# トライ木を用いたXMLキーワード検索法

吉永　光†　古川　孝則††　山本　博章††　金子　美博†††

† 信州大学大学院理工学系研究科　〒 380-8553 長野県長野市若里 4-17-1
†† 信州大学工学部　〒 380-8553 長野県長野市若里 4-17-1
††† 岐阜大学工学部　〒 380-8553 岐阜県岐阜柳戸 1-1
E-mail: †{12tm546j,10t5068j}@shinshu-u.ac.jp, ††yamamoto@cs.shinshu-u.ac.jp, †††fau@gifu-u.ac.jp

**あらまし**　XML は情報を交換したり保存するための標準的な形式として急速に広まってきている．そのため，XML 形式のデータを検索する技術の開発は重要な研究課題である．XML データを検索するための手法として，XPath や XQuery などを利用した方法があるが，これらの手法は XPath や XQuery を理解する必要がある．より簡単な検索法としてキーワードによる検索法が開発されている．これはクエリとして，単にキーワードを与えるだけで検索できるもので，初心者でも簡単に検索することができる．XML のキーワード検索に対しては，検索結果として SLCA(Smallest Lowest Common Ancestor) が広く採用されており，SLCA を検索するアルゴリズムが開発されてきた．本論文は，情報検索で用いられているトライ木を採用し，SLCA を見つける効率的な手法を提案する．

**キーワード**　XML，キーワード検索，SLCA，トライ木

## 1. まえがき

### 1.1 背景と目的

XML は情報を交換したり保存するための標準的な形式として急速に広まってきている．そのため，XML 形式のデータを検索する技術の開発は重要な研究課題である．XML データを検索するための手法として，XPath や XQuery などを利用した方法があるが，これらの手法は XPath や XQuery を理解する必要がある．より簡単な検索法としてキーワードによる検索法が開発されている．これはクエリとして，単にキーワードを与えるだけで検索できるもので，初心者でも簡単に検索することができる．XML のキーワード検索に対しては，検索結果として意味のある結果を一意に決定することが難しいため，いろいろな定義が導入されてきた．その概要については文献 [2] で見ることができる．その中で，検索結果として，SLCA(Smallest Lowest Common Ancestor) が広く用いられており，SLCA を検索するアルゴリズムが開発されてきた [3], [4], [5], [6], [7], [8]．本論文の目的は，情報検索で用いられているトライ木を利用し，SLCA を見つける効率的な手法を開発することである．

