices which aim for high-speed processing of both full-text and structural queries on XML documents. An important design principle of our indices is the use of B^+-tree. To represent structural information of XML trees, each node in the XML tree is labeled an identifier. The identifier contains an integer number representing the path information from the root node. We have designed two types of indices, STB-tree and COB-tree using B^+-tree. Index entries of the COB-tree are a pair of text fragment in the XML document and the identifier of the leaf node which contains the text, whereas index entries of the STB-tree are an identifier of nodes. We have implemented COB-tree and STB-tree using GiST. We will show the efficiency of our indices in the experimental study.

Key words　XML query processing, full-text search, B^+-tree, node labeling scheme

1. はじめに

XML [1] 文書に対して，XPath [2] や XQuery [3] などで記述された問合せを高速処理することは重要な研究課題である．XML の論理構造は木であり，問合せ処理を行う上で，ノードの親子関係あるいは子孫関係を高速に判定することは重要である．そのために，構造的な結合手法，索引付け，ノード番号付けなどこれまでに多くの提案がなされてきた [4]〜[13].

一方，XML 文書の全文検索に関しては，W3C により全文検索に関するワーキングドラフト [14], [15] が発表されたり，TeXQuery [16] といった全文検索言語の提案も行われるなど，注目され始めているが，その高速化に関しては，まだ十分な研究が行われていない．特に文書指向 (document-centric) な XML 文書においては，XML 文書に対する検索としては，その構造に焦点をあてた検索だけでなく，それに加えて，XPath において contains() 関数を用いる検索のような全文検索の要求は高い．そのため，XML 文書の全文検索に対する索引技術は非常に重要になってくると思われる．

本研究では，XML 文書の構造と全文検索の両方に関係する問合せを高速処理するための索引を B+木上に実現する方法を

提案する．B+木上に索引を構築することは実際に使用する観点から見て重要である．B+木を利用する理由は，多くのデータベースで利用されているため，それを利用した索引は広く普及する可能性があるためである．

B+木を利用した，XML 文書のための索引はすでにいくつかのものが提案されている．たとえば，XISS [8] はノードを単位として索引を構築する索引手法である．ノードに対して，(*order,size*) で表現される識別子を付与する．ノードを単位としているため、柔軟性があるが，問合せに答えるためには問合せを分解し，結果を結合する必要がある．また，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 [9] 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，B+木を拡張した索引を構築する．しかし，これらの索引は，ノードの先祖/子孫関係，親子関係を効率的に得ることを目的としており，全文検索は対象としていない．また，XR-Tree については，B+木を独自拡張しており，実装が困難である．

我々は，XML の構造検索と全文検索をともに高速に処理するために，テキストにそのテキストが現れるノード識別子を付加したものを探索キーとする B+木を構築する．ここで重要な点は，ノード識別子は，構造による検索の絞込みができるように，根ノードからの経路を表現する番号と，根ノードからの経路中のノードの兄弟番号情報の二つから構成される点である．このような探索キーで B+木を構築することにより，索引中で，同じテキストを持っているノードに関するエントリがクラスタ化され，さらにそのようなクラスタの中では，同様の構造を持っているノードに関するエントリを近い位置に保持することができる．我々は，この索引を COB-tree (COntent B^+-tree) と呼ぶことにする．たとえば，“//title[contains(.,'XML')]” のような XPath 問合せは，COB-tree を一度たどるだけで答を得ることができる．

しかし，COB-tree は，最初にテキストによってエントリがソートされているため，“//title” のような構造のみによる問合せを高速に処理することはできない．そこで，上述のノード識別子を探索キーとするもう一つの B+木を構築する．これを STB-tree (STructure B^+-tree) と呼ぶ．ただし，ノード識別子を構成する経路番号の大小関係に従って B+木を構築すると，一般的にエントリの順序に意味がなくなってしまう．そこで，我々は，*search-key mapping* という手法を用いる．これは，経路番号の大小ではなく，その経路番号が表現する逆経路式 (ノードから根ノードへさかのぼった経路) の辞書順で，エントリをソートしておき，探索時にも経路番号から逆経路式を参照しながら探索を進める手法である．逆経路を用いるのは，XPath 問合せにおける “//” を含む問合せに対して，その子孫ノードからの絞込みが有効であると考えたためである．これにより，同じタグ名を持つノードに関するエントリは索引中で並んだ位置に存在するようになり，“//title” といった問合せにも高速に答えることができる．

本研究では，部分的な更新・削除等の操作が行われない，静的な XML 文書を対象とする．XML 文書集合からあらかじめ索引を生成し，検索の際に利用して検索処理の高速化を図るが，更新等の操作に対してはここでは考えない．

一般に，XML 文書に対する contains() 関数を含むような XPath 問合せを処理する際には，問合せ中のテキストを用いて絞り込んでいく手法と問合せ中の構造情報を用いて絞り込んでいく手法が考えられる．COB-tree と STB-tree により，これら二種類の絞込みをともに高速化することができる．

