データベースシステム 128−31
（２００２．７．１８）

更新に強いXML節点数え上げ手法とその管理

江田 毅晴† 天笠 俊之† 吉川 正俊†† 植村 俊亮†

†奈良先端科学技術大学院大学情報科学研究科 〒630–0101 奈良県生駒市高山町8916-5
††名古屋大学 情報連携基盤センター 〒464–8601 名古屋市千種区不老町

あらまし 前置順と後置順を利用したXML節点数え上げ手法において, 効率の良い更新を可能にする節点番号管理を提案する. 節点に番号の対を用いてある範囲を与えることによって, その対の比較のみで先祖子孫関係を判定する手法を, 範囲数え上げ手法という. 大量かつ大容量のXMLデータが既に登場しつつあり, 近い将来頻繁に更新が起きる状況でのXMLデータの利用が予想される. 節点番号に割り当てることの出来る容量は有限であり, その中で更新に対して節点間の関係を維持するには, 番号の付け直しが必要になる. 本研究は, この番号の付け直しに焦点を当て, 問合せだけではなく更新も可能なXML木ラベル付手法の基礎を与える. まず, 前置順と後置順を間隔を空けて数えることによって, 更新に備える. 更新が起こった際には, 更新箇所周辺の節点番号間の間隔の分布が平衡状態を保つように節点番号をずらす. これにより, 節点を全て数え直すことを避けながらXMLデータを更新し続けることを可能にする.

A robust XML Node Numbering Scheme and its Management

Takeharu EDA†, Toshiyuki AMAGASA†, Masatoshi YOSHIKAWA††, and Shunsuke UEMURA†

†Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630–0101 Japan
††Information Technology Center Nagoya University Furo-cho, Chikusa-ku, Nagoya
464–8601 Japan

Abstract This paper proposes a scheme for XML node numbers based on pre- and postorders, that enables efficient updating. We exploit the **Range Labeling**, where each node in an XML tree is assigned a range, consisting of two numbers, and the ancestor-descendant relationship between two nodes can be detected by the labels. Huge and many XML data have already appeared, and in time to come, we make use of such XML data that would be updated frequently. Because spaces available for **ranges** are limited, when updating XML data, we need **renumbering** in order to retain relationships among node numbers. In this paper, we concentrate on the issue of **renumbering**. First, we make the basis for update by giving sparse numbers to nodes. Next, by managing node numbers, it become possible to update XML data persistently. Our method will be the basis of full-feachered XML Labeling schemes.

1 はじめに

アプリケーションの設定ファイルや企業の顧客情報, あるいはウェブページに至るまで, 今までそれぞれ独自に設計されていた様々なデータが, XML(Extensible Markup Language)[2] という共通のフォーマットを利用して記述されはじめている. 急激に増えつつある多様なXMLデータを効率よく管理, 運用するには, XMLデータベースが不可欠である.

XMLデータベースの要件として, XMLデータを効率良く格納し, 内容に関する検索及びXMLの持つ構造を利用した検索を提供するこ

とがあげられる. さらに, よりデータベースとしての機能を追い求めるのなら, 更新操作も実現することが必要である. しかしながら, 構造問合せを高速化する格納手法は確立されつつあるものの, それらの手法の更新への対応については現状ではそれほど研究されていない. このことから, XML データベースにおける更新操作の実現, あるいは実現可能性を探ることは現時点での課題であると言える.

一般に, XML データを格納する際には, 問合せ処理高速化のために様々な手法で XML データ中の各節点にラベル付けをおこなうことが多い. いくつかのラベル付手法が提案されているが, 代表的なものの一つとして範囲数え上げ手法 (Range Labeling) があげられる [6]. 範囲数え上げ手法では, 節点のラベルに番号の対を用いる. 前置順, 後置順によって決まる範囲を与えることによって, その対の比較のみで先祖子孫関係を判定することができ, これにより, XPath[5] のような正則パス表現を含む問合せの処理を高速化することが可能となる.