### 1.2 関連研究

一般に，XML データは木構造として表すことができる．XML に対するキーワード検索では，与えられたキーワードに対しどのような結果を返すかは，一意に定義することができないため，いろいろな解釈のもとで研究されてきた．その中で，検索結果として SLCA を採用した研究が活発に行われている．まず，複数キーワードからなるクエリ $Q = \{w_1, \ldots, w_k\}$ が与えられたとき，それらのキーワードに対応する LCA は，各キーワードに対応するノードの共通の祖先の中で最も葉に近いノードと定義される．そのとき，SLCA は LCA の中でその子孫に LCA をもたない LCA である．Xu 他 [4] は，SLCA を見つけるためのアルゴリズムとして IL 法（Index Lookup Eager) を提案した．IL 法は $O(kd|S_k|)$ 時間で走る．Sun 他 [3] は，IL 法を改良し，BMS 法（Basic Multiway Search) および BMS を高速化した IMS 法（Imcremental Multiway Search）を開発した．BMS 及び IMS はどちらも $O(kd|S_1|\log|S_k|)$ 時間で走るが，IMS のほうが計算を最適化しているため，XML データによっては速くなる．なお，$k$ はキーワード数，$d$ は XML を木構造で表したときの高さ，$S_1, \cdots, S_k$ を各キーワードに一致するノードの集合としたとき，$|S_1| \leqq \cdots \leqq |S_k|$ である．Wang 他 [8] はハッシュ表を用いた手法を提案している（ハッシュ法）．彼らの手法は，XML に対応する木のすべてのノードに対し，そのノードを根とする部分木に含まれるすべてのキーワードをハッシュ表に登録し，検索に使うものである．ハッシュ表の探索を $O(1)$ 時間で行えるならば，彼らの手法は $O(|S_1|(k + \log d))$ 時間で走る．ただし，検索中，常にハッシュ表に問い合わせる必要があるため，巨大なハッシュ表をメモリに持っている必要がある．Zhou 他 [7] は，集合の共通集合演算を計算するアルゴリズムを用いた手法を提案した．彼らの手法は，各キーワードごとに，そのキーワードに一致するノードの集合を求め，集合の共通部分を求めることによって SLCA を計算する（インターセクション法）．彼らの手法は，$O(k|L_1|\log|L|)$ 時間で走る．ここで，$L_1, \cdots, L_k$ は各キーワードに一致するノードを含む部分木の根の集合であり，$|L_1| \leqq \cdots \leqq |L_k|$ である．$S_i$ $(1 \leqq i \leqq k)$ と比較すると，$|S_i| \leqq |L_i| \leqq d|S_i|$ なる関係が成り立つ．また，Zhou 他 [5] は，トップダウン的な手法として，根から順に各ノードを根とする部分木が各キーワードに一致するすべてのノードを含むかどうかをチェックする手法を提案している．この手法は，$O(k|S_1|\log|S_k| + dk|S_1|\log\frac{|S_k|}{|S_1|})$ 時間で走る．

### 1.3 本論文の結果

本論文では，トライ木を利用した新たな SLCA 計算法を与える．従来の手法である IL 法，BMS 法，IMS 法の問題点は，SLCA を見つけるために LCA の計算を何回もしなけれ

ばならないことである．ハッシュ法，インターセクション法，トップダウン法はその点を改善している．本論文では，Dewey number をトライ木と呼ばれるデータ構造を使って管理することにより，LCA の計算を高速化している．すなわち，クエリ $Q = \{w_1, \ldots, w_k\}$ に対し，キーワード $w_i$ に対するノードの Dewey Number の集合 $S_i$ をトライ木で管理する．そのとき，ノード $v \in S_1$ に対し，$S_2$ のノードとの LCA で SLCA となりうる可能性のあるノードを $O(d)$ 時間で見つけることができる．この操作を繰り返し用いると，$O(kd|S_1|)$ 時間ですべての SLCA を見つけることができる．このように，我々の手法の理論値は，$d$ が大きくなければ，ハッシュ法の最適な時間に匹敵する．我々は，DBLP を用いて，実験的にその性能も評価する．

## 2. 準　　備

本章では必要な定義について述べる．XML データは要素をノードとする木構造で表すことができ，木の各ノードは要素名でラベル付けされている．本論文における木構造では，テキスト部分についてはキーワードに分割し，各キーワードを 1 つのノードに対応付けて表す．今，$T$ を XML に対応する木とする．そのとき，木 $T$ のノード $v$ のラベルがキーワード $w$ に一致するとき，$v$ は $w$ に一致すると言う．各ノードの深さは，根を 1 とし，根以外は「その親ノードの深さ+1」で定義される．木の高さは深さの最大値とする．$T$ の各ノードに対し，DeweyNumber が次のように定義される．まず，$T$ の根は 1 である．ノード $v$ に対し，$v$ の DeweyNumber が $d$ ならば，$v$ の $i$ 番目の子の DeweyNumber は $d.i$ となる．このように DeweyNumber は数字が深さごとにドットによって区切られた形になっている．ノード $v_1$ と $v_2$ の LCA (Lowest Common Ancestor) とは，$v_1$ と $v_2$ の共通の祖先の中で最も葉に近いものであり，$lca(v_1, v_2)$ と表す．一般に $k$ 個のノード $v_1, \ldots, v_k$ に対する LCA を $lca(v_1, \ldots, v_k)$ と表すことにする．