2. 関 連 研 究

XML 文書に索引を付与する研究は近年盛んに行われている．ノードに対する索引 [8], [9]，経路に対する索引 [4]〜[7]，ノードに対して付与された識別子の系列を元にした索引 [10], [11] など，索引付けは，様々な手法で行われている．

ノードを単位として索引を構築する索引手法はノード同士の関係 (先祖/子孫関係，親子関係など) を高速に求めることができ，問合せ中の構造に対して柔軟に対応できるが，問合せに答えるためには問合せを分解し，結果を結合する必要がある．

XML 文書の経路に着目した索引は問合せ中の経路情報を効率的に処理することができる．単純に根ノードからの経路に対して索引付けするもの [4] だけでなく，問合せ中に頻繁に出現するパターンを考慮したもの [5] や，枝分かれ経路も索引付けの対象とするもの [7] などがある．

ViST [10] や PRIX [11] は XML 文書と問合せをともに系列で表現し，部分系列一致により問合せを扱う手法を提案している．索引を B+木にのせて，効率的に部分系列一致に答えることにより問合せを処理している．

以上のような索引手法は，いずれも主に XML の構造を効率的に扱うためのものであり，等号条件によるテキストの検索を考慮しているものもあるが，全文検索には対応していない．近年，XML 文書に対する構造索引と転置リストを統合して全文検索にも対応した索引を構築するアプローチとして [17] が提案された．本研究では，ノードに対して経路情報を含んだ識別子を付与し，逆経路情報を用いた search-key mapping を行うことで XML 文書の構造をより効率的に扱うことができ，索引構造として B+木を利用することでディスクへのアクセスを軽減する．またテキストの保持方法を工夫することでフレーズ検索も考慮している点でこの研究と異なる．

3. ノードへの識別子付け

本節では，我々の索引で用いるノードへの識別子付け手法を図 1 の XML 文書を例として説明を進める．図 1 はの XML 文書の木構造を表したものである。楕円はエレメントノードを，三角は属性ノードを，菱形はテキストノードを示しており，四角に入っている文字列がテキスト値である．

XML 問合せを高速に処理する上で，XML 木のノードに識別子を付与することは重要であり，多くの研究がされている [8], [12], [13]．最もよく使われるノード識別子としてノードの preorder と postorder の対がある．preorder と postorder の対は，XML 木を一意に復元することができる．しかし，このようなノード識別子は要素名や根からの経路に関する情報を

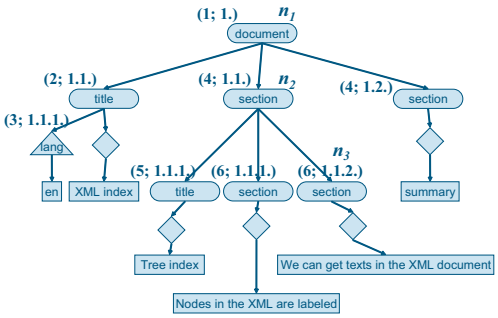

図 1　XML 文書の例

表 1　経路と Path Identifier の対応

経路	Path Identifier
/document	1
/document/title	2
/document/title/@lang	3
/document/section	4
/document/section/title	5
/document/section/section	6

含んでいない．我々は，XML 問合せの経路式に合致するノードを高速に取得するためには，ノード識別子から要素名や根からの経路に関する情報を容易に得られることが重要であると考える．そこで，我々は，ノード識別子に根ノードからの経路式を一意に識別する *Path Identifier* を含むことにした．表 1 は図 1 の XML 文書中のそれぞれの経路に対して付与された Path Identifier の一例である．しかし, Path Identifier のみでは XML 文書内のノードを一意に識別できない．たとえば，図 1 の /document/section に合致する二つのノードは同じ Path Identifier を付与される．

そこで，我々は兄弟順序情報を保持する *Sibling Dewey Order* を導入する．Sibling Dewey Order は着目したノードの根ノードからの経路上にあるそれぞれのノードに対し，同名のタグ名を持つノード間での兄弟順序番号を得て，その兄弟順序番号を連結することで得られる．

これら Path Identifier と Sibling Dewey Order の二つを組み合わせたものを *PSP* (*Path Sibling Pair*) と呼ぶこととする．PSP をノードに対する識別子とすることでノードは一意に識別できる．たとえば，表 1 のような Path Identifier が与えられたとする．図 1 において各ノードに付加されている識別子 (x;y) は PSP を表し，x が Path Identifier，y が Sibling Dewey Order を表す．たとえば，図 1 の n_3 のノードの根からの経路式は /document/section/section であり，表 1 から，Path Identifier は 6 となる．また，n_3 のノードは同じ要素名を持つ兄弟の中では順序が 2 番である．したがって，親である n_2 のノードの Sibling Dewey Order である 1.1. に 2. を連結した，1.1.2. が n_3 のノードの Sibling Dewey Order となる．