範囲数え上げ手法を更新に対応させる一つの方法として, 節点番号を間隔をあけて数えることによって, 挿入を可能にする手法がある. しかし, 範囲に割り当てられるビット数は有限であるため, 挿入, 削除の繰り返しにより番号が偏った場合には挿入を続けることが不可能になる. その結果, 大規模な節点番号の付け直しが必要になる. また, XML データが大量にあり, かつ大容量化していることを考えると, 格納効率, 処理効率両面を向上させるために範囲を割り当てるのに用いるビット数は, 出来るだけ少ないほうが良い. ビット数が少ない際には, 当然ながら番号の付け直しをおこないながら, XML データへの問合せを処理しなければならない.

本研究は, XML データ全体に渡る数え直しを回避するために, 節点番号の付け直しに焦点を当てた. 挿入及び削除をおこなうときに周辺の番号の混み具合を判定して, 常に間隔の均衡が保たれるように周辺の番号をふり直す. これによって後々の更新操作に柔軟に対応することが出来る.

本稿の構成は次の通りである. 2 節では関連研究, 3 節では範囲数え上げ手法とその問題点及び XML の更新操作について述べ, 4 節ではこの問題を回避するための節点番号管理について述べる. 5 節では具体的なアルゴリズムの動きについて述べ, 最後に 6 節で本稿をまとめ, 今後の展望について述べる.

2 関連研究

前置順と後置順の対によって木構造における節点間の先祖子孫関係を判定する手法は, Dietz[7] による. この手法を Li 等 [10] は XML に適用した. 前置順と後置順の性質は, 全文索引にみられる単語の先頭からの順番 [15] や, 先頭からのバイト数 [14] などにもあらわれるため, XML データベースに関する研究の文脈で同様の構造判定手法が用いられている. [10] では, 間隔を空けて前置順を数える拡張前置順を導入し更新に対応した. しかし, 間隔を空けただけでは多くの挿入が起きた際に破綻が起きる. それを回避するために我々は節点番号管理を提案する.

前置順と後置順を利用した正則パス表現を含む XPath 式 [5] 評価アルゴリズムが多数研究されている [1, 3, 4]. これらは, 構造結合 (Structural Join) あるいは, 小枝結合 (Twig Join) と呼ばれ, 3 節の定理 2 の不等式二つと等式を含む XML に特有の結合操作である. 関係データベースでこれらの結合操作を実行させた場合, 最適なアルゴリズムを適用しないことが分かっており [15], 将来的にこれらの XML 問合せ処理を高速化するアルゴリズムが関係データベースに実装されることが予想される. また [3] では, 前置順と後置順に対して XB-tree と呼ばれる B-tree を改良した索引を構築することによって, 構造結合のさらなる高速化をはかっている.

[6] では, 範囲数え上げ手法だけでなく接頭辞ラベル付手法なども含めた一般的なラベル付手法を議論し, 静的, 動的両方の場合において, 最小容量のラベルを与えるアルゴリズムを提案している. 実装する際には, 範囲に使えるビット数は限られているため, 我々の手法はそうしたアルゴリズムを利用した際にも, 使える領域を不均衡に使い果たした際の数え直しの指針を与えるものとなる.

我々の手法は番号をふり直すことによって, 外部からの参照や索引に対する耐参照性 (durable reference)[13] を失うことになる. しかし節点番

号上に張られる B-tree や R-tree といった索引に関しては, 提案する節点番号管理では順序関係を保存する形で番号をふり直すので, 再構成の際のコストは小さいものと予想される. また, 文書の版管理のような永続的な外部参照が必要な場合もあるが, 我々は続々と挿入, 削除が起こるような状況を想定している. その場合は, 外部参照はあまり重要ではない. また番号に当てられるビット数が限られる状況を想定すると, いかに少ないビット数で効率的に XML データを管理するかが重要になってくる. 提案手法は永続的な耐参照性を失うかわりに, 容量内であれば永続的な更新を可能にしている. 永続的な外部参照が必要な場合は別に識別子を用意することも考えられる.