さて，クエリ $Q = \{w_1, \ldots, w_k\}$ に対する SLCA(Smallest LCA) について定義する．今，$S_i$ $(1 \leqq i \leqq k)$ をキーワード $w_i$ に一致するノードの集合とする．そのとき，ノード $v$ が $Q$ に対する SLCA とは次の 2 つの条件を満足することである．

（1） ノード $v_1 \in S_1, \ldots, v_k \in S_k$ が存在し，$v = lca(v_1, \ldots, v_k)$ となる．

（2） 任意のノード $v_1 \in S_1, \ldots, v_k \in S_k$ に対し，$lca(v_1, \ldots, v_k)$ は $v$ の子孫にはならない．

本論文では，与えられたクエリに対し，対応する SLCA をすべて見つける問題を考える．

## 3. トライ木を用いた SLCA 検索法

本手法ではトライ木と呼ばれるデータ構造を用いて SLCA の検索を行なう．トライ木については 3.1 節で紹介する．

本手法の検索は，

（1） 各検索キーワードごとの DeweNumber からトライ木を生成

（2） トライ木を用いて，検索キーワード同士の LCA を計算

（3） 得られた LCA から SLCA の特定

という手順に分けられる．

検索を効率化するために (1) のトライ木生成は，検索対象である XML データに含まれるすべてのキーワードのトライ木を作成し，データベースに保存する．検索を行う際には，検索したいキーワードのトライ木をデータベースより読み出した後，検索に使用する．

### 3.1 トライ木

トライ木とは順序付き木構造の一種であり，プレフィックス木 (Prefix Tree) とも呼ばれる．任意のノードの配下の全ノードは，自身に対応する文字列に共通するプレフィックスが存在する．本手法ではこの特徴を利用し，各キーワードに対する DeweyNumber のセットをトライ木に変換する．

SLCA の検索においては，その過程で LCA 何度も計算する必要がある．作成したトライ木を用いることで，LCA の探索を高速化することができ，結果として SLCA の計算速度向上に貢献している．例として，任意のキーワードの DeweyNumber セットをトライ木にしたものを図 1 に示す．このトライ木に対

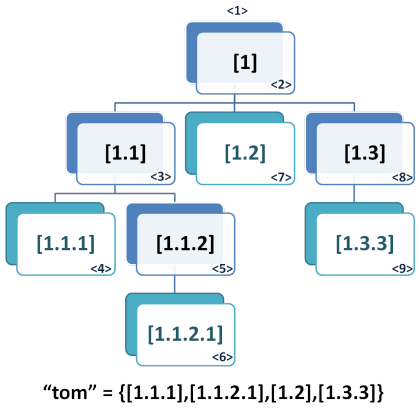


図 1　DeweyNumber を用いたトライ木の例

して，任意のノードが持つ DeweyNumer と共通部分が存在するかを調べる．DeweyNumber の共通部分は，そのままノード同士の LCA となる．

例として，図 1 のトライ木とノード [1.1.2.3] の共通部分を探索する．トライ木の根から順に探索すると，[1.1.2] までが共通部であることが分かる．ここで，図 1 のトライ木を構築しているキーワード”tom”の DeweyNumber セットとノード [1.1.2.3] の LCA を考えると，同じ [1.1.2] となっていることが分かる．

このようにトライ木を用いることで，任意ノードとトライ木に変換したキーワードとの LCA を高速に求められる．最悪のケースを考えたとしても，トライ木の高さ分の計算量で探索を終えることができる．

### 3.2 トライ木遷移表

DeweyNumber からトライ木を構築するにあたり，検索をしやすくするためトライ木構造を持つ遷移表を作成する．ここではその遷移表をトライ木遷移表と呼ぶこととする．図 1 のトライ木から，トライ木遷移表を作成したものが図 2 である．

