構造更新に対応したXML木のラベル付け方式とその評価

Bei LI† 川口 勝也† 都司 達夫† 樋口 健†

†福井大学大学院工学研究科 〒910-8507 福井県福井市文京 3-9-1

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

あらまし 本論文では，XML木の構造更新に対応したラベル付けの方式を提案する．この手法は，経歴オフセット法と呼ばれる多次元データセットのエンコード方式に基づいており，XML木を多次元拡張可能配列へと埋め込むことによりエンコードを行うが，XML木の構造更新に対し，再ラベル付けを行う必要はない．本方式の最も優れた点として，同種の他方式と比べ，ノードの追加場所と順序に関わらず，ラベルの記憶コストが格段に小さくて済むことが挙げられる．一方，他方式では，同じ場所に複数ノードの追加が起こると，生成されるラベルのサイズは膨大になってしまう．我々のラベル付け方式について説明した後，ラベルサイズ・ラベル記憶コスト・ノード検索コストについて，他方式のDLN,ORDPATH,QEDとの比較実験を行い，我々の手法が有効であることを示す．

キーワード XML木，ラベル付け，構造更新

1. はじめに

近年，半構造データを効率よく扱える言語として，XML(Extensible Markup Language)[1]が注目されており，それに伴い，大規模のXML文書データを効率よく記憶・管理することの重要性が高まってきている．XML文書の論理構造は木構造を用いて表現することが可能であるが，XML文書データの管理において，文書構造を効率よく管理するために，文書から得られるXML木ノードへの効率のよいラベル付けの方法を提供することが重要である．

代表的なラベル付けの手法として，XML木をpre-order順とpost-order順にたどって番号付けを行う範囲ラベル付けの方式[2]や，本の章・節立てのようにラベル値を割り当てていく接頭辞ラベルの方式[3][4]，ノードに素数を割り当てるラベル付けの手法[5]など，多くの手法が提案されている[6]．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．

また，XML木をk-ary完全木にマッピングするラベル付けの手法[7]およびその改良手法[8][9][10]が提案されている。これらの手法ではラベル値の簡単な加減乗除算により高速に各軸に対応するノード（集合）を求めることができる．しかし，基本的には静的なXML木ノードに対するラベル付けであり，動的なノード追加に対しては，ノード全体のラベル値の再計算が必要となることがあり，また，効率的なラベル値空間の使用が不可能であり，ラベル値空間があふれやすい．

一方，効率的な構造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[11], QED[12], DLN[13]などが存在する．これらの最大の利点は，XML木のあらゆる更新に対し，文書順を保持したまま，再ラベル付けを行うことなくラベル付けが可能な点である．しかし，ノードの追加が特定の場所に集中的に行われる場合，ラベルサイズが極端に大きくなるという欠点がある．

[14]では，k-ary完全木にマッピングするラベル付けの手法の長所を保持しつつ，これらの欠点を解消するために，我々が提案している多次元データのエンコード方式である“経歴オフセット法”[15]を利用した，XML木ノードのラベル付けの方式を提案している．この方式では，比較的小さい補助データ構造を用いるのみでXML木の動的な構造更新に対して，ラベル値の再計算を必要とせずに，効率よく対処できる．また，ラベル値空間も有効に利用できる。

本論文では，経歴オフセット法によるラベル付け方式について，その文書順保持の方式を提案し，ORDPATH, QED, DLNなどの構造更新に対応した手法と，記憶コストや検索コストの比較評価を行う．本方式では，集中的なノード追加の場合にも，格段に小さい記憶コストで効率よくラベル付けができる．

2. 経歴オフセット法

本方式の基盤となる多次元データの実装方式である経歴オフセット法について概説する．多次元データの代表例として関係テーブルがあるが，本節では，関係テーブルを対象として多次元データの実装を論じる．

2.1. 拡張可能配列

プログラミング言語で用いられる従来の固定サイ

ズ配列は，実行時に動的に配列の各次元サイズの変更を行うことができない．配列を動的に拡張する必要が生じた場合，より大きな新たな記憶域を確保し，元の配列データを新たなアドレス関数により再配置しながら，要素ごとにコピーする必要があるため，大きい時間的コストがかかる．これに対して，拡張可能配列[16]では，配列サイズの動的変更に対して，すでに確保している領域はそのまま使用し，新たな領域を差分領域として，必要な分だけ付加することができる(図 1)．

一般に n 次元拡張可能配列は，n-1 次元の固定サイズの部分配列の集合で表される．拡張可能配列では，各部分配列の先頭アドレスおよび，部分配列が何番目の拡張の際に確保されたかを表す経歴値を補助テーブルで管理している．また，部分配列の要素へ高速にアクセスできるように，n が 3 以上の場合には，補助テーブルを用いて各部分配列のアドレス関数を計算するための n-2 個の係数値を係数ベクトルとして記録している．図 1 では 2 次元拡張可能配列を第 1 次元方向にサイズを一つ拡張している．第 2 次元方向のサイズが 3 であるので，サイズ 3 の部分配列を確保する．次に，拡張を行う前の経歴値の最大値よりも 1 つ大きい値を新たに確保した部分配列の経歴値として，また，新たに確保した部分配列の先頭アドレスを拡張次元方向の補助テーブルに記録する．