3 範囲数え上げ手法

XML データは, ラベル付順序木 [12] とみなせる. 順序木とみなして議論するときには, XML データを **XML** 木と呼ぶことにする. 更に, 順序木の部分木に対応させて XML 木の部分木を部分 **XML** 木と特に呼ぶ. 部分 XML 木は XML 木である. 範囲数え上げ手法では, XML 木中の節点を前置順及び後置順で数えており, その際に間隔をあけて数えることによって, 後々の更新操作に対応する基盤を与える. 議論を分かり易くするため, 以後 XML 木の各節点の型は区別しないものとする. つまり要素節点も, 文節点も一つの節点としてそれぞれ唯一に番号をふる. 属性節点は今回は無視することにする.

T を XML 木とする. T が n 個の節点から成ることを $|T| = n$ と書く. T の根節点を $root(T)$ で表す. 節点 e の前置順を $pre(e)$, 後置順を $post(e)$ で表す. また, 根節点から節点 e への距離を節点 e の深さという. $root(T)$ に関しては, 次の定理が成り立つ.

定理 **1** *XML* 木 T 中の全ての節点の中で $root(T)$ が最も前置順の値が小さく, 後置順の値が大きい.

次に節点番号を定義する.

定義 **1** 節点番号
XML 木 T において, *(pre(e), post(e))=(i,j)* で

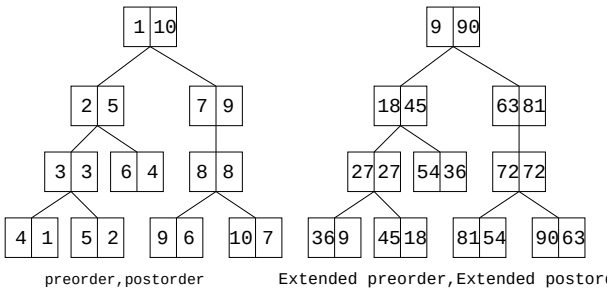

図 1: 前置順, 後置順及び拡張前置順, 拡張後置順の例

ある節点 e の節点番号は,

$$(a_i, b_j)$$

とする. 但し, a_i, b_j はそれぞれ i, j に対して狭義単調増加数列である.

a_i,b_j をそれぞれ拡張前置順, 拡張後置順と呼び, $epre(e)$, $epost(e)$ で表す. 節点に節点番号を与えるとは, 図 1 のように XML 木中の節点を前置順及び後置順に何らかの間隔を空けて数えることである.

T 中の全ての節点に節点番号を与えることにより, T の拡張前置順列を $(a_1, a_2, a_3, \ldots, a_{n-1}, a_n)$ と定義する. 同様に T の拡張後置順列は $(b_1, b_2, b_3, \ldots, b_{n-1}, b_n)$ と定義される. 今後, この節点番号と拡張前置順列及び拡張後置順列を用いて節点番号管理をおこなう.

定理 **2** 包含定理
XML 木 T 中の任意の節点 $x = (a_i, b_j), y = (a_k, b_l)$ に対して次の条件を満たす時のみ, x は y の先祖となる.

$$a_i < a_k, \text{かつ } b_j > b_l$$

これは前置順と後置順の順序関係を拡張前置順と拡張後置順においても保存していることから明らかである. この定理を用いることによって, 定数時間で任意の節点間の先祖子孫関係が判定可能になる. 包含定理を用いた構造結合により, 正則パス表現による問合せの処理を高速化することが出来る. さらに, 深さを用いることによって, 親子関係や固定長の先祖子孫関係を判定することも可能になる.

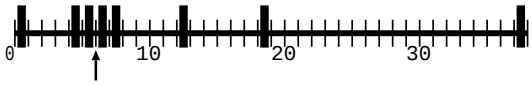

図 2: 不均衡に番号を使い尽くして矢印のところに挿入出来ない例

3.1 問題点

拡張前置順, 拡張後置順では, 間隔を空けてXML 木中の節点を数えているため, 間隔が空いている箇所にはそのまま XML 木を挿入することが出来る. しかし実際には, 節点番号が固定であるため, 隣り合う拡張前置順（拡張後置順）の間隔以上の節点を挿入することが出来ない. また図 2 のように挿入, 削除を繰り返すことによって, 番号が不均衡に使い尽くされてしまうこともある. この場合, XML 木全体に渡る節点番号の数え直しを行なう必要がある.