トライ木遷移表は縦軸に ”ノード番号 ”，横軸に ”DeweyNumber 値 ”，要素に ”遷移先のノード番号 ”を保持している．

DeweyNumber値

| ノード番号 | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 2 | ／ | ／ |
| 2 | 3 | ϕ | 5 |
| 3 | ϕ | 4 | ／ |
| 4 | ϕ | ／ | ／ |
| 5 | ／ | ／ | ϕ |

“tom” = {[1.1.1],[1.1.2.1],[1.2],[1.3.3]}

図 2　図 1 のトライ木遷移表

実際のこの遷移表がトライ木の構造を持つ例を示す．図 1 に含まれるノード [1.1.2.1] が図 2 ではどのように得られるかを考える．

最初の 1 行目において，1 列目に要素 2 が存在するため，2 行目に遷移する．遷移先が存在する場合，その列番号を保存する ([1])

2 行目において，同じく 1 列目に要素 3 が存在するため，3 行目に遷移する．([1.1])

3 行目において，同じく 2 列目に要素 4 が存在するため，4 行目に遷移する．([1.1.2])

4 行目において，1 列目に終端の記号 $\phi$ が存在するため，現在の列の番号を保存して終了する．([1.1.2.1]) このように図 2 はトライ木の構造を保ったまま，遷移表となっていることが分かる．

実際に LCA を探索する際には，ノードの DeweyNumber を先頭から遷移表に当てはめていき，要素が存在しない，または終端の記号が現れるまでの列番号が LCA となる．

また，リストを用いてトライ木遷移表を表したものを図 3 に示す．

| 番号 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 要素 | 1: 2 | 1: 3,<br>2: ϕ,<br>3: 5 | 1: ϕ,<br>2: 4 | 1: ϕ | 3: ϕ |

“tom” = {[1.1.1],[1.1.2.1],[1.2],[1.3.3]}

図 3　リストを使用したトライ木遷移表

### 3.3　SLCA の計算

トライ木遷移表より LCA を求めるとした上で，SLCA を計算するアルゴリズムを「Algorithm1」に示す．ここでアルゴリズム内で使用する用語を以下に定義する．

$S_1$：検索キーワード中で最も出現頻度の低いノードの集合
$ST$：すべての検索キーワードのトライ木遷移表の集合
$v$：任意のノード (DeweyNumber)
$stt_i$：$i$ 番目のキーワードに対するトライ木遷移表
$d_{root}$：検索対象 XML データの根のノード (DeweyNumber)

**Algorithm 1** トライ木を用いた SLCA 検索法

**Input:** $S_1 = \{v_1, v_2, ..., v_m\}, ST = \{stt_1, stt_2, ..., stt_n\}$
**Output:** The result of SLCAs.

```
1: initialize: x = 1; LCAs[m] = {}; SLCAs[m] = {}
2: LCAs[0] ← d_root
3: for i ← 1 to m do
4:     SLCAs[i − 1] ← v_i
5: end for
6: for j ← 2 to n do
7:     for k ← 1 to m do
8:         α ← GetLCA(SLCAs[k − 1], stt_j)
9:         if (LCAs[x − 1] ⪯_a α) then
10:            LCAs[x − 1] ← α
11:        else if (LCAs[x − 1] ⋠_a α) then
12:            x ← x + 1 ;  LCAs[x − 1] ← α
13:        end if
14:    end for
15:    SLCAs[x] = {}
16:    for i ← 1 to x do
17:        SLCAs[i − 1] ← LCAs[i − 1]
18:    end for
19:    m ← x ;  x ← 1
20: end for
21: return SLCAs
```

