社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2007－DBS－143（12）
2007／7／2

漸増的なパストライ構築に基づく高速・軽量XML文書フィルタリング

萩尾 一仁† 御手洗秀一†† 石野 明††† 竹田 正幸†

† 九州大学大学院 システム情報科学府 情報理学専攻
†† 九州大学 情報基盤研究開発センター
††† 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: †{kazuhito.hagio,takeda}@i.kyushu-u.ac.jp, ††mitarai@cc.kyushu-u.ac.jp, †††ishino@ecei.tohoku.ac.jp

あらまし XPath式によるストリーム型XML文書フィルタDXAXENを開発したので報告する．DXAXENは，パストライの動的構築に基づき，高速・省メモリかつ質問式の増大に頑健な性能を実現している．代表的なストリーム型XML文書フィルタであるXMLTKと比較したところ，実行速度で2〜5倍高速であり，メモリ使用量も5〜20%と圧倒的に少ないことを示した．

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, XPath

Fast and Lightweight Filtering of Streaming XML Documents Using Incrementally Constructed Path-trie

Kazuhito HAGIO†, Shuichi MITARAI††, Akira ISHINO†††, and Masayuki TAKEDA†

† Department of Informatics, Kyushu University
†† Research Institute for Information Technology, Kyushu University
††† Department of System Information Sciences, Tohoku University
E-mail: †{kazuhito.hagio,takeda}@i.kyushu-u.ac.jp, ††mitarai@cc.kyushu-u.ac.jp, †††ishino@ecei.tohoku.ac.jp

Abstract In this paper, we present a streaming XML document filter named DXAXEN which is based on incremental construction of path-trie. It runs very fast, and processes a large number of XPath queries efficiently. Experimental comparison with XMLTK, a well-known streaming XML document filter, shows that DXAXEN is 2–5 times faster and needs only 5–20 percent of memory.

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

1. まえがき

近年，XMLの利用範囲が拡大し，センサデータや株式市場データなど，リアルタイム性の高い情報の伝達にXMLデータストリームが用いられる傾向にある．また，携帯電話や情報家電など，計算機としてのリソースが少ない環境におけるXMLデータ処理の需要も高まりつつある．

XMLデータ処理の中でも，大量のデータから必要な部分を抽出するフィルタリングは，XMLデータ処理の基礎であり，非常に重要である．フィルタリングシステムの入力には，XPath [5]が用いられ，XPathを高速に処理するための手法の研究が数多くなされている [2], [4], [6], [13].

1.1 先行研究

代表的なXMLデータストリーム処理器にYFilter [6]とXMLTK [4]がある．YFilterは，XFilter [2]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る手法を用いている．これにより，XFilterと比べ，質問数に関する高い規模耐性を得ている．しかし，NFAは同時に複数の状態遷移を処理しなければならず，処理速度は遅い．一方で，NFAを等価な決定性有限オートマトン(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データ走査時に必要に応じてNFAを部分的にDFAに変換するlazy DFA [8]と呼ばれる手法を採用している．これにより，XMLTKは質問数に関して規模耐性が高く高速なものとなっているが，DFAの状態数は依然大きく，大量のメモリを必要とするという欠点がある．

著者らは[18]において，パストライとパスプルーニングを用いてDFAの状態数の増大を抑え，効率的にパス照合処理を行なう手法を提案した．パストライは，XMLデータ中の根から要素節点へ向かうパスに沿ったタグ名列の全体を表わすトライ構造である．この手法は，まず入力XMLデータを一度走査して

パストライを得ておき，これを用いて質問式に含まれる//や*を展開し，状態数の少ない DFA を直接構築するもので，フィルタ処理はこの DFA を用いた二度目の走査によって行なうことになる．パストライをいったん作成しておけば，DFA の状態数はパストライの節点数を超えないため，非常に高速なフィルタ処理が実現できる．

