経歴・パターン法に基づく XML 文書の実装と評価

西野 裕臣† 渡部 優太郎† 都司 達夫† 樋口 健†

†福井大学工学研究科 〒910-8507 福井県福井市文京 3 丁目 9 番 1 号

E-mail: †{h_nisino, watabe, tsuji, higuchi}@pear.fuis.u-fukui.ac.jp

あらまし 本論文では大規模な XML 文書に対して，我々が提案している経歴・パターン法と呼ぶ多次元データセットのエンコード方式を利用した実装方式を提案する．この方式では XML 木の動的な構造更新に対して再ラベル付け等の再編成等を行う必要が無い．また，ラベルは経歴・パターン法を利用してコンパクトに表現されていることから低記憶コストな XMLDB を実現できる．本論文では，経歴・パターン法を利用した多次元データセットの実装方式の 1 つである HPRD について説明し，XML 木を木構造と経路式に分けてラベル付けを行うことで要素名付きの木構造を HPRD に格納する方法について述べる．通常，検索速度向上のために多くの XMLDB では，高い記憶コストの索引付けが行われるが，本方式では，コンパクト性と高速性を両立させながら，動的な XML 文書の実装を行う．構築したプロトタイプシステムについて，eXist-db および NeoCore の 2 種類の MLDB システムとの比較評価を行った結果について示す.

キーワード XML，XMLDB，経歴・パターン法,

Hiroomi NISHINO† Yutaro Watabe† Tatsuo TSUJI† and Ken HIGUCHI†

†Graduate School of Engineering, University of Fukui, 3-9-1, Bunkyo, Fukui, 910-8507 Japan

E-mail: †{h_nisino, watabe, tsuji, higuchi}@pear.fuis.u-fukui.ac.jp

1. はじめに

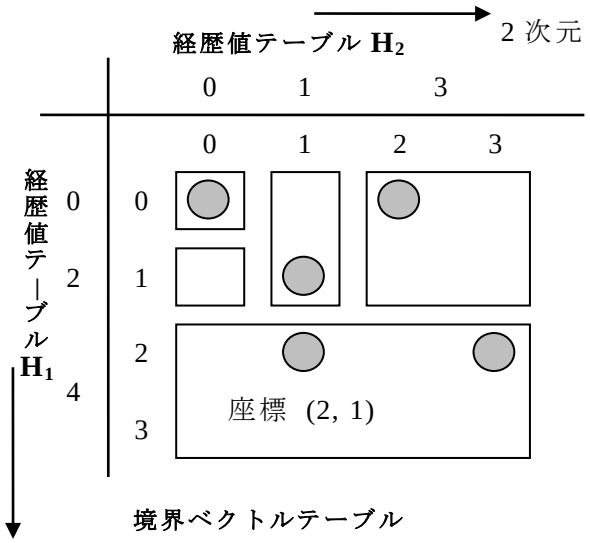
我々は，経歴・オフセット法と呼ぶ多次元データセットのエンコード方式とその実装方式を提案している[2]。この方式は動的に拡大する多次元データセットをコンパクトに表現し，高速な検索を可能にしている。[3]は経歴・オフセット法を利用した，XML 文書の実装方式を提案しているが，本研究では，より規模の大きいデータセットを扱うことができる経歴・パターン法を利用した XML 文書の実装方式を提案する。XML 文書を表現する XML 木を経歴・パターン法による多次元データセットの実装データ構造に埋め込むことにより，経歴・パターン法の利点を享受する。本方式では，XML 木に対して動的な構造変更が行われた場合にもノードの再ラベル付けの必要がなく，文書順を保持した更新を実現する．本文では経歴・パターン法を利用した多次元データセットの実装方式の一つである HPRD[2]について説明し， XML 木の圧縮格納方式とそのデータ構造の概要を述べる．さらに，このデータ構造に対して XPath[4]による検索の手順を説明した後，作成したプロトタイプシステムを XMLDB としてよく知られている eXist-db[6]および NeoCore[7] と性能比較評価した結果を示す.

2. 経歴・パターン法

経歴・パターン法は拡張可能配列[2]の概念を基にした多次元データセットのエンコード方式である．一般に多次元データセット M の各次元を多次元配列 A の次元に対応させ，各次元属性値を A の当該次元の添字に対応させれば，M のタプルは A の配列要素を表す添字座標で表現できる．M のタプルが動的に増加する場合には，A は拡張可能配列として，各次元サイズは動的に拡張できる必要がある．ところが，一般に M のタプルに対応しない配列要素が A に多く含まれ，A は疎配列となるという問題がある．

