Vol.2013-DBS-158 No.5
2013/11/26

XML 文書の一実装方式とその性能評価

渡部優太郎[†1]　西野裕臣[†2]　都司達夫[†1]　樋口健[†1]

本論文では経歴・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 XML 文書の一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を行う必要が無い．ラベルは経歴・パターン法の特長により，コンパクトに表現されるため，低い記憶コストで XMLDB を実装できる．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，最小限の索引付けによりコンパクト性と高速性を両立させながら，動的な大規模 XML 文書を実装することを目的としている．本文では本方式について述べた後，作成したプロトタイプシステムを他のネイティブ XMLDB システムと比較して性能評価する．

An implementation Scheme of XML Documents and Its Evaluation

YUTARO WATABE[†1] HIROOMI NISHINO[†2] TATSUO TSUJI[†1] KEN HIGUCHI[†1]

This paper proposes a new implementation scheme of XML documents based on the encoding scheme for multidimensional datasets called history-pattern encoding scheme. Our implementation scheme requires no node relabeling against dynamic structural updates of XML trees. Since the node labels can be represented compactly due to the history-pattern encoding, we can implement an XMLDB of low storage cost. In many XMLDB systems, indexes of large storage cost are attached to promise the high speed retrieval. In contrast, our scheme is able to implement large scale XML documents achieving both low storage and retrieval costs. In this paper after describing implementation scheme, the performance of our proposed scheme is evaluated. In the evaluation, our system is compared with other native XMLDB systems.

1. はじめに

XML 文書を効率よく処理するためには，内在する XML 木の構造を反映した要素ノードのラベル付けや要素名パス式のエンコーディングが重要である．筆者等は，XML 木の動的な構造更新に対して再エンコードを行う必要のない XML 木ノードのラベル付け方式を提案している [7]．このラベル付け方式は動的に拡大・縮小する XML 木をコンパクトに表現し，高速な検索を可能にしているが，ラベル値がマシン語長を超える規模の XML 木は性能の大幅な劣化なしに扱えない．

本論文では，この問題に対して，筆者等が提案している経歴・パターン法と呼ぶ多次元データのエンコード方式 [13] を利用した XML 文書の実装方式を提案して，評価する．経歴・パターン法では，エンコード結果のラベル値がマシン語長を超えても，ほとんど性能劣化なしに，高速にエンコード/デコードが可能である．また，エンコード結果のラベル値の格納は逐次ファイルに効率よく格納できる．本方式では 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･パターン法の利点が活かされている。また，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，文書順が保持されている．

†1 福井大学工学研究科
Graduate School of Engineering, University of Fukui
†2 日本通信特機(株)
Nihontsushintokki Co.,Ltd.

多くの XMLDB システムでは検索の高速化を達成するために，インデクスを多用しており，データベースサイズが肥大する傾向にある．これに対して，本研究では経歴・パターン法を利用することにより，必要なインデクスを最小限（パスインデクスのみ）にとどめ，コンパクトなデータベースサイズを保証すると同時に検索性能を維持できるような XMLDB システムを提供することを目的としている．

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について説明し，XML 木の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[15]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性能評価し，ネイティブ XMLDB システムとして知られている eXist-db[9]および NeoCore[11]と比較する．

2. 経歴・パターン法

経歴・パターン法は拡張可能配列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 n 次元データセット M の各次元を n 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 n 次元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．

2.1 経歴パターン法における拡張可能配列のモデル

経歴・パターン法は，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 n 次元拡張可能配列のモデル [14]では新たな属性値の出現により，その次元方向に垂直な断面である n-1 次元の部分配列が付加され，拡張されるのに対して，経歴・パターン法では図 1 に示すように拡張可能配列 A の拡張は拡張する直前の配列と同形のサイズの n 次元部分配列が拡張次元方向に付加される．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”H_i (i=1, …, n)に格納される（図 1）．部分配列中の要素は，その部分配列の経歴値と，A の拡張可能配列空間中でのその要素の座標のエンコードの対で表される．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呼ぶ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 B に格納される．図 1 において，○は M のタプルに対応する拡張可能配列の要素を表すが，○中の数字はタプルが M に追加された順番を表す．

