

XML 文書フィルタリングのための軽量な高速化技法

御手洗秀一[†] 石野 明^{††} 竹田 正幸[†]

[†] 九州大学大学院 システム情報科学研究院 情報理学部門

^{††} 東北大学大学院 情報科学研究科 システム情報科学専攻

E-mail: [†]{mitarai,takeda}@i.kyushu-u.ac.jp, ^{††}ishino@ecei.tohoku.ac.jp

あらまし XML データに極めて単純な前処理を施すことで、質問数の増大に頑健な XML フィルタリング方式を開発した。その手法は、データ配信側で XML データを事前に走査し、根から葉へ向かうパスに沿ったタグ名の列全体を表すパストライと、XML 文書中の開始タグと終了タグを特殊な記号で置き換えたバイナリ XML データを作成する。開始タグを表すコードの直後には、タグを表す番号ではなく、対応するパストライ節点の番号を埋め込む。データ受信側では、このパストライとバイナリ XML データを受け取るものとする。質問式に現れるパスパターンの照合は、パストライを対象として行う。パストライは XML データに比べ非常に小さいため、高速に処理を完了することが可能である。パスパターンの照合結果はバイナリ XML データ走査時に参照することができ、これをテキスト節点を対象としたキーワード照合の結果と結合させることで質問処理は完了する。代表的なストリーム型 XML プロセッサである XMLTK との比較において、実行速度で 4~6 倍高速であり、メモリ使用量は 1/6 程度であることが判明した。キーワード ストリーム処理, XML, XML フィルタリング, パストライ, 決定性有限オートマトン

Light-weight acceleration for streaming XML document filtering

Shuichi MITARAI[†], Akira ISHINO^{††}, and Masayuki TAKEDA[†]

[†] Department of Informatics, Kyushu University

^{††} Department of System Information Sciences, Tohoku University

E-mail: [†]{mitarai,takeda}@i.kyushu-u.ac.jp, ^{††}ishino@ecei.tohoku.ac.jp

Abstract This paper proposes a scalable XML filtering basing on preprocessing of XML data. XML data is pre-processed and transformed into a pair of a path trie and a binary XML data. The path trie is the trie representing the set of strings of tag names along with root-to-leaf paths, and the binary XML data is obtained from the original XML data by replacing every start tag and the corresponding end tag with special byte codes, respectively. Each occurrence of the special byte code for a start tag is followed by ID of the corresponding node of the path trie. Path pattern matching is performed against the path trie. Since the path trie is much smaller than the XML data, a drastic speedup is possible. Query pattern processing is done by combining the keyword occurrences found in scanning of the binary XML data with the information added to path trie node implied by node IDs embeded in the binary XML data. Experimental results show that, the processing time and memory requirement of our method are, respectively, 1/6-1/4 and 1/6 compared with XMLTK, a state-of-the-art streaming XML processor.

Key words stream processing, XML, XML filtering, Path-trie, DFA

1. ま え が き

インターネット上のニュース記事や広告等のメディア、日々観測されるセンサー情報や株価表示器など、膨大な量の情報が絶え間なく生成されている。これらの情報を有効に活用するための技術が強く求められている。SDI (Selective Dissemination of Information) は、そのようなサービスの一つであり、日々生成される大量の機械可読文書を購読者が選択的に、

かつ、リアルタイムに取得することができる [6]。このような publish/subscribe 型のデータ配送システムは、従来、キーワードや “bag-of-words” に基づくフィルタリングによる比較的単純なものであった。

しかし近年、XML (eXtensible Markup Language) の台頭により、XPath [20] を用いたより高度なフィルタリングが可能となった。XPath は、W3C の勧告による、XML 文書の特定の部分 (要素、属性、テキストなど) を指定するための言語で、

事実上の標準となっている。

XML データストリームのフィルタリングに関する研究の初期段階では、DOM 木を用いた手法が主流であった。しかし、DOM 木の構築に多大な時間と主記憶領域を要するため、このような手法では、50–200MB 程度の XML データになると事実上適用不可能である。そこで、近年はストリーム処理に基づくフィルタリング手法が主流になっている (たとえば [3], [4], [8], [15])。ストリーム処理とは、XML データを一方向に走査することによって出力を得る方式を指し、DOM 木を用いた方式に比べある一定サイズのバッファしか必要とせず、次々と生成されるデータに対して即座に対応可能といった特徴をもつ。

代表的な XML データストリーム処理器である YFilter [8] や XMLTK [4] は、非常に高速に XPath を処理できるが、扱う軸は順方向軸 (forward axes) のみと限定している。これに対して、最近では先祖軸 (ancestor axes)、兄弟軸 (sibling axes) にも対応したストリーム処理手法が提案されている [5] が、順方向軸のみに限定した手法に比べ規模耐性、処理速度等を犠牲にしている。本論文で提案する手法は、扱う軸は順方向軸のみで、オートマトンを用いて XML データを走査する点では YFilter や XMLTK と同様であるが、XML データに前処理を施すことによって高速化・軽量化を可能とした点で異なる。

YFilter は、XFilter [3] のパス照合処理を改良したもので、質問式毎に非決定性有限オートマトン (NFA) を構築するのではなく、質問式全体の共通接頭辞パスを共通の状態として束ねたもので、XFilter よりも質問数に関して規模耐性があり、高速に動作する。しかしながら、NFA は同時に複数の状態遷移を処理しなければならないため、決定性オートマトン (DFA) と比較して照合速度は遅い。NFA を等価な DFA に変換すると、一般に状態数が指数的に増大する。そこで、XMLTK では、XML データ走査中に必要に応じて NFA を部分的に DFA に変換する *Lazy-DFA* [4] と呼ばれる手法を採用している。XMLTK は、質問数に関して規模耐性があり高速であるが、一方、DFA の状態数が増大し大量のメモリを必要とするという欠点がある。