経歴・パターン法は，拡張可能配列の概念を取り入れつつ疎配列問題を解消する多次元データのエンコード方式であり，データセット中のタプルをコンパクトに表現し，また，高速に次元添字を取り出すことができる．従来の拡張可能配列のモデルとは異なり，経歴・パターン法では拡張する直前の配列と同形のサイズの部分配列を拡張次元方向に付加する(図 1)．拡張時に付加される各部分配列はそれが何番目に拡張・付加さ

れたものであるかを示す経歴値で識別される．この経歴値は各次元毎に用意される経歴値テーブルに格納される(図１)．部分配列中に属する要素は，その部分配列の経歴値と，配列空間中でのその要素の座標のエンコードの 2 値で表される．座標は各次元の添字サイズのビット長を記録した境界ベクトルと呼ぶベクトルを当該部分配列の経歴値により引当て，このビット長にしたがって，各次元添字を結合することにより得られる座標パターンというビット列にエンコードされる．この境界ベクトルおよび，拡張された次元は境界ベクトルテーブルと呼ぶ単一の配列に格納される．



境界ベクトルテーブル

経歴値	境界ベクトル
0	[0, 0]
1	[1, 0]
2	[1, 1]
3	[1, 2]
4	[2, 2]

座標 (2, 1) ⇨ 〈4, 1001〉

経歴値 ：4

座標パターン ：$1001_{(2)}$

図１．経歴・パターン法のエンコード例

二次元の座標空間における経歴・パターン法による座標値のエンコードの一例を示す．図１は境界ベクトルによって二次元の論理拡張可能配列を表現している．空間中には 5 つのタプルを表す配列要素があり，そのうち座標 (2, 1) の要素は図１に示すように経歴値が 4，座標パターンは $1001_{(2)}$にエンコードされる．このエンコードの手順は，まず座標 (2, 1) の要素がどの部分配列に属するかを求める．これには各次元の座標値を表すのに必要なビット数を求めるが，これは，それぞれ２ビット，１ビットである．経歴値テーブル H_1 および，H_2 を参照し，$H_1[2] = 4$, $H_2[1] = 1$ であり，$H_1[2] > H_2[1]$ であるから，(2, 1) の要素は経歴値 4 の部分配列に属することがわかる．次に境界ベクトルテーブルよりその経歴値 4 に対応する境界ベクトルを求め，これをもとに座標値の結合処理を行う．この例では経歴値 4 の境界ベクトルは [2, 2] であるから，1 次元目，2 次元目ともに 2 ビットずつ用いて座標値の結合を行う．また，座標値は左から 1 次元目，2 次元目として結合を行っている．1 次元目の座標 $10_{(2)}$ と 2 次元目の座標 $01_{(2)}$ をビットシフトとマスク処理によって結合して座標パターン $1001_{(2)}$ が生成される．

3. HPRD

図１に示したデータ構造をタプルのエンコード/デコードのための核として，関係テーブルにみるような任意のデータ型の属性からなる多次元データセットを効率良く実装することができる．すなわち，各次元について，その属性の値を 2 節で示した拡張可能配列の対応する次元の添字にマッピングするための CVT(key-subscript ConVersion Tree)と呼ぶ B+木，および，属性値テーブルと呼ぶ属性値の一次元配列を各次元ごとに用意する．エンコードされた＜経歴値，座標パターン＞の組は RDF と呼ぶ逐次ファイルに生成順に格納される．経歴・パターン法の実現に使用される 2 節で述べたデータ構造に加え，上記 3 種類のデータ構造による多次元データセットの実装方式およびこれらの実装データ構造を HPRD (History-Pattern implemantation for Real Data)と呼ぶ．図２は“会社”と“商品”の 2 つの属性からなる 2 次元の関係テーブルを実装する HPRD の例である．

HPRD において通常境界ベクトルはレコードの入力順に必要に応じて拡張される．図２の上部に示される関係テーブルのデータレコードを順に入力していった場合には図のような状態になる．この例では，経歴値および，境界ベクトルテーブルは図１の例と同一であるので，省略している．

CVT のキーとなるデータ型を変えることで HPRD は各次元に任意の型のデータを対応させることができる．また、経歴・パターン法を利用しているためカラムの追加が低コストで行えるという特徴がある．このテーブルに新しく「価格」という整数型カラムを追加したいとする．その場合，境界ベクトルと経歴値テーブルは以後 3 次元に拡張され，整数キーの CVT が追加される(図３)．すでにエンコード済の＜経歴値，座標パターン＞は再エンコードの必要はなく，また，境界ベク

トルテーブルを再構成する必要もない.

XML 文書の実装時には XML 木の構造を多次元空間にマッピングすることとなるが，低コストで論理空間の拡張を行えるという特徴は，その上で非常に大きな利点となる.