さらに，この Path Identifier と Sibling Dewey Order の組合せを用いることで，二つのノード間の親子関係，先祖/子孫関係を簡単に判定できる．Path Identifier 同士を比較することで，対応する経路の包含関係からその二つのノードの経路関係を得ることができ，Sibling Dewey Order の部分系列一致を判定することでインスタンスレベルで親子関係，先祖/子孫関係を得ることができる．PSP を用いると，ノード間の親子関係，先祖/子孫関係は以下のような手順で求められる．たとえば，図 1 において，“6;1.1.2.” という PSP を持ったノード n_3 と “4;1.1.” という PSP を持ったノード n_2 の親子関係を判定したい場合，まず，“6;1.1.2.” の “6” という Path Identifier から，このノードが “/document/section/section” の経路上にあることが分かる．そして “4;1.1.” の “4” という Path Identifier から，このノードが “/document/section” の経路上にあることが分かる．これらの経路情報からこの二つのノードは経路的に親子関係にあることが分かる．そして “/document/section” までにあたる二つ目までの Sibling Dewey Order を照らし合わせる．この場合どちらとも “1.1.” で等しいのでこれら二つのノードはインスタンスレベルで親子関係にあることが分かる．

このように Path Identifier と Sibling Dewey Order の組合せである PSP を用いることで，問合せに柔軟に対応し，インスタンスレベルでのノードの識別を行うことができる．

また，Sibling Dewey Order は兄弟番号を指定する問合せにおいてそのままの形で利用できる．例えば，“/document[1]/section[1]/section[2]” といった兄弟番号を指定した問合せにおいて，“/document/section/section” の経路に相当する Path Identifier と “1.1.2.” という Sibling Dewey Order から求めるノードを簡単に一意に特定できる．

Path Identifier としては SPIDER [18] や P-labeling [19] 等のような経路識別子も用いることができる．これらの手法を用いることで，より効率的に経路同士の関係を取得することが考えられる．本研究はディスクの入出力が問合せ処理コストの大部分を占めるという考えの下に行っており，表 1 のような表をメモリ上に保持することができれば，Path Identifier に対応する経路情報を取得し，経路同士の関係を取得するコストは Path Identifier を付与する手法に依存しないと考える．一般的に経路と Path Identifier の対応表は XML 文書に比べて十分に小さく，メモリ上に保持することができる．

4. 索引の構築

4.1 索引のエントリ

我々は COB-tree と STB-tree という B+木上の二種類の索引を提案する．COB-tree は，テキストとそのテキストが現れるノードの PSP を探索キーとする．3. 節でも述べたように，PSP は，ノードの根からの経路情報を持っている．COB-tree は，XML 文書中のテキストと構造の両方に着目した問合せに答えるための索引として用いられる．

また，STB-tree は，すべてのエレメントおよび属性ノードの PSP を探索キーとし，XML 文書の構造のみに着目した問合せに答えるための索引として用いられる．

全文検索に対応し，かつフレーズ情報を保持するために COB-tree のエントリ中の探索キーに含まれるテキストとしては，もとのノードに含まれていたテキストの接尾語集合を用いることにする．接尾語集合とは，たとえば，テキスト「Nodes in the

XML are labeled」の，

```
Nodes in the XML are labeled
in the XML are labeled
the XML are labeled
XML are labeled
are labeled
labeled
```

のようなテキスト集合である．このようなテキスト集合のそれぞれに対して，このテキストを含んでいたノードの PSP を組み合わせて COB-tree のエントリとすることで「Nodes in the XML are labeled」中の任意の単語からはじまるフレーズを含んでいるノードを検索することができる．

しかし，このようなテキストのフレーズを全てを索引中に保持することは索引容量の増加を招く．特に長いテキストを多く含む XML 文書に関してその問題は顕著である．そこで，テキストの保持に関して，他のテキストの出現と十分に区別できるだけのフレーズのみを保持することにする．たとえば，「the XML are labeled」という接尾語に関しては，図 1 を見ると,「the XML」で始まるフレーズとして「We can get texts in the XML document」というテキストノード中に,「the XML document」という出現があるが，この出現と区別するためには「the XML are」という部分までで十分である．このようにすることで，検索の際に，索引に保持されているテキストよりも長いフレーズで検索したいときでもそのフレーズが出現する可能性のあるノードを高い精度で絞り込むことができる．

しかし，このようにしても保持するフレーズが長くなる場合があるため，実際には，索引容量の増加を抑えるために，索引に含むテキストのフレーズ長はある閾値 L までに抑えることが有効であると思われる．閾値 L はどれくらいの長さのフレーズ検索を行う可能性があるかを考慮して設定される．特にフレーズ長を閾値で区切らない場合は閾値 L を無限大に設定する．

このようなテキストとそのテキストを含むノードの PSP との組合せが COB-tree のキーとなる．この組合せを探索キーとすることで，そのテキストを含むようなノードを COB-tree をたどって得ることができる．

また，B+木の葉ノードページに含まれるエントリのポインタとして，どのようなものを保持するかも重要な問題の一つである．ここでは，それぞれのキーに含まれる PSP に相当するノードのディスク上の位置をポインタとして保持する．このキーとポインタの組合せが索引のエントリとなる．たとえば，キーが “6;1.1.2.” であるようなエントリのポインタはノード n_3 のディスク上の位置である．