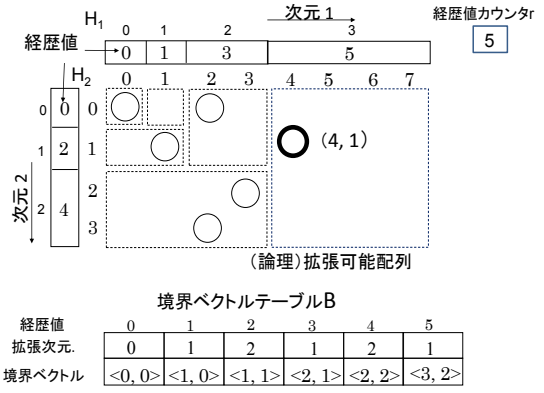

図 1　経歴・パターン法の説明図

2.2 経歴パターン法による座標値のエンコード例

2 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 2 次元の論理拡張可能配列を表現している．空間中には 6 つのタプルを表す配列要素がある．例えば，座標(4, 1)の要素は図 1 に示すように経歴値が 5，座標パターンが $100.01_{(2)}$にエンコードされる（“.”は各次元添字の境界位置を表す）．このエンコードの手順は，まず座標(4, 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 3 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，$H_1[3] = 5$，$H_2[1] = 1$ であり，$H_1[3] > H_2[1]$であるから，(4, 1)の要素は経歴値 5 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 5 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 5 の境界ベクトルは[3, 2]であるから，1 次元目の添字に 3 ビット，2 次元目の添字に 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 $100_{(2)}$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 $100.01_{(2)}$が生成される．以上より，座標(4, 1)のエンコードは<5,17>となる．

逆にエンコードされた＜経歴値，座標パターン＞の対をエンコード結果の例<5, 17>を用いて，座標にデコードする手順を説明する，経歴値が 5 であるから，境界ベクトルテーブルより，B[5] = <3, 2>を引き当てる．座標パターンが $17 = 10001_{(2)}$であり，これを 3 ビットと 2 ビットに分け座標(4, 1)を得る．

2.3 拡張可能配列の拡張と境界ベクトルテーブル

A は論理的な拡張可能配列であり，M 中のタプルに対応する配列要素のエンコード結果のみ記憶域を使用するため A の疎配列問題は回避される．M の各属性のカージナリティを A の当該次元の実サイズという．また，A のある次元 k (k=1, 2, …, n) の実サイズが s_k ($s_k \geq 1$）であるとき，$2^{\lceil \log_2 s_k \rceil}$を A の次元 k の論理サイズという．図 1 では，A の実サイズは[4,3]であり，論理サイズは[7, 3]である．M への新たなタプルの挿入により，次元 k の実サイズが論理サイズを超えたとき，次元 k 方向への A の拡張が起こる．例えば，図 1 において(1, 4)要素が追加された場合には，次元 2 の方向に拡張が起こる．このとき，次元 k の経歴値テーブルには最大経歴値より 1 大きい経歴値 h が書き込まれ，境界ベクトル B[h]の次元 k には，B[h-1]の次元 k の値より 1 大きい値が書き込まれる．座標パターンの各次元の添字を固定長とはせずに，境界ベクトルにより，各添字の境界を柔軟に設定している．境界ベクトルテーブルは，経歴・パターン法の中心的なデータ構造であり，従来の拡張可能配列のモデルには存在しない．

2.4 経歴・パターン法の長所と短所

経歴パターン法において，経歴値として，固定サイズの 1 バイトを割り当てても, 256 ビットまでの座標パターンを表現可能である．マシン語長を 64 ビットとすると，座標パターンの最大サイズはマシン語長を超える．「」の経歴オフセット法をはじめ他の同種研究の多くは配列または部分配列のアドレス関数を使用して，先頭要素からのオフセットを求めている．これには，乗徐算を必要とし，マシン語長を超える場合には多倍長演算ライブラリを使って，ソフトウエアで行う必要があり，大幅に性能が劣化する．経歴パターン法の場合にはアドレス関数は使用せず, エンコード/デコードはシフト命令とマスク命令のレジスタ命令の組み合わせのみで行われ，乗除算によらない．したがって，マシン語長を超える座標パターンに対しても最小限のオーバヘッドでエンコード/デコードが可能である．これは大規模な多次元データセットについても，実用速度の検索性能を保証し得る可能性があることを示している．

経歴・パターン法では座標のエンコード値である＜経歴

値,座標パターン＞を可変長サイズで効率よく表現できる．2.3 節で述べた拡張可能配列の拡張方式により，経歴値が常に座標パターンのビット長を表しているために，＜経歴値，座標パターン＞をレコードとして記憶する際に経歴値をレコードのヘッダとして用いればレコードサイズを知ることができる．したがって，タプル集合 M の各タプルを可変長レコードとして，効率よく記憶域に詰め合わせることができる．経歴・パターン法の以上のような長所は，本研究における XML 文書の実装方式の性能向上に寄与し得る．

経歴・オフセット法では拡張可能配列の論理サイズと実サイズは常に一致していているが,経歴･パターン法では，最悪，論理サイズは実サイズの 2 倍近くになり，拡張可能配列の論理空間を大きく消費してしまうことが欠点である．しかし，これにより座標パターンのビット長が無駄に大きくなることはない．

3. 経歴・パターン法の実装

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，2 節で示した，

① 経歴値テーブル H_i (i=1, …, n)

② 境界ベクトルテーブル B

に加えて，各次元について，その属性の値を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

③ “CVT_i (i=1, …, n)”（key-subscript ConVersion Tree）と呼ぶ B^+木，および,

④ “属性値テーブル”と呼ぶ属性値の一次元配列 A_i (i=1, …, n)