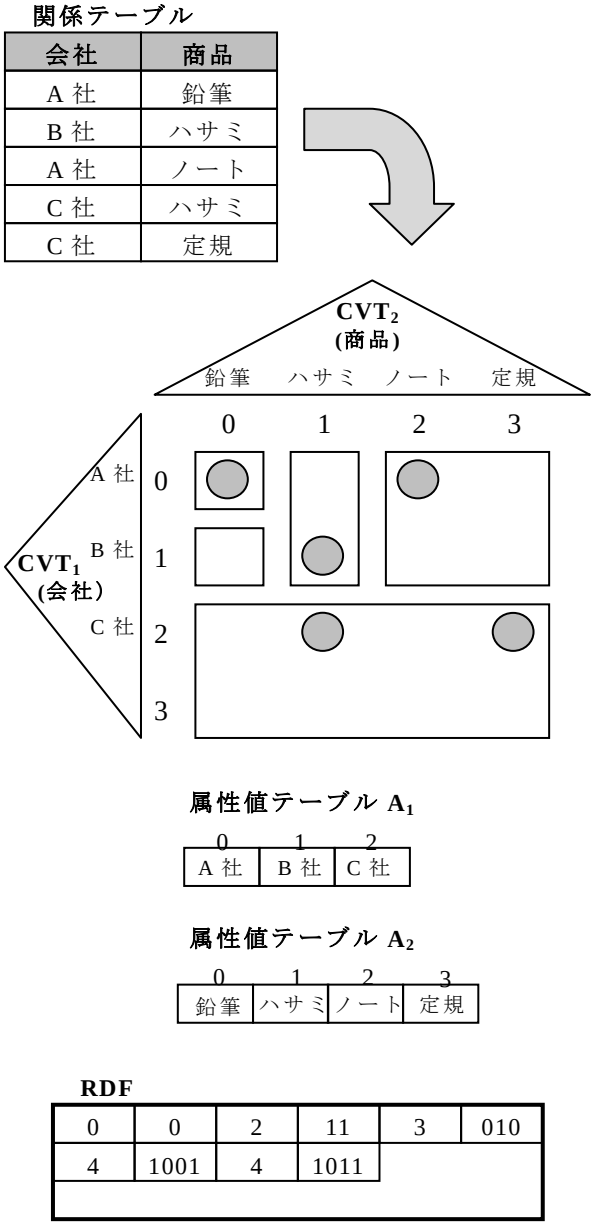


図２．2 次元の HPRD

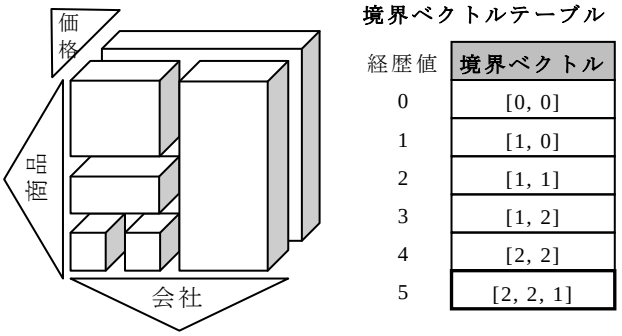


図３．図２を 3 次元に拡張した後の状態

4. XML 木の HPRD へ格納方法

この節では XML 木を HPRD に格納する手法を述べる. XML 文書の実装にあたって XML 木の構造を多次元空間にマッピングした後，HPRD により，XML 木を実装する．低コストで論理空間の次元拡張を行えるという特徴を 3 節で述べたが，その際に大きな利点となる.

XML 木を表現するためには木そのものの形と，各要素に対応付けられた要素名を表し，管理できなければならない．ここでは XML の各高さレベルを HPRD における拡張可能配列の論理空間の次元に対応させる.

XML 木の形(構造)を表すには相対ラベリング方式を用いる．この方法では XML 木の各ノード毎にその子ノードに対して 1 から順に添字を割り振ることで，その子ノード要素の高さレベル数分の多次元データとして子ノード要素の位置を表す．こうして XML 木から木構造のみ取り出して考え，整数値カラムのみからなる HPRD に各ノード要素の位置を＜経歴値，座標パターン＞にエンコードして格納する. この HPRD は XML 木全体の構造を表現することになる．このようにして XML 木から木構造を取り出したものを構造木と呼び，これを実装する HPRD を node HPRD という(図４). node HPRD では座標パターンから取り出せる添字がそのまま相対ラベルの各添字となるため，CVT と属性値テーブルを持つ必要がない．そのかわり，子要素の文書順の変化を保持するために OS(Ordered Sequence) テーブル[3]と呼ぶ各添字の文書順を記憶しておくテーブルを持つようにする.