4.2 Search-Key Mapping

索引構造として使用する B+木のエントリとして 4.1 節で提案したものを用い，索引を構築する．ただし，ここで注意が必要な点は，B+木中のキーの順序である．B+木では葉ノードページに含まれるキーの順序は非常に重要である．隣り合った葉ノードページを取得するコストは非常に小さく，離れた場所にある葉ノードページをそれぞれ取得するコストは高くなる．

ここで表 1 のように付与した Path Identifier を単純にその値の大小で並べた場合，その順序にはほとんど意味がない．これはこの例にかかわらず，一般的に経路に対して値を割り当てたときに，その大小に意味を持たせることは困難である．たとえば “//title” といった問合せを考えたとき，この経路に適合する Path Identifier は B+木上で並んでいることが望ましいが，単純に Path Identifier の値の大小でキーを並べると，一般的に “//title” に適合する Path Identifier 値は B+木上で離れた場所に分布してしまう．

そこで本研究では *search-key mapping* を提案する．search-key mapping ではキーの順序はそのキー自身ではなく，そのキーに関連付けられた情報 (Mapping Information) によって決定される．ここでは，キーのうち Path Identifier の順序付けを Path Identifier の値ではなく，Path Identifier に対応する経路情報に基づいて行うことを提案する．Path Identifier と経路の対応として表 1 を用いると，“//title” のように “//” を含んだ問合せを考えたとき，“//title” に適合する Path Identifier 値である 2 と 5 は索引中で隣あっていることが望ましい．このような状況を考えたとき，B+木中で Path Identifier は対応する経路の逆順に基づいて順序付けを行うことが適切であると考える．つまり，この例においては，1 に対して “document\”，2 に対して “title\document\”，3 に対して “@lang\title\document\”，4 に対して “section\document\”，5 に対して “title\section\document\”，6 に対して “section\section\document\” という Mapping Information を用意し，この逆経路情報の辞書順に基づいて対応する Path Identifier の順序を決定する．この例では Mapping Information を用いて Path Identifier に順序付けすると 3 <1 <4 <6 <2 <5 である．

一般的に Mapping Information の大きさは XML 文書に比べて非常に小さく，メモリ上に保持することができる．従って，Mapping Information を用いた順序付けは高速に行うことができる．これにより同じタグ名を持つノードに関するキーは索引中の一箇所で並び，XPath における “//” を含むような問合せを高速に処理できる．

また，Sibling Dewey Order は根ノードから順番にその兄弟順序の大小を比較することで順序付けするものとし，STB-tree ではキーは Mapping Information を用いた Path Identifier の順序でまず順序付けし，Path Identifier が等しい場合は Sibling Dewey Order で順序付けする．

COB-tree の B+木中でのキーはテキストの大小でソートされ，テキストが等しいキーに関しては STB-tree と同様の手法でキーを順序付けする．

このような search-key mapping の技法を用いることで，索引中で，同様のテキストを持っているノードに関するエントリ，さらに，同様の構造を持っているノードに関するエントリを近い位置に保持することができる．

4.3 Prefix-Diff COB-tree

B+木においてキーの長さは短い方が望ましい．COB-tree においてキーにはテキストが含まれており，含まれるテキストのフレーズ長によってはキーが長くなってしまう．

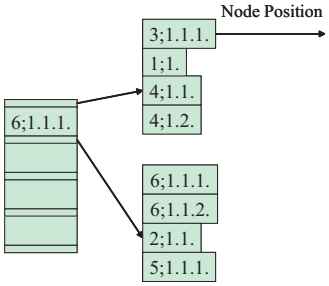

図 2　STB-tree の概念図

そこで，COB-tree の B+木の葉ノードページに含まれるキーのテキストは辞書順に並んでおり，同じフレーズを持つようなテキストを含むキーが索引中で並んでいることを利用して，テキストは一つ小さい隣のキーに含まれるテキストとの共通する文字数と，共通する文字列との差分のテキストのみを保持することにする．これにより，もとのテキストを復元することができ，キーの長さを短くできる．葉ノードページに含まれるキーのうち，一番小さいものはそのまま全てのテキストを保持しなければならないが，それ以外のキーに含まれるテキストは差分情報のみを保持すれば良い．

B+木ではエントリはノードページ単位でディスクから取得されるため，葉ノードページが取得されると，その中に含まれるキーのテキストを復元し，検索したい文字列を探すことになる．また，葉ノードページ以外のノードページにおけるキーは Prefix B-trees [20] にならい，どの子ノードページをたどるかを決定するために必要かつ十分な情報のみを保持する．

テキストの保持に関してこのような工夫を施した COB-tree を Prefix-Diff COB-tree と呼ぶこととする．工夫を施さない COB-tree を Prefix-Diff COB-tree と特に区別したい場合は Normal COB-tree と呼ぶ．