また，要素<i_1, i_2, ..., i_n>が与えられたとき，各次元の添字に対応する経歴値の内，最大経歴値の次元を m とすると，次元 m の補助テーブルの i_m番目の先頭アドレスの部分配列が，要素を含む．その係数ベクトルを知って，要素のアドレスが計算できる．

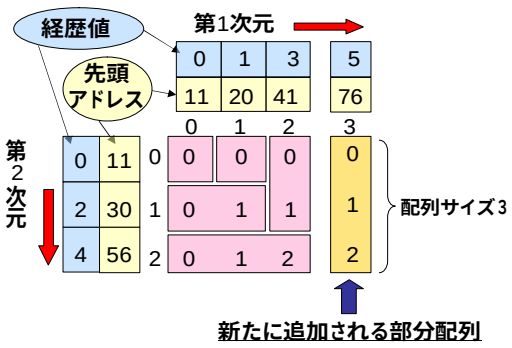

図 1　拡張可能配列

2.2. 拡張配列を用いた多次元データの表現

関係テーブルの各カラムを従来の多次元配列の各次元に，カラム値を配列の添字に対応付けることによって，関係テーブルのレコードを，配列要素の位置で表現することができる．2.1 節で述べた，拡張可能配列を用いれば，関係テーブルの動的なレコードの追加に対処することができる．図 2 に拡張可能配列を用いた，関係テーブルの表現例を示す．

2.3. 経歴オフセット法を用いた多次元データの実装

2.2 節で述べたように，多次元配列を用いて関係テーブルを表現することは可能であるが，通常，関係テーブルのレコードに対応しない配列要素が多く存在してしまうため，使用しない配列要素を多く確保しなければならないという問題がある．そこで，配列次元数に依存することなく配列要素の位置情報を保持し，また，無駄な配列要素を保持しないために，拡張可能配列の経歴値と部分配列内のオフセットを配列要素の位置情報として保持する．この位置情報は対応するレコードそのものを表すことになる．このような多次元データセットのエンコーディングの方法を，経歴オフセット法と呼ぶ[15]．図 3 に，経歴オフセット法を用いた関係テーブルの実装例を示す．

図 3 では，関係テーブルのカラム値から配列の添字に高速に変換できるよう，配列の各次元において添字変換用の B$^+$木（CVT：key-subscript Conversion Tree と呼ぶ）を利用している．また、配列要素の位置情報を表す，経歴値とオフセットの組を B$^+$木（RDT： Real Data Tree と呼ぶ）を用いて管理している．このように，経歴オフセット法を用いて多次元データを実装するためのデータ構造を HOMD(History-Offset implementation for Multidimensional Datasets)と呼ぶ．

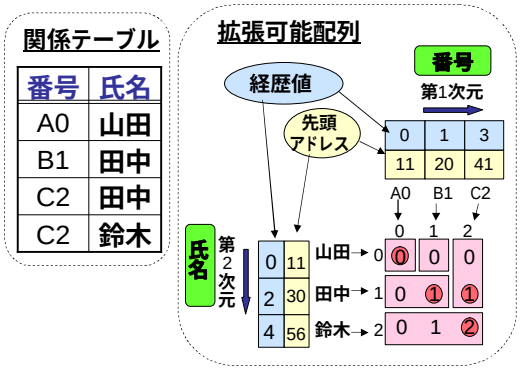

図 2　拡張可能配列を用いた関係テーブルの表現

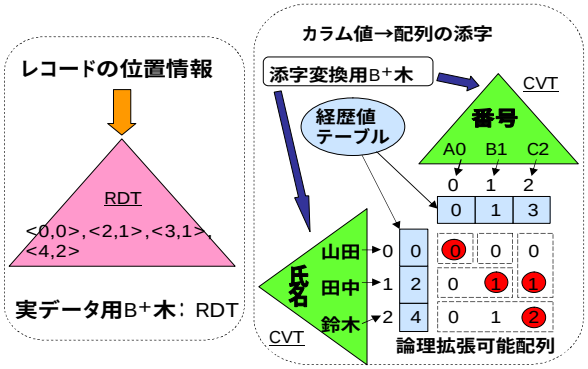

図 3　経歴オフセット法による関係テーブルの実装例

3. XML 木ノードのラベル付け

本節では，2.3 節で述べた経歴オフセット法を用いた，XML 木ノードに対するラベル付けの方式を提案する．

3.1. 固定サイズ配列を用いたラベル付け

図 4 に示すように，木の深さレベルを配列の各次元に対応付ける．また，各ノードの子ノード集合について，最左の長子ノードから順に番号を 1，2，3，…と割り当て，その番号を配列添字に対応付ける．このようにすることで，各ノードを配列要素の位置(座標値)に一意的に対応付けることができる．さらにこの座標値に対して，配列のアドレス関数を計算した結果の値をこのノードのラベル値として与える．ただし，木の深さレベルの昇順に配列次元を次元 1，2，3，…と割り当てる．また，ルートノードには特別に，ラベル値として 0 を割り当てる．