本論文では、配信する XML データにあらかじめ簡単な前処理を施しておくことにより、さらなる規模耐性と高速化・省メモリ化を達成する。XML データに前処理を施してフィルタリングに都合のよい形式で配信するというアイデアは、たとえば [7] などに見られる。また、XML のバイナリフォーマットに関する提案も多くなされている [2]。さらに、XMLTK によるストリーム処理を加速するために用いられるストリーム索引 (Stream Index; SIX) [4] も、XML データの前処理によって得られる情報である。

本論文で提案する前処理手法は、XML データをパストライとバイナリ XML データの対に変換する。パストライは、XML データ中の根から葉へ向かうパスに沿って現れるタグ名の列全体を表すトライ構造である。また、バイナリ XML データは、XML データ中の開始タグと終了タグを各々特殊コードで置き換えたもので、開始タグを表すコードの直後に、タグ名を表す番号ではなく、対応するパストライ節点を識別するための番号 (以下、パストライ節点番号) を埋め込んでいる。データ送信

サーバは、このパストライとバイナリ XML データを対として送信する。データ受信側では、まず、パストライを対象にパス照合を行い、その結果をパストライの節点に付加する。これにより、バイナリ XML データの走査時にはもはやパス照合の必要はなく、バイナリ XML データ中に埋め込まれたパストライ節点番号を頼りにパス照合の結果を参照することが可能となる。通常、パストライは XML データ全体の木に比べて十分小さいため、処理速度は劇的に向上する。この前処理で行うデータの変換は、以下の望ましい性質を有する。

(1) 変換に要する時間が極めて小さい。

(2) 変換して得られたデータの大きさは、もとの XML データ以下に抑えられる。

さらに、この手法を SIX と併用することで、さらなる高速化が達成できる。

パストライは、DataGuide [9] と呼ばれるデータ構造の一種である。XML データ処理の効率化のために DataGuide を用いるアイデアは新しくないが、本論文のような使い方は、著者らの知る限り、これまで存在していない。

本論文の構成は以下の通りである。2. 節で、フィルタ処理を高速化するための手法として、XML データに対する前処理方法と、フィルタ処理手法について述べる。次に 3. 節で、実験により YFilter、XMLTK との比較を行う。また、SIX を本手法に適用し、XMLTK における SIX 効果と比較する。4. 節では、論理演算子や述語 (predicate) の入れ子、*count* などの集計処理への拡張にふれる。最後に 5. 節でまとめを行う。

2. パストライを用いた高速なフィルタ処理

石野ら [22], [23] では XQuery の部分族をパスブルーニングと DFA を用いて効率的に処理する手法を提案した。その手法は、まず入力となる XML データを事前に調べてパストライを得る。このパストライを用いることで、質問式に含まれるパスパターン中の *//* や *** を展開し、これによって DFA の状態爆発を抑えるというものであった。本論文では、石野らの手法をさらに発展させ、パスパターン照合をデータ走査時に実行するのではなく、質問式を得た段階で完了させておくことによってより高速なフィルタ処理を実現することに成功した。

以下では用語を定義したあと、フィルタ処理に必要な問題を定式化し、本手法のアルゴリズムを紹介する。

2.1 定義

Σ を文字の有限集合とし、 \mathcal{N} をタグ名から成る集合とする。XML 木とは、 \mathcal{N} のタグ名を内部節点のラベルとし、 Σ 上の文字列を葉 (テキスト節点) のラベルとするようなラベル付き順序木をいう。^(注1)

単純パスパターンとは \mathcal{N} のタグ名と特殊な記号 “***” から成る記号列をいい、各々の記号は “*/*” または “*//*” で区切られているものとする。ここで、“*/*” と “*//*” は、それぞれ、親子関係、先祖-子孫関係に対応する。単純パスパターン π が XML 木

(注1): 本論文では、“*name=value*” に対応する属性節点は、“*value*” をラベルとする葉を唯一の子とし、“*@name*” というラベルをもつ内部節点として扱う。

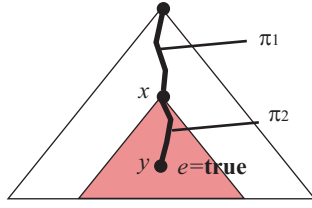


図 1 XML 木の位置 (x, y) における XPath パターン $\pi_1[\pi_2 : e]$ の生起

のパスにマッチするとは、“*”を任意のタグ名にマッチするワイルドカード、“//”を任意のタグ名列にマッチする可変長ワイルドカードとみなした際に、 π がそのパスに沿ったタグ名列に合致するときをいう。単純パスパターン π 中に現れるタグ名と“*”の個数を π の長さといい、 $|\pi|$ で表わす。

パスパターンとは単純パスパターン π_1, π_2 の順序対であり、 $\pi_1[\pi_2]$ と表す。XML 木の任意の節点 x とその子孫 y ($x = y$ の場合も含む) に対し、パスパターン $\pi_1[\pi_2]$ が位置 (x, y) に出現するとは、 π_1, π_2 が、それぞれ、根から x へのパスと x' から y へのパスにマッチするときをいう。ここで、 x' は、根から y へ向かうパスに沿った x の子節点である。

$e = e(w_1, \dots, w_m)$ を、 Σ 上の空でない文字列 w_1, \dots, w_m の生起の有無に関する論理式であるとする。 Σ 上のテキスト d が論理式 e を充足するとは、 d におけるキーワード w_1, \dots, w_m の生起に対応した真偽値割り当て (truth assignment) のもとで e が真となることをいう。

XPath とは、パスパターン $\pi_1[\pi_2]$ と論理式 e の組をいい、 $\pi_1[\pi_2 : e]$ と表す。XPath パターン $\pi_1[\pi_2 : e]$ が XML 木の節点 x に出現するとは、 x の子孫 y が存在して、パスパターン $\pi_1[\pi_2]$ が位置 (x, y) に出現し、かつ、 y の子であるテキスト節点の少なくとも一つが e を充足するときをいう。この定義より、パスパターン $\pi_1[\pi_2]$ は、XPath パターン $\pi_1[\pi_2 : \text{true}]$ と見なすことができる。図 1 に、XPath パターン $\pi_1[\pi_2 : e]$ とその生起位置 (x, y) との関係を図示した。

本論文では、以下の問題に取り組む。

[定義 1]

Given: XML データ T .