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

⑤ “ETF”（Encoded Tuple File）と呼ぶ逐次ファイル

に生成順に格納される．上記①～⑤の 5 種類のデータ構造による多次元データセットの実装データ構造を HPMD (History-Pattern implementation for Multidimensional Datasets) 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 HPMD の例である．

HPMD では，境界ベクトルテーブルはタプルの入力順に必要に応じて拡張される．図 2 はその上部に示される関係テーブルのタプルデータを順次入力した後の HPMD の状態である．この例では，経歴値テーブルおよび，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．

HPMD では新しい属性を低コストで追加することができる．例えば，図 2 のテーブルに新しく「価格」という属性を追加する場合，HPMD は 2 次元から 3 次元に次元拡張が行われる．このとき,「価格」次元用に経歴値テーブルが設けられ， 新しく CVT が追加される（図 3）．また，境界ベクトルは以後，3 次元ベクトルとなる．次元拡張以前にエンコード済の＜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．次元拡張以前の境界ベクトルの次元 3 の値は次元拡張後は 0 として扱うことで，次元 3 の添え字のビット列は空列となるからである．XML 木の構造は HPMD の多次元空間にマッピングされる．次節で述べるように上記の次元拡張に関する HPMD の利点により，XML 木の高さレベルの動的な拡大と縮小に低コストで対応することができる．

	商品	座標	<経歴値，座標パターン>
A 社	鉛筆	→ (0, 0)	→ <0, . > = <0, 0>
B 社	ハサミ	→ (1, 1)	→ <2, 1.1> = <2, 3>
C 社	鉛筆	→ (2, 0)	→ <3, 10.0> = <3, 4>
D 社	定規	→ (3, 2)	→ <4, 11.10> = <4, 13>
C 社	消ゴム	→ (2, 3)	→ <4, 10.11> = <4, 11>
E 社	ハサミ	→ (4, 1)	→ <5, 100.01> = < 5, 17>

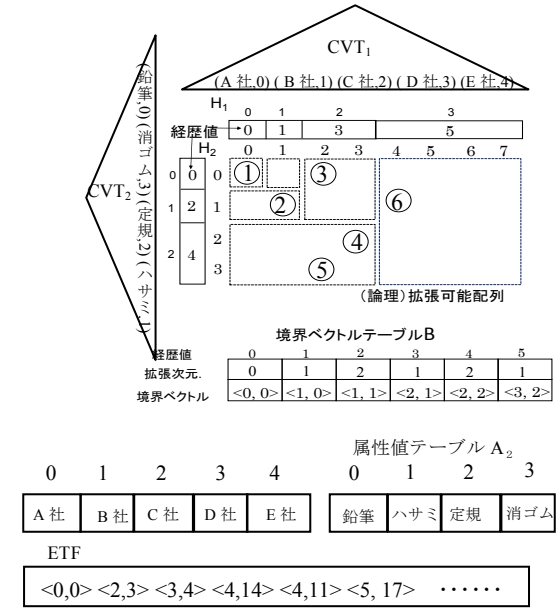

図 2 2 次元の HPMD

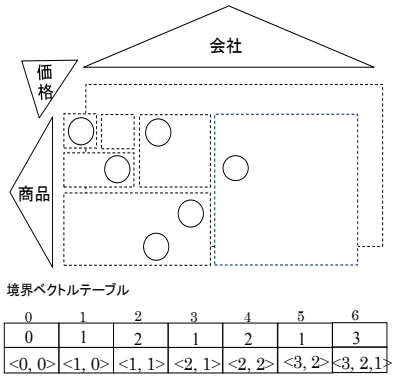

図 3 の HPMD を 3 次元に拡張した後の状態

4. HPMD による XML 木の実装

XML 木は拡張可能配列の多次元空間にマッピングした後，HPMD により，実装される．

4.1 構造木と経路木

XML 木を操作するためには木の構造そのものと, 各ノードに対応付けられた要素名やルートノードからの要素名経路式を管理できなければならない. ここでは図 4 に示すように, XML 木の構造と経路式の情報を分離した後, それぞれ, 構造木と経路木として, 個別の HPMD により管理する. いずれの HPMD においても, 木の各高さレベルを当該の拡張可能配列の次元に対応させる. また, 図 5 に示すように木の内部ノード p について, その高さレベル *l* に対応する次元 *l* の添字を 0 とする. また, p の子ノードの次元 *l* の添字を 1 から順に割り振ることで, 木の各子ノード要素の n 次元の位置座標を決定する. ここで, n は XML 木の最大高さレベルである. この位置座標はそれぞれの HPMD で＜経歴値, 座標パターン＞の組にエンコードされる.