4.4　構築される索引の例

図 1 の XML 文書に対して構築された STB-tree の概念図を図 2 に示し，Normal COB-tree の概念図を図 3 に，Prefix-Diff COB-tree の概念図を図 4 に示す．ここではテキストと PSP を “,” で区切り，さらに Path Identifier と Sibling Dewey Order を “;” で区切って表現している．また，COB-tree に含まれるテキストのフレーズ長の閾値 L を 3 とした．これらの図では 1 ページに最大 4 つのエントリが入るものとして索引が構築されているが，実際は 100 近くの数のエントリが入り，B+木の高さは低く抑えられる．search-key mapping を用いることでもともと意味のなかった Path Identifier の順序を非常に有用なものにすることができる．図 4 におけるキーは，たとえば “[8]document,10;1.1.2.” であればそのテキスト部分は 8 文字目まで一つ小さい隣のキーのテキストと同じであり，それ以降の文字列が “document” であることを意味する．

5.　問合せ処理

XML を検索する際には XPath [2] を用いるのが一般的である．XPath による検索要求があると，B+木索引をたどって索引中のエントリのうち検索要求に適合するノードに関するエントリを取得して問合せに答える．

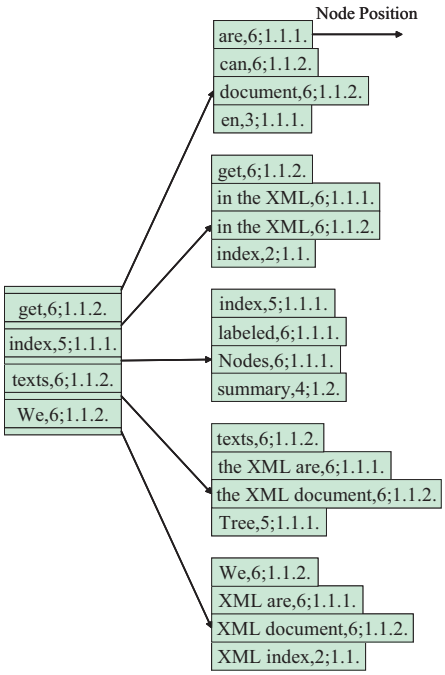

図 3　Normal COB-tree の概念図

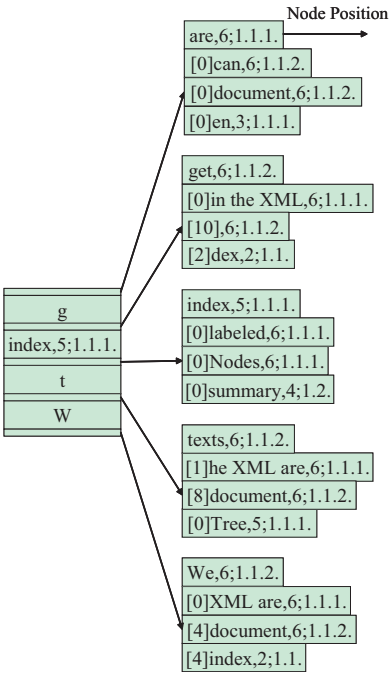

図 4　Prefix-Diff COB-tree の概念図

本研究で提案する索引は，主に child や descendant 軸を用いた検索を対象としている．これは，一般的に XML 文書に対してなされる検索のほとんどがこれらの軸を対象としたものであるからである．following-sibling と preceding-sibling に関してはコンテキストノードと取得したいノードのタグ名が同一の場合においては Sibling Dewey Order を用いることで同じタグ名を持つ兄弟ノード間での順序を求めることができ，問合せを処理できるが，その他の場合においては following や following-sibling 等の軸に関してノード間の順序を求めること

はできず，問合せを処理することができない．

単純に経路のみを問題とする単純経路問合せ，求めたいノードに対し contains() 関数を用いた述語を含んだ全文検索問合せ，問合せ経路中の一つ以上のノードに対して述語を含んだ複合問合せの三種類の問合せに対して，それぞれ問合せ処理手法を説明する．

これらの手法を組み合わせることで複雑な問合せにも答えていくことができる．[14] にあるような全文検索問合せに対してもその問合せの基本要素となっている contains() 関数を用いた検索や構造検索を高速に処理することができ，問合せ処理の高速化が狙えると考えている．

本節で説明のために用いる例は図 1 の XML 文書に対する XPath 問合せの処理例である．ノードに付与される PSP に含まれる Path Identifier と経路の対応は表 1 を用いる．

5.1 単純経路問合せ

単純経路は s_i を “/” または “//” であるとし，l_i をタグ名として，“$s_1 l_1 s_2 l_2 \ldots s_k l_k$” のような形をしていると考える．このような問合せは，まずこの単純経路に対応する Path Identifier を得る．いずれかの s_i が “//” である場合は複数の Path Identifier が得られることになる．そしてこれらの Path Identifier の集合から 4.2 節で定義した Path Identifier の大小に基づいて最も小さな Path Identifier と最も大きな Path Identifier を抜き出して，それら二つの Path Identifier で STB-tree 中で検索開始地点と検索終了地点を決定する．そして，その間に含まれる全てのエントリに関して問合せに適合するものを抜き出す．単純経路問合せの処理例を以下に挙げる．