次に著者らは [11], [17] において，XML データを事前にパストライとバイナリ XML データに変換しておき，それを配信するという設定のもとで高速なフィルタ処理手法を実現し，XAXEN (eXtreamly-Accelerated XML filtering ENgine) と名付けた．変換の際，バイナリ XML データには，開始タグ相当部分に対応するパストライの節点へのポインタを埋め込んでおく．また，パス照合処理は，バイナリ XML データ走査時に実行するのではなく，パストライを対象にして質問式を得た段階で完了させておく．こうすることにより，バイナリ XML データ走査時に，埋め込まれたパストライの節点へのポインタを用いて，パス照合処理の結果を参照できる．なお，パストライに対するパス照合処理にはビット並列化手法 [12] を用いた NFA を採用している．

しかし XAXEN では，パストライとバイナリ XML データを作成する前処理が必要であり，このため，XML データをそのまま配信する設定でのフィルタ手法としては適さない．そこで，本論文では，漸増的にパストライを構築しながらパス照合処理を行なうことで，XAXEN の処理速度をそれほど低下させることなく，前処理を排することに成功した．本論文では，この手法を DXAXEN (dynamic XAXEN) と呼ぶことにする．

1.2 XPath 部分族

XAXEN および DXAXEN は，XPath 式 [5] の大きな部分族を処理できる．表 1 に示したのは，その例である．5. 節でふれるように，複雑な XPath 式は，基本となる単純な XPath 式の結合として表現できる．そこで，本論文では，基本的な XPath 式として単純パスパターン，パスパターン，XPattern に焦点を絞って，その効率的な処理手法について論じる．ここで，単純パスパターンとは，タグ名およびタグ名に関するドントケア記号 “*” からなる記号列を，親子関係を表わす “/” や先祖-子孫関係を表わす “//” で区切ったものをいう．また，パスパターンとは，二つの単純パスパターン π_1，π_2 を組み合わせた $\pi_1[\pi_2]$ という形式の XPath 式をいい，XPattern とは，これにキーワードの生起に関する論理式を付加したものである．

1.3 本論文の構成

本論文の構成は以下のとおりである．2. 節では，パストライの性質とそれを用いたパス照合手法について述べる．3. 節では，本論文で提案する，漸増的なパストライ構築に基づくフィルタリング手法について述べる．次に 4. 節で，実験により XMLTK との比較評価を行なう．5. 節では，複雑な質問式に対する処理手法を述べる．最後に 6. 節でまとめを述べる．

表 1 サポートする XPath 式の例

/*/order//sender	単純パスパターン
//receiver[name]	パスパターン
/order//[contains(name, “mickey”)]	XPattern
//sender[count(//region) > 2]	集約
/order//address[street or region]	論理演算
//address[//region[//country and zipcode]]	入れ子
/order[//address]//zipcode	パス途中の述語

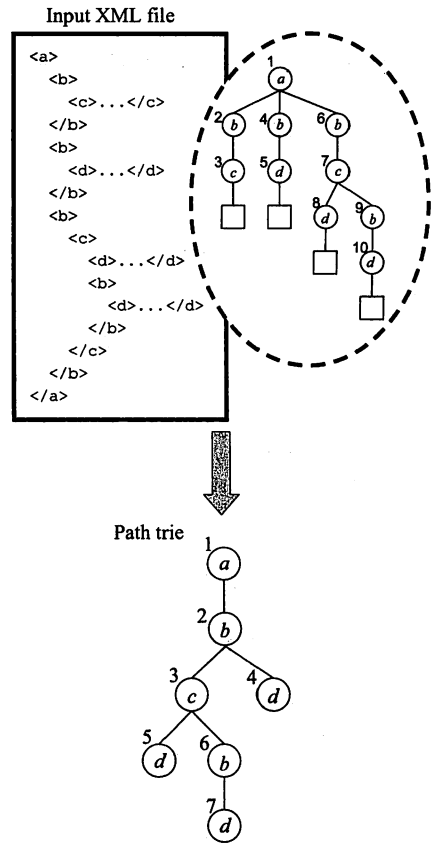


図 1 XML データと生成されるパストライ

2. パストライを用いたフィルタ処理

2.1 パストライ

XML 木の根から要素節点へ向かうパスに沿ったタグ名の系列を**パス文字列**とよぶことにする．**パストライ**とは，ある XML 木におけるパス文字列全体の集合をトライ (trie) として表現したものをいう．パストライは，DataGuide [7] と呼ばれるデータ構造の一種である．DataGuide はグラフ構造によってデータベースにおけるデータの構造を示したものであるが，XML データは木構造をしているため，XML データに対する DataGuide すなわちパストライは木構造となる．