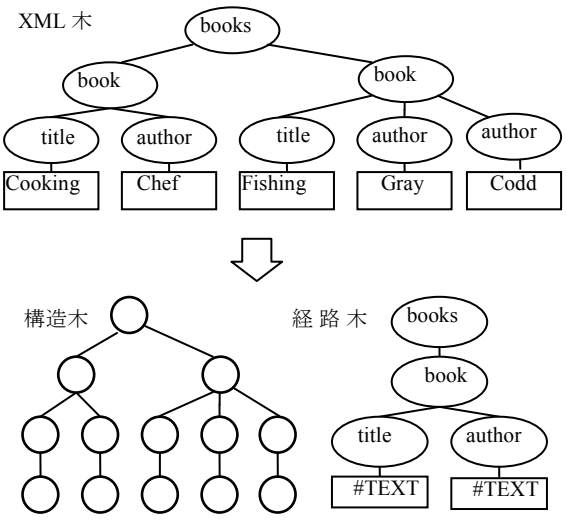

図 4　構造木と経路木

4.2 構造木と経路木の実装

XML 木のノード集合の構造を表現する木構造を構造木といい, 構造木を実装するための HPMD を node HPMD という. node HPMD では, 要素名が捨象されるため, 属性値テーブルや CVT を持つ必要がない. 座標パターンから取り出せる各次元の添字がそのまま node HPMD の拡張可能配列の座標としてノードの位置を表す. ただし, 各内部ノードについて, 必要に応じて, その子要素の文書順を保持するために OS(Ordered Sequence)テーブルと呼ぶ子要素の各添字の文書順を追跡するためのテーブルを持つ. XML 文書の格納時には添字と文書順は一致するが, 要素の追加挿入などにより添字と文書順が一致しなくなる状況においては, OS テーブルを構築し, 親要素とハッシュテーブルにより対応付けて管理する.

経路木では, 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に集約する. 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 HPMD に格納する. この HPMD を path HPMD と呼ぶ. path HPMD においては 3 節で示した①～⑤のすべてのデータ構造を実装している. ただし, node HPMD のように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.

path HPMD において XML におけるテキストノードにあたる要素は, 過剰な次元サイズの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テキスト保存用ファイルに別途保持する. node HPMD, path HPMD とも, 各高さレベル *l* における子要素集合の最大サイズが増加した場合に, 拡張可能配列の *l*+1 次元の実サイズが 1 増加する. また, XML 木の最大高さが増加すれば, 次元の拡張が起こる.

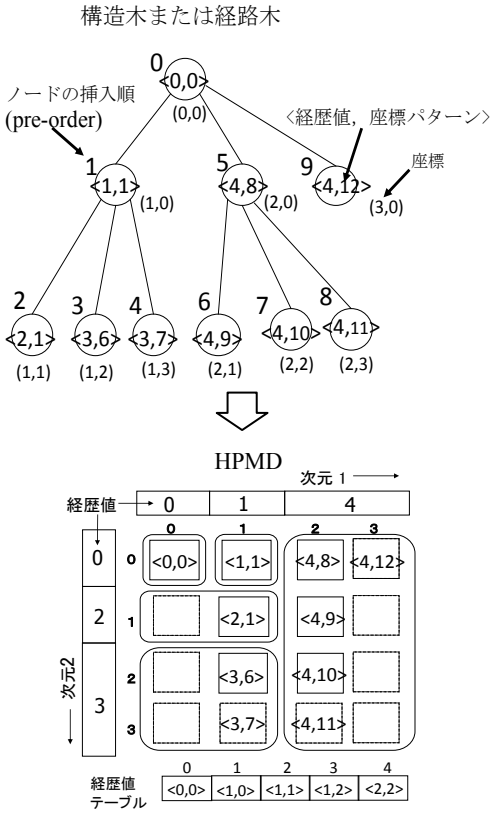

図 5　構造木／経路木の HPMD 拡張可能配列へのマッピング

4.3 node ID と path ID

node HPMD および path HPMD 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. node ID を node HPMD を利用してデコードすれば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 HPMD を利用してデコードすれば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 は一意な位置情報であるのに対して, path ID について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, XML 木のすべてのノードについて node ID と当該の path ID を相互に対応付ける必要がある. そのために node HPMD と path HPMD の ETF は 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 ETF を node HPMD と path HPMD で共有し, 固定サイズのページで区切られた構造とする．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．さらに，path ID をキーとして対応するページの先頭へのオフセットを保持する経路式索引を構築し, path ID からページへのアクセスを高速化する（図 6）. node ID をキーとして, 対応する path ID を検索する索引も望まれるが，node ID の数は path ID の数に比較して一般にははるかに大きく，この索引は膨大になる．これは本研究が目指すコンパクトな XMLDB の構築の方針を阻害することになり，ここではこのような索引は作成しないこととする.

経路式索引にはすべての path ID がキーとして検索できるため，ETF のページ先頭には path ID を格納する必要はな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軸検索パスも想定しうる．その際に読み取ったページの先頭から経路式を迅速に特定できるようにするために，ここ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 9 節で評価するが，ETF のサイズに対して path ID の占める領域は一般にごくわずかである.

さらに，node ID, path ID の固有の問題点を解消し，ETF をさらにコンパクト化するため，また，ID の検索を高速化するために次節で述べるように，ETF を XML 木のレベルごとに分割して個別のファイルに格納する.