$GetLCA(v, stt_i)$：$v$ と $stt_i$ を用いて LCA を求める関数

$S_1$ は DeweyNumber 値，$ST$ はトライ木内に存在するノードの個数で，それぞれ降順にソートされているものとする．また，比較演算子 ”$\preceq_a$ ”は，2 つのノードの子孫関係を比べる演算子である．$x \preceq_a y$ とすると，$y$ は $x$ の子であり，$x$ は $y$ の親である．また，$x$ と $y$ が同一のノードである場合も演算子は成り立つとする．

例：[1.3] $\preceq_a$ [1.3.2]，[1.4.1] $\not\preceq_a$[1.5.3]，[1.2] $\preceq_a$ [1.2]

まず，各検索キーワードに対応する DeweyNumber のセットを出現頻度の低い (DeweyNumber の数が少ない) 順にソートをかける．次に最も頻度の低いキーワードを基準として，その次に頻度の低いキーワードのトライ木に対して，各ノードごとに LCA の探索を行う．求められた LCA は子孫関係を調べ，SLCA の条件とマッチするものを候補として残す．再び SLCA 候補となったものを基準にして，次に頻度の低いキーワードのトライ木に対して，各ノードごとに LCA の探索を行う．この走査を繰り返し行うことで SLCA 候補を絞り込んでいき，キーワードがなくなると検索を終了する．

本手法では，このように SLCA 候補を絞り込んで次のキーワードに対する検索を行うことで，キーワードの数が増加しても無駄な LCA の探索を減らすことができ，検索時間を短縮できる．

さて，アルゴリズム Algorithm1 の計算時間について詳しく考察する．入力として，$k$ 個のキーワードからなるクエリ $Q = \{w_1, \ldots, w_k\}$ を考える．このとき，各キーワードに対する

トライ木遷移表が与えられている．ここで，$S_i$ をキーワード $w_i$ に一致するノードの集合とする．$|S_1| \leqq \cdots \leqq |S_k|$ であるから，Algorithm1 は，$S_1$ の各ノードに対し，$S_2$ のノードとの SLCA 候補を求め，次にその SLCA 候補と $S_3$ との SLCA 候補を求め，というように $S_k$ までの計算していく．各 SLCA 候補を求める計算時間は，トライ木を使うと $O(d)$ 時間である．したがって，$S_1$ の 1 つのノードに対し，$S_k$ まで順に計算するから，$O(kd)$ 時間かかる．よって，最終的に，アルゴリズムは $O(kd|S_1|)$ 時間かかる．このようにトライ木遷移表が与えられると，高速な検索が可能である．

## 4. 実　　験

### 4.1 準　　備

本手法の有効性を評価するために，実際に大規模なサンプル XML データを用いて実験を行った．実験に用いた XML データは，DBLP [1](1,008MB) を使用した．比較対象とした SLCA 検索アルゴリズムは BMS(Basic Multiway-SLCA Algorithm) [3] である．

なお，検索効率化のため対象の XML データはパーサーにかけたのち，各キーワードごとに DeweyNumber を付けた後データベースに保存している．検索時は入力されたキーワードに対応する DeweyNumber を読み込んだ後，各アルゴリズムを用いて検索を行うものとする．

また 3. 章で述べたように，提案法はトライ木遷移表もデータベースに保存している．比較対象の BMS アルゴリズムは，検索キーワードの DeweyNumber を逐次比較し処理を行う手法であり，遷移表などは用いていない．その点を考慮し本実験においては，提案法の実行時間をアルゴリズム本体の検索時間に加え，検索キーワードのトライ木遷移表データを DB から読み込み，プログラム上で参照できる遷移表の状態に復元する時間も計測し，実験結果に加えた．

実験環境を表 1 に示す．

表 1　実 験 環 境

| | |
|---|---|
| CPU | Intel Core i7 @ 2.60GHz |
| Memory | 12.0GB |
| OS | Mac OS X(10.9) Mavericks |
| Programming Language | Python 2.7.5 |
| Database | Berkeley DB 11gR2(BTree) |