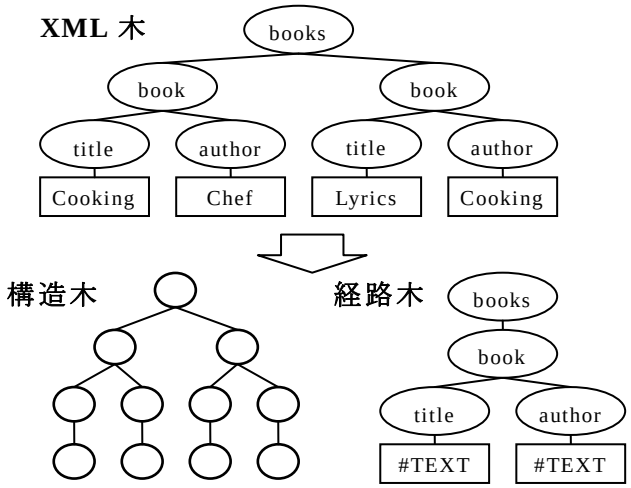
次に，XML 木の各ノードに至る経路式が同一のものをすべてまとめて単一の経路式とする経路木に XML 木を変換する．この経路木においては要素ノードの文書順などの構造情報は考慮する必要はない．経路木のすべてのノードには要素名が付けられているので，これを文字列を属性とする HPRD に格納する．この HPRD を path HPRD と呼ぶ.

path HPRD において XML におけるテキストノード

にあたる要素は，過剰な拡張を抑えるために単にテキストノードとして扱うことにし，要素が保持しているテキストデータ自体は位置情報と関連付けられた形でテキスト保存用ファイルに別途保持する．

node HPRD では各レベルにおける子要素集合の最大サイズが増加した場合に，path HPRD では各レベルにおいて新しい要素名が出現した場合に空間の拡張が起こることがある．また，どちらの HPRD でも XML 木の最大高さが増加すれば必ず次元の拡張が起こる．

XML 木の木構造と経路式の情報を分離して 2 つの HPRD に格納した．ここでは，node HPRD および path HPRD によってエンコードされる＜経歴値，座標パターン＞の組をそれぞれ node ID，path ID と呼ぶ．



要素値 (エンコードした ID を各々HPRD で保持)

/1, /1/1, /1/2, /1/1/1, /1/1/2, /1/2/1, ：	/books, /books/book, /books/book/title, ：

図 4．構造木と経路木

node ID を node HPRD を利用してデコードした場合は拡張可能配列の当該要素の位置情報としてその添字座標が得られる．また，path ID を path HPRD を利用してデコードした場合にはルート要素から当該要素までの経路式が得られる．したがって，node ID と path ID の組み合わせにより構造検索および経路式検索に必要な XML 木における当該ノード情報が得られる．

node ID は一意な位置情報なのに対して，path ID は一般に同一の経路式を持つノードは多数存在するため，node ID と path ID の対応付けは多対一である．

したがって，軸指定による構造検索と経路式検索を組み合わせた検索を可能とするために，すべてのノードについて node ID と path ID を相互に対応付ける必要がある．そのためには node HPRD と path HPRD の RDF はそれぞれ独立して確保せずに，node ID と path ID を相互に参照できる形で保持されなければならない．

このための ID 保持手法はいくつか考えられるが，本研究では node HPRD と path HPRD が共有の RDF を持ち，RDF は固定サイズのページで区切られた構造とする．そして各ページの先頭に path ID を 1 つ格納し，残りのページ領域をそれに対応する node ID の保持に使用する方式とする．

また，path ID をキーとして対応する各ページの先頭へのオフセットを保持しておく Path index と呼ぶ索引を構築し，path ID からページへのアクセスを高速化する(図 5)．この索引にはすべての path ID が含まれているため，RDF のページ先頭には path ID を格納する必要は無いように考えられる．しかし，要素名を含まず位置情報だけを指定するような検索パスも想定しうる．その際には RDF を先頭から逐次的に読み込む方が高速に行えるため，読み取ったページの先頭から経路式を特定できるようにするために今回の方式ではページの先頭には path ID を常に記録している．この path ID の記憶コストについては 8 節で示すが，RDF のサイズに対して path ID の占める領域はごくわずかである．

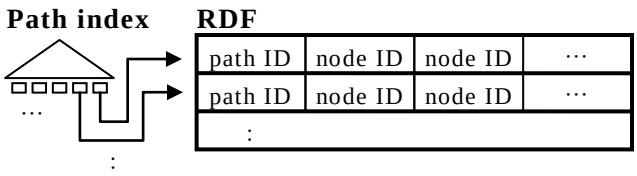


図 5．ページ区切りにした RDF と Path index

5. ID の問題点と改善

node ID および path ID を RDF に格納する際，それぞれの HPRD における境界ベクトルテーブルの拡張順によっては，座標パターンの下位のレベルのビットがすべて 0 で埋まり記憶コストが無駄になるという問題が発生する．この問題は，最大次元以外の次元で論理空間が拡張された場合に，その時の最大次元より低い次元かつ拡張された次元以上の要素で，拡張された経歴値以上の経歴値を持つ ID において常に発生する．