XML データと生成されるパストライとの関係を図 1 に示した．図 1 右上は XML 木を示しており，正方形はテキスト節点を表わしている．下に示したのがパストライである．パストライの節点横の数はその節点の ID である．

2.2 パストライのサイズ

実際の多くの XML データが小さな DataGuide をもつこと

表 2 パストライのサイズ

	データサイズ	タグ数	タグ名数	パストライサイズ
(a)	2.2GB	198,887,920	92	160
(b)	377MB	22,215,944	41	166
(c)	116MB	4,096,360	83	549

が示されており [9]，パストライも同様となる．一般に，対象となっている XML データが規則正しい構造をもつ，すなわち構造化されているほど，パストライは元のデータサイズに比べて小さなものとなる．(a) UniProtKB/Swiss-Prot [3] の XML データ，(b) DBLP [10] の XML データ，(c) XMark ベンチマーク [15] の xmlgen を用いて生成したランダムデータ，の三つについてパストライのサイズ等を表 2 に示す．

また，これらの XML データを先頭から処理する際にパストライのサイズが増加する様子を，図 2 に示した．データに含まれるタグ数及びタグ名数の増加の様子も併せて示している．(a) の UniProtKB/Swiss-Prot の場合，データは規則正しい構造をもっており，パス文字列のバリエーションは XML データの先頭部分ですぐに出尽くしてしまい，パストライは飽和していることが分かる．また，(b) の DBLP の場合，途中 240MB 付近で傾向が変わっているが，それ以外の部分は比較的安定している．(c) のランダムデータは，根の直下で 6 個のカテゴリに分かれ，それぞれが独立して異なるスキーマをもつために，パストライのサイズもそれに沿って変化している．

2.3 パストライを用いたパス照合

図 1 の 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質問式 Q_1 = //b/c と Q_2 = a/b//d の生起の様子を図 3 に示す．この例では，質問式 Q_1 はパストライ中の節点 3，質問式 Q_2 は節点 4, 5, 7 に生起することが分かる．さて，パストライにおける Q_1 の出現位置である節点 3 に対応する XML 木の節点は二つあるが，そのいずれとも Q_1 の出現位置である．Q_2 の出現位置についても，同様のことがいえる．2.2 節で述べたように，パストライのサイズは XML 木のサイズに比べて非常に小さい．したがって，XML 木に対してでなく，パストライに対してパス照合を行ない，それを参照することで XML 木における出現位置を得るようにすれば，パス照合処理に要する時間を大幅に削減することが可能となる．

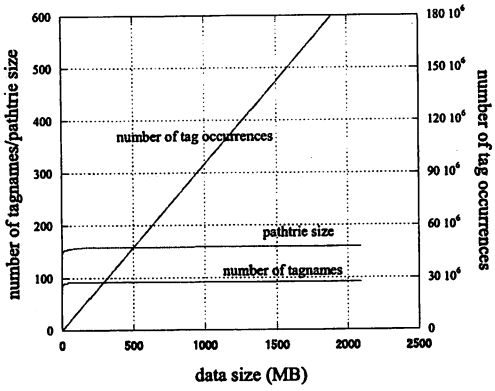
3. 漸増的パストライ構築に基づくフィルタ処理

本節では，XML データを走査しながらパストライ構築とパス照合を同時に行なうことによって，XML ストリームに対して高速にフィルタ処理する手法を述べる．

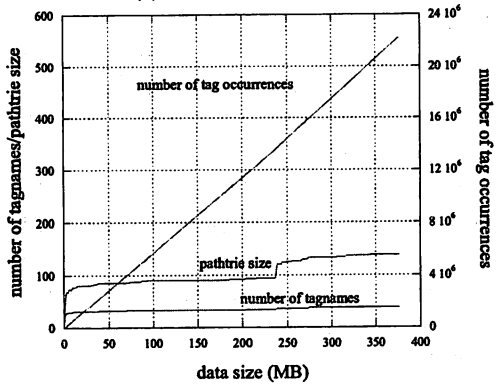
3.1 漸増的パストライ構築とパス照合

与えられた XML データはまず XML パーサによって逐次的に構文解析される．XML パーサはタグの出現を確認すると，そのタグの情報をイベントとして発行する．このイベントを用いて，パストライの構築とパス照合を行なう．

