DEIM Forum 2017 I2-2

木文法に基づき圧縮されたXML文書に対する データ値を考慮した直接更新法

高山隆之介[†]　　橋本　健二[†]　　関　　浩之[†]

† 名古屋大学 大学院情報科学研究科　〒 464–8603 名古屋市千種区不老町

E-mail: †takayama@trs.cm.is.nagoya-u.ac.jp, {k-hasimt, seki}@is.nagoya-u.ac.jp

あらまし　XML 文書の圧縮法の 1 つとして木文法に基づく手法が知られている．また，その手法による圧縮文書に対して解凍せずに更新を行う手法が提案されている．しかし，これらの手法では要素の階層構造のみに注目しているため，要素がもつデータ値は考慮されておらず，更新位置の指定には構造に関する条件しか利用できない．本稿では，木文法を用いた既存の圧縮法および直接更新法を拡張し，構造情報だけでなく，データ値の参照，およびそれらに依存した構造やデータ値の更新が可能な手法を提案する．また，提案手法の評価実験として，更新に要した時間・メモリについて計測し，代表的な XML データベース管理システムの 1 つである BaseX 上での更新との比較を行った結果について報告する．

キーワード　XML, 木オートマトン, 木文法, 圧縮, 更新

1. まえがき

XML は構造化データの一つであり，タグによって構造を指定するデータの交換・蓄積書式である．XML 文書には同じ構造が頻出することが多く，サイズが大きくなりやすい．そのため，XML 文書に対する圧縮法が盛んに研究されている [4] [5] [8] [9] [10].

圧縮された XML 文書に対して更新を行う場合，一度解凍してから更新し，再圧縮するという方法は実行時間，使用メモリにおいて非常に非効率的である．Straight-Line Context-Free Tree Grammar (SLCFTG) [5] に基づいて圧縮された XML 文書は圧縮された状態のまま走査が可能であり [8]，その性質を利用して解凍せずに直接更新を行う手法が提案されている [12] [13] [14].

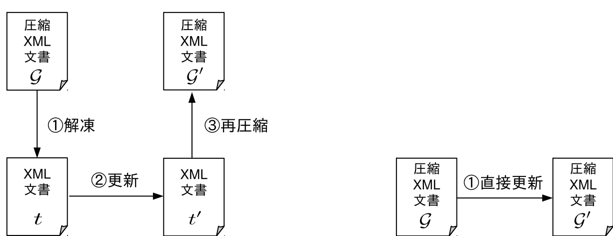

図 1　解凍 → 更新 → 再圧縮

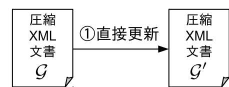

図 2　直 接 更 新

圧縮したまま処理を行うことで，同じ構造をもつために共有されている部分に対して重複した処理を回避することができ，効率よく更新を行うことができる (図 1，2)．しかし，それらは XML 文書に含まれるデータ値（属性値や CDATA）は考慮されていなかった．具体的に，更新位置を指定する際には構造に関する条件のみしか指定できず，文書に含まれているデータ値に関しての条件を指定することができなかった．また，データ値自体を更新することもできなかった．

本稿では，既存手法 [14] で用いられていた更新位置を指定するための選択木オートマトンや更新位置の検索アルゴリズム，更新アルゴリズムを拡張することで，構造情報だけでなくデータ値の参照，およびそれらに依存した構造やデータ値の更新が可能な手法を提案する．また，データ値を扱うための工夫としてデータ値部分に関する情報の分割を行い，高速化を図る．また，いくつかの XML 文書に対して提案手法に基づいて実装したシステムを用いて評価実験を行った．更新に要した時間・メモリ量について計測し，代表的な XML データベース管理システムの 1 つである BaseX [1] 上での更新との比較結果について報告する．

以降の構成は次の通りである．まず 2 節では諸定義を与える．3 節では XML 文書の圧縮について述べ，4 節では想定する更新ならびに直接更新を行うアルゴリズムについて説明する．5 節では実際の XML 文書に対する実験の結果を述べる．6 節ではまとめと今後の展望について述べる．

2. 準　　備

2.1 データ木

ランクとよばれる自然数をもつ記号の有限集合 Σ をランク付きアルファベットとよぶ．記号 $a \in \Sigma$ のランクを $\mathrm{rank}(a)$ と表記し，$\Sigma_n = \{a \in \Sigma \mid \mathrm{rank}(a) = n\}$ とする．Att を属性とよばれる記号の集合とする．

［定義 2.1］ (Σ, Att) 上のラベル付き順序データ木（以降，データ木）は以下を満たす 3 項組 $t = (D, \lambda, \delta)$ である．

（1） $D \subseteqq \mathbb{N}_+^*$．ただし，$\mathbb{N}_+ = \{1, 2, 3, \ldots\}$.

（2） λ は D から Σ への全域関数である．

（3） δ は $D \times Att$ から $\mathbb{N}$ への部分関数である．

（4） $w = vv' \in D$ かつ $v, v' \in \mathbb{N}_+^*$ のとき，$v \in D$.

（5） $v \in D$ かつ $\lambda(v) \in \Sigma_n$ のとき，$1 \leqq i \leqq n$ である全ての i について $vi \in D$. □

D の要素を t の頂点（または位置）とよぶ．λ をラベル付け

関数，δ をデータ値関数とよぶ．頂点 $v, vi \in D$（$v \in \mathbb{N}_+^*, i \in \mathbb{N}_+$）について，$vi$ を v の i 番目の子頂点，v を vi の親頂点とよぶ．子頂点をもたない頂点を葉とよぶ．また，ϵ を t の根頂点という．頂点 $v \in D$ を根頂点とする t の部分木を t_v と表す．$\delta(v, a)$ が定義されていてかつ $\delta(v, a) = d$ のとき，頂点 v は属性 a に関するデータ値 d をもつという．(Σ, Att) 上のデータ木からデータ値を除いたものを Σ 上の木とよぶ．Σ 上の木 t は，データ木の定義から δ を除いて，$t = (D, \lambda)$ のように 2 項組で表す．