Query: XPath パターン P_1, \dots, P_ℓ .

Answer: 各 $i = 1, \dots, \ell$ に対し、 T に対応する XML 木において P_i が生起する節点 x を返せ。

ここで、入力として与えられる XML データ T は、複数の XML 文書の系列 (XML 木の系列) となっていることを仮定している。なお、フィルタリング問題では、該当の XML 文書が質問に合致するかどうかだけを考えればよいので、上の定義で節点 x を求める必要はないように見える。しかし、4. 節で述べるように、複数の XPath を組み合わせた複雑な質問を処理するためには、このような節点 x を求めることが必要である [19]。

2.2 パストライとバイナリデータ

本手法では XML データに対して一度だけ前処理を行い、パストライと、バイナリ XML ファイルを作成する。XML データと前処理によって作成されるパストライ及びバイナリ XML

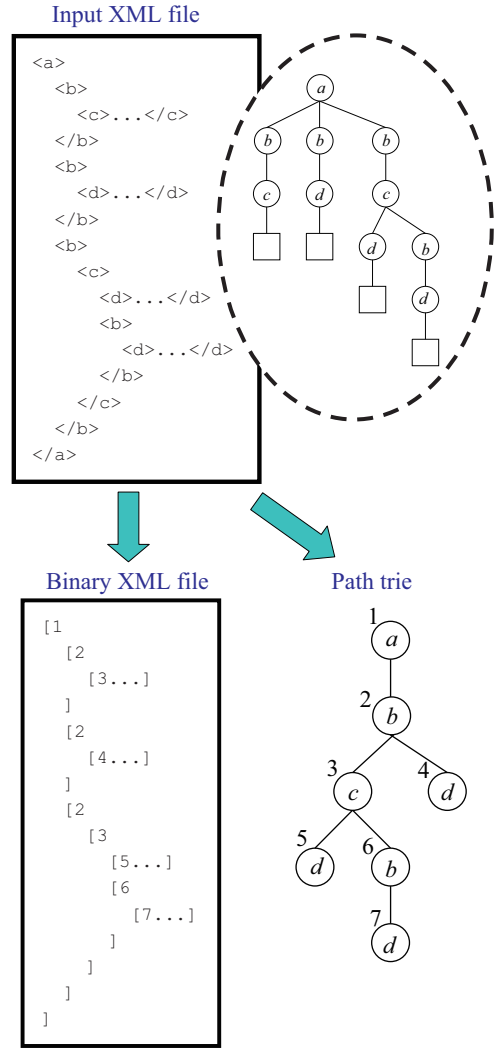


図 2 XML データと前処理で作成されるパストライ、バイナリ XML データ

データとの関係を図 2 に示す。図 2 右上は XML 木を示しており、正方形はテキスト節点を表している。また、この XML データから前処理を経て生成されるバイナリ XML データとパストライをそれぞれ、左下と右下に示した。パストライの節点の傍にある数は、その節点の ID (パストライ節点番号) である。バイナリ XML データにおいて、開始タグと終了タグはそれぞれ、特殊記号 “[” と “]” に置き換えられ、“[” の直後にはタグを表す番号ではなく、根からのパスに沿ったラベルの列を表すパストライ節点番号が埋め込まれていることに注意されたい。

このことによって、パスパターン照合はパストライを対象としたパターン照合を行うだけで良く、パストライが XML データに比べ非常に小さいため、高速に処理を完了することが可能となる。

この前処理に要する時間は $O(n \log |\mathcal{N}| + |T|)$ であり、表 2 に示すとおり高速な処理が可能である。ここで、 n は XML データ T 中の開始タグの個数を表し、 $|\mathcal{N}|$ はタグ集合の要素数を、 $|T|$ は XML データのサイズを表す。また、XML データと生成されるデータサイズの関係を表 1, 2 に示す。このことから、パ

表 1 DBLP [11] と xmlgen [17] によってランダムに生成された入力 XML データの大きさ

	XML データ		
	サイズ (MB)	節点数	タグ数
DBLP	352	8,632,812	35
random	111	1,666,310	74

表 2 前処理で構築されるバイナリ XML データとパストライのサイズ及びその構築時間

	バイナリ XML ファイル	パストライ	CPU 時間
	サイズ (MB)	節点数	(sec)
DBLP	208	138	100.41
random	84	515	73.26

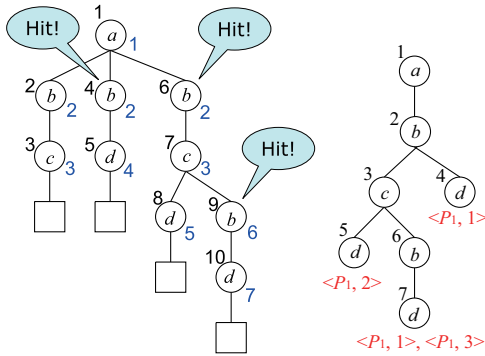


図 3 パストライ・XML 木の関係と、パスパターン $P_1 = a//b[/d]$ の照合結果

ストライは、XML 木に対して十分に小さく、バイナリ XML データに関しても入力ファイルサイズの 76%と小さくなっていることが分かる。

2.3 パスパターンの照合

本論文では、XML 木に対するパスパターン照合の問題を、パストライに対するパスパターン照合に置き換える。

図 2 で示した XML データを例にとり、XML 木中、および、パストライ中におけるパスパターン $P_1 = a//b[/d]$ の生起の様子を図 3 に示した。