- `//title`

Path Identifier を経路に変換したとき，最後が “title” であるような Path Identifier は 2 と 5 である．索引をたどり，Path Identifier が 2 から 5 の間であるようなエントリを取り出すと “2;1.1.”，“5;1.1.1.” というキーを持つ二つのエントリが得られる．これらのキーは実際に問合せに沿ったものであるため，これらのキーに関連付けられている “title” ノードの位置を得ることで検索を行うことができる．search-key mapping を用いることで索引中で “2;1.1.”，“5;1.1.1.” が隣り合っているため，効率的に索引をたどることが可能であると言える．

5.2 全文検索問合せ

全文検索問合せは “$s_1 l_1 s_2 l_2 \ldots s_k l_k[contains(.,'text')]$” のような形をしていると考える．問合せにテキストが含まれている場合は，そのテキストを用いて COB-tree をたどり，索引中の検索開始地点と検索終了地点を決定する．そして，その間に含まれる全てのエントリに関して，問合せ中の構造に関する部分も考慮して問合せに適合するものを抜き出す．全文検索問合せの処理例を以下に挙げる．

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

まず “Tree” というテキストを用いて，COB-tree 中で検索開始地点と検索終了地点を決定する．そしてその間に含まれるエントリを取り出すと “Tree,5;1.1.1.” というキーを持つエントリが得られる．このキーは “//section/title” という構造に適合する．よって，このキーに関連付けられている “title” ノードの位置を得ることで検索を行うことができる．

5.3 複合問合せ

複合問合せは “$s_1 l_1[Pred_1] s_2 l_2[Pred_2] \ldots s_k l_k[Pred_k]$” のような形をしていると考える．それぞれの $Pred_i$ は単純経路問合せもしくは全文検索問合せの形をしている．複合問合せに対して基本的には二つのアプローチが考えられる．一つは，複合問合せを複数の単純経路問合せもしくは全文検索問合せに分解してそれぞれを索引を用いて処理し，結果を結合することで問合せに答えるアプローチである．このアプローチを Join Approach と呼ぶこととする．この場合は，まず，与えられた複合問合せを “$s_1 l_1 Pred_1$”，“$s_1 l_1 s_2 l_2 Pred_2$”，...，“$s_1 l_1 s_2 l_2 \ldots s_k l_k Pred_k$” のような k 個の問合せに分解する．そして，分解されたそれぞれの問合せの結果を結合することにより，もともとの問合せに答えることができる．

もう一つは，複合問合せを複数の単純経路問合せもしくは全文検索問合せに分解し，それらの分解された問合せのうち一つを選んで，索引を用いて処理した後に，その問合せ結果を利用して，順番に次の分解された問合せに答えていくことでもとの問合せに答える手法である．このアプローチを Cascade Approach と呼ぶこととする．複合問合せの処理例を以下に挙げる．

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

全文検索問合せの例と同様にまず COB-tree を用いて “Tree,5;1.1.1.” というキーを持つエントリを取得する．しかし，この場合は取得したいノードが “title” ノードではなく “section” ノードであるため，“5;1.1.1.” に対応するノードの親ノードである “4;1.1.” という PSP を持つノードの位置を STB-tree を用いて取得する必要がある．Join Approach を用いる場合は，“//section” に適合するエントリを STB-tree をたどって取得し，“4;1.1.”，“4;1.2.”，“6;1.1.1.”，“6;1.1.2.” というキーを持つ四つのエントリが取得された中で，“Tree,5;1.1.1.” の経路上にある “section” ノードに相当するエントリのみを得る．Cascade Approach を用いる場合は，“Tree,5;1.1.1.” から求めたい “section” ノードが “4;1.1.” であることを計算し，STB-tree をたどって問合せに答える．

Cascade Approach は分解した問合せの一つに適合するエントリが非常に少ない場合等に，その絞込み情報を有効に使うことができるため高速である．しかし，分解した問合せの中に適合するエントリが少ないものがない場合は，Cascade Approach を用いると一つ目の問合せに適合するエントリの数だけ次の問合せに関して索引をたどらなければならず，非効率的である．そのような場合は Join Approach を用いた方が高速となる．

三つ以上に分解される複合問合せに関しては Join Approach と Cascade Approach を適切に組み合わせて利用することでより効率的に問合せを処理できると考えられる．

6. 実　　験

本研究で提案する索引を実装し，その有効性を検証した．B+木索引を実装するにあたっては，GiST (Generalized Search Tree) [21] を利用した．