XML 文書は，頂点のラベルによってその子頂点の数が固定されないランクなしラベル付き順序木でモデル化できる．ランクなし木は fcns 符号化 [6] によって 2 分木に符号化できることが知られている．fcns 符号化によって，(Σ, Att) 上のランクなしデータ木を $(\Sigma \cup \{\#\}, Att)$ 上の 2 分木に符号化できる．ここで，# は rank(#) = 0 かつ Σ に含まれない記号であり，2 分木に符号化された木における Σ 中の記号 a はすべて rank(a) = 2 である．以下の図 3 のランクなし木に fcns 符号化を施すことで，図 4 の 2 分木を得ることができる．以降，fcns 符号化によって 2 分木化されたデータ木を扱う．

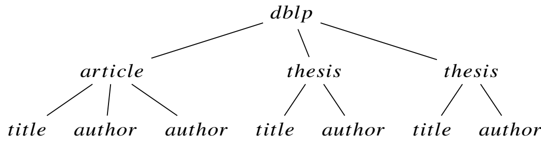

図 3　ランクなし木の例

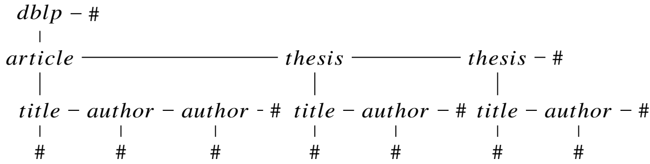

図 4　fcns 符号化された 2 分木の例

2.2 データ木オートマトン

本稿で扱うデータ木オートマトンは，ボトムアップ先読み付き決定性選択トップダウン木オートマトン (Deterministic Selecting Top-down tree automaton with Bottom-up look-ahead, DSTA^B) [14] にデータ値の比較を追加したものである．

［定義 2.2］ 2 分木上の**データ木オートマトン**は 8 項組 $\mathcal{A} = (\Sigma, Q_b, Q_t, q_{b0}, q_{t0}, \Delta_b, \Delta_t, \mathcal{S})$ である．ここで，

- Σ はランク付きアルファベット
- Q_b, Q_t は状態の有限集合
- $q_{b0} \in Q_b, q_{t0} \in Q_t$ は開始状態
- Δ_b は $w(q_{b1}, q_{b2}) \rightarrow (a, \varphi)?q_T : q_F$ という形のボトムアップ遷移規則の有限集合，ただし，$w \in \Sigma, q_T, q_F, q_{b1}, q_{b2} \in Q_b, a \in Att, \varphi$ は頂点のもつ属性 a のデータ値に関する述語
- Δ_t は $(w, q_b, q_t) \rightarrow (q_{t1}, q_{t2})$ という形のトップダウン遷移規則の有限集合，ただし，$w \in \Sigma,\ q_t, q_{t1}, q_{t2} \in Q_t, q_b \in Q_b$
- $\mathcal{S} \subseteq \Sigma \times Q_b \times Q_t$ は選択指定の有限集合
- Δ_b, Δ_t それぞれにおいて，任意の異なる 2 つの遷移規則は同一の左辺を持たない □

［定義 2.3］ $\mathcal{A} = (\Sigma, Q_b, Q_t, q_{b0}, q_{t0}, \Delta_b, \Delta_t, S)$ をデータ木オートマトンとする．(Σ, Att) 上のデータ木 $t = (D_t, \lambda_t, \delta)$ に対して，$\Sigma \times Q_b \times Q_t$ 上の木 $r = (D_r, \lambda_r)$ が以下の条件を満たすとき，r は $\mathcal{A}$ による t の受理実行木であるという．

- $D_r = D_t$
- 各頂点 $p \in D_t$ は以下を満たす．

– p が葉であるとき，ある状態 $q_t \in Q_t$ が存在して $\lambda_r(p) = (\#, q_{b0}, q_t)$．

– p が葉でないとき，$\lambda_t(p)(q_b, q_t) \rightarrow (q_{t1}, q_{t2}) \in \Delta_t$ と $(\lambda_t(p), q_{b1}, q_{b2}) \rightarrow (a, \varphi)?q_T : q_F \in \Delta_b$ が存在して，

$$
\begin{aligned}
&\lambda_r(p) = (\lambda_t(p), q_b, q_t) \\
\wedge\ &\exists w_1.\lambda_r(p1) = (w_1, q_{b1}, q_{t1}) \\
\wedge\ &\exists w_2.\lambda_r(p2) = (w_2, q_{b2}, q_{t2}) \\
\wedge\ &(\varphi(\delta(p, a)) \Rightarrow q_b = q_T) \wedge (\neg\varphi(\delta(p, a)) \Rightarrow q_b = q_F)
\end{aligned}
$$

□

ボトムアップ遷移規則について，その右辺にある φ が恒真である場合，$w(q_{b1}, q_{b2}) \rightarrow q_T$ のように略記する．$\mathcal{A}$ は決定性であるので，$\mathcal{A}$ による t の受理実行木は葉頂点のラベルの第 3 成分を除き高々 1 つである．r が $\mathcal{A}$ による t の受理実行木であるとき，$\mathcal{A}$ によって選択される t の頂点集合を $\mathcal{A}(t) = \{p \in D_t \mid \lambda_r(p) \in \mathcal{S}\}$ とする．

データ木オートマトン $\mathcal{A} = (\Sigma, Q_b, Q_t, q_{b0}, q_{t0}, \Delta_b, \Delta_t, S)$ に対し，$\mathcal{A}_b = (\Sigma, Q_b, q_{b0}, \Delta_b)$ を $\mathcal{A}$ に付随するボトムアップ木オートマトンという．$\mathcal{A}_b$ による受理実行木を同様に定義する．