この例では、パスパターン $P_1 = a//b[/d]$ は XML 木中の位置 $(x, y) = (4, 5), (6, 8), (6, 10), (9, 10)$ で生起している。これらの位置は、図右に示したパストライの節点に付加された情報から得ることができる。一方、パストライ中においては、生起位置は $(x, y) = (2, 5), (2, 7), (6, 7), (2, 4)$ である。そこで、パストライ中の生起位置 (x, y) の節点 y にパスパターンの番号と節点 x, y の深さの差の情報を付加する。すると、XML 木中の生起位置は、パストライに付加された情報から得ることができる。たとえば、XML 木中の位置 $(6, 8)$ における P_1 の生起は、パストライの節点 5 のもつ出力 $\langle P_1, 2 \rangle$ から得ることができる。すなわち、バイナリ XML データ中において、XML 木の節点 $y = 8$ に対応する箇所にはパストライ節点番号 5 が埋め込まれており、パストライの節点 5 に付加された情報 $\langle P_1, 2 \rangle$ によって、 y から 2 つ上の先祖である節点 $x = 6$ が得られる。

次に、パストライに対するパスパターン照合の手法について

述べる。質問式に含まれる各パスパターンに対し、パストライ中の出現位置 (x, y) をすべて求めなければならない。このために、竹田ら [21] が XML 木に対するパスパターン照合に用いた方法を採用する。

すなわち、各パスパターンから非決定有限オートマトン (NFA) をそれぞれ作成し、パストライ中を深さ優先で巡回しながら、それらの NFA を用いて照合を行う。NFA の実装にはビット並列化手法 [14] を用いる。単純パスパターン π に対する NFA の状態数は $|\pi| + 1$ となる。多くの場合、各 NFA の状態数はワード長である 32 (あるいは 64) 以内に収まるため、非決定的な状態遷移を高速に並列実行できる。また NFA の構築時間は $O(|N| \cdot |\pi|)$ であり、実用上も極めて高速である。

パストライの節点数と高さを、それぞれ、 n, h とするとき、本照合法の実行時間は一つのパスパターンに対して $O(h \cdot n)$ となる (NFA の構築時間を除く)。これは、パスパターン $\pi_1[\pi_2]$ の $\pi_1\pi_2$ を受理する NFA M_f^{12} と、 π_2 の転置パターン π_2^R を受理する NFA M_b^2 とを作成し、 M_f^{12} が受理するたびに、そこから根に至るパス上を M_b^2 が走る必要があるためである。文書指向の XML データなどではパストライのサイズが大きくなることも考えられるが、このビット並列化手法による照合法は極めて高速であるため、パスパターン処理時間は実用上問題にならない。

これら二つの NFA M_f^{12}, M_b^2 を用いて、パストライ中を深さ優先で巡回し、パスパターン $P_i = \pi_1[\pi_2]$ の出現位置 (x, y) をすべて求める。出現位置 (x, y) が求まるたびに、節点 y に出力 $\langle P_i, d \rangle$ を付加する。ここに、 d は節点 x, y の深さの差である。

本手法を用いることで、バイナリ XML データ走査時のパスパターン照合処理は不要となり、XML データ走査時にパスパターンの照合を行っている YFilter や XMLTK と比べ、処理時間を大幅に短縮することができる。

2.4 キーワード照合

パスパターン照合は、フィルタリング問題の一部に過ぎない。XPath の仕様から明らかに、述語はパス以外に、テキストデータや属性、ポジションを含むことができる。このうち、テキストデータや属性は、(i) $a/b[\text{contains}(\text{name}, \text{"mickey"})]$, (ii) $a/b/\text{name}[\text{./text}() = \text{"mickey mouse"}]$, (iii) $a/b[\text{@month} = \text{December}]$ などのように記述され、(i) は $a/b/\text{name}$ のテキスト節点に、キーワード “mickey” が含まれること、(ii) は name のテキスト節点が “mickey mouse” であること、(iii) は $\text{./text}()$ の属性値が December であることを調べるときに用いられる。

フィルタリングにおいてこのような問い合わせの処理を考えるとき、XML データ中のすべてのテキスト節点における複数キーワードの出現を求める問題として捉えることができる。著者らは、この問題に対して古典的手法である Aho-Corasick (AC) 法を用いた。AC 法は同時に複数キーワードの照合を行うことができ、キーワード数の増加に伴うテキスト走査速度の低下は特にない。また、索引を用いる方法では、キーワード毎に出現するテキストの番号のリストを求め、ブール式に応じた集合演算を行うが、AC 法で複数のキーワードを同時に照合する場合

には、テキスト節点毎にブール演算を行えばよい、集合演算のために領域・時間を消費しないという利点もある。フィルタリングにおいては、大量の質問式から集めたキーワードの総数は膨大なものになり得るため、これらの利点のもつ意義は大きい。