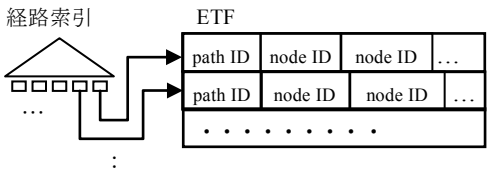

図 6　ページ区切りした ETF と Path Index

5. ID 格納の問題点と改善

node ID および path ID を ETF に格納する際, 動的に高さが増大し得る木構造のエンコードに HPMD を使用するときの固有の問題が発生し得る．すなわち，それぞれの HPMD における次元拡張やサイズ拡張の順序によっては，座標パターンの下位のレベルのビットがすべて 0 で埋まり記憶領域が無駄になることがある．この問題は，最大次元以外の次元で拡張可能配列の論理サイズが拡張された場合に生じ得る.

図 7 は木構造の成長とその境界ベクトルテーブル B の例を表しており，木構造のノード中の番号はノードの追加順を表している．ノード 1，ノード 2，ノード 3 の追加により，それぞれ，動的に次元拡張が行われる．3 節で示した HPMD の次元拡張の方法と 4.1 節で説明した，木構造の HPMD への埋め込みの方法により，ノード 5，ノード 6，ノード 7 が含まれる部分配列の経歴値はそれぞれ，5, 6, 6 となり，座標は(1, 2, 0), (2, 0, 0), (3, 0, 0)となる．したがって, ID はそれぞれ, <5, 1.10.00>, <6, 10.00.00>, <6, 11.00.00> となり，下位次元は無駄な 0 で占められる.

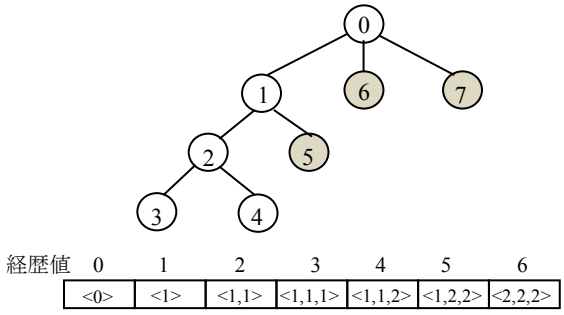

図 7　木構造の成長と境界ベクトル

この問題に対して，ID にノードのレベル値 *l* を加え，*l* を超える次元については添字パターンを登録しないようにすれば，無駄な部分を削減できる．これにより，上記の ID はそれぞれ, <5, 1.10>, <6, 10>, <6, 11>となり, それぞれ, 2, 4, 4 ビット削減できる．ただし，レベル値を格納する必要があるため，かえって記憶コストが増大してしまう場合が多い．そこで，ETF を木のレベルごとに複数，確保することとして，同じレベルのノードの ID は同一の ETF に格納する（図 8）．これにより，ETF 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あり，かつ，座標パターンにはレベルに対応する次元までのパターンでよくなるため，記憶コストを大きく削減できる．さらに，検索対象のレベルを絞り込める場合にはそのレベルの ETF のみ検索すればよくなり，検索を高速化できる.

各 ETF 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レベルの次元数分だけ参照してデコードを行うようにする．境界ベクトルには従来どおりであり，全次元分の添字サイズを記録しておく.

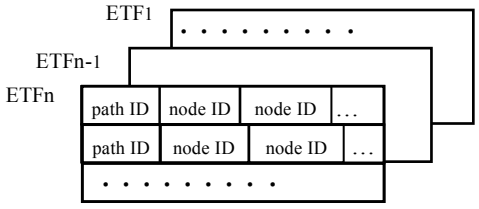

図 8　木のレベルによる ETF の分割

6. 検索手順

4.3 節で述べたように ETF には node ID と path ID の対応

付けをして各 ID が格納されている．ユーザの検索要求を受けて，条件と一致する両 ID を求めるために ETF を探索する．例として次の XPath について検索手順を説明する（図 9）．これは books 子要素である author 要素の 1 番目のノードに含まれるテキスト値を得る．

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

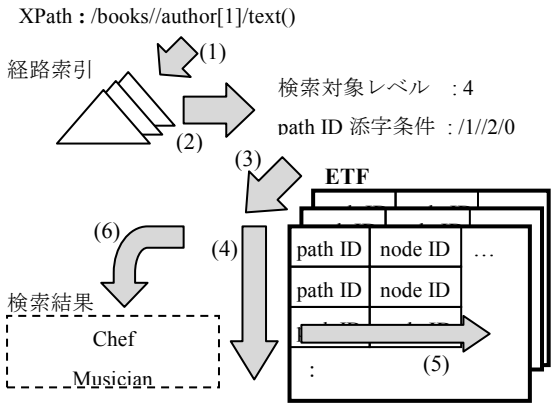

図 9　検索の流れ

まず path HPMD 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 としている．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，条件に一致する path ID は複数存在し得る．