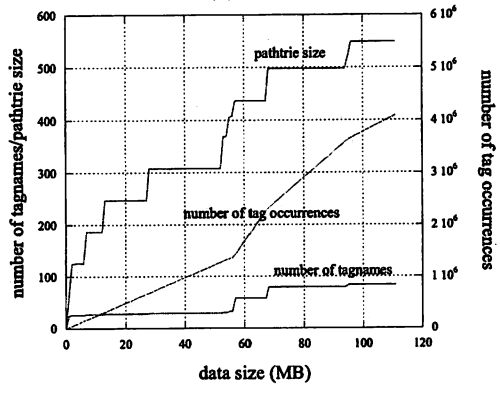
再び図 1 で示した XML データを例にとり，本手法のアルゴリズムを述べる．いま，XML 木中の節点 1 から 7 までの処理が完了しているものとする．このとき，パストライは 1 から



(a) UniProtKB/Swiss-Prot



(b) DBLP



(c) ランダムデータ

図 2 パストライサイズの増加の様子

4 までの節点をもつ．また，パストライを DFA として見立て XML 木の根からのパス上を走らせているため，パストライの節点 3 が現在の位置となる．XML パーサによって XML 木中の節点 8 が読まれると，パストライの節点 3 からそのタグ名 d で「状態遷移」する．今回のように，もしそのタグ名 d で遷移できなければ，遷移先となる新たな節点 5 を現在の節点 3 の子として作成する．その場合，パストライに根から節点 5 に至る新たなパスが発生したことになるため，すべての質問式についてこのパスに対する必要な照合を行なう．以上を XML 木のすべての節点を読み終えるまで繰り返すことで，漸増的なパスト

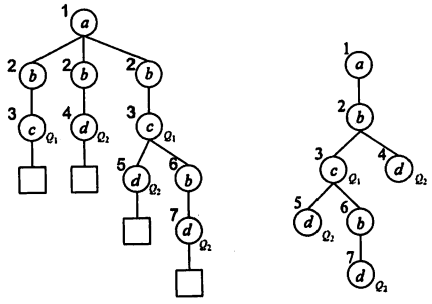


図 3 XML 木及びパストライにおける Q_1 = //b/c と Q_2 = a/b//d の生起.

ライの構築とパスの照合を同時に行なう.

2.2 節で述べたように，実際な XML データでは，パストライへの節点の追加が早い段階でほとんど行なわれなくなるため，このようにすることで高速なパスパターン照合が期待される.

3.2 キーワード照合

本手法では，SAX [14] などの既存の XML パーサを用いず，XML パース処理とキーワード照合を同時に行なう．具体的には，Aho-Corasick のパターン照合機械 [1] を XML データのテキスト要素に対して適用することで複数キーワードの照合を行ない，タグの開始を表わす "<" を検出する度に独自の XML パーサを呼び出し，終了を表わす ">" に出合うまでの間，タグ名の切り出しや属性の処理等を行なう．これにより，高速なストリーム処理が可能となる.

3.3 NFA によるパス照合

パストライに新たな節点が追加された際に行なう，パス照合の詳細について述べる．ここでは 3 種類の手法を提案する.

DXAXEN: 1 番目の手法として，質問式ごとにビット並列化手法 [12] を用いて NFA を作成し，これを動作させる方法が考えられる．この方法は，質問の個数だけ NFA を走査させなければならない反面，NFA の状態遷移をビット並列化により高速に行なえるという利点をもつ．パストライの各節点 n には，根から n まで NFA を走らせた際の active な状態の集合を表わすビットベクトルをもたせておく．このビットベクトルのすべてのビットが 0 になると，それ以上照合を続けても受理することはありえないため，ビットベクトルを保持する必要はない．すなわち，パストライの各節点は，受理の可能性のある NFA の番号とそのビットベクトルの組をもつことになる.