### 4.2 結　　果

提案法と BMS を比較した結果を図 4〜図 9 に，検索に使用したキーワードクエリを表 2,3 に示す．実験で使用した検索キーワードは，XML データに含まれる語句の中で，出現頻度を満たすものを抽出した．検索キーワードの個数を 2 個もしくは 4 個とし，キーワード出現頻度が 100〜10000 であるクエリに分けて実験を行った．

なお，$k$ はキーワードの個数，$Fq$ は各キーワードの出現頻度をそれぞれ表している．また $Fq$ の中で $n1000$ は頻度が 950〜1050，$n10000$ は頻度が 9500〜10500 のものを示している．$k = 4$ の場合は，一つ目のキーワードのみ出現頻度を変更して実験を行い，残り三つのキーワードは $n10000$ とした．

図中グラフにおいて，縦軸が検索時間 (msec)，横軸が検索に使用したクエリである．凡例は,「Create_PtStt」が提案手法における，DB から各キーワードに対応するトライ木遷移表を取り出して，リスト構造の遷移表を構築する時間,「Search」が各手法の検索アルゴリズム実行時間を表している．

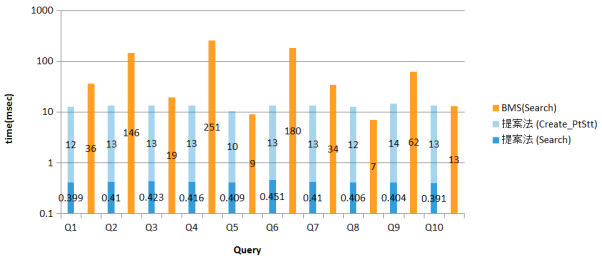


図 4　k=2, Fq={100,n10000}

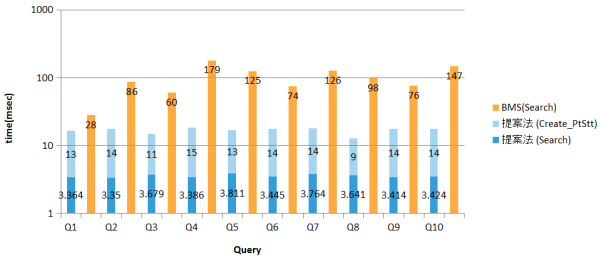


図 5　k=2, Fq={n1000,n10000}

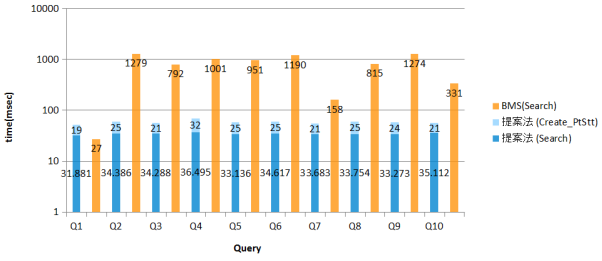


図 6　k=2, Fq={n10000,n10000}

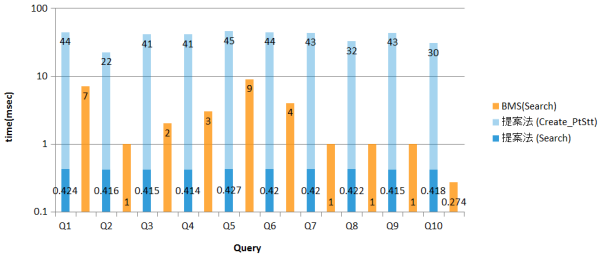


図 7　k=4, Fq={100,n10000,...}

表 2　実験に用いたキーワードクエリ (k=2)