3. 圧　　縮

3.1 文法による木構造圧縮

$\Sigma \cap \mathcal{X} = \emptyset$ であるような**変数**の可算集合を $\mathcal{X} = \{x_1, x_2, \ldots\}$ としたとき，$\mathcal{X}$ に属する変数をランクが 0 の記号として含む木の全体集合を $T(\Sigma, \mathcal{X})$ と表記する．$t \in T(\Sigma, \mathcal{X})$ 中のどの変数も高々 1 回しか出現しないとき，t は線形であるという．$t \in T(\Sigma, \{x_1, \ldots, x_n\})$, $t_1, \ldots, t_n \in T(\Sigma, \mathcal{X})$ としたとき，$i \in \{1, 2, \ldots, n\}$ について，$\lambda_t(p_i) = x_i$ である頂点 p_i を木 t_i で置き換えて得られた木を $t[x_1/t_1, \ldots, x_n/t_n]$ と表記する．つまり，$[x_1/t_1, \ldots, x_n/t_n]$ で $\sigma(x_i) = t_i (1 \leqq i \leqq n)$ であるような代入 σ を表す．$z \notin (\Sigma \cup \mathcal{X})$ について，z がちょうど 1 度だけ出現するような木 $C \in T(\Sigma, \mathcal{X} \cup \{z\})$ を**文脈**とよぶ．$C[z/t]$ の代わりに $C[t]$ と表記する．

［定義 3.1］ **文脈自由木文法** (Context-Free Tree Grammar, CFTG) は 4 項組 $\mathcal{G} = (N, \Sigma, P, S)$ である．ここで，

（1） N は非終端記号の有限集合

（2） Σ はランク付きアルファベット

（3） P は $A(x_1, \ldots, x_n) \rightarrow t$ という形の生成規則の有限集合．ここで，$A \in N$, rank$(A) = n, t \in T(\Sigma \cup N, \{x_1, \ldots, x_n\})$ であり，t には $x_1, \ldots, x_n$ が現れる．

（4） $S \in N$ は rank$(S) = 0$ の開始非終端記号． □

生成規則 $A(x_1, \ldots, x_n) \rightarrow t \in P$(ただし rank$(A) = n$), 文脈 $C \in T(\Sigma \cup N, \mathcal{X} \cup \{z\})$, $s = C[A(t_1, \ldots, t_n)]$, $s' = C[t[x_1/t_1, \ldots, x_n/t_n]]$

であるような木 $t_1,\ldots,t_n \in T(\Sigma\cup N,\mathcal{X})$ が存在するとき，$T(\Sigma\cup N,\mathcal{X})$ 上の関係 $s \Rightarrow_{\mathcal{G}} s'$ が成り立つと定義し，$\Rightarrow_{\mathcal{G}}$ の反射推移閉包を $\Rightarrow^*_{\mathcal{G}}$ と書く．また，$\mathcal{G}$ によって生成される木言語を $L(\mathcal{G}) = \{t \in T(\Sigma) \mid S \Rightarrow^*_{\mathcal{G}} t\}$ と定義する．さらに，2 項関係 $\rightsquigarrow_{\mathcal{G}}$ を $\{(A,B)\in N\times N \mid A \to t \in P$ かつ B が t に現れる$\}$ で定義する．

［定義 3.2］ **直線的文脈自由木文法** (Straight-Line CFTG, SLCFTG) は以下の条件を満たすような CFTG $\mathcal{G} = (N,\Sigma,P,S)$ である．

（1） 全ての $A \in N$ について，ただ 1 つの生成規則 $A(x_1,\ldots,x_n) \to t \in P$ が存在し，t が線形である

（2） 関係 $\rightsquigarrow_{\mathcal{G}}$ は非循環

非終端記号 $A \in N$ を左辺にもつ生成規則の右辺の木を t_A と表記する． □

本稿では一般性を失うことなく，規則の右辺の木 t において変数は左から右に $x_1,\ldots,x_n$ の順番で現れると仮定する．また，関係 $\rightsquigarrow_{\mathcal{G}}$ の推移閉包を $\rightsquigarrow^+_{\mathcal{G}}$ と書き，$A \rightsquigarrow^+_{\mathcal{G}} B$ のとき，A は下の階層に B をもつという．

$\mathcal{G}$ が SLCFTG であるとき，$\mathcal{G}$ によって生成される木言語 $L(\mathcal{G})$ の要素はただ 1 つとなる．与えられた木 t のみを生成する SLCFTG $\mathcal{G}$，すなわち $L(\mathcal{G}) = \{t\}$ を満たす $\mathcal{G}$ を t の圧縮という．与えられた t から $L(\mathcal{G}) = \{t\}$ を満たす $\mathcal{G}$ を構成することを，t を圧縮するという．逆に $\mathcal{G}$ から t を計算することを，$\mathcal{G}$ を解凍するという．

［例 3.1］ $\mathcal{G} = (\{S,A,B\},\{dblp, article, thesis, title, author, \#\}, P, S)$ は図 4 の 2 分木を圧縮した例である．ここで

$$P = \left\{\begin{array}{l} S \to dblp(article(A(B), thesis(A(\#), thesis(A(\#),\#))),\#) \\ A(x) \to title(\#, author(\#, x)) \\ B \to author(\#,\#) \end{array}\right\}$$

□

3.2 圧縮データ木

(Σ, Att) 上のデータ木を圧縮する際，木構造部分とデータ値部分を分離したものを考える．

［定義 3.3］ $t = (D,\lambda,\delta)$ をデータ木とし，t を木構造圧縮して得られた SLCFTG を $\mathcal{G}$ とする．このとき，2 項組 $(\mathcal{G},\delta)$ を**データ木 t に対する圧縮データ木**とよぶ．

δ の表現として，次のような位置と属性の組をキーとするようなマップを考える．文書の構造にもよるが，fcns 符号化を行っていることから位置には “2” が頻出するためラン・レングス符号化を施し圧縮する．