この方法を用いると，ノードの経路式から，配列のアドレス関数を利用して，ノードのラベル値を求めることができる．さらに，アドレス関数の係数値を係数ベクトルとして記憶しておけば，これを利用して，逆にノードのラベル値から簡単な割り算によりそのノードの経路式を求めることができる．また，ノードのラベル値から，その親子や兄弟のノードのラベル値を直接算出することも可能である．

しかし，この方法では固定サイズ配列の各次元のサイズが不変であることを前提としている．したがって，XML 木への新しい要素の追加や，既存要素の削除などの動的な構造更新には対処不能である．すなわち，このような場合には配列の動的な拡張や縮小が必要になり，これは新たなアドレス関数により XML 木ノードをすべて再配置しなければならないことを意味し，高いコストが必要となる． さらに，垂直方向に沿った XML 木の伸張を必要とする場合には対応する新たな配列次元を用意する必要があり，これについても，XML 木の全ノードの再配置を行う必要がある．

以上より，固定サイズ配列を用いた XML 木ノードのラベル付けは多くの利点を有するものの，静的な XML 木にのみ適用できる方式であるといえる．ノードのラベル値そのものが，ルートノードからそのノードに至る経路式情報を担っており，ラベル値に対する簡単な計算により，隣接ノードのラベル値を求めることができる他のラベル付けの方式［8］［9］[10]についてもこの利害得失は共有されている．

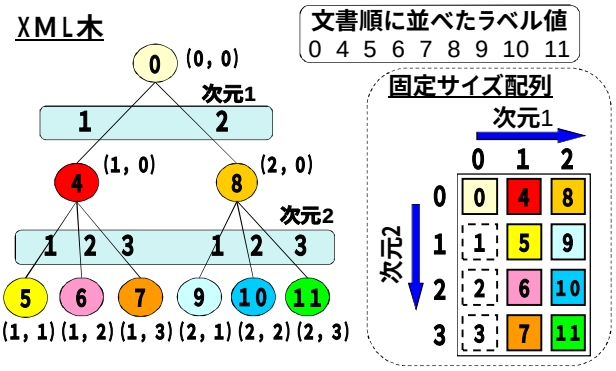

図 4　固定サイズ配列を用いた XML 木表現

3.2. 経歴オフセット法を用いたラベル付け

固定サイズ配列を用いたラベル付けの方法の問題点を解決するために、XML 木のマッピングに 2.4 節で述べた経歴オフセット法を用いる．これにより、XML 木は HOMD データ構造で表現される．

図 5 に，HOMD による XML 木の表現例を示す．HOMD を用いた場合も，3.1 節で述べたように，木の深さを配列次元に，ノードに番号付けした値を配列添字に対応付けることで，ノードの座標値を得る．この座標値を経歴オフセット法により、そのノードが含まれる部分配列の経歴値と，部分配列内のオフセットの組にエンコードしてラベル値として RDT に格納する．また，固定サイズ配列の場合では，係数ベクトルを配列全体について唯一保持していたが，HOMD の場合は，部分配列ごとに係数ベクトルを持つ．この係数ベクトルはノードの親子や兄弟ノードのラベル値の算出に用いられる．図 5 では XML 木が pre-order の順に成長したとして，対応する HOMD の部分配列の拡張の様子とノードのラベル値である <経歴値，オフセット> 対を示している．このラベル値は HOMD の RDT に格納される．

経歴オフセット法を用いた上記のラベル付けの方式では新たなノードの追加により，部分配列の拡張が行われても既存ノードの配列内での再配置は必要なく，したがって，ノードのラベル値も不変である．さらに，XML 木の要素のみそのラベル値を RDT に格納するので，配列のランダムアクセス性は利用するものの，無効要素が記憶域を占めることはない．さらに，大きな利点として，XML 木の高さが深くなっても（次元が大きくなっても）ラベル値は固定長データの対で表現でき，システム内部では固定長として扱えることの利点は大きい．