| | Fq={100,n10000} | Fq={n1000,n10000} | Fq={n10000,n10000} |
|---|---|---|---|
| Q1 | hiromu,trees | mipro,stochastic | dexa,cluster |
| Q2 | cbms2004,automated | reynolds,philippe | mohamed,shen |
| Q3 | rens,evolution | classroom,link | icalt,andrea |
| Q4 | tsmcb28,tang | technological,processes | type,roberto |
| Q5 | gallant,eprint | parts,chris | delay,code |
| Q6 | jmlr7,patterns | zhigang,query | path,jung |
| Q7 | juyang,solving | maurer,aisel | icpp,matthew |
| Q8 | Tarn,1984 | 780,miccai | gao,functional |
| Q9 | fon,aisnet | petersen,single | probabilistic,distribution |
| Q10 | bijective,ben | multichannel,andrea | cost,sigcse |

表 3　実験に用いたキーワードクエリ (k=4)

| | Fq={100,n10000,...} | Fq={n1000,n10000,...} | Fq={n10000,...} |
|---|---|---|---|
| Q1 | gone,functional,choi,philippe | mbec,part,nar,jonathan | trees,smith,chris,francisco |
| Q2 | compsystech2009,matthew,edcation,iccs | jco,mohamed,dexa,path | processes,patterns,eprint,choi |
| Q3 | martinovic,practice,link,path | kwak,patterns,distribution,trees | nar,shen,aisnet,jonathan |
| Q4 | 1516241,allocation,probabilistic,icpp | blake,1984,education,functional | distribution,probabilistic,statistical,andrea |
| Q5 | 2830,single,distribution,point | 849,sigcse,gao,probabilistic | 1984,icalt,operational,single |
| Q6 | iorio,jian,evolution,francisco | 0905,philippe,delay,processes | object,cluster,gao,miccai |
| Q7 | bebis,andrea,1984,jung | cgiv,eprint,icpp,practice | icpp,tang,query,sigcse |
| Q8 | xintao,source,icalt,sensing | dominic,operational,source,reliability | point,evolution,solving,automated |
| Q9 | 0412,secure,aisel,trees | iscapdcs,tang,miccai,jian | path,jian,practice,type |
| Q10 | xueying,aisnet,sigcse,wsc | 824,code,icalt,aisnet | jung,delay,cost,roberto |

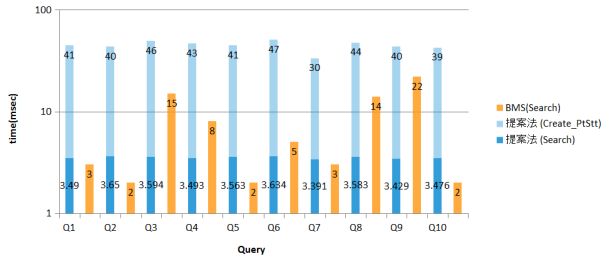


図 8　k=4, Fq={n1000,n10000,...}

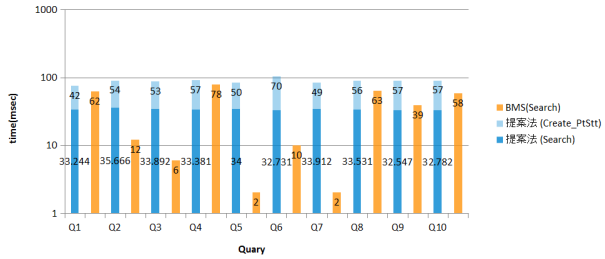


図 9　k=4, Fq={n10000,...}

### 4.3　考　　察

実験結果より，BMS アルゴリズムはクエリによって検索時間に大きな差が見られるのに対し，提案手法はほぼ一律に同じ検索時間であることが分かる．これは，BMS が SLCA の候補を検索する際に二分探索を行うため，探索対象のデータによって実行速度に差が出てしまう．一方，提案手法は SLCA の候補をトライ木を用いて探索するため，対象のデータ構造に関わらず木の高さの時間で求められるからだと考えられる．すなわち，DBLP に対する木は，その高さが低いため高速に計算できる．