DXAXEN-Y: 2 番目の手法として，YFilter [6] のように，複数の質問のパターンを表わすトライ構造を作成し，これを NFA に変換して用いる方法が考えられる．すなわち，パターン中の "*" や "//" をタグ名と同様に 1 個の記号とみなした記号列に対するトライを作成しておき，NFA に変換する際には，"*" でラベル付けされた辺を任意のタグ名による遷移に置き換え，"//" でラベル付けされた辺については，これを ε-遷移で置き換えるとともに，行き先である節点に任意のタグ名によるループを付加する．パストライの各節点 n には，根から n までこの NFA を走らせた際の active な状態の集合をもたせておく．こうすることにより，パストライに新たな節点が追加された場合，その部分だけ NFA を動作させればよい.

DXAXEN-S: 以上二つの手法は，パストライの節点 m の子として節点 n が追加された際に，根から m まで NFA を走らせた際の active な状態の集合を保持しておくことにより，m から n へ向かう部分についてだけ照合を行なうことで計算時間を減らすものである．反面，active な状態を保持するための領域が必要である．そこで，非効率は覚悟の上で，active な状態の集合を保持せず，根から新しい節点 n へのパスに対して，質問式の数だけ毎回ビット並列化による NFA を動作させる手法も考えられる.

次節では，これら三つのパス照合手法の比較を行う.

表 3 実行時間の比較

(a) ランダムデータ

質問数	経過時間 (sec)			
	DXAXEN	DXAXEN-Y	DXAXEN-S	XMLTK
1	1.49	1.46	1.42	2.22
10	1.49	1.50	1.48	2.58
100	1.58	1.63	1.56	3.04
1000	1.79	1.77	1.73	4.12
10,000	2.85	2.87	3.14	11.56
100,000	39.79	49.13	45.09	283.34

(b) DBLP

質問数	経過時間 (sec)			
	DXAXEN	DXAXEN-Y	DXAXEN-S	XMLTK
1	5.82	5.81	5.80	11.34
10	6.05	6.15	6.00	11.29
100	6.72	6.89	6.61	18.06
1000	7.30	7.38	7.36	21.26
10,000	14.98	15.58	14.73	59.95
100,000	195.37	175.71	207.41	1242.58

4. 実装実験

XML データとして，2.2 節で述べた 377MB の DBLP のデータと 116MB の XMark ベンチマークのランダムデータを用いた．質問式には，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ることとし，YFilter の pathgenerator [16] によってランダムに生成した．ここで，//，*の生起確率はそれぞれ (1%, 1%) とした．実験に用いた計算機環境は，2.4 GHz Intel Pentium 4, 2.0 GB RAM, Red Hat Linux Advanced Server 2.1 である.

以下では，フィルタリング手法としての有効性を検証するために，実際の実行時間，メモリ使用量，スループットの三つの観点から XMLTK との比較実験を行なった結果を述べる．なお，XFilter, YFilter や XSQ は，いずれも，速度，使用メモリ量，質問数に関する規模耐性の点で XMLTK を下回るため [13]，ここでは，XMLTK のみを比較の対象とした.

4.1 実行時間の比較

表 3 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，すべ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．

DXAXEN と XMLTK を比較すると，おおよそ 2〜6 倍 DXAXEN が高速である．質問数が増えるほど，その差は顕著に現れる．DXAXEN と DXAXEN-S では，わずかに DXAXEN が高速な程度にとどまっており，このことから，ビット並列化による NFA の動作が非常に高速であることが分かる．

4.2 メモリ使用量の比較

表 4 は，ランダムデータに対する質問数が 10,000 と 100,000 のときのメモリ使用量を表わしている．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かではなく，プログラム実行中に実際に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．

DXAXEN を XMLTK と比較すると，質問数 10,000 のとき 1/5 程度，質問数 100,000 では 1/20 程度しかメモリを使用しないことが分かる．また，DXAXEN-S は DXAXEN と比較して 1〜2 割ほど領域を節約できた．DXAXEN-Y は DXAXEN と比較して 2 倍程度のメモリ使用量となった．