表 1 データ値関数 δ の表現

位置	属性	データ値
12	lang	jp
121	from	us
⋮	⋮	⋮

4. 直接更新法

4.1 更新の定義

与えられた木のどの頂点をどのように変更するのかを指定するため，更新を 2 項組 $(\mathcal{A}, op)$ で指定する．ここで，$\mathcal{A}$ はデータ木オートマトンであり，更新対象の木を $t = (D,\lambda,\delta)$ とするとき，$\mathcal{A}$ の選択する頂点集合 $\mathcal{A}(t)$ が更新位置集合となる．また，op は更新操作を表し，relabel, delete, insert-before, insert-after, update-value のどれかである．それぞれの操作は fcns 符号化を行う前のランクなし木における “自然な” 更新操作に対応している．

relabel(s') $p \in \mathcal{A}(t)$ である全ての p について，$\lambda_t(p) = s'$ とする．

delete() $p \in \mathcal{A}(t)$ である全ての p について，p 以下の部分木 $t_p = \lambda_t(p)(t_1,t_2)$ を t_2 に置き換える．

insert-before$(t_i = r(t_{i1},\#))$ $p \in \mathcal{A}(t)$ である全ての p について，p 以下の部分木 $t_p = \lambda_t(p)(t_1,t_2)$ を $r(t_{i1},\lambda_t(p)(t_1,t_2))$ に置き換える．

insert-after$(t_i = r(t_{i1},\#))$ $p \in \mathcal{A}(t)$ である全ての p について，p 以下の部分木 $t_p = \lambda_t(p)(t_1,t_2)$ を $\lambda_t(p)(t_1, r(t_{i1},t_2))$ に置き換える．

update-value$(attr, opr, v)$ $p \in \mathcal{A}(t)$ である全ての p について，$\delta(p, attr)$ を $opr(\delta(p,attr), v)$ で上書きする．

relabel, delete, insert-before, insert-after は，選択頂点を post-order(帰りがけ順) で更新していき，update-value は pre-order(行きがけ順) で更新する．

delete, insert-before, insert-after については，構造が変更されることにより位置が変化する頂点が存在するため，δ の更新が必要となる．

4.2 提案手法

この節では，(Σ, Att) 上のデータ木 $t = (D,\lambda,\delta)$ に対する圧縮データ木 $t' = (\mathcal{G},\delta)$ とデータ木オートマトン $\mathcal{A}$ が与えられたときに，t' を解凍することなく $\mathcal{A}$ によって選択される t の頂点集合を求め，その選択頂点に対して更新を行う手法を提案する．[14] と同じように，更新位置の検索と更新処理の 2 段階で行われる．

4.2.1 更新位置の検索

更新位置の検索の基本方針は，$\mathcal{G}$ の開始規則の右辺 t_S の各頂点に状態を割り当てながら受理実行木を構成することである．このとき，頂点のラベルが非終端記号の場合は，その非終端記号を左辺にもつ生成規則の右辺の木に移り同様に受理実行木を構成していく．

更新位置の検索は，[14] と同じようにアルゴリズム bu_eval と td_eval の 2 段階で行われる．bu_eval は，与えられた木を post-order で再帰的に評価し，$\mathcal{A}$ に付随する $\mathcal{A}_b$ による受理実行木を構成し ARmap というマップに記憶していく．td_eval は，bu_eval の結果に基づき $\mathcal{A}$ のトップダウン部の評価を行う．ただし，構造情報だけでなくデータ値も参照するように拡張する．

bu_eval の擬似コードを Algorithm 1 に示す．[14] のように構造情報だけを考える場合は，非終端記号の引数に割り当てられ

た状態が全て等しいとき，構成される受理実行木は等しくなり重複回避が可能である．しかし，データ値も考慮する場合は，非終端記号の引数に割り当てられた状態が全て等しくても，データ値の条件判定を行った結果に依存して，異なる受理実行木が構成されることがある．

非終端記号の引数に割り当てられた状態から，データ値の条件判定が行われないとわかり，かつその引数に割り当てられた状態に対して受理実行木が構成済みの場合は，既存研究と同じように受理実行木の構成を回避することができる．それ以外の場合はまず受理実行木を構成し，その後構成した受理実行木と同じものがすでに構成済みであれば，その受理実行木を削除することで同じ受理実行木がメモリ上に存在しないようにする．非終端記号とデータ値の条件判定が適用された頂点の状態列，引数の状態列の組が定まれば受理実行木は一意に定まるので，これらの組を ARmap のキーとすることで $\mathcal{A}_b$ による同一の受理実行木の構成を回避する．また，$\mathcal{G}$ を走査する際には，δ によってデータ値を参照するためにもとのデータ木上での位置を計算しながら走査する．非終端記号 A の子頂点の位置は t_A における変数の位置に対応するため，$\mathrm{Vmap}(A, i)$ を非終端記号 $A \in N$ の i 番目の変数の位置としてあらかじめ計算しておく．ここで，非終端記号 A の変数 x_i の位置は，$A(x_1, \ldots, x_i, \ldots, x_n) \Rightarrow^*_{\mathcal{G}} t$ かつ，$t \Rightarrow_{\mathcal{G}} t'$ である t' が存在しない（すなわち t に非終端記号が出現しない）t における変数 x_i の位置である．$\mathrm{Jmap}(A, \pi)$ は，データ木上の位置 π に対応する非終端記号 A に対して A を左辺にもつ規則の右辺 t_A を評価したときに，条件判定をもつ遷移規則が適用された頂点に割り当てられた状態列を記憶するためのマップであり，$\mathrm{ARmap}(A, \cdot, \cdot)$ の第 2 成分を決定するために用いる．

データ木オートマトン $\mathcal{A} = (\Sigma, Q_b, Q_t, q_{b0}, q_{t0}, \Delta_b, \Delta_t, \mathcal{S})$ と SLCFTG $\mathcal{G} = (N, \Sigma, P, S)$ が与えられたとき，まず bu_eval を実引数値 $(t_S, \epsilon, \perp, (S, \epsilon), \epsilon)$ で呼び出して $\mathcal{A}_b$ による受理実行木を構成する．一般に $\mathrm{bu_eval}(t, p, \sigma, (\ell, c), \pi)$ は第 1 引数で与えられた木 t の頂点 p の子頂点に対して bu_eval を再帰的に呼び出し（2-9 行目），その後，次のように頂点 p を評価する．σ は変数に割り当てる状態を表す代入，ℓ は $t = t_\ell$ を右辺にもつ生成規則の左辺の非終端記号，c は ℓ のデータ木上の位置，π は p のデータ木上の位置を表す．