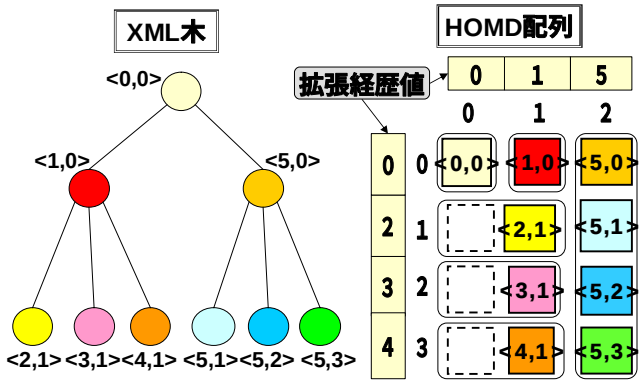

図 5　HOMD を用いた XML 木表現

3.3. 要素の挿入と削除による配列次元サイズの拡張･縮小

XML 木に対して要素が挿入されたとき，配列の拡張が起こり得る．拡張されるのは，XML 木ノードの挿入に際して，その高さレベルにおける分岐数の最大が増加する場合である．例えば，図 5 の XML 木について，次元 1 の高さレベルにノードが追加されたときには，次元 1 の方向に経歴値 6 の部分配列が確保されラベル値 <6,0> のノードが配列座標 (3,0) の位置に置かれる．また，XML 木に対して要素が削除されたときには，配列の縮小が起こり得る．例えば，ラベル値<5,0>をルートとする部分木ノードがすべて削除されたときには，経歴値 5 の部分配列が削除され 1 次元方向に配列が縮小される．

3.4. 配列次元数の増加･減少

XML 木に，要素を挿入した結果，木の高さレベルが伸張したときには，対応する HOMD 配列の次元数が 1 つ増加する．配列の次元拡張に対しては，2.1 節で説明した拡張可能配列は極めて効率よく対処可能である．一般に n 次元拡張可能配列に新たな次元を増設するときには，あらたに，HOMD テーブルをその次元に対して用意し，初期化するだけでよい．既存の n 次元拡張可能配列の各次元番号は，1 つずつ増加し，増加した次元の番号は 1 となる．この時，拡張前の配列の次元 1 の添字は 0 となる．図 6 に XML の高さレベルの伸張に対応した HOMD 配列の次元拡張の様子を示す．

逆に高さレベル *n* の XML 木の要素を削除した結果，高さレベルが 1 減少したときには，拡張可能配列の次元数の縮小が起こり，*n*-1 次元配列となる．すなわち，次元 1 の HOMD テーブルが破棄されるとともに，既存の *n* 次元拡張可能配列の各次元番号は，1 つずつ減少する．

通常の関係テーブルにおけるスキーマ進化操作における add column や drop column ではテーブル全体の再編成が必要となるのに対して，HOMD で実装される XML 木ではこのような再編成処理を必要としない．

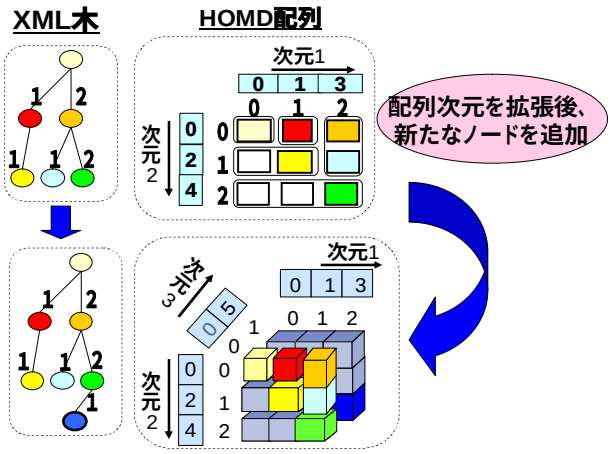

図 6　HOMD の次元拡張

3.5. 文書順の保持のためのデータ構造

図 7 に,兄弟ノード間の文書順保持のためのデータ構造(1 次元配列)の例を示す．以後,このデータ構造を os (ordered sequence) テーブルと呼ぶ．このテーブルの添字は拡張可能配列におけるノードの次元添字を表しており，また，テーブル要素の値は文書順で次順の弟ノードの論理拡張可能における添字を表す．このテーブルを順にたどることで,兄弟ノード集合にアクセスできる．

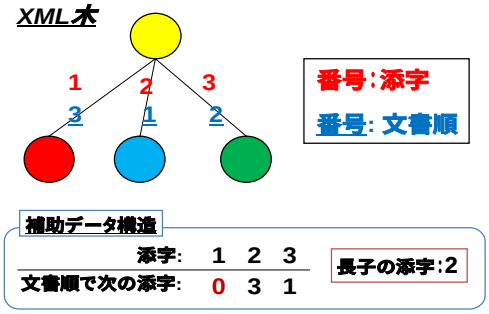

図 7　文書順保持のためのデータ構造(os テーブル)

4. 軸の計算

XPath[17]では，XML 木ノードの検索条件として，軸(Axis)を用いている．本節では，3 節で述べたラベル付けの方法における，XPath の軸の計算について述べる．