このような大規模な節点番号の数え直しを避けるために, 4 節で提案する節点番号管理を行なう.

節点番号管理の説明をする前に, XML 木の更新操作について説明する.

3.2 XML の更新操作

XML 木の更新操作には, 下記の基本的な操作が必要とされる [9] [11].

- 挿入 (n, k, X) $\cdots$ 節点 n の k 番めの子供に XML 木 X を挿入する.
- 削除 (X) $\cdots$ 部分 XML 木 X を削除する.
- 修正 (n, v) $\cdots$ 節点 n の値を v に更新する.
- 移動 (n, k, X) $\cdots$ 部分 XML 木 X を節点 n の k 番めの子供に移動する.

このうち, 修正は XML 節点の名前及び値を修正するが, その際には XML 木の構造は変化しないので, 節点番号は修正する必要がない. また移動は挿入と削除によって実現可能でもあるので, 今回は考慮しないことにした. よって, 挿入と削除について考える.

ここで, 挿入と削除の基本単位は XML 木とする. XML 木及びその部分 XML 木に関しては図 3 のように, 次の性質が成り立つ.

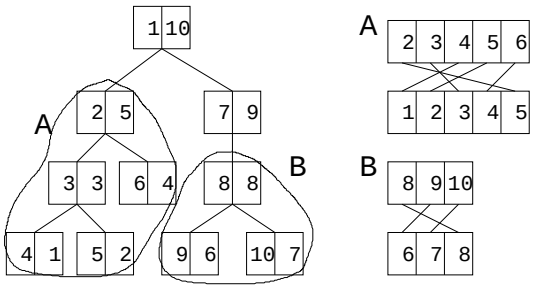

図 3: 部分 XML 木は連続した前置順, 後置順を持つ

定理 3 *任意の挿入する XML 木及び削除する部分 XML 木は連続した前置順及び後置順を持つ.*

定理 3 により, 挿入する際には, 拡張前置順列および拡張後置順列のある一箇所の間隔に XML 木, つまり番号列を挿入すればいいことが分かる. また, 削除に関しても, ある連続した部分拡張前置順列および部分拡張後置順列を削除すれば良い.

4 節点番号管理

我々の提案する節点番号管理の基本的流れを説明する. XML 木への操作がおこなわれると, まず節点番号が平衡状態であるかどうかを, 更新箇所周辺の節点番号の分布から確認する. 次に, (1) 挿入操作の場合, 挿入先の拡張前置順（拡張後置順）の間隔が狭ければそれを広げて挿入をおこなう. (2) 削除操作の場合, 削除によって生じる間隔が広過ぎればそれを狭める. これによって常に節点番号の間隔の平衡状態を保つことにより, 後々の挿入操作にも柔軟に対応することが出来る.

節点番号の均衡の確認及び調整をするタイミングについては種々考えられるが, 今回は最も基本的なタイミングとして一括読み込み, 挿入, 削除の際にアルゴリズムを適用することを想定した.

スキーマや類似した XML データの統計情報等が利用できる場合が考えられるが, 今回は利用しないものとした. 情報を利用した状況に我々の手法を適用するのも容易と考えられる. 逆に, スキーマ情報が利用出来なくても, 常に間隔の均衡を保つので, 理論上は節点数が利用

可能な節点番号の最大値を越えない限り次々と起きる連続した挿入に対応することが出来ることが提案手法の利点である.

$Width$ を使用可能数の最大値とする. $T(|T| = n)$ を更新される XML 木, $ExPreList(T)$, $ExPostList(T)$ をそれぞれ T の拡張前置順列, 拡張後置順列, $X(|X| = m)$ を挿入 XML 木とする. 使える数値の幅を等分割した幅 $BestInterval\left(:= \left[\frac{Width}{|T|+|X|+1}\right]\right)$ 間隔で番号がふられている時を最も良い均衡状態とする. 挿入可能条件として, この幅が 1 以上でなければならない.