このようになる理由は，木構造を多次元配列にマッピングする上で，ある要素はそのレベル以下の座標値を必要としないためである．即ち，ID には経歴値と座標パターンに加えてその要素のレベル情報があれば座標パターンの無駄な部分を削減できる(図 6)．ただし，node ID と path ID にレベルの情報を加えるとすると，レベル値の格納には 1 バイトは必要になるため，座標パターンを削減できた部分が 8 ビット以下の要素につ

いては記憶コストが増大してしまう．大規模な XML 文書では一般にそのようになる要素が占める割合は非常に少ない．しかし，レベル値の記憶コスト自体はある程度大きい(XML 木中の全ノード数分のバイト数を必要とする)．

そこで，その中に格納される ID のレベルを統一して最大レベルの数だけ RDF を確保する(図 6)．これにより，RDF の中に格納する ID は従来通り経歴値と座標パターンのみでよくなりレベル値を格納する必要がなくなるため,記憶コストを大きく削減できる.また，検索対象のレベルを絞り込める場合にはそのレベルの RDF のみ検索すればよくなるので検索の高速化ができるといった利点がある．

各 RDF から取り出される ID のデコードには経歴値とレベルの 2 つの情報から，境界ベクトルを必要な次元数分だけ参照してデコードを行うようにする．

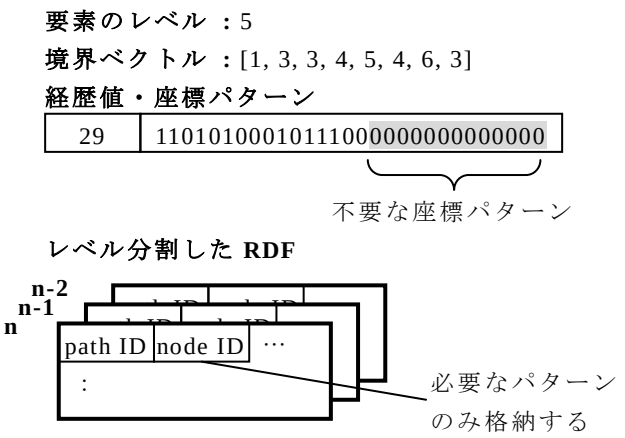


図 6 ． RDF の分割

6. 検索手順

RDF には node ID と path ID の対応付けをして ID が格納されている．ユーザーの検索要求を受けてここから条件と一致する両 ID を探すことになる．

例として以下のような XPath について検索手順を説明する(図 7).

/books//author[1]/text()

books 以下の author 要素の 1 番目であるノードに含まれる文書値を得る XPath である．

まず path HPRD の当該次元の CVT によって books と author の要素名を添字に変換する．テキストノードは特別に添字は 0 と決まっている．books は 1 次元目と確定しているが，author は 2 次元目以降，複数の次元に存在する可能性がある．したがって，この時点で条件に一致する path ID は複数存在する可能性がある．

2 次元目以降の CVT を検索し，author が見つかった場合に，その配下のテキストノードを検索するためそれより 1 つ大きい次元の RDF が検索対象となる．

検索対象になる次元の範囲が決定したら，次に当該次元の Path index を参照して検索条件に合致する経路式を持つ要素があるか調べる．もし Path index から条件に合う path ID が見つかった場合､データ部から RDF のページを参照するオフセットを得ることができる．RDF 内の各ページの node ID を順次探索し､node ID は構造検索が必要な場合に該当するレベルのみデコードして比較する．この例では author の要素名のレベルで 1 番目である要素を探すために node ID を検索する．これには，node ID をデコードして得られた座標値から OS テーブルを参照し，author の 1 番目のノードであることが判った場合，この node ID に関連付けられた文書値をテキスト保存用ファイルから読み取り，この node ID と path ID および文書値は検索結果となる．このような手順を検索対象の RDF ファイル全てについて順次行うことですべての検索結果を得る(図 7).