次にキーワードの個数ごとに結果を見てみる．BMS は k=2 の場合，検索時間が増えてしまうことが多いが，k=4 の場合は余分な計算を省く効果が出ているため，k=2 よりも比較的に高速になっている．提案手法は k=2,k=4 のいずれの場合も，結果は殆ど変わらず安定した検索時間であり，k=2 では多数のクエリで BMS より高速になっている．しかし k=4 では検索時間では勝っているものの，トライ木遷移表構築時間が増えてしまい，結果として全体の速度が BMS より遅くなってしまうことがある．これは本手法が各キーワードがもつ DeweyNumber すべてをトライ木の要素としてデータベースに保存しているため，検索キーワードが増えるに連れて，必然的にトライ木遷移表を構築する時間が増えてしまうからである．

加えて，手法ごとの計算量を比較する．

$k$：キーワードの個数，$|S_1| \leqq \cdots \leqq |S_k|$
$d$：XML データ対応する木の高さ とすると，

BMS は $O(kd|S_1|log(|S_k|))$，提案手法は，検索部は $O(kd|S_1|)$ となる．ただし，現在の実装では，提案法は，検索時間の他に，DB よりトライ木遷移表を取り出し，構築する時間が掛かる．これは，すべてのキーワードに対し行っているため，キーワードの数が増えるほど時間がかかる．

## 5. まとめと今後の課題

本研究では，トライ木を使用した XML データに対する，より高速なキーワード検索法を提案した．本手法を用いることで，XML データからより高速で安定した速度で SLCA の検索が行うことができた．しかし，検索時間に加え，トライ木遷移表を DB から取り出し復元する時間がキーワードが増えるに連れて余計に掛かってしまい，結果的にそこがネックとなり検索全体の速度低下につながっている．要因として考えられるのは，現在トライ木遷移表をそのままの状態でデータベースに保存していることである．結果データベースより取り出した遷移表データを，プログラム上で扱える構造に戻すのに時間が掛かってしまっている．

今後の課題としては，トライ木遷移表をデータベース格納する際に最適な形で保存することが挙げられる．遷移表データの最適化を行うことでトライ木遷移表作成速度の向上する可能性がある．

また，今回の実験は速度面のみの比較であったが，得られた結果がどれだけ効果的なもであるかを検証する実験も必要であるといえる．加えて，現在は SLCA の検索のみであるが，SLCA を更に拡張したノードの検索にも対応できるよう改良を行うことも，今後の展望として挙げられる．

## 文　献


[1] The DBLP Computer Science Bibliography : <http://dblp.uni-trier.de/db/>

[2] Z. Liu and Y. Chen. Processing keyword search on XML: a survey, Proc. of WWW, pp. 671-707, 2011.

[3] C. Sun, C.Y. Chan, and A.K. Goenka. Multiway slca-based keyword search in xml data, Proc. of WWW, pp. 1043-1052, 2007.

[4] Y. Xu, Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Database, Proc. of SIGMOD, pp.537-538, 2005.

[5] J. Zhou, Z. Bao, Z.Chen, G. Lan, X. Lin and T.W. Ling: Top-Down SLCA Computation Based on List Partition, Proc. of DASFAA, pp. 172-184, 2012.

[6] J. Zhou, G. Lan, Z.Chen, and X. Tang, Fast Smallest Lowest Common Ancestor Computation Based on Stable Match, J. Computer Science and Technology, 28(2), pp. 366-381, 2013.

[7] J. Zhou, Z. Bao, W. Wang, T.W. Ling, Z. Chen, X. Lin, and J. Guo, Fast SLCA and ELCA Computation for XML Keyword Queries based on Set Intersection, Proc. of ICDE, pp.905-916, 2012.

[8] W. Wang, X. Wang and A.Zhou. Hash-Search: An Efficient SLCA-Based Keyword Search Algorithm on XML Documents, Proc. of DASFAA, LNCS 5463, pp.496-510, 2009.