定義 **2** 挿入可能
$BestInterval \geq 1$ の時, 挿入可能とする.

$BestInterval$ に対して, ある許容範囲までは, 均衡が保たれているとして数え直しをしない. この定数を min, max とする. $0 < min \leq 1, 1 \leq max$ である.

定義 **3** 平衡
現在注目している間隔 x が, 次の条件を満たしている時, x は平衡であるという.

$$min \cdot BestInterval < x < max \cdot BestInterval$$

min, max を今回は定数にしたが, スキーマ情報や XML データの統計情報及び挿入箇所を引数にとる関数とすることによって, 更新が頻繁に起きる箇所の情報を節点番号の分布状態に反映させることも可能である.

4.1 一括読み込み

一つの XML データを最初に静的に数え上げる, 一括読み込みアルゴリズムについて説明する. 単純に節点数で使える番号の領域を割って前置順, 後置順にそれぞれ番号を付ける.

Algorithm *BulkLoad*
入力 : XML 木:T
出力 : $ExPreList(T), ExPostList(T)$
1 $BestInterval := \left[\frac{Width}{|T|+1}\right]$;
2 **if**(挿入可能でない) **exit**;
3 $b := BestInterval$;
4 **return**
$ExPreList(T) := (b, 2b, 3b, \ldots, |T| \times b)$,
$ExPostList(T) := (b, 2b, 3b, \ldots, |T| \times b)$;

4.2 挿入アルゴリズム

ここでは, 拡張前置順列についてのみ説明するが, 拡張後置順列についても同様である.

挿入する際のアルゴリズム $Insert$ は下記のようになる.

Algorithm *Insert*
入力 :拡張前置順列 : $\mathbf{T}$, 挿入 XML 木の節点数 : m, 挿入箇所の直前の前置順 : k
出力 :挿入を完了した $n+m$ 個の拡張前置順列
1 **if** （挿入可能でない） **exit**;
2 **if** $(k = |T|)$ $b := \dfrac{Width - a_k}{m+1}$;
3 **if** $(k = 0)$ $b := \dfrac{a_1}{m+1}$;
4 **else** $b := \dfrac{a_{k+1} - a_k}{m+1}$;
5 **if** (b が平衡でない)
6 **if**$(k = 1)$
7 $\mathbf{T} := (a_3, \ldots, a_{n-1}, a_n)$;
8 $Insert(\mathbf{T}, m+2, 1)$;
9 **if**$(k = |T|)$
10 $\mathbf{T} := (a_1, a_2, \ldots, a_{n-3})$;
11 $Insert(\mathbf{T}, m+2, |T|)$;
12 **else**
13 $\mathbf{T} := (a_1, a_2, \ldots, a_{k-1}, a_{k+2}, \ldots, a_n)$;
14 $Insert(\mathbf{T}, m+2, k-1)$;
15 **else**
16 **return** $\mathbf{T} := (a_1, a_2, \ldots, a_k, a_k + [b], a_k + 2[b], \ldots, a_k + m[b], a_{k+1}, \ldots, a_n)$;

ここでは, 入力として挿入箇所を特定する変数として k を入れた. k は, 挿入箇所の間隔を定める拡張前置順の少ない方の前置順である. この k を求めるのはそう簡単ではない. 挿入する際には, 親節点からみて何番目の子節点かを特定する必要があるが, 今回議論している範囲数え上げ手法では, この XPath における $position$ 関数に相当する問合せの高速化を実現することが出来ないからである. 子 XML 木を先頭から全てみていけば, 検索することは可能であるが, 非常にコストがかかるものと予想される. 解決策としては, 兄弟節点同士をリンクでつなげることが考えられるが, 詳細については今後の課題である.

4.2.1 挿入アルゴリズムの改良

上記では挿入箇所の間隔を一つ一つ両脇に広げて平衡状態かどうかを判定しながら挿入操作

を実現したが, 挿入 XML 木 X の節点数からあらかじめ節点間の間隔を見積もることが出来る. すなわち, 挿入先の間隔が $(m+1)\cdot min\cdot BestInterval$ だけは最低限必要であるので, その幅を先にとってから番号付けを行なう.