2.5 バイナリ XML データ走査アルゴリズム

バイナリ XML データを走査してキーワード照合を行い、その結果とパスパターン照合の結果を結合して質問処理を行うアルゴリズムの概略を以下に示す。

与えられた XPattern P_1, \dots, P_ℓ の論理式 e 中に含まれるキーワードからなる集合を $W = w_1, \dots, w_m$ とする。 W から AC 機械 M を構築する。バイナリ XML ファイル先頭からの位置を表す変数 $offset$ と、入力を XML 木として見たときの節点の深さを表す変数 $depth$ を用意する。ブール値を格納する 2 次元配列 Occ と Q を用意する。配列 Occ は、現在巡回している深さ d の節点の子であるテキスト節点にキーワード w_i が生起するときに $Occ[d][i]$ の値が真となるというものであり、配列 Q は、現在巡回している深さ d の節点に XPattern P_q が出現するときに $Q[d][q]$ の値が真となる。本アルゴリズムは、バイナリ XML ファイルを 1 バイトずつ走査し、読み込んだバイト c に対して以下を実行する。

- c が開始タグを表す特殊記号 “[” であるとき、パストライ節点番号 v を読み込み、 $(v, offset)$ をスタック S にプッシュする。 $depth$ を 1 インクリメントする。 $Occ[depth][1 \dots m]$ と $Q[depth][1 \dots l]$ を偽に初期化する。 M の状態を初期状態に設定する。
- c が終了タグを表す特殊記号 “]” であるとき、スタック S より $(v, offset)$ をポップする。もしパストライのノード v が出力をもっていたら、それぞれの $\langle q, d \rangle$ に対して、XPattern P_q の論理式 e を、 $Occ[depth][1 \dots m]$ の値に基づいて計算し、もし e が真ならば、 $Q[depth - d][q]$ を真と設定する。 $depth$ を 1 デクリメントする。 M の状態を初期状態に戻す。
- c がそれ以外のバイトコードであるとき、 M を文字 c で状態遷移させる。もし遷移先の状態が出力をもつならば生起したキーワードに対応して $Occ[depth][1 \dots m]$ を更新する。

3. 実装実験

実験に用いた計算機環境は、RedHat Linux Advanced Server 2.1, CPU 2.4GHz Intel Pentium4, 2.0GB RAM で、入力に用いた XML データは、xmlgen [17] によって生成されたランダムデータでサイズは 111MB である。質問式は、XMLTK が任意の位置ステップ (location step) での述語の使用を許していないため、単純パスパターンを用いた。パスは、YFilter によって提供されている pathgenerator [1] によってランダムに生成され、//, * の生起確率はそれぞれ (1%, 1%), (10%, 10%) としている。

3.1 節では、パスパターン照合に要する時間を計測した。

3.2 節では、バイナリ XML データ走査時間を計測した。また、パストライを用いることによる効果を検証するために、パストライを用いない手法を実装し、その走査時間との比較を

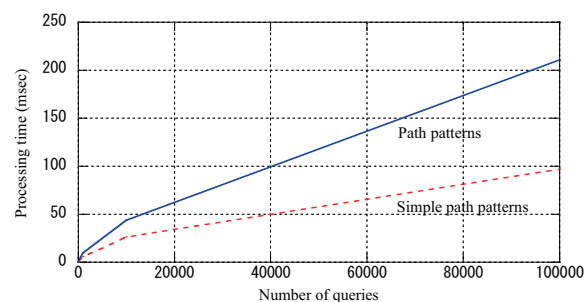


図 4 パストライとパスパターン照合に要する時間

行った。

3.3 節では、フィルタリング手法としての有効性を検証するために、YFilter と XMLTK との比較を、実際の実行時間、使用メモリ量という観点から実施した。

3.4 節では、本手法に SIX を組み込んだ際の高速化の効果について検証した。

3.5 節では、AC 法を用いた大量キーワード照合技術の高速性を検証するため、最新のテキスト索引技術との性能比較を行った。

3.1 パスパターン照合に要する時間

図 4 は、質問数を変化させたときのパスパターン照合に要する時間を測定したものである。NFA の構築に要する時間も含んでいる。この結果より、質問数が多くなっても実際にバイナリ XML データを走査する時間 (表 5) と比較して、ほとんど無視できる時間であることが分かる。これは、パストライのサイズが入力 XML 木の節点数 1,666,310 個に対して 515 個と小さい (表 2 参照) ことと、ビット並列化技術による NFA の構築・照合がきわめて高速に動作することによる。

3.2 バイナリ XML データ走査時間とパストライ効果

バイナリ XML データを走査する時間を計測し、表 3 に「本手法」として示した。また、パストライによる高速化の効果を検証するために、パストライを使用しない手法 (ナイーブ法) による走査時間も併せて示してある。ここで、ナイーブ法では、バイナリ XML データとして、パストライ節点番号を埋め込む代わりに、対応するタグの番号を埋め込んだものを用い、このバイナリ XML データを走査する際に、文献 [21] に示した手法でパスパターン照合を行った。

走査時間は、何を出力するかに依存する。ナイーブ法とパス

表 3 本手法とナイーブ法の走査時間の比較

	質問数	経過時間 (sec)	
		本手法	ナイーブ法
単純パスパターン Prob(//)=Prob(*)=0.01	1	0.51	0.86
	10	0.52	1.29
	100	0.54	4.30
	1,000	0.53	21.81
パスパターン Prob(//)=Prob(*)=0.01	1	0.51	0.86
	10	0.52	1.36
	100	0.52	5.51
	1,000	0.54	34.50

トライを用いた手法とで出力に要する時間は同じであるので、ここでは出力時間を除いた時間、すなわちパスが見つかったか否かを求める実装になるよう条件を揃えた。

この表の結果より、ナイーブ法は質問式の個数 ℓ と等しい個数の NFA を作成し、タグを読み込む度にそれぞれの NFA に対して状態遷移を行う必要があるため、走査時間は ℓ に関して線形に増加しているが、パストライを用いた手法では、パスパターン照合の段階ですでにどのパスでどの入力パスがマッチするかは分かっており、タグが現れても状態遷移を行うわけではないため、入力質問式の数に依らず高速に処理できることを確認できた。