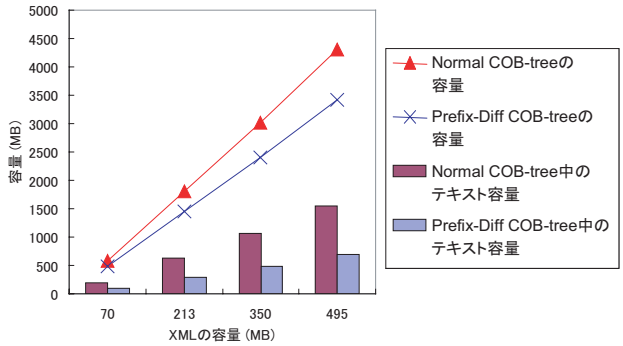

図 5　COB-tree の容量

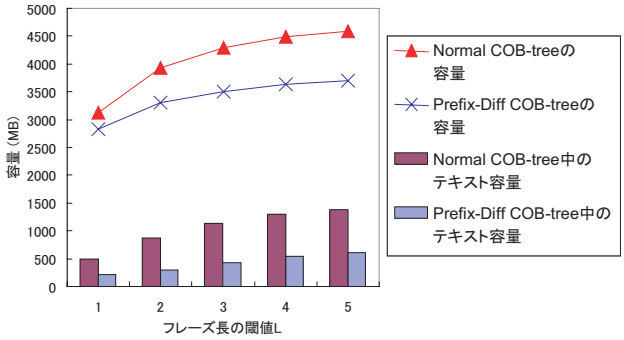

図 6　L を変化させたときの COB-tree の容量

実験には INEX Project [22] によって提供される XML 文書集合を使用した．この XML 文書集合は IEEE によって出版される論文を XML 文書化したものである．実験は，全体の XML 文書集合から XML 文書を抜き出していくつかの大きさの XML 文書集合を生成して行った．1995 年までに発行された論文の XML 文書集合 (約 70 MB)，1997 年までの集合 (約 213 MB)，1999 年までの集合 (約 350 MB)，XML 文書集合全体 (約 495 MB) の 4 種類の集合を作成した．

実験環境は CPU: Pentium M 1.40 GHz, メモリ: 512 MB, OS: Windows XP SP1 である．ディスクは 5400 rpm のものを使用した．また，実行環境として Visual Studio .NET を利用した．

6.1　索 引 容 量

それぞれの XML 文書集合に対して STB-tree と Normal COB-tree，Prefix-Diff COB-tree を生成した．

STB-tree の容量は，ほぼ XML 文書の容量に等しかった．XML 文書の容量が増加すると共にノード数が増加し，STB-tree のエントリ数が増えるため，XML 文書の容量に比例する形で容量が増加した．

閾値 L を無限大としたときの COB-tree の容量と，COB-tree の容量のうち葉ノードのテキストの容量のみを取り出したものを図 5 に示す．また，図 6 に閾値 L を変化させたときの COB-tree の容量と葉ノードのテキスト容量を示す．図 6 は 495 MB の文書集合を用いた場合の容量である．それぞれ Normal COB-tree と Prefix-Diff COB-tree の容量を示す．

Prefix-Diff COB-tree にすることで Normal COB-tree と比較して索引中のテキスト容量を半分以下にすることができている．索引容量全体としては，Prefix-Diff COB-tree は Normal COB-tree と比べて約 80 %の容量となっている．COB-tree のエントリを構成する要素としてはテキスト，PSP，ポインタの 3 つがあるが，Prefix-Diff COB-tree は，この中のテキストの容量のみを減少させる．PSP やポインタの容量減少に関しては今後の課題である．

図 6 を見ると，Prefix-Diff COB-tree にすることで L が大きくなっても索引容量の増加をある程度抑えることができている．どれくらいのフレーズ長で検索する可能性があるのかが事前に分からない場合，閾値 L はできるだけ大きく設定したい．Prefix-Diff COB-tree を用いることで索引容量を抑えつつも長いフレーズ情報を保持することができるようになる．

表 2　XPath 問合せ

	XPath
Q1	/books/journal/title
Q2	//sec/st
Q3	//article/fm/abs/p[contains(., 'software process')]
Q4	//article[contains(./fm/abs/p, 'software process')]
Q5	//sec[contains(./st, 'animation')]
Q6	/books/journal/article/bdy/sec [contains(.//*, 'stemming')]
Q7	/books/journal/article/bdy/sec [contains(.//*, 'information integration')]

6.2　問合せ処理時間

表 2 の XPath 問合せを用いて問合せ処理時間を調べた．問合せ処理時間を図 7 に示す．Prefix-Diff COB-tree を使用し，複合問合せ (Q4〜Q7) に関しては Join Approach を用いた処理時間を示す．それぞれの問合せは 10 回行い，その平均時間を問合せ処理時間として得た．なお，Normal COB-tree を用いた場合の問合せ処理時間は Prefix-Diff COB-tree を用いた場合のものとほとんど違いがなかった．Prefix-Diff COB-tree では葉ノードページを取得した際にキー中のテキストを復元するコストが余計にかかるが，ディスク入出力にかかるコストに比べて非常に小さいため，問合せ処理時間に違いが見られなかったものと思われる．

実験に使用した全ての問合せにおいて，XML 文書集合の容量が 495 MB になっても 500 ms 以内という非常に高速な処理を実現できた．Q1，Q3 は，問合せに適合するノードの数が比較的少なく，問合せ処理にかかる時間のほとんどが B+木索引を根ノードページから葉ノードページへたどる時間であり，XML 文書の容量が大きくなっても非常に高速に問合せを処理できている．Q2 の場合は問合せに適合するノード数が多く，B+木索引を根ノードページから葉ノードページへたどる時間よりも葉ノードページのリンクをたどっていく時間が主な処理時間であると思われる．このような場合は問合せに適合するノードの数に比例するような形で処理時間が長くなる．Q2 のように "//" を含む問合せは *search-key mapping* によって効率的に処理されている．"//" を含む問合せに適合する可能性のあるエントリは STB-tree 中で一箇所に並んでおり，B+木索