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 ETF が検索対象となる．

検索対象になる次元（集合）が決定したならば，次に対象次元の経路索引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．もし経路索引から条件に合う path ID が見つかった場合、データ部から ETF のページを参照するオフセットを得ることができる．ETF 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 に関連付けられたテキスト値をテキスト保存用ファイルから読み取る．この node ID と path ID およびテキスト値は検索結果に含まれる．上記の手順を author を含む検索対象の次元の ETF 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 9）．

7. システムの構成

本節ではシステムの全体的な流れと構成を示す．本システムでは XML 文書の入力を受け付け，それを SAX[16]で解析して，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 HPMD と path HPMD に格納する．

XML 木の構造情報は node HPMD によって管理され，要素名や属性名などは path HPMD の CVT および属性値テーブルに格納される．テキストノードのテキスト値は 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．

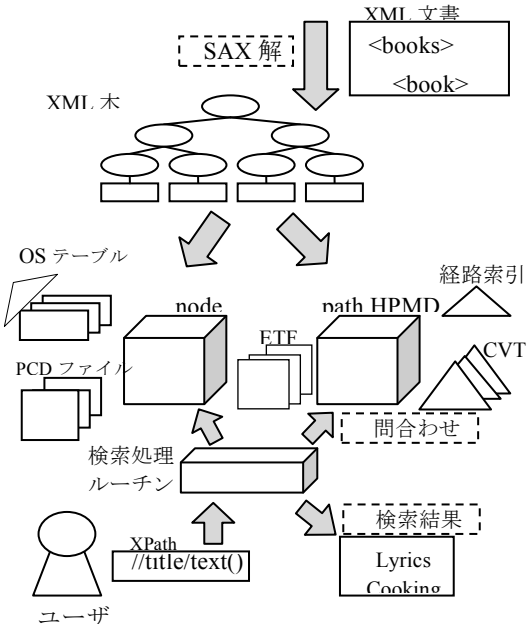

図 10　システムの構成

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 HPMD と node HPMD に問い合わせを行って ETF から検索条件に一致する node ID と path ID およびテキスト値を検索する．こうして得た node ID は node HPMD で，path ID は path HPMD 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 10）．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，ユーザは文字列として検索結果を受け取るだけで良い．

8. 関連研究

よく知られているように，XMLDB の実装方式には，XML 文書を関係テーブルにマッピングして，RDB のテーブルとして格納する方式と，XML 文書をそのまま木構造で格納するネイティブ XMLDB の方式がある．前者では，XML 文書は RDBMS の配下にあり RDBMS が提供する諸機能がそのまま使用できることが大きな利点であるが，マッピング

に要する格納コストおよび検索・操作コストが増大することが欠点である．後者は，逆に，XML 文書の格納・検索コストは低いが, DBMS としての機能を実装する必要があり，システムの開発コストが高い．eXist-db[9]，Xindice[10]，NeoCore[11], DB2[12]など，多くのネイティブ XMLDB システムが開発されている．

以上のようなシステム開発と同時に，軸指定による構造検索のために, XML 木の構造を反映した多くのノードラベリングの手法が提案されている．このようなラベル付けにより，ノード間の親子関係や兄弟関係，先祖子孫関係などをラベル値の検査により探索することができるため，効率よくノードを検索することが可能となる．これらのラベル付けの手法のひとつとして， XML 木を k-ary 完全木にマッピングするラベル付けの手法やその改良手法 [1)2)]が提案されている．しかし，これらの手法は静的なラベル付けであり， 動的なノード追加に対しては, 追加の位置によってはノード全体のラベル値の再計算が必要となる．

これに対して，構造の動的な更新に着目したラベル付けやエンコーディングの手法として，ORDPATH[3]，QED[4], DLN[5], 素数ラベル付け [6)]などが存在する. これらの利点は，XML 木の更新に対し，文書順を保持したまま，再ラベル付けを行う必要はない．しかし，いずれもノードの追加が動的に行われる場合，ラベルサイズが大きくなるという欠点がある．

経歴オフセット法を用いたラベル付 [7)]では先に述べた，k-ary 木の問題点と動的更新に強い，ORDPATH 等のラベル付けの欠点が解消されているが，経歴・オフセット空間のあふれにより，規模の大きい文書を格納することができない [8)]．経歴・パターン法によるラベル付けではこの欠点が軽減されており規模の大きい文書も格納処理できる．

9. 評価実験

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．

9.1 実験環境

実験に用いた計算機は以下の仕様である．

OS : CentOS 6.4
CPU : Intel Xeon X5690 @3.47[GHz]
RAM : 48[GB]
HDD : 450[GB]，15,000[rpm]

本システムと比較する XMLDB システムの対象として，eXist 2.0[9)]および NeoCore XMS 3.1.3[11)]の 2 つの XMLDB を利用した．

格納する XML 文書は XMark[17)]を利用して自動生成されたベンチマーク用 XML 文書を使用する．xmark.xml は XMark の設定値として scaling factor に 100 を指定して生成した．以下に XML 文書情報を示す．