3.3 XMLTK および YFilter との性能比較

表 4 は、質問数が 10,000, 100,000 のときのメモリ使用量 (//, * の生起確率は (1%, 1%), (10%, 10%) の 2 通り) 表している。メモリ使用量は、論理空間上にどれだけメモリを確保したのかではなく、プログラム実行中に実際に物理メモリにアクセスしたサイズの最大値を測定している。これは、例えば Linux 環境ならば、RSS (Resident Set Size) の値を参照することで測定できる。

質問数が 100,000 のとき、YFilter は本計算機環境では、メモリ枯渇により動作不能となった。質問数が 10,000 のとき (//, * の生起確率は 10%) を見ると、本手法が 4,925KB であるのに対して、XMLTK 34,412KB, YFilter 1,494,845KB と優に 6 倍以上であることが分かる。

次に、表 5 は、単純パスパターンを 1~100,000 個ランダムに与えたときの実行時間を測定したものである。ここでは公平な測定とするため、全てのプログラムで出力形式を質問式毎にマッチした節点の個数を求めるよう条件を合わせている。

この結果より、本手法が YFilter, XMLTK と比較して圧倒的に高速であることが分かる。また、質問数が 100,000 のときに、本手法と XMLTK の処理時間が大幅に遅くなっているのは、マッチする節点数が多くなりすぎて、出力依存の要因が全

表 4 メモリ使用量の比較

	質問数	メモリ使用量 (KB)		
		本手法	YFilter	XMLTK
単純パスパターン 1%	10,000	3,320	1,169,975	30,288
	100,000	18,632		285,328
単純パスパターン 10%	10,000	4,952	1,494,845	34,412
	100,000	28,543		318,560

空欄はメモリ枯渇のため、実施不能であったことを示す。

表 5 実行時間の比較

質問数	経過時間 (sec)		
	本手法	YFilter	XMLTK
1	0.57	39.24	2.27
10	0.57	42.54	2.57
100	0.57	45.22	3.09
1,000	0.67	61.30	4.14
10,000	1.85	155.30	11.83
100,000	142.04		270.81

空欄はメモリ枯渇のため、実施不能であったことを示す。

体の処理時間の大半を占めるようになっていたため、マッチする節点数の増加率と実行時間の増加率はほぼ一致することを確認している。

3.4 SIX による高速化検証

本手法をさらに高速化させるための技法として、Stream Index (SIX) [4] がある。SIX とは、開始タグとそれに対応した終了タグの位置情報の組から成る集合であり、本手法もこのデータ構造を用いることで、関係のない部分木を読み飛ばすことが可能である。

表 6 が表 5 で示した実験に SIX を追加したときの実行時間である。この結果より、質問数が少ないときには 1.6~7.2 倍高速化されており、たとえ質問数が多くほとんど読み飛ばしができない場合であっても性能劣化がないことが分かる。また、表 5 との比較より、同じ入力データを用いているにも関わらず XMLTK に比べ、SIX による効果が高いことが分かる。これは、XMLTK が // や * に相当する状態に遷移した後、どのようなタグが現れてもその状態から抜けられない限り読み飛ばせるか否かの判断がつかないのに対し、本手法ではそれらをパストライにより予め展開しており、任意のタグで判別可能であるためと考える。

表 6 SIX による高速化の比較

質問数	経過時間 (sec)	
	本手法	XMLTK
1	0.00	0.14
10	0.07	1.53
100	0.18	2.49
1,000	0.39	4.37
10,000	1.81	12.25
100,000	139.62	275.74

3.5 AC 法と索引技術との性能比較

本論文で提案する手法では、テキスト節点に対する大量のキーワード照合に AC 法に基づく一方逐次処理技術を採用している。その性能がきわめて高いことを、最新のテキスト索引技術を用いた場合の性能と比較してみよう。

テキストデータの任意の部分文字列を扱う索引技術としては、接尾辞配列 (SA) [13] 圧縮接尾辞配列 (CSA) [10], [16] が知られている。接尾辞配列は、テキスト中に現れるすべての部分文字列の出現位置を接尾辞の辞書式順に格納したものであり、入力テキストと併せた索引データとして $5N\text{byte}$ の主記憶領域を要する。ここに、 N は入力テキスト長である。また、圧縮接尾辞配列は、接尾辞配列を圧縮したもので、入力テキストは索引データから復元可能であることと、索引サイズを入力テキストサイズ以下に抑えられることから、注目を集めている。

表 8 では、フィルタリングシステムに必要な処理として、キーワード集合とテキスト (複数の文書を接続したもの) が与えられたときに、キーワードごとに、(1) 出現回数、(2) 出現位置リスト、(3) 文書番号リスト、を求める問題を考え、それぞれについて、性能比較を行った。また、それぞれの問題に対する時間計算量と必要となる索引データサイズ (テキストデー

タを含む)を表7に示す．ここで, occ はキーワードの出現回数, m はキーワード長, D は文書数である．CSA 構築時にはパラメータ d, l の値を定める必要があるが, ここでは, 実験に用いたデータに対し, CSA のサイズが入力テキストサイズ N を超えない範囲で最も高速に照合が可能なパラメタの組である $(d, l) = (8, 128)$ についての結果を示した．また, CSA, SA を用いて文書番号リストを求める際には, 各文書の開始オフセット値から成る配列に対する二分探索法を用いている．

実験には, <http://pizzachili.dcc.uchile.cl/texts/nlang> より入手したテキスト 100MB を使用した．キーワード集合は, 単語単位で頻度を測定したときの上位 100 語 (高頻度語), 中央値前後 50 語ずつ (中頻度語), 下位 100 語 (低頻度語) を用いた．