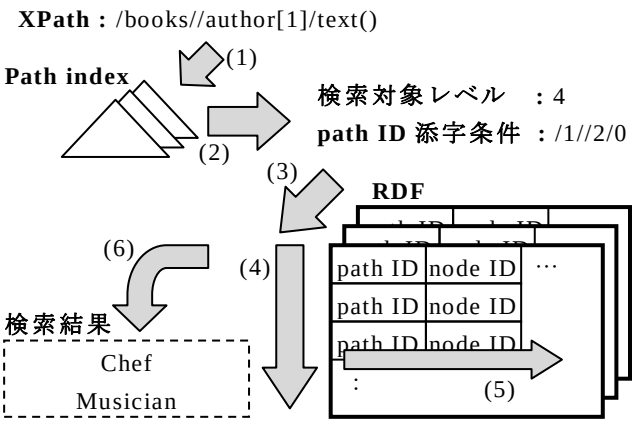


図 7 ． 検索の流れ

7. システムの構成

ここでシステムの全体的な流れと構成を示す．

これまでに述べた通り，本システムは node HPRD と path HPRD によって構成される．

本システムでは XML 文書の入力を受け付け，それを SAX[5]で XML 文書を解析して XML 木に変換しながら経歴・パターン法により順次エンコードし，その結果を node HPRD と path HPRD に格納する．

XML 木の構造情報は node HPRD によって管理され，要素名や属性名などは path HPRD の CVT および属性値テーブルに格納される．テキストノードの文書値は node ID に関連付けて PCD(pcdata)ファイルと呼ぶテキストデータ保存用ファイルに格納する．

この時点で XML 木はシステムによって表現されて

いるので，これに対してノードの検索を行うことができるようになる．

ユーザーからは XPath による検索クエリを受け付け，本システムの XPath 構文解析ルーチンによって検索条件を確認し，path HPRD と node HPRD に問い合わせを行って RDF から検索条件に一致する node ID と path ID および文書値を検索する．こうして得た node ID は node HPRD で，path ID は path HPRD でそれぞれデコードすることができ，これによって検索結果の文字列を取得できる(図８)．このデコード処理等の問い合わせは検索処理ルーチンが自動で行うため，ユーザーは文字列として検索結果を受け取るだけで良い．

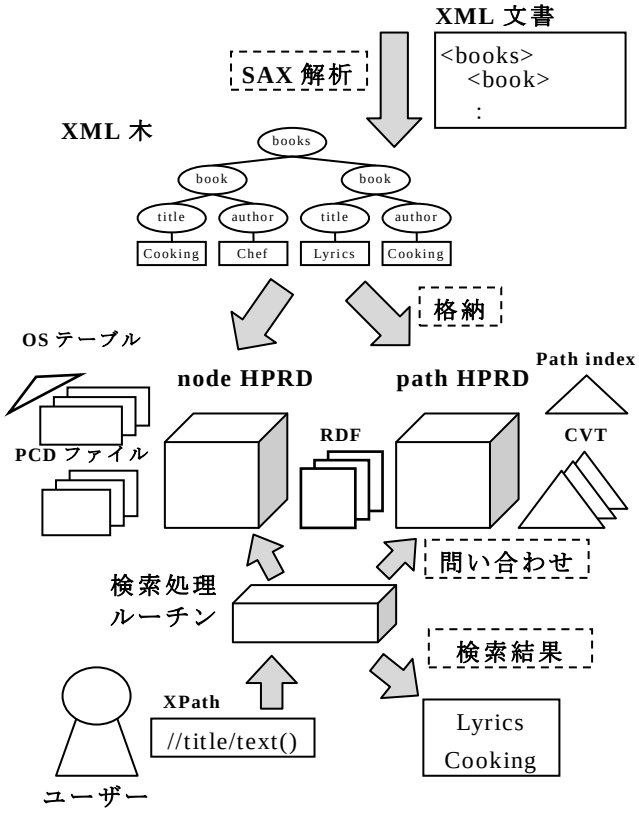


図８．システムの構成と流れ

8. 評価

本研究で提案する方式に従って実際に XML データベースシステムを構築し，XML 文書を格納して検索を行う実験を行った．実験に用いた計算機は以下の仕様である．

OS : CentOS Linux
CPU : AMD Athlon(tm) X2 4000+ 2.67[GHz]
RAM : 3.84[GB]
HDD : 160[GB], 5400[rpm]

本システムと比較する XMLDB システムの対象として，eXist 1.4.2[6] および NeoCore XMS 3.1[7] の 2 つの XMLDB を利用した．

格納する XML 文書は dblp.xml [8] および XMark[9] を利用して自動生成されたベンチマーク用 XML 文書の 2 種類を使用する．dblp.xml は DTD を含まないものである．xmark.xml は XMark の設定値をデフォルト値で生成した．以下に各 XML 文書情報を示す．

dblp.xml
　ファイルサイズ ：754,453,578[byte]
　XML 木深さ ：7
　総ノード数 ：39,051,918
　要素名数 ：47
xmark.xml
　ファイルサイズ ：116,517,691[byte]
　XML 木深さ ：13
　総ノード数 ：3,240,011
　要素名数 ：93

dblp.xml は XML 木の最大深さは浅めだがレベル 2,3 における兄弟要素の幅が非常に広く横に大きい構造になっている．これに対して xmark.xml は XML 木の最大深さは比較的大きくなっている．