4.1. 固定サイズ配列の場合

本論文で提案しているラベル付け方式では，3.1 節で述べたように,アドレス関数を計算することにより，ノードの配列座標に対応したアドレス値をそのノード

のラベル値とする．その値を利用して，ノードのラベル値から，その親や兄弟などのノードのラベル値を算出することができる．

ここで，ラベル値が m のノードをノード m と呼び，ノード m のレベルを $l(m)$と記す．また，レベル l に対応する係数ベクトルの項の値を C_l とすると，以下の方法で，ノード mの親子･兄弟のラベル値を算出できる．

・ノード m の親のラベル値：

0 ($l(m) = 1$)

$m - m \% C_{l(m)-1}$ ($l(m) \geqq 2$)

・ノード m の第 k 子のラベル値 ($k \geqq 1$)：

$m + k * C_{l(m)+1}$ ($l(m)+1 <$ 最大レベル)

$m + k$ ($l(m)+1 =$ 最大レベル)

・ノード m の弟のラベル値：

$m + C_{l(m)}$ ($l(m) <$ 最大レベル)

$m + 1$ ($l(m) =$ 最大レベル)

このような方法を用いることで，ノード間の親子・兄弟や祖先・子孫の軸を求めることができる．

図 8 に，いくつかの軸を示す．図 8 では，コンテキストノードの座標を(2,2,0,0)としている．なお，このコンテキストノードの属するレベルは 2 である．

4.2. HOMD の場合

HOMD を利用する場合でも，求めたいノードのラベル値が，コンテキストノードのラベル値と同じ経歴値を持つのであれば，同じ部分配列(固定配列)にそのラベル値が含まれる．したがって，4.1 節とほぼ同様の考え方で，親子・兄弟ノードの算出が可能である．

しかし，求めたいノードがコンテキストノードのラベル値とは異なる経歴値を持つ場合は，4.1 節の計算式ではラベル値を求めることができない．そこで，その場合はノードに対応している配列座標を利用することでラベル値を算出する．

例えば，図 8 では、コンテキストノードは配列座標(2,2,0,0)に対応しているが，コンテキストノードの親の配列座標は，コンテキストノードの属するレベルに対応する座標値を 0 にした(2,0,0,0)となるので，この座標から経歴値とオフセットを求めれば親のラベル値が得られる．

ラベル値である<経歴値,オフセット>対から，配列座標$(i_1,i_2,\ldots,i_n)$への変換は，以下の(a1),(a2)で行う．

(a1)ラベル値の経歴値が，どの部分配列のものかを調べる．その部分配列が，配列全体において，どの次元のどの位置に属するかを知ることで，座標の値 i_p がひとつ定まる．

(a2) オフセットを $o(o > 0)$, (a1)の部分配列の係数ベクトルを$[c_1,c_2,\ldots,c_{n-2}]$とすると，o は

$$o = c_1i_1+c_2i_2+\ldots+c_{p-1}*i_{p-1}+c_p*i_{p+1}+\ldots+c_{n-2}*i_{n-1}+i_n$$

と表される．

従って，座標値 i_k は，

$i_1 = o/c_1$

$i_k = (o\%c_{k-1})/c_k$ ($k < p$ の場合)

$i_k = (o\%c_{k-2})/c_{k-1}$ ($k > p$ の場合)

$i_n = o\%c_{n-2}$

として求められる．

また，配列座標から経歴値とオフセットへの変換は，次のように行う．

(b1) 配列座標の各値に対応する部分配列を求め，各部分配列の経歴値を比較し，一番経歴値が大きい部分配列を求める．これがラベル値の経歴値となる．

(b2) 配列座標から(b1)の部分配列の属する次元の座標値を除き，(b1)の部分配列から得られるアドレス関数から，オフセット値を算出する．

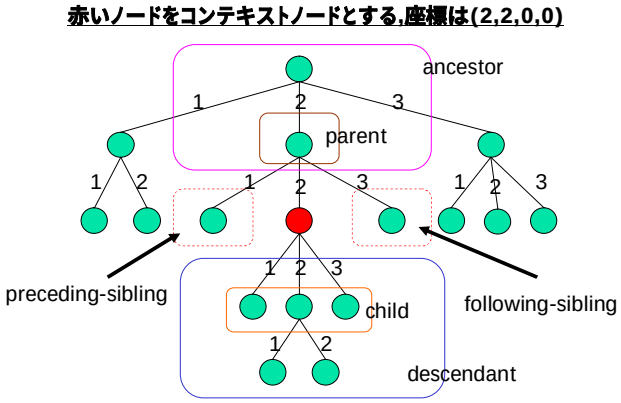

図 8　XML 木と軸

5. 他方式の概要と特徴

本節では，XML 木の構造更新に対応した他のラベル付け方式の概要と特徴を述べる．

5.1. 概要

XML 木の構造更新に強いラベル付けの手法として,ORDPATH[11]や DLN[13]などが挙げられる．また，辞書式順を利用したエンコード方式である QED[12]を利用して,柔軟に構造更新に対応できる．また，例えば，本の章・節立てのようなラベル付けの接頭辞ラベルにおいて,ラベルの整数部(XML 木の各レベルの値)を QED コードに置き換えるようなラベル付けの手法 (以降,この方法を QED-PREFIX と呼ぶ) が利用できる．