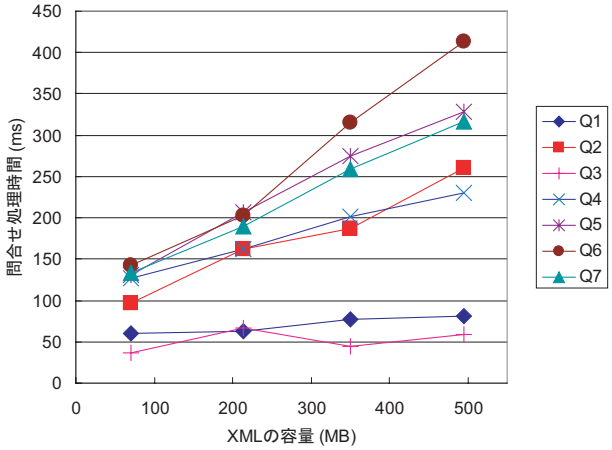

図 7　問合せ処理時間

引中を根ノードページから葉ノードページへたどる操作は一度行うだけでよい．Q4〜Q7 は複合問合せであり，問合せを分解して，中間結果の結合操作を行うことにより処理される．本論文中では Join Approach での処理時間のみを示したが，Join Approach と Cascade Approach は問合せによって使い分けることが重要である．使い分けには XML 文書に対する統計情報の導入や，分解された問合せの一つでまず索引をたどり，得られたエントリの数でどちらの手法を使うかを決定することなどが考えられる．

7. おわりに

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．XML 文書中のテキストに特に着目し，XML 文書中の各ノードに付与した経路情報に基づく識別子とノードに含まれるテキストを組み合わせて COB-tree を生成した．また，STB-tree では逆経路情報に基づいて B+木索引上でエントリが並ぶ順序を工夫することで “//” を含むような問合せにも対処した．COB-tree は B^+-tree 中でのテキスト保持の工夫を施し，Prefix-Diff COB-tree とすることでテキスト情報を失うことなく索引容量を減少させた．また，それらの索引を使用して XPath を処理する手法について説明した．そして実験を行い，本研究で提案する索引の性能を示した．実験では INEX Project によって提供される XML 文書集合を用い，索引の容量と問合せ処理時間について調べた．

今後の課題として，PSP としてどのようなものを用いることがより適切かを検討することや，索引上のエントリの順序を決定する情報として逆経路情報以外のものを導入することなどがある．また，本索引を利用する場合の結合操作を必要とするような問合せに対する高速な処理アルゴリズムの検討，XML 文書や問合せの統計情報の導入，ここでは考えなかった XML 文書の更新への対処などが課題としてあげられる．

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] W3C. XQuery 1.0: An XML Query Language http://www.w3.org/TR/xquery/ 2004.

[4] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistrucutred databases. In *VLDB*, pages 436-445, August 1997.

[5] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *VLDB*, pages 341-350, September 2001.

[6] Chin-Wan Chung, Jun-Ki Min, and Shim Kyuseok. APEX: An Adaptive Path Index for XML data. In *ACM SIGMOD*, pages 121-132, June 2002.

[7] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *ACM SIGMOD*, pages 133-144, June 2002.

[8] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions, In *VLDB*, pages 361-370, September 2001.

[9] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi: XR-Tree: Indexing XML Data for Efficient Structural Joins. In *ICDE*, pages 253-264, March 2003.

[10] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *ACM SIGMOD*, pages 110-121, June 2003.

[11] Praveen R. Raw and Bongki Moon. PRIX: Indexing and querying XML using prüfer sequences. In *ICDE*, pages 288-300, March 2004.

[12] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and 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, pages 110-141, 2001.

[13] Xiaodong Wu, Mong Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *ICDE*, pages 66-78, March 2004.

[14] W3C. XQuery 1.0 and XPath 2.0 Full-Text Use Cases http://www.w3.org/TR/xmlquery-full-text-use-cases/. July 2004.

[15] W3C. XQuery and XPath Full-Text Requirements. http://www.w3.org/TR/xmlquery-full-textrequirements/. May 2003.

[16] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. In *Proceedings of the 13th international conference on World Wide Web*, pages 583-594, May 2004.

[17] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. In *ACM SIGMOD*, pages 779-790, June 2004.

[18] Dao Dinh Kha and Masatoshi Yoshikawa: XML Query Processing using a Schema-based Numbering Scheme. Second International XML Database Symposium, XSym 2004, pages 21-34, August 2004.

[19] Yi Chen, Susan B. Davidson, and Yifeng Zheng. BLAS: An Efficient XPath Processing System. In *ACM SIGMOD*, pages 47-58, June 2004.

[20] Rudolf Bayer and Karl Unterauer, Prefix B-trees. ACM Transactions on Database Systems (TODS), Volume 2 , Issue 1, pages 11-26, March 1977.

[21] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *VLDB*, pages 562-573, September 1995.

[22] INitiative for the Evaluation of XML Retrieval (INEX) http://inex.is.informatik.uni-duisburg.de:2004/