また、今回の実験で比較に用いる XMLDB の特徴を簡単に以下に列挙しておく．

eXist
・オープンソースの XMLDB
・低記憶コストかつ高速な構築時間
NeoCore
・商用の高機能な XMLDB
・豊富なインデックス機能による高速検索
・様々な言語に対応した API

いずれの実験においても、データベースの各種設定値は，システムインストール時のデフォルトの設定による値とした．

8.1. 記憶コスト

まず 2 種類の XML 文書を両方とも格納した時点での構築サイズを比較する．

本システムでは node HPRD と path HPRD の各種構成要素のサイズ，実データを保持する RDF と PCD ファイルのサイズを測定する．eXist-db では eXist-db のインストールフォルダ内にあるデータベース情報を格納しているフォルダ内のファイルサイズを調べた．NeoCore についても，データベース情報に関連したファイルのサイズを調べるが，ファイル数が多いのでファイルの種別ごとにバイト数を合計する．表 1 に提案方式，表 2 と表 3 ではそれぞれ eXist-db と NeoCore で

の記憶コストの詳細な測定結果を示す.

さらに、表４では各 XML 文書ごとについての全体のサイズを示す.

表１．提案方式の記憶コスト測定結果

データ	サイズ[Byte]
全体	827,855,316
├node HPRD	11,656
│ ├OS テーブル	0
│ ├境界ベクトル	11,376
│ └経歴値テーブル	280
├path HPRD	8,564
│ ├境界ベクトル	5,344
│ ├経歴値テーブル	215
│ ├CVT	1,677
│ └属性値テーブル	1,328
├Path index	3,707
├RDF	321,060,864
│ ├node ID	316,818,652
│ ├path ID	154,103
│ └内部構成情報	4,088,109
└PCD ファイル	506,770,525

表２．eXist の記憶コスト測定結果

データ	サイズ[MB]
全体	1,879,361,057
├dom.dbx	1,655,312,384
├elements.dbx	224,018,432
├collections.dbx	12,288
├values.dbx	8,192
├ngram.dbx	8,192
└symbols.dbx	1,569

表３．NeoCore の記憶コスト測定結果

データ	サイズ[MB]
全体	8,102,476,386
├*.crf	82,845,696
├*.inx	404,807,680
├*.dup	704,274,432
├*.dct	451,952,640
├*.map	3,287,408,640
├*.adm	1,243,960
└*.lmf	3,169,943,338

表４．記憶コストの XML 文書別測定結果

データベース	サイズ[Byte]	
	dblp.xml	xmark.xml
提案方式	716,174,931	111,680,385
eXist	754,453,578	116,517,691
NeoCore	7,283,286,348	819,190,038

提案方式では RDF 内に記憶されている ID は pathID はページの先頭にのみ保持されることから node ID に比べてサイズは小さくなる．node ID の記憶コストが RDF の 99%を占める結果となった.

また，ここでは XML 文書を一括格納後，XML 木に編集を加えていない状態の記憶コストを測定しているため，OS テーブルのサイズは無い．OS テーブルはノード間に動的に要素追加操作を行う度に少しずつ増加するが，RDF の再編成処理を行うことでサイズを 0 に初期化することができる.

提案方式では，全体のサイズとしても各 XML ファイルのサイズよりコンパクトに実装できている．これは経歴・パターン法によって各要素の ID を非常に小さくエンコードできていることによる.

eXist と NeoCore では NGram インデックスによる全文検索機能に対応しているが，測定では全文検索のためのオプションは付けていない．本研究システムでも全文検索への対応は今回は研究対象外である.

8.2. 構築時間

各 XML 文書を入力としてデータベースを構築するまでにかかる時間を測定する．表５に各 XML 文書を入力として，データベースを構築するまでにかかる時間の測定結果を示す.

表５．構築時間結果

データベース	時間[sec]	
	dblp.xml	xmark.xml
提案方式	697	66
eXist	755	71
NeoCore	13,260	1,672

dblp.xml については, 提案方式では 697[sec], eXist-db では 755[sec]と約 12 分で構築できるのに対し NeoCore では 13,260[sec]と 3 時間以上かかる結果となった. ore では自動で複数の索引を作成しており，構築時に様々な索引ファイルに書き込みを行うためディスクアクセスが激しくなる．これによって構築時間は長くなったと考えられる.