（1） p のラベルが変数 x のとき（10-11 行目）
代入 σ を用いて状態 $\sigma(x)$ を割り当てる．

（2） p のラベルが終端記号のとき（12-27 行目）
Δ_b に従い状態を割り当てる．ただし，ラベルが#のときは初期状態である q_{b0} を割り当てる．このとき，データ値の条件判定をもつ遷移規則によって状態 q_b が割り当てられた場合は $\mathrm{Jmap}(\ell, c)$ に q_b を追加する．

（3） p のラベルが非終端記号 A のとき（28-34 行目）
生成規則 $A(x_1, \ldots, x_n) \to t_A \in P$ の右辺 t_A を再帰的に評価する．$t_p = A(t_1, \ldots, t_n)$ とするとき，t_A 中に現れる変数 $x_1, \ldots, x_n$ に割り当てる状態は部分木 $t_1, \ldots, t_n$ の根頂点に割り当てられた状態 (左から順に $q_{b1}, \ldots, q_{bn}$ とする) となるので，代入 $[x_1/q_{b1}, \ldots, x_n/q_{bn}]$ を構成して bu_eval を実引数値 $(t_A, \epsilon, [x_1/q_{b1}, \ldots, x_n/q_{bn}], (A, \pi), \pi)$ で呼び出す．もし，$\mathrm{ARmap}(A, \epsilon, q_{bs})$ が記憶済みなら，t_A の再評価は行わない．t_A の評価後に $\mathrm{ARmap}(A, \mathrm{Jmap}(A, \pi), q_{bs})$ に得られた $\mathcal{A}_b$ による受理実行木を記憶する．もし，$\mathrm{ARmap}(A, \mathrm{Jmap}(A, \pi), q_{bs})$ が記憶済みなら，同一の受理実行木が構成済みであるため，得られた受理実行木を削除する．$\mathrm{Jmap}(A, \pi)$ が ϵ でなかったら，t_A の根頂点に割り当てられた状態を $\mathrm{Jmap}(\ell, c)$ に追加する．

Algorithm 1 bu_eval(t : 木, p : t 上の位置, $\sigma : \mathcal{X} \to Q_b$, ($\ell$: 非終端記号, c: データ木上の位置), π: データ木上の現在位置)

```
1: s = λ_t(p) ∈ Σ_n, 各 i(1 ≦ i ≦ n) について q_bi = λ_t(p · i)[2] とおく．また，q_b1 ··· q_bn = q_bs とおく．
2: if s が非終端記号 then
3:     for i = 1 to n do
4:         bu_eval(t, p · i, σ, (ℓ, c), π · Vmap(s, i))
5:     end for
6: else if s が終端記号かつ s ≠ # then
7:     bu_eval(t, p · 1, σ, (ℓ, c), π · 1)
8:     bu_eval(t, p · 2, σ, (ℓ, c), π · 2)
9: end if
10: if s が変数 then
11:     λ_t(p) = (s, σ(s)) に上書き
12: else if s が終端記号 then
13:     if s = # then
14:         λ_t(p) = (#, q_b0) に上書き
15:         return
16:     end if
17:     if s(q_b1, q_b2) → (a, φ)?q_T : q_F ∈ Δ_b then
18:         if φ(δ(π, a)) then
19:             λ_t(p) = (s, q_T) に上書き
20:             Jmap(ℓ, c) に q_T を追加
21:         else
22:             λ_t(p) = (s, q_F) に上書き
23:             Jmap(ℓ, c) に q_F を追加
24:         end if
25:     else if s(q_b1, q_b2) → q_b ∈ Δ_b then
26:         λ_t(p) = (s, q_b) に上書き
27:     end if
28: else if s が非終端記号 A then
29:     if ARmap(A, ϵ, q_bs) が未記憶 then
30:         A(x_1, ..., x_n) → t_A ∈ P である t_A について
31:         bu_eval(t_A, ϵ, [x_1/q_b1, ..., x_n/q_bn], (A, π), π)
32:         A_b による t_A の受理実行木 t'_A を ARmap(A, Jmap(A, π), q_bs) に記憶
33:         Jmap(A, π) が ϵ でなかったら，λ_{t'_A}(ϵ)[2] を Jmap(ℓ, c) に追加する
34:     end if
35:     λ_t(p) = (s, λ_{t'_A}(ϵ)[2]) に上書き
36: end if
```

td_eval は ARmap に記憶された $\mathcal{A}_b$ による受理実行木を pre-order で再帰的に評価して $\mathcal{A}$ による受理実行木を構成し，選択頂点を UDmap というマップに記憶していく．非終端記号のボトムアップでデータ値の条件判定が適用された頂点の状態列と

各引数に割り当てられた状態列，非終端記号にトップダウンで割り当てられた状態が全て等しいとき，構成される $\mathcal{A}$ による受理実行木は等しくなる．そこで，それらを UDmap のキーとすることで同一の受理実行木の重複した構成を回避する．

重複回避のために UDmap は 4 つの要素 (リスト $\mathcal{V}$，リスト $\mathcal{T}$，リスト $\mathcal{N}$，L) を記憶する．$\mathcal{V}$ は t_A 中の変数に割り当てられた状態のリスト，$\mathcal{T}$ は t_A 中の選択頂点の位置のリスト，$\mathcal{N}$ は t_A 中の非終端記号で下の階層に選択頂点をもつ非終端記号と割り当てられた状態と位置の組のリストである．L は更新を行う際に使用する．UDmap を辿ることで，選択頂点までの経路を得ることができる．

[14] と同様に ARmap に記憶されたボトムアップの受理実行木を，データ木上の位置を計算しながら再帰的に評価し，受理実行木を構成していく．その際は，Jmap の情報を用いて ARmap から bu_eval で構成したボトムアップの受理実行木を得て評価していき，必要な情報を UDmap に記憶していく．bu_eval の終了後，td_eval を実引数値 $(t_S, q_{t0}, \epsilon, (S, \epsilon, \epsilon, q_{t0}))$ で呼び出して $\mathcal{A}$ による選択頂点集合を求める．