Algorithm *Insert'*
入力 :拡張前置順列 : **T**, 挿入 XML 木の節点数 : m, 挿入箇所の直前の前置順 : k
出力 :挿入を完了した $n+m$ 個の拡張前置順列
1 **if** （挿入可能でない） **exit**;
2 $a_{max} := a_k + \frac{(m+1)\cdot min\cdot BestInterval}{2}$;
3 $a_{min} := a_k - \frac{(m+1)\cdot min\cdot BestInterval}{2}$;
4 **if**$(a_{max} > Width)$
5 $a_{min} := Width-(m+1)\cdot min\cdot BestInterval$;
6 $a_{max} := Width$;
7 **if**$(a_{min} < 0)$
8 $a_{min} := 0$;
9 $a_{max} := (m+1)\cdot min\cdot BestInterval$;
10 $l := \{[a_{min}, a_{max}]$ に含まれる番号の個数 $\}$;
11 $i := max(a_i)\{$ 但し, $a_i < a_{min}\}$;
12 **T** $:= (a_1, a_2, \ldots, a_i, a_{i+l+1}, \ldots, a_n)$;
13 $Insert(\mathbf{T}, m+l, i)$;

このアルゴリズムを実装するには, 間隔木索引等を使って範囲に対する検索を高速化することが考えられる.

4.3 削除アルゴリズム

削除は, 拡張前置順列と拡張後置順列を往復する必要があるため, 引数は XML 木 T と削除する部分 XML 木 X になる. 番号を削除した後に発生した間隔が, 平衡かどうか判定し数え直しをするが, その際には $Insert$ アルゴリズムの挿入節点の数を 0 とすることによって実現している.

Algorithm *Delete*
入力 : XML 木 T, 削除する部分 XML 木 X
出力 : 削除後の T の拡張前置順列, 拡張後置順列
1 $DeleteNumber(T, root(X))$;//$root(X)$ 以下の番号を消す.
2 $Insert(ExPreList(T), 0, pre(root(X))-1)$;
3 $Insert(ExPostList(T), 0, post(root(X))-1)$;

$DeleteNumber(T, d)$ は, $ExPreList(T)$, $ExPostList(T)$ それぞれの d 節点以下の番号を削除する. 片方のみでは, 削除する範囲が分からないため定理 1 及び定理 3 を用いて同時に処理している.

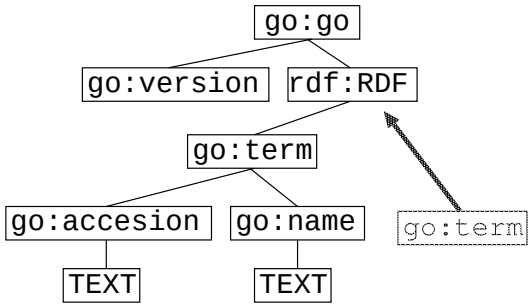

図 4: XML 木 T

Algorithm *DeleteNumber*(T, d)
1 $i := pre(d)$;
2 $j := post(d)$;
3 $b_{max} := epost(d)$;
4 **while** $(0 < b_j \leq b_{max}$かつ, $i \leq n)$
5 $deletenodePre(i)$;
6 $i++$;
7 $j :=$ 前置順が i の節点の後置順;

$deletenodePre(i)$ は, 前置順が i 番めの節点番号（前置順, 後置順も含めて）を削除する関数である. 上記の通り, $DeleteNumber$ は, 拡張前置順列と拡張後置順列を往復するので, 実装上は往復を高速化するファイル構造等を用いて工夫する必要がある.

以上が節点番号管理を構成するアルゴリズムになる.

5 アルゴリズムの具体的動作

[8] で公開されている XML データを利用して, 提案手法の具体的動作を説明する. 今, 図 4 のように XML 木 T が与えられたとする.