これらの手法の最大の利点は,再ラベル付けを行うことなく,あらゆる更新に文書順を保ちながら対応できることである．例えば ORDPATH の場合,ラベルが'1.3'と'1.5'のノード間に新たなノードを追加する場

合は,'1.4.1'を一意なラベルとして割り当てる．また,DLN では'1.2'と'1.3'の間に'1.2/1'を,QED-PREFIX では,2 進数で表現された 2 つのラベル'10.10'と'10.11'の間には'10.1010'を割り当てる．

これらの手法はすべて接頭辞ラベルを基本としており,ラベル値の大小順がそのまま文書順に対応するので,大小関係だけでノード間の文書順を判定することが可能である．

5.2. 更新におけるラベルサイズ

5.1 節で述べた各々の手法は,構造更新に柔軟に対応できる反面,ノードの追加更新に対してラベルの長さが増加しやすいという欠点がある．特に,一箇所に集中的にノードが追加されると,ラベル長は極端に増加する．

図 9 に,ORDPATH を例として,ラベル長が極端に増大するような集中的なノードの追加例を示す．ノード間へ次々と追加することで,ラベル長が長くなっている．DLN についても同様であるが，QED-PREFIX ではラベルを構成する QED コードの長さが増加する．

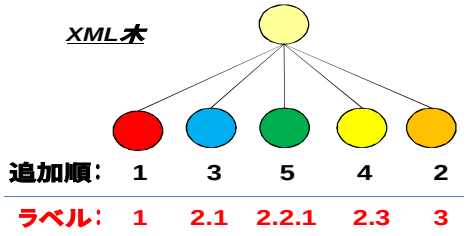

図 9 ノードの集中的な追加

6. 比較実験

本節では,3 節で述べた我々の提案手法と,ORDPATH, QED(QED-PREFIX),DLN との比較実験について述べる．実験には Sun Blade 1000 (CPU: UltraSparcⅢ(クロック速度 750MHz)，メモリ容量:512MB，ディスク:208GB，OS: Solaris 8) を用いた。

6.1. 実験条件

6.1.1. 使用 XML データとラベルの格納・検索

実験用の文書データ作成として Xmark[18]を利用し，さらに,データ解析時に読み取る木の高さを変えることで 4 種類の XML 木データを作成した．文書データの解析には SAX[19]を利用した．表 1 に，使用した 4 つの XML 木のノード数とレベルを示す．なお，使用した文書データのサイズは 16214640 [Byte]である．

XML 木ノードラベルの動的な追加に対応するため，また，ラベル値の逐次アクセスとランダムアクセスの双方を効率よく行うためにラベルの格納にディスク上の B^+木を用いた。

検索時には,XML 木の総ノードのうち 10%のノードをコンテキストノードとしてランダムに選択する．また,いかなるノードの追加順にかかわらず,最終的な XML 木の形は同一である．

表 1 XML 木ノード数とレベル

木の高さ	総ノード数	平均ノードレベル
3	62474	2.889298
5	261048	3.996242
6	297163	4.239764
7	334176	4.545485

6.1.2. 長大ラベルの格納

ラベルサイズが比較的大きい場合は,ディスクファイル上にラベルを格納する．長大ラベルには文書順に従った識別子を与え,その識別子を B^+木のキーとし,データ部には長大ラベルを格納する上記ファイルの当該ラベル位置を格納する．ラベル値を得る場合は,識別子をキーとして B^+木を検索し,得られたデータ部の位置を知り，ラベル値にアクセスする．

6.2. ラベル記憶コストの測定

本節では,ランダムなノード追加と 5.2 節で述べた集中的なノード追加の 2 種類についてラベル記憶コストを評価する．

6.2.1. ランダムなノード追加の場合

図 10 に,各ラベル付け手法における,ラベルサイズの測定結果を示す．

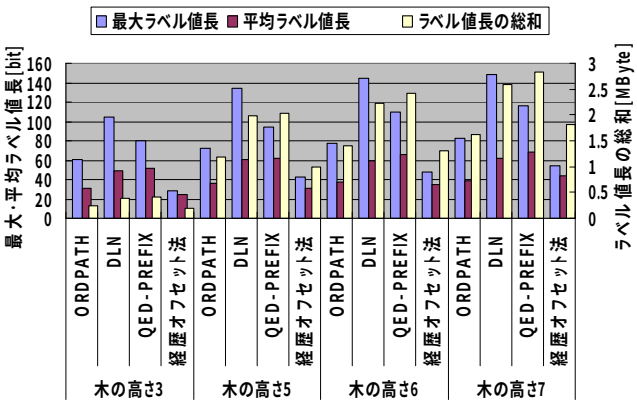

図 10 ランダム追加時のラベル記憶コスト

また,経歴オフセット法における，os テーブルの合計サイズは，XML 木の高さ 3, 5, 6, 7 において，それぞれ，0.186，0.981，1.290，1.660MB であり，図 10 のラベル値長の総和にそれぞれ，加えたサイズが記憶コストとなる．