4.2.2 更 新 処 理

入力の圧縮データ木 $(\mathcal{G}, \delta)$ と更新 $(\mathcal{A}, op)$ に対して，bu_eval と tu_eval によって得られた UDmap を用いて，$\mathcal{A}$ による各選択頂点に更新操作 op を適用した結果の圧縮データ木 $(\mathcal{G}', \delta')$ を構成する．$\mathcal{G}$ から $\mathcal{G}'$ への木文法の更新については文献 [14] にあるアルゴリズム update と同様に実行できる．

今回新たに追加した update-value(att, opr, v) については，すべての選択頂点 p に対して，$\delta(p, attr)$ を $opr(\delta(p, attr), v)$ で上書きすることで実現する．

また，delete, insert-before, insert-after によって構造情報が更新された場合に δ の更新が必要となる．選択頂点前後への挿入や削除が，選択頂点と子孫関係にある頂点の位置を変化させるからである．具体的には，delete, insert-after は選択頂点の右の部分木以下の位置，insert-before は選択頂点以下の部分木の位置が変化する．各選択頂点につきその頂点での更新に影響を受けるすべての頂点位置を毎回更新していくのは非効率であるので，各頂点につき位置の更新は一回で済むようにする．UDmap を用いることで選択頂点位置は pre-order で列挙可能であるので，その順で δ 中の位置を更新していく．その際に，更新しようとしている位置より先祖での更新による位置のずれを覚えておくことによって一度の走査で更新可能である．

4.3 データ値関数の分割

XML 文書が多くのデータ値を含む場合，それらを 1 つのファイルにまとめておくと更新には不要な部分の読込みや書込みに時間がかかる場合がある．そのため，データ値関数のファイルを複数に分割しておくことにより，なるべく必要な部分のファイルだけを読み込んで更新するようにする．

4.3.1 属 性 分 割

属性分割では，データ値関数を実現するデータ構造（ファイル）を属性にしたがって分割し，更新時に必要なファイルのみを読み込むことで，更新処理の効率を向上させる．表 2 を属性分割した例を，表 3,4 に示す．これらに示すとおり，分割後のファイルを関連付けるための属性を導入する．これは関係データベースにおける 2 次キーに相当する．また，ファイルを表 2 のような表形式で表すことを前提とすると，属性分割は垂直分割ということもできる．

表 2　分 割 前 δ

位置	属性名	データ値
11	text	long text
112	text	long long text
1122	text	super long text
⋮	⋮	⋮

表 3　分 割 後 δ_1

位置	属性名	ID
11	text	1
112	text	2
1122	text	3
⋮	⋮	⋮

表 4　分 割 後 δ_2

ID	データ値
1	long text
2	long long text
3	super long text
⋮	⋮

4.3.2 レコード分割

レコード分割は，データ値関数の定義域を複数の部分集合に分割し，その分割にしたがって定まる複数のデータ値関数ごとにデータ値関数を保存する方法である．属性分割と同様，レコード分割により，データ値の更新において必要な部分だけを読み込むことにより更新処理の効率化を図る．属性分割を垂直分割ともよぶのに対して，レコード分割を水平分割とよぶこともある．

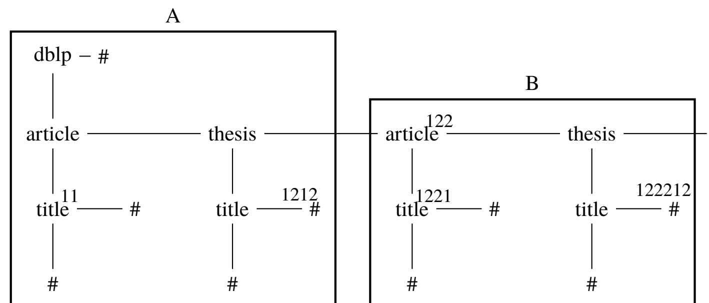

表 5　$\delta_A^{(\epsilon,1212)}$

位置	属性	データ値
11	text	a
⋮	⋮	⋮

表 6　$\delta_B^{(122,122212)}$

位置	属性	データ値
1	text	c
⋮	⋮	⋮

データ値関数をレコード分割する際にいくつに分割されたかを分割数とよぶ．分割された各データ値関数 δ はそれに含まれるレコードの位置集合 P_δ について以下の条件をみたすようにする．

- ある位置 p_{root} が存在して，他のどの位置 $p' \in P_\delta$ も p_{root} の子孫である．

- どの位置 $p \in P_\delta$ についても，p の左の子 $p1$ の子孫はすべて P_δ に含まれる．

位置 p_{root} を根の位置とよび，P_δ 中で辞書式順序に関して最大のものを δ の終端位置とよぶ．分割された各データ値関数 δ ごとにこれらの 2 つの絶対位置を記憶することによって，選択頂点がどのデータ値関数に含まれているかを決定することができる．また，各レコードの位置は根の位置に対する相対位置として記憶する．根の位置の（真の）先祖頂点において構造の更新が起こった場合は，この根の位置と終端位置を修正すればよく，各レコードを読み込む必要がないので位置の更新の効率化が見込める．

5. 実　　験

本節では，ベンチマークとなる XML 文書 [3] を圧縮して得られた圧縮データ木 (以降，圧縮文書) に対して，提案手法で更新位置を検索し更新を行った際に要した時間および最大使用メモリなどを示す．また，一旦解凍した XML 文書 (以降，非圧縮文書) に対して更新を行った場合の結果も示し，比較する．データ値関数を保持するファイルを分割しなかった場合の結果や分割数を変化させた時の結果も示し，分割や分割数の効率への影響を考察する．代表的な XML データベース管理システムの 1 つである BaseX とも更新に要した時間，最大使用メモリを比較する．

5.1 実験環境と用いたデータ

実験に用いた実行環境は OS: Ubuntu 14.04 LTS，プロセッサ: Intel® Core™ i5-2400 3.10GHz，メモリ: 8GB である．用いた XML 文書に関する情報を表 7 に，更新位置の指定を XPath 式で表したものと行った更新操作を表 8 に示す．なお，XML 文書の圧縮には，圧縮ツール TreeRePair [8] を使用し，分割数は 1-4 の文書が 50，5-8 の文書が 100 となるようにした．圧縮前のサイズ，圧縮後のサイズの定義は文献 [14] による．