5.1 一括読み込み

$Width = 500$, $min = \frac{1}{2}$, $max = 2$, $|T| = 8$ とする.

$$BestInterval = \left[\frac{500}{8+1}\right] = 55$$

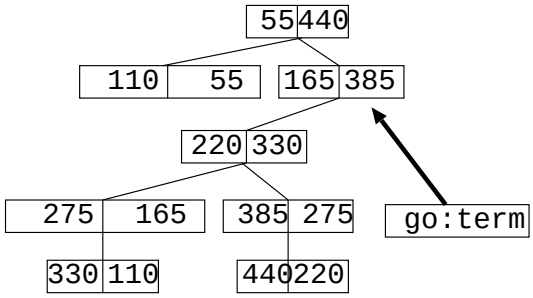

図 5: 初期節点番号

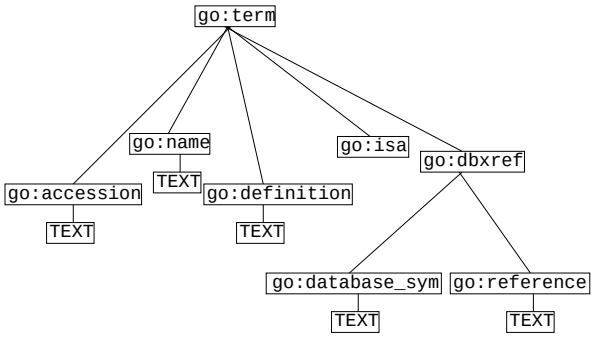

図 6: 挿入 XML 木

であるので,

$$\begin{aligned} ExPreList(T) &= (55, 110, \ldots, 440) \\ ExPostList(T) &= (55, 110, \ldots, 440) \end{aligned}$$

と図 5 のように初期節点番号を与える.

5.2 挿入

点線で示された部分に go:term 要素 (図 6) が挿入されるものとする. 挿入箇所の左端を与える前置順及び後置順は, $k = 8$ (前置順), $k = 7$ (後置順) となる. $|X| = 13$, $\mathbf{T} = ExPreList(T)$ として
$Insert(\mathbf{T}, 13, 8)$ を適用すると,

$$BestInterval = \left\lceil \frac{500}{8 + 13 + 1} \right\rceil = 22 \geq 1$$

であるので, 挿入可能である. 今, $|T| = 8 = k$ より,

$$b = \frac{500 - 440}{14} = 4 < min \cdot BestInterval = 11$$

より平衡でないので,
$\mathbf{T} = (55, 110, , \ldots, 300)$ として
$Insert(\mathbf{T}, 15, 6)$ を呼ぶ.

$$b = \frac{500 - 330}{16} = 10 < 11$$

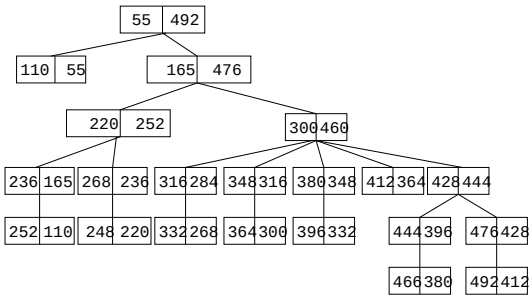

図 7: 挿入後

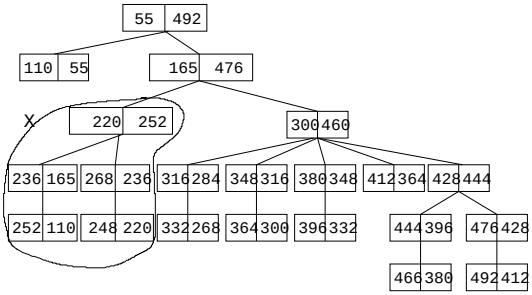

図 8: 削除部分 XML 木

より平衡でないので,
$\mathbf{T} = (55, 110, 165, 220)$ として
$Insert(\mathbf{T}, 17, 4)$ を呼ぶ.

$$\begin{aligned} max \cdot BestInterval &= 44 > \\ b &= \frac{500 - 220}{17} = 16 > 11 \end{aligned}$$

より平衡となる. よって
$\mathbf{T} = (55, 110, 165, 220, 220 + 16, 220 + 2 \times 16, \ldots, 220 + 17 \times 16)$ を返し, 挿入が完了する. $ExPostList(T)$ についても同様であり, 図 7 のようになる.