6.2.2. 集中的なノード追加の場合

図 11 に，集中的なノードの追加が生じた場合の，他方式のラベルサイズについて示す．また，経歴オフセット法における各ラベルサイズを図 12 に示す．

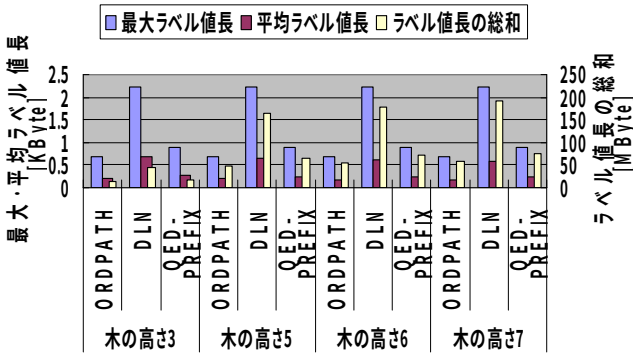

図 11　集中追加時のラベル記憶コスト(関連研究)

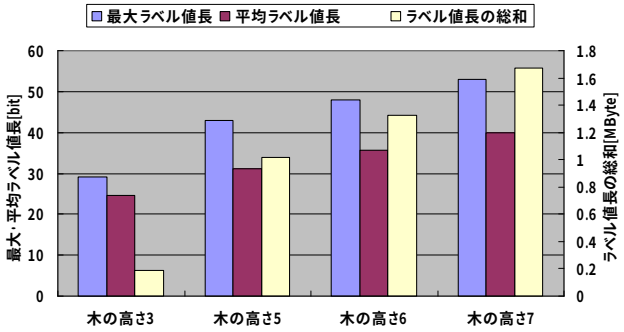

図 12　集中追加時のラベルサイズ(経歴オフセット法)

この結果から，ランダムな追加の場合と比べ，集中追加時にはラベル長が極端に増大していることがわかる．一方，我々の提案する手法では，ラベルサイズはほとんど XML 木の形に依存するため，ノードの追加方法によってサイズが増加することはなくほぼ一定である．また，os テーブルの記憶コストもノード追加の順序に依存せず．各ノードの子ノード数にのみ依存するので，変化はない．

例えば，高さ 7 の XML 木の場合，ORDPATH と QED-PREFIX の最大ラベル長,平均ラベル長,ラベル長の総和は，それぞれ

ORDPATH　0.67[KByte],　0.17[Kbyte],　57.1[MByte]
QED-PR.　0.90[KByte],　0.23[Kbyte],　76.4[MByte]

であり，我々の提案手法の方は殆ど無視できるほどサイズは小さい．

6.3. 構築コストの測定

図 13 に，各ラベル付け手法における，構築時間コストを示す．使用データは集中的な追加を行ったときの表 1 の木である．

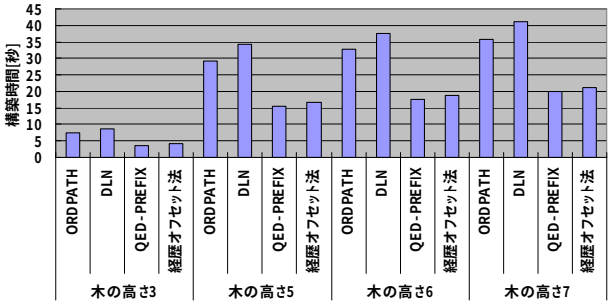

図 13　構築コスト

この結果では，我々の提案手法は DLN,ORDPATH には勝り，QED-PREFIX には劣っている．DLN,ORDPATH では集中追加によりラベルそのものの長さが増加するが，QED-PREFIX ではラベルを構成する QED コードのサイズが増加するため，ラベル値となるビット列の作成コストは比較的小さい．そのため，我々の手法は，配列の拡張や os テーブルの作成に伴うコストの分，QED-PREFIX と比べ不利になったと考えられる．

6.4. 検索コストの測定

図 14, 図 15 に，それぞれ child(子)・descendant(子孫)・preceding-siblings(兄集合)・following-siblings(弟集合)の検索時間を示す．使用データは集中的な追加を行ったときの表 1 の木である．

この検索結果より，我々の提案する手法は，他方式と比べ，特に子供や兄弟ノードの検索において有利となっていることがわかる．これは，他方式がラベルを文書順にたどって検索を行うのに対し，我々の手法は os テーブルを参照し，直接ラベル値を求めていることが原因と考えられる．他方式のこの欠点を回避するためには，適切な構造化索引を導入することによって，回避できるがそのための記憶コストが増大する．