表 7　実験に用いたデータ

文書名	ファイルサイズ (kB)	圧縮前のサイズ	圧縮後のサイズ
BaseBall	652	28,305	500
Shakespeare	7,710	179,689	17,747
Nasa	25,212	476,645	22,781
DBLP	134,315	3,332,129	150,720
SwissProt	115,467	2,977,030	247,873
TreeBank	87,466	2,437,665	524,743

5.2 非圧縮文書と圧縮文書に対する更新の比較

圧縮文書に対する実験結果を表 9 に，非圧縮文書に対する実験結果を表 10 に示す．

表 9 と表 10 の最大使用メモリを比較すると全ての更新において圧縮文書に対する更新の方が少ないメモリ使用量で行えることがわかる．これは，非圧縮文書の場合はデータ木をメモリ上に展開する必要があるのに対して，圧縮文書は同一の受理実行木はメモリ上に複数存在することがないためである．

実行時間についても非圧縮文書よりも圧縮文書の方が短い時間で更新が行えると言える．これは，圧縮文書に対しては同じ状態割当になることがわかっている部分木に対しては状態割当を行わずに結果を再利用することができるためである．

表 8　更新位置と更新操作

#	文書名	更新位置を表す XPath 式	更新操作
1	BaseBall	//PLAYER[HOME_RUNS ≧ 10]	delete
2	BaseBall	//LEAGUE// PLAYER[WINS ≧ 15]	relabel
3	Shakespeare	//ACT[text() = ACT I]	delete
4	Shakespeare	//ACT[text() = ACT III]	update-value
5	Nasa	//dataset[@subject = astronomy] /altname[1]	insert-after
6	DBLP	//proceedings[year ≧ 2000]/url	insert-before
7	SwissProt	//Entry[@seqlen > 2500]/Species	update-value
8	TreeBank	/FILE[@name = "head.cmb"] //VP[.//S//NP//VP //PP//NP//NP//NNP]//_PERIOD_	delete

表 9　圧縮文書に対する実験結果

#	ボトムアップ (ms)	トップダウン (ms)	更新 (構造) (ms)	更新 (データ値) (ms)	合計 (ms)	使用メモリ (kB)
1	29	0	2	25	56	3,380
2	16	0	0	0	16	2,813
3	22	14	0	35	71	14,436
4	21	18	0	23	62	14,129
5	89	30	0	2,210	2,329	65,903
6	289	434	0	19,891	20,614	326,457
7	531	835	39	21,481	22,886	411,673
8	804	1,084	53	16,064	18,005	501,133

表 10　非圧縮文書に対する実験結果

#	解凍	ボトムアップ (ms)	トップダウン (ms)	更新 (構造) (ms)	更新 (データ値) (ms)	圧縮 (ms)	合計 (ms)	使用メモリ (kB)
1	22	18	10	2	24	51	129	12,365
2	22	15	14	0	0	48	147	14,251
3	222	30	56	0	48	462	818	54,687
4	222	30	56	0	21	462	791	56,992
5	397	136	216	0	6,499	1,066	8,314	214,587
6	8,390	2,323	2,890	0	19,121	12,381	45,105	1,087,122
7	9,547	3,290	3,850	9	21,436	15,702	53,834	1,158,403
8	12,601	5,652	6,537	15	17,084	15,041	56,930	1,337,513

5.3 分割による更新の効率への影響

以下の表 11 に，分割をしなかった場合 (分割数 1) の結果を示す．表 11 と表 9 を比較すると，5-8 の文書に対してデータ値の更新に多くの時間がかかっている．これは，文書のサイズが大きいことや，データ値としてテキストデータを持っているため，データ値部分に関する情報が巨大になっており，加えて，分割されていないため更新に関係がないデータを大量に読み込んでしまうからである．分割することで更新に関係がない部分についてはある程度読み込まずに済ますことができるようになり，効率が向上することがわかる．

以下の表 12 に分割数を変化させたときの更新に要した時間を示す．なお，1-4 の文書に関しては，分割数を変化させてもあまり変化が見られなかったので省略した．

表 11　分 割 な し

#	ボトムアップ (ms)	トップダウン (ms)	更新 (構造) (ms)	更新 (データ値) (ms)	合計 (ms)	使用メモリ (kB)
1	38	0	2	23	63	3,208
2	19	0	0	0	19	2,987
3	25	16	0	42	83	15,229
4	24	18	0	26	68	15,924
5	95	30	0	6,328	6,492	68,885
6	295	447	0	45,152	45,894	354,113
7	533	870	37	44,831	46,271	418,437
8	814	1,077	49	49,845	51,785	490,057

表 12　分割数 n による効率への影響

#	n = 25 (ms)	n = 50 (ms)	n = 100 (ms)	n = 200 (ms)
5	2,003	2,127	2,329	2,459
6	27,938	26,032	20,614	21,201
7	27,498	23,540	22,886	22,421
8	23,130	21,437	18,005	17,165

表 12 を見ると，5 の文書に関しては，更新位置が文書全体に広がっており，結局分割したものすべてを読み込む必要があり，分割数が少ないほうが効率が良くなったと考えられる．

また，6-8 の文書に関しては非常に大規模な文書であり，分割された 1 つ 1 つのサイズが大きくなり，更新の際に必要ない情報を読み込んでしまう．そのため，ある程度分割数を大きくしたほうが効率よく更新が行えると考えられる．6 の文書については分割数が 200 のときよりも，100 のときの方が若干速く更新が行えている．これは，更新位置が文書の前半部分に集中して出現しており，分割数を増やしたことによって連続して出現している更新位置が途中で分断され，より多くの情報を読み込んでしまったためだと考えられる．