5.3 削除

図 8 が示す XML 部分木 X を削除する. $|T| = 21$, $|X| = 5$ である.

$Delete(T, X)$ を適用すると,
まず, $DeleteNumber(T, root(X))$ を呼びだし,
$i := pre(d) = 4, j := post(d) = 6$,
$b_{max} := epost(d) = 252$ であり,
$deletenodePre(4)$ で $(220, 252)$ を消去し, 次に $i = 4 + 1 = 5$, $j = 3$ とし, $b_3 = 165 \leq b_{max}$, $5 \leq 21$ であるので, $deletenodePre(5)$ で $(236, 165)$ を消去する. 次に $i = 6$, $j = 2$ となり, $b_2 = 110 \leq b_{max}$, $6 \leq 21$ であるの

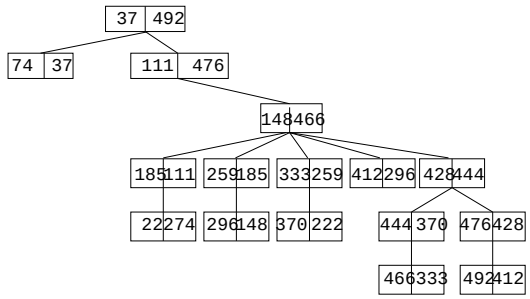

図 9: 削除後

で, $deletenodePre(6)$ で, $(252, 110)$ を消去する. 次に $i = 6+1 = 7$, $j = 5$ とし, $b_5 = 236 \leq b_{max}$, $7 \leq 21$ であるので, $deletenodePre(7)$ で $(268, 236)$ を消去する. 次に $i = 7+1 = 8$, $j = 4$ とし, $b_4 = 220 \leq b_{max}$, $8 \leq 21$ であるので, $deletenodePre(8)$ で $(284, 220)$ を消去する. 次に $i = 8+1 = 9$, $j = 19$ とするが, $b_{19} = 460 > b_{max}$ であるので, **while** ループが止まり, $DeleteNumber(T, root(X))$ が終了する. 次に, 発生した間隔を調整するため, $Insert(ExPreList(T)), 0, 3)$ 及び, $Insert(ExPostList(T)), 0, 1)$ を呼びだし, 調整した後, 結果を出力する (図 9).

6 まとめ

本稿では, XML データベース上で効率の良い更新を実現するため, XML 索引技術の一つである範囲数え上げ手法における節点番号の管理手法を提案した. 提案手法では, 挿入及び削除をおこなうときに周辺の番号の混み具合を判定して, 常に間隔の均衡が保たれるように周辺の番号をふり直す. これによって後々の更新操作に柔軟に対応することが出来る.

今後の課題として, 提案手法の実装及び評価が挙げられる.

参考文献

[1] Shurug Al-Khalifa, H. V. Jagadish, et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE Conference*, 2002.

[2] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, et al. Extensible Markup Language (XML) 1.0 (Second Edition).

[3] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD Conference*, 2002.

[4] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed xml documents. In *Proceedings of the 28th VLDB Conference*, 2002.

[5] James Clark and Steve DeRose. XML Path Language(XPath) version1.0.

[6] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML Trees. In *PODS*, 2002.

[7] Paul F.Dietz. Maintaining order in a linked list. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pp. 122–127, 1982.

[8] http://www.geneontology.org. Gene ontology consortium.

[9] Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language.

[10] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *The VLDB Journal*, pp. 361–370, 2001.

[11] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S̃. Weld. Updating XML. In *SIGMOD Conference*, 2001.

[12] Alfred V.Aho, John E.Hopcroft, and Jeffrey D.Ullman. Data Structures and Algorithms.

[13] Shu-Yao Chien Vassilis. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In *Proceedings of The 2nd International Conference on Web Information Systems Engineering*, pp. 270–279, 2001.

[14] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel:A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases.

[15] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD Conference*, 2001.