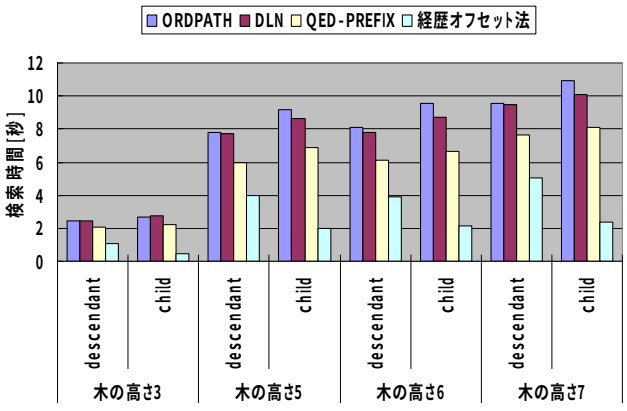

図 14　child 及び descendant 検索時間

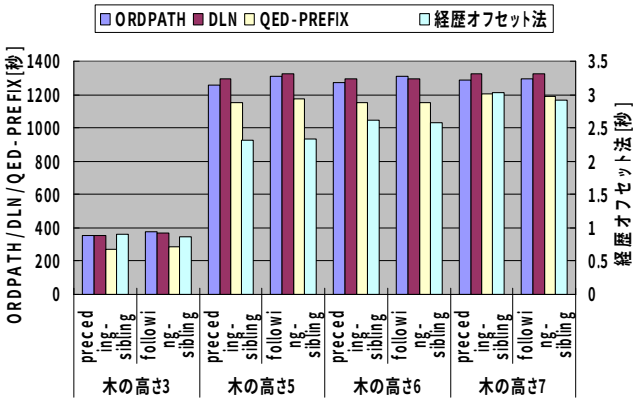

図 15　sibling 検索時間

7. むすび

本論文では，我々が提案している多次元データのエンコーディング方式である経歴オフセット法を用いた，構造更新に強い XML 木ノードに対するラベル付けの手法を提案した．他方式との記憶・検索コストの比較を行い，本方式が，特に一箇所へのノードの集中的な追加が起こる場合に記憶コストの面で有利となり，また，検索速度も高速であることを確認した．

文　献

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

[2] T. Gust, “Accelerating XPath Location Steps”, Proc. of ACM SIGMOD, pp.109-120, 2002.

[3] E. Cohen, H. Kaplan, and T. Milo, “Labeling Dynamic XML Trees”, Proc. of PODS, pp.271-281, 2002.

[4] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, C. Zhang, “Storing and Querying Ordered XML Using a Relational Database System”, Proc. of ACM SIGMOD, pp.204-215, 2002.

[5] X. Wu, M. L. Lee, and W. Hsu, “A Prime Number Labeling Scheme for Dynamic Ordered XML Trees”, Proc. of ICDE, pp.91-99, 2004.

[6] 清水敏之,鬼塚真,江田毅晴,吉川正敏: XML データの管理とストリーム処理に関する技術，電子情報通信学会論文誌　Vol.J90-D No.2 pp.159-184, 2007.

[7] Yong Kyu Lee, Seong-Joon Yoo and Kyougro Yoo: “Index Structures for Structured Documents”, Proc. of ACM Conference on Digital Libraries, pp.91-99, 1996.

[8] Wolfgang Meier, “eXist: An Open Source Native XML Database”, Proc. of Web, Web-Services, and Database Systems, pp.169-183, 2002,

[9] W.M. Meier, “eXist.”, http://exist.sourceforge.net/

[10] 佐藤隆士,里本智彦,小畑喜平，潘洪涛:階層構造を識別可能な木節点の番号付け，電子情報通信学会，データ工学ワークショップ DEWS2002, A4-8, 2002.

[11] O’Neil, P.E., O’Neil, E.J., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: Insert-friendly XML node labels. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD’04), pp. 903–908(2004)

[12] Li, C., Ling, T.W.: QED: a novel quaternary en-coding to completely avoid re-labeling in XML updates. In: Proceedings of the 14th International. Conference on Informatio and KnowledgeManagement (CIKM’05), pp. 501–508 (2005)

[13] B o h m e, T., E. R a h m. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. – In: Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb), 2004.

[14] 川口勝也，Bei LI，都司達夫，樋口健:構造更新に対応した XML 木ノードのラベル付けの一方式，電子情報通信学会，データ工学ワークショップ DEWS2008, C8-3, 2008.

[15] K. M. A Hasan, T. Tsuji, K. Higuchi, “An Efficient Implementation for MOLAP Basic Data Structure and Its Evaluation”, Proc. of DASFAA 2007, pp.288-299, 2007.

[16] E. J. Otoo and T. H. Merrett, “*A* storage scheme for extendible arrays”, Computing, Vol.31, pp.1-9, 1983.

[17] XML Path Language (XPath): http://www.w3.org/TR/xpath/

[18] XMark - An XML Benchmark Project: http://monetdb.cwi.nl/xml/

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