この結果より, AC 法は文書番号リストを求める問題においては, 低頻度語の場合を除くと, 他の 2 手法に比べ高速であることが分かる．このことから, フィルタリングのように大量のキーワードを同時に照合する場合には, AC 法による逐次処理は, テキスト索引を用いた手法に匹敵する性能を示すことが分かる．

表 7 AC, SA, CSA の時間計算量及び索引データサイズの比較．
AC 法の実行時間は $O(m + N + occ)$, 索引は不要．

	SA	CSA
(1)	$O(m \log N + occ)$	$O(m \log N + occ \log^\epsilon N)$
(2)	$O(m \log N + occ)$	$O(m \log N + occ \log^\epsilon N)$
(3)	$O(m \log N + occ \log D)$	$O(m \log N + occ (\log^\epsilon N + \log D))$
索引	$5N$ (byte)	N (byte) 以下

表 8 AC, CSA ($d = 8, l = 128$), SA の実行時間比較．キーワードの総出現数は, 高頻度語 39,670,500, 中頻度語 2,532,290, 低頻度語 2,159．

	出現回数 (sec)		
	AC	CSA	SA
高頻度 100 語	1.31800	0.00450	0.00161
中頻度 100 語	0.81000	0.00525	0.00156
低頻度 100 語	0.36300	0.00630	0.00051
	出現位置リスト (sec)		
	AC	CSA	SA
高頻度 100 語	1.31800	445.866	0.00161
中頻度 100 語	0.81000	31.027	0.00156
低頻度 100 語	0.36300	0.0394	0.00051
	文書番号リスト (sec)		
	AC	CSA	SA
高頻度 100 語	1.59500	517.113	22.880
中頻度 100 語	0.91300	36.364	1.460
低頻度 100 語	0.44900	0.0429	0.002

4. 拡張

4.1 ほかの質問への参照

以下の XPattern 形式の質問群が入力されたとして．

$$\begin{cases} P_1 &= \pi_1[\pi_2^1 : e^1] \\ &\vdots \\ P_\ell &= \pi_1[\pi_2^\ell : e^\ell] \end{cases}$$

表現 e^i は, キーワード w_1, \dots, w_m に関するブール表現である．ここで, e^i を w_1, \dots, w_m と P_1, \dots, P_{i-1} を変数とするブール表現に拡張することを考えよう．表現 e^i の真偽値は, 2.5 節で述べたアルゴリズムにおいて現在の $depth$ の値に対する $Occ[depth][1..m]$ 及び $Q[depth][1..i-1]$ の真偽値によって定まる真偽値割り当ての下で決定される．このように拡張することで, 以下に示すように, 論理演算子や述語の入れ子の導入が可能となる．

a) 論理演算子

例として, $\pi[\pi_1 \text{ and } \pi_2]$ という XPath 表現を考えよう． $P_1 = \pi[\pi_1 : \text{true}]$, $P_2 = \pi[\pi_2 : \text{true}]$ とおき, $P_3 = \pi[\varepsilon : P_1 \wedge P_2]$ とする．このとき, XPath 表現 $\pi[\pi_1 \text{ and } \pi_2]$ が生起するのは, $Q[depth][3]$ の値が真になるときであり, かつ, そのときに限る．

同様の手法により, XPath 表現 $\pi[\pi_1 \text{ or } \pi_2]$ や, $\pi[\pi_1 \text{ and } \pi_2 \text{ or } \pi_3]$ などが可能となる．

b) 述語の入れ子

XPath 表現 $\pi_1[\pi_2[\pi_3 \text{ and } \pi_4]]$ を考える． $P_1 = \pi_1\pi_2[\pi_3 : \text{true}]$, $P_2 = \pi_1\pi_2[\pi_4 : \text{true}]$, $P_3 = \pi_1[\pi_2 : (P_1 \wedge P_2)]$ とおく．すると, XPath 表現 $\pi_1[\pi_2[\pi_3 \text{ and } \pi_4]]$ が生起するのは $Q[depth][3]$ の値が真になるときであり, かつ, そのときに限る．

$P_4 = \pi_1[\pi_2\pi_3 : \text{true}]$, $P_5 = \pi_1[\pi_2\pi_4 : \text{true}]$, $P_6 = \pi_1[\varepsilon : (P_4 \wedge P_5)]$ とおくとき, XPath $\pi_1[\pi_2[\pi_3 \text{ and } \pi_4]]$ が生起しなくても, $Q[depth][6]$ の値が真になりうることに注意しよう．

4.2 ブール関数から算術関数へ

質問式 P_i は XML 木の節点へ真偽値を割り当てる写像とみなせる．そこで, これを整数値を割り当てる写像に拡張する．第 1 に, ブール表現を e^i を整数上の算術表現へ拡張する．真偽値 $\text{true}, \text{false}$ は整数 1, 0 として表し, 論理演算子 \wedge, \vee , and \neg も整数上の算術的関数とみなす．第 2 に, 質問 $P_i = \pi_1[\pi_2 : e^i]$ が節点 x で生起したとき, 写像 P_i は, (x, y) が P_i の生起位置となるようなすべての子孫 y において算術表現 e^i の値を評価し得られた値の合計を節点 x に割り当てる． P_i が生起しないような節点 x に対しては, 0 を割り当てる．

以上の拡張により, XPath 表現 $\pi_1[\text{count}(\pi_2) > 1]$ のように, 集計関数 count の実装も可能となる．同様に, sum , max , min , and avg (average) などの集計関数も容易に実装できる．

5. まとめと今後の課題

本論文では, パストライ上のパス照合処理と AC 法を用いたキーワード照合処理を結合したストリーム型のフィルタリング技法を提案した．XPath の複数の質問式を同時に処理する手法について述べ, 3. 節で示したとおり, 一連の実験によって, パストライがフィルタ処理の高速化に寄与することを示し, 本手法が YFilter や XMLTK と比較して高い性能を有することが判明した．