ファイルサイズ : 116,517,691 [byte]
XML 木の最大高さレベル : 13
総ノード数 : 3,240,011
要素名数 : 93

比較評価には，共にネイティブ XMLDB システムである eXist-db と商用の NeoCore を用いた．前者はオープンソースシステムとして，広く使用されており，また，後者は，XML データ格納時に, インデクスを付与するフルオートインデクス機能を有している．

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．なお，eXist と NeoCore では Ngram インデクスによる全文検索機能に対応しているが，ここでは全文検索のオプションは付けていない．本システムでは全文検索への対応は研究対象外として実装していない．

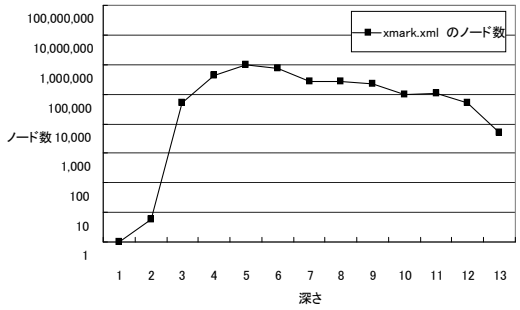

図 11 XML 木のレベル別ノード数

9.2 記憶コスト

本システムでは node HPMD と path HPMD の各種構成要素のサイズ，実データを保持する ETF と PCD ファイルのサイズを測定し,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測定する．ETF については当該 XML 木の各レベルの ETF サイズの合計値である．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 に本システム，eXist-db，NeoCore の 3 システムについて xmark.xml を一括格納したときの記憶サイズの合計と内訳を示す．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 0 である．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，ETF の再編成処理を行うことでサイズを 0 に初期化できる．

提案方式では ETF 内に記憶されている ID について, path ID 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，無視できるほどサイズは小さくなり，その記憶コストは ETF 全体の 0.1%以下である．また，提案方式では，全体のサイズは各 XML 文書サイズよりコンパクトに実装できており，9.1 節で示した元の文書サイズより，やや小

さくなっている．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．一方, eXist-db では元の文書サイズの 1.95 倍となっており，NeoCore では，5.61 倍となっている．

表 2 は表 1 に示したデータ本体サイズ以外の本システムにおける補助データ構造のサイズである．合計サイズはいずれも本体サイズに比べて非常に小さく無視できる．

表 1　記憶コスト

		xmark.xml (byte)
本システム	ETF	28,147,712
	└ node ID の領域	25,693,904
	path ID の領域	22,410
	Path Index	3,133
	PCD	83,514,149
	合計	111,664,994
eXist-db	collections.dbx	159,744
	dom.dbx	184,463,360
	ngram.dbx	2,928,640
	sort.dbx	8,192
	structure.dbx	39,202,816
	symbols.dbx	18,033
	values.dbx	208,896
	words.dbx	8,192
	合計	226,997,873
NeoCore	*.crf (Cross ref)	5,251,072
	*.inx (Core index)	27,320,320
	*.dup(duplicate index)	89,874,432
	*.adm	1,243,960
	*.map(物理的構造情報)	267,509,760
	*.lmf	262,003,459
	合計	653,203,003

表 2　本システムにおける補助データの構造の記憶コスト

		xmark.xml (byte)
node HPMD	境界ベクトルテーブル	8,624
	経歴値テーブル	181
	小計	8,805
path HPMD	境界ベクトルテーブル	4,256
	経歴値テーブル	142
	CVT	1,235
	属性値テーブル	952
	小計	6,585
	合計	15,390

9.3 構築時間

表 3 に各 XML 文書を入力として，データベースを構築するのに要した時間を示す．NeoCore ではフルオートインデクス機能によりデータ要素格納時に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ために，構築時間は長くなっている．

表 3　データベース構築時間

	xmark.xml (sec)
本システム	15.1
eXist-db	24.6
NeoCore	324.7

9.4 検索時間

パス式検索には XPath を用いるが，XPath のすべての仕様は満たしていない．現時点で軸には child，descendant，descendant-or-self，attribute に対応している．また，述語においては属性，text()，position()に対応しているが，単一条件のみの対応などの制約がある．

9.4.1 単純なパス式検索

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，最短時間を得た．経路式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．表 4 にその結果を示す．提案方式では，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．

表 4　単純 XPath 式の検索時間

	最長[ms]	平均[ms]	最短[ms]
本システム	68.1	3.41	0.0160
eXist-db	412	47.7	4.22
NeoCore	576	36.9	0.935

9.4.2 複雑なパス式検索

述語や子孫軸の指定を含む XPath（表 5）による検索要求をそれぞれ 5 回ずつ行い，平均時間を得た．表 6 にその結果を示す.