この結果から文書の構造やどの位置の頂点を更新するかなどによって最適な分割数は変化すると考えられる．現状，あらかじめ分割されたものに対して問い合わせを行っているため，問い合わせによっては分割したことにより，分割しなかった場合よりも時間がかかる場合も考えられる．どのように分割するかについてはなるべく効率が良くなるような方法を検討する必要がある．例えば，更新位置と文書構造を見てから効率が良くなるように分割することや，あらかじめユーザが行うであろう問い合わせを予測した上で分割しておくことなどを考えている．

6. BaseX との比較

BaseX [1] は軽量，高速，高機能な XML データベース管理システムである．ここでは BaseX と更新に要する時間と使用メモリの観点から比較を行う．

BaseX は，まず XML 文書を読み込んで DB を作成してから，それに対して XML に対する問い合わせ言語である XQuery [2] を実行して更新を行う．実験時の Java 仮想マシンの最大ヒープサイズは 2048MB とした (-Xmx2048m)．以下の表 13 に BaseX で XML 文書を読み込んで DB を作成したときの結果，表 14 に BaseX で作成した DB に対して更新を行ったときの結果を示す．また，表 15 に木構造圧縮したときの結果，表 16 に圧縮文書に対して直接更新法を行ったときの結果を示す．

表 13　DB 作成 (パース，索引付け)

文書名	時間 (ms)	使用メモリ (kB)
BaseBall	623	70,812
Shakespeare	1,006	100,040
Nasa	2,099	160,204
DBLP	11,928	766,008
SwissProt	12,049	819,354
TreeBank	9,230	1,022,877

表 14　XQuery 実行

#	時間 (ms)	使用メモリ (kB)
1	842	108,764
2	875	110,212
3	1,207	131,828
4	1,215	129,708
5	3,923	211,020
6	15,930	863,592
7	8,402	1,011,542
8	5,359	1,072,793

表 15　圧縮 (木構造圧縮，データ値抽出)

文書名	時間 (ms)	使用メモリ (kB)
BaseBall	159	3,396
Shakespeare	971	9,816
Nasa	2,388	12,972
DBLP	14,212	72,916
SwissProt	22,767	180,548
TreeBank	21,665	159,768

表 16　圧縮文書に対する直接更新（表 9 抜粋）

#	時間 (ms)	使用メモリ (kB)
1	56	3,380
2	16	2,813
3	71	14,436
4	62	14,129
5	2,329	65,903
6	20,614	326,457
7	22,886	411,673
8	18,005	501,133

表 14,16 を比較すると，使用メモリの点では，提案手法がより少ないメモリで更新できているが，更新に要した時間は，巨大な XML 文書である DBLP, SwissProt, TreeBank に対して，BaseX の方がより速く更新できており，とくに SwissProt と TreeBank に対しては更新に要した時間が大幅に短い．これらは，BaseX

の優れた XQuery の最適化や索引付け [7] [11] によるものだと考えられる．提案手法ではデータ値の更新が大部分を占めており，この部分を改善することができれば，大幅に更新に要する時間を削減することができる．

7. あとがき

本稿では，木文法によって圧縮された XML 文書に対して解凍を行わずに，既存手法を拡張する形で文書の構造情報だけでなく文書に含まれるデータ値を考慮し，それらに依存した更新位置の検索と更新を行う手法を提案した．

実験結果から，解凍してから更新するよりも，提案手法を利用して圧縮した状態のまま処理を行った方が少ないメモリで高速に更新できることが分かった．また，巨大なデータ値部分に関する情報を分割することでより高速に更新が行えることを確認した．BaseX との比較では，要した時間に関しては大きな文書に対しては BaseX がより高速に動作したが，要したメモリに関しては提案手法を利用して更新を行ったほうが少ないメモリ量で更新が行えた．

今後の研究では，実用化に向けて，高速化や省メモリ化を目指し，より良いデータ値の表現方法や，圧縮の方法，分割の方法などを検討，考察していきたいと考えている．

文献

[1] BaseX | the xml database. http://basex.org.

[2] W3C XML Query (XQuery). https://www.w3.org/XML/Query/.

[3] XMLCompBench. http://xmlcompbench.sourceforge.net/Dataset.html.

[4] S. Böttcher, M. Grohe, and C. Krislin. Clux - clustering XML subtrees. In *Proc. of the 12th International Conference on Enterprise Information Systems (ICEIS 2010)*, Vol. 1, pp. 142–150, 2010.

[5] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Information Systems*, Vol. 33, pp. 456–474, 2008.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2008. http://tata.gforge.inria.fr/.

[7] L. Kircher, M. Grossniklaus, C. GrÃijn, and M. H. Scholl. Efficient structural bulk updates on the pre/dist/size xml encoding. In *Proc. of the 31st International Conference on Data Engineering (ICDE 2015)*, pp. 447–458, 2015.

[8] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Information Systems*, Vol. 38, pp. 1150–1167, 2013.

[9] M. Lohrey, S. Maneth, and E. Noeth. XML compression via DAGs. In *Proc. of the 16th International Conference on Database Theory (ICDT 2013)*, pp. 18–22, 2013.

[10] S. Maneth and G. Busatto. Tree transducers and tree compressions. In *Proc. of the 7th International Conference on Foundations of Software Science and Computation Structures (FoSSaCs 2004)*, pp. 363–377, 2004.

[11] L. Wörteler, M. Grossniklaus, C. Grün, and Marc H. Scholl. Function inlining in XQuery 3.0 optimization. In *Proc. of the 15th Symposium on Database Programming Languages (DBPL 2015)*, pp. 45–48, 2015.

[12] 尾上栄浩, 橋本健二, 関浩之, 伊藤実. 木文法による圧縮 XML 文書に対する問合せと更新手法. 信学技報, Vol. 114, No. 271, SS2014-28, pp. 17–22, 2014.

[13] 後藤健志, 尾上栄浩, 橋本健二, 関浩之. 木文法に基づく XML 圧縮文書に対する直接更新手法の評価. 信学技報, Vol. 114, No. 416, SS2014-45, pp. 73–78, 2015.

[14] 後藤健志, 高山隆之介, 橋本健二, 関浩之. 圧縮構造化文書のための更新手法の拡張とその実験的評価. 信学技報, Vol. 115, No. 486, LOIS2015-75, pp. 69–74, 2016.