また, 本手法が効率的に大量の質問式を同時に処理すること

ができることを実験により実証した。これにより、節 1. で述べたように、SDI サービスにおいて大量のクライアントが質問式をサーバに登録するような大規模 SDI システムの構築も可能である。また、質問式が少ない際は、SIX によりさらにフィルタリング性能を向上させることができる。

また、配信するべきクライアント数が急増し、単一計算機で処理をまかなえなくなるような場合でも、同じ XML データを複数の計算機に重複して担当させることで、対応可能である。この場合、バイナリ XML データは配信元となるマスターサーバでただ一度作成するだけでよく、前処理の時間は問題にならない。

今後の課題としては、(1) 単一計算機環境におけるフィルタ処理をさらに高速化する手法を検討することと、(2) 本論文の成果をもとに、前処理を施さない XML データをそのまま扱うタイプのフィルタリング手法を開発すること、(3) 分岐パターン (twig pattern) や親軸・先祖軸などへの対応が挙げられる。

(1) については、著者らの研究グループが開発した圧縮パターン照合技術 [12], [18] をテキスト照合処理部分に組込むことによって、圧縮率にほぼ比例した割合で、キーワード照合時間の削減を達成できると考えている。

(2) については、XML データに対する前処理を行わず、必要に応じて動的にパストライを更新しながら質問処理を行うことを考えている。本論文の手法に比べると処理速度は低下するが、石野ら [22], [23] と同様に DFA の状態爆発を抑えることができるため、XMLTK と比較すると省メモリのフィルタリングが可能となろう。

(3) の分岐パターンについては、節 4. の考え方と同様に、質問式を複数の *XPattern* の結合として表現することと、遅延評価を組み込むことにより対応可能である。

文 献

- [1] : Filtering and Transformation for High-Volume XML Message Brokering, <http://yfilter.cs.berkeley.edu/code.release.htm>.
- [2] : Report From the W3C Workshop on Binary Interchange of XML Information Item Sets, <http://www.w3.org/2003/08/binary-interchange-workshop/Report.html> (2003).
- [3] Altinel, M. and Franklin, M.: Efficient filtering of XML documents for selective dissemination, *VLDB'00*, pp. 53–64 (2000).
- [4] Avila-Campillo, I., Green, T. J., Gupta, A., Onizuka, M., Raven, D. and Suciu, D.: XMLTK: An XML Toolkit for Scalable XML Processing, *PLANX'02* (2002).
- [5] Barton, C., Charles, P., Goyal, D., Raghavachari, M., Josifovski, V. and Fontoura, M.: Streaming XPath processing with forward and backward axes, *ICDE*, pp. 455–466 (2003).
- [6] Carzaniga, A., Rosenblum, D. and Wolf, A.: Design and Evaluation of a Wide-Area Event Notification Service, Vol. 19, No. 3, pp. 332–383 (2000).
- [7] Chen, Y., Mihaila, G. A., Davidson, S. B. and Padmanabhan, S.: EXPedite: A System for Encoded XML Processing, *CIKM'04*, pp. 108–117 (2004).
- [8] Diao, Y., Altinel, H., Franklin, M. J., Zhang, H. and Fischer, P. M.: Path Sharing and Predicate Evaluation for HighPerformance, *ACMTOD* (2003).
- [9] Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, *VLDB'97*, pp. 436–445 (1997).
- [10] Grossi, R. and Vitter, J. S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching, *STOC'00*, pp. 397–406 (2000).
- [11] Ley, M.: DBLP Computer Science Bibliography, <http://dblp.uni-trier.de/>.
- [12] M. Takeda, et al.: Speeding up string pattern matching by text compression: The dawn of a new era, *Trans. Information Processing Society of Japan*, Vol. 42, No. 3, pp. 370–384 (2001). Special issue for IPSJ 40th anniversary award papers.
- [13] Manber, U. and Myers, G.: Suffix arrays: A new method for on-line string searches, *SIAM J. Computing*, Vol. 22, No. 5, pp. 935–948 (1993).
- [14] Navarro, G. and Raffinot, M.: *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press (2002).
- [15] Peng, F. and Chawathe, S. S.: XPath queries on streaming data, *SIGMOD'03*, pp. 431–442 (2003).
- [16] Sadakane, K.: Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array, *Proc. of 11th International Symposium on Algorithms and Computation (ISAAC'00)*, LNCS 1969, pp. 410–421 (2000).
- [17] Schmidt, A., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I. and Busse, R.: XMark: A benchmark for XML data management, *VLDB'02*, pp. 974–985 (2002).
- [18] Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T. and Arikawa, S.: Speeding up pattern Matching by Text Compression, *CIAC'00*, LNCS 1767, pp. 306–315 (2000).
- [19] Takeda, M., Ishino, A. and Mitarai, S.: A Light-Weight XML Query Processor for a Large Number of Structural and Textual Patterns, Technical Report DOI-TR-CS-226, Department of Informatics, Kyushu University (2006).
- [20] W3C: XQuery 1.0 and XPath 2.0 Full-Text Use Cases, <http://www.w3.org/TR/xmlquery-full-text-use-cases>.
- [21] 竹田正幸, 石野 明, 辻 寿嗣, 宮本 哲: ストリーム指向の高速 XML データ処理技法について, *DBWeb2003* (2003).
- [22] 石野 明, 竹田正幸: パスブルーニングによる決定性有限オートマトンを用いたストリーム指向の XQuery 処理, *DBWeb2005* (2005).
- [23] 石野 明, 竹田正幸: パスブルーニングによる決定性有限オートマトンを用いた XQuery 処理の提案, 日本データベース学会 Letters, Vol. 4, No. 4 (2006).