8.3. 検索時間

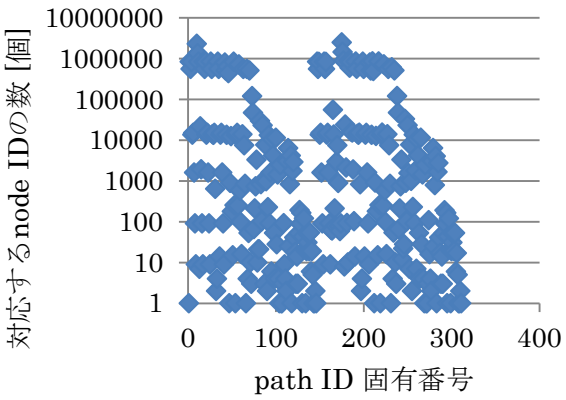
検索は XML 木の全レベルからランダムにノードを選択し，その経路式を取り出し XPath として検索要求を実行することを 500 回行い，平均，最長，最短時間を得ることにする．表６にその実験結果を示す.

表６．検索時間結果

データベース	最長[sec]	平均[sec]	最短[sec]
提案方式	0.678	0.0221	0.00132
eXist	3.504	1.072	0.0620
NeoCore	0.0162	0.00559	0.00131

提案方式では，述語が無く省略構文で表せるような場合の要素名のみで構成される検索クエリにおいて，非常に高速に処理することができる．

軸に following や preceding を指定する場合や，述語に position()を指定する場合などでは位置情報の照合が必要になるため node ID のデコードを必要とする．この比較回数が多くなる場合には検索時間が長くなる．部分木ごとの要素や形状に関する情報を持つような索引は保持していないため，経路式に適応するノードは常に RDF 内を全検索する必要がある．そのため，path ID に対応する node ID の数によって検索時間は大きく変動する．次に，今回の実験での XML 木における path ID とそれに対応する node ID の個数の分布を図 8 に示す．これは dblp.xml の XML 木におけるデータである．



対応 node ID 数範囲	path ID 数
1 ～ 10	73
11 ～ 100	68
101 ～ 1000	37
1001 ～ 10000	38
10001 ～ 10000	46
100001 ～ 1000000	45
1000001 ～ 10000000	4

図 9．path ID に対応する node ID の分布

path ID は全部で 311 件存在し，その内 264 件は対応する node ID 数が 10 万件以下である．これらの path ID に適合した検索クエリにおいては提案方式は高速に検索を行える．

しかし，50 万件以上対応 node ID が対応する path ID も 45 件存在する．これらの path ID に適合する検索では検索時間が 0.5 秒以上かかる傾向にあった．また，xmark.xml についても同様に対応する node ID が多いほど検索時間が増大した．表 7 にその一例として具体的なパス式とヒット数に対する検索時間を示す．

表 7．検索例とヒット件数及び検索時間

経路式	件数[件]	時間[sec]
/dblp/article/year	561215	0.0786
/dblp/inproceedings/cite	120822	0.0173
/dblp/incollection/crossref	13950	0.00202

これに対して eXist や NeoCore では多くのクエリで検索速度がある程度安定しており，検索条件によって速度が大きく変化することは少ない．特に、NeoCore は速度のぶれ幅も小さく，全体的に高速な検索を行えることが確認できた．

9. おわりに

本論文では，経歴パターン法を使用した XML 文書の実装方式を提案した．本実装方式では，XML 木に対して構造木および経路木をそれぞれ，node HPRD および path HPRD と呼ぶ 2 種類の実装データ構造を用いている．これにより，XML 木を効率よく表現し，XPath に対応した検索を行うことができる．本提案方式によって実装される XMLDB では nodeID の検索は逐次検索を基本とし，索引を最低限に抑制し低記憶コストでありながら，検索速度についても比較的高いパフォーマンスを示すことが判った．ただし前述した通り，path ID に対応する node ID の数による検索速度のぶれが大きく，兄弟要素が多いほど速度が低下する問題がある．

今後の課題としてはこうした速度低下を抑え，かつ述語などの指定で多く生じる構造検索に対応するために，node ID に関連した低記憶コストの索引を構成することなどが考えられる．

参　考　文　献

[1] Extensible Markup Language, http://www.w3.org/XML/

[2] 前田卓哉，水野広治，都司達夫，樋口健，多次元データのコンパクトな実装とその性能評価，Proc. of DEIM 2011, E1-5, 2011.

[3] Tsuji, T., Amaki K., Nishino H., Higuchi K., History-offset Implementation Scheme of XML Documents and Its Evaluations, Proc. of DASFAA 2013, April,2013 (to apper).

[4] XML Path Language, http://www.w3.org/TR/xpath/

[5] the Simple API for XML (SAX) http://www.saxproject.org/

[6] eXist-db Open Source Native XML Database http://exist.sourceforge.net

[7] XML データベース NeoCore http://www.neocore.jp/

[8] The DBLP Computer Science Bibliography http://www.informatik.uni-trier.de/~ley/db/

[9] XMark An XML Benchmark Project http://exist.sourceforge.net/