表 5　検索例

#	XPath
1	/site/regions/africa/item[@featured='yes']/description
2	/site/open_auctions/open_auction[2000]/bidder/*/text()
3	/site/regions//text

表 6　ヒット件数および検索時間

#	ヒット件数 [件]	本システム [ms]	eXist-db [ms]	NeoCore [ms]
1	54	3.49	56.5	6.7
2	3	590	811	588
3	61823	1201	254	2308

本システムにおいて，XPath のロケーションステップ中に述語や子供以外の軸が含まれる状況においては、Path Index による絞り込みが完全に行えない．このため，単純なパス式と見なせるステップまでの検索を行い，それ以降はステップごとに検索と絞り込みを行う必要があり，単純パス式の検索に比べて時間を要する．これに対し，eXist-db では子孫軸を含む#3 の XPath 検索を高速に行えた. eXist-db においては，単純パス式のようにロケーションステップを重ねるよりも子孫軸を用いて少ないロケーションステップでクエリを記述することで，参照するインデクスの数を抑え，高速な検索が行われるためである．

9.4.3 軸検索

XML 木の全レベルからランダムにノードを選択し, そのノードを軸として各種軸検索要求を実行することを 500 回行い、平均時間を得た．図 12 に結果を示す.

軸検索においては，軸に指定した要素ノードが持つ経路式情報を用いることで，ETF の探索範囲を絞り込めるため高速な検索を行えた．子孫検索については，ルートに近いレベルの要素を軸とした場合において，子孫の数が膨大となるため，他の軸に比べて時間を要した.

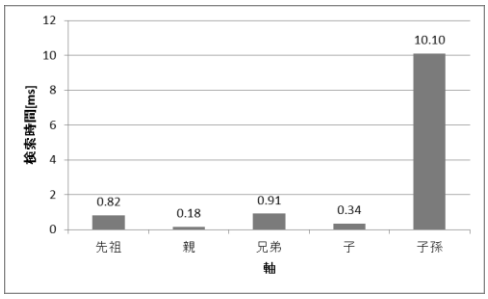

図 12　本システムにおける軸検索時間

9.5 ノードの追加挿入

表 5 の XPath が指すノードの直後に空要素ノードを追加するのに要した時間と，すべての XPath に対して挿入を行った後での検索要求に要した時間を表 7 に示す.

本システムにおいて，挿入時には OS テーブルの構築が行われるが，このときノードの最大添字を求めるために兄弟ノード数を求めるための検索が行われるため, 大量の OS テーブルが構築される#3 の XPath においては挿入時の時間コストは大きくなってしまう．また，挿入後の検索時間は挿入前（表 6）と比べると，#1 や#2 の XPath においては参照する OS テーブルの数が少ないため，検索時間にはほぼ影響がなかった．しかし，#3 の XPath においては OS テーブルの参照量が多いために 1.2 倍程度の検索時間を要した.

表 7　ノード挿入時間と挿入後の検索時間

#	挿入時間[ms]	挿入後検索時間[ms]
1	19.0	3.50
2	11.5	587
3	46877	1424

10. むすび

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 HPMD および path HPMD 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 node ID 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.

文献

1) Meier, W. :eXist-db: An Open Source Native XML Database", Proc. of Web, Web-Services, and Database Systems, pp.169-183, 2002.
2) Rung-Ren Lin, Ya-Hui Chang, Kun-Mao Chao: A Compact and Efficient Labeling Scheme for XML Documents, Proc. of DASFAA, pp.269-283, 2013.
3) O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: Insert-friendly XML node labels, Proc. of the ACM SIGMOD, pp. 903–908, 2004.
4) Li, C., Ling, T.W.: QED: a novel quaternary en-coding to completely avoid re-labeling in XML updates, Proc. of International. Conference on Information and Knowledge Management (CIKM'05), pp. 501–508 (2005)
5) Bohme, T. , Rahm, E. : Supporting Efficient Streaming and Insertion of XML Data in RDBMS, Proc. 3rd Int. Workshop on Data Integration over the Web, 2004.
6) Wu, X., Lee, M. L. and Hsu, W. : A Prime Number Labeling Scheme for Dynamic Ordered XML Trees, Proc. of ICDE, pp.91-99, 2004.
7) Li, B., Kawaguchi K., Tsuji T., Higuchi K.: A Labeling Scheme for Dynamic XML Trees Based on History-offset Encoding, 情報処理学会論文誌: データベース, Vol.3, No.1, pp.1-17, 2010.
8) Tsuji, T., Amaki K., Nishino H., Higuchi K., History-offset Implementation Scheme of XML Documents and Its Evaluations, Proc. of DASFAA 2013, pp.315-330, 2013.
9) eXist-db Open Source Native XML Database: http://exist.sourceforge.net
10) The Apache Foundation: Xindice 1.1 Developer Guide: http://xml.apache.org/xindice/guide-developer.htm
11) XML データベース NeoCore: http://www.neocore.jp/
12) DB2 pureXML - Intelligent XML database management ALT: http://www-01.ibm.com/software/data/db2/xml/
13) 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
14) Tsuchida, T., Tsuji, T., Higuchi, K. : Implementing Vertical Splitting for Large Scale Multidimensional Datasets and Its Evaluations, DaWaK 2011, pp.208-223, 2011..
15) XML Path Language, http://www.w3.org/TR/xpath/
16) The Simple API for XML (SAX) : http://www.saxproject.org/
17) XMark - An XML Benchmark Project: http://exist.sourceforge.net/