4.3 スループットの比較

図 4 は質問数が 100 と 10,000 のときのスループットを比較したグラフである．x 軸は XML データの先頭からのデータサイズを表わす．

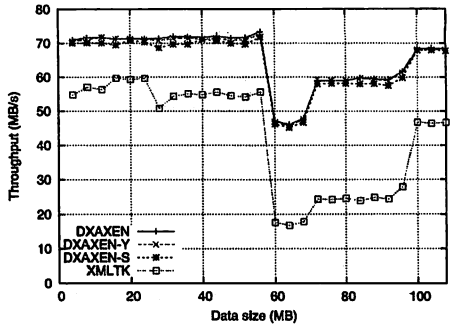
(a), (b) のランダムデータでは，どの手法もスループットが安定しない．データ前半の DXAXEN-S と XMLTK の局所的なスループットの低下は，図 2 (c) に示したパストライの節点数の増加に対応しており，節点の追加によるパス照合処理が増大していると考えられる．これより，DXAXEN と DXAXEN-Y はパストライの節点数の増加に対して比較的堅牢であることが分かる．後半部分の全体的なスループットの低下は，図 2 (c) に示したデータ中のタグの出現頻度に対応しており，テキスト部分の処理に対してタグ部分の処理に時間がかかるためと考えられる．しかしながら，DXAXEN は常に XMLTK より高いスループットであることが分かる．

(c), (d) の DBLP では，データが比較的規則正しいスキーマをもっており，XMLTK と DXAXEN の両方に有利な条件であるが，DXAXEN が圧倒的に高いスループットを示している．

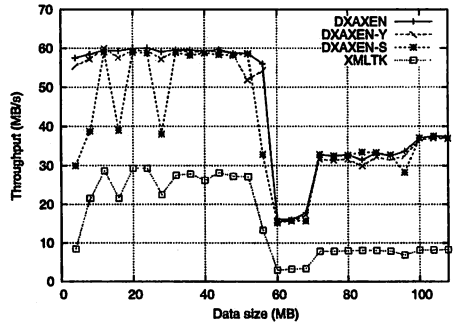
5. 複雑な質問式の処理

ここまでは，単純パスパターンとよぶ，非常に制限された XPath 式の処理について述べてきた．この節では，より複雑な XPath 式を扱うための拡張について述べる．

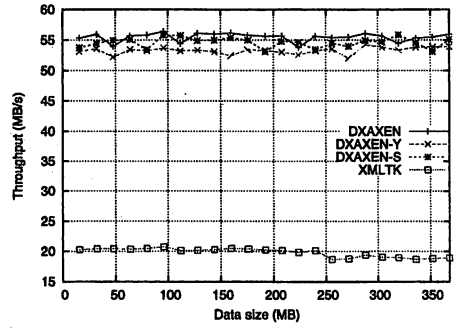
パスパターンとは単純パスパターン π_1，π_2 の順序対であり，$\pi_1[\pi_2]$ と表わす．XML 木の任意の節点 x とその子孫 y ($x = y$ の場合も含む) に対し，パスパターン $\pi_1[\pi_2]$ が**位置** (x, y) **に出現する**とは，π_1，π_2 が，それぞれ，根から x へのパスと x' から y へのパスにマッチするときをいう．ここで，x' は，根から y へ向かうパスに沿った x の子節点である．($\pi_2 = \varepsilon$ とす



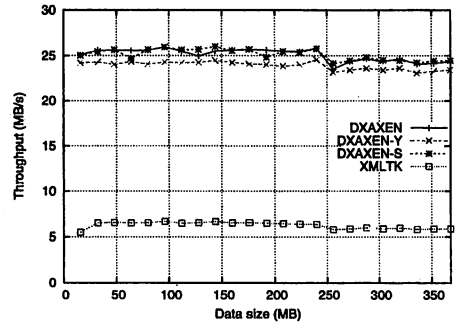
(a) ランダムデータ, 質問数 100



(b) ランダムデータ, 質問数 10,000



(c) DBLP, 質問数 100



(d) DBLP, 質問数 10,000

図 4 スループットの比較

表 4 メモリ使用量の比較

質問数	メモリ使用量 (KB)			
	DXAXEN	DXAXEN-Y	DXAXEN-S	XMLTK
10,000	7,268	14,400	6,670	38,780
100,000	17,452	35,216	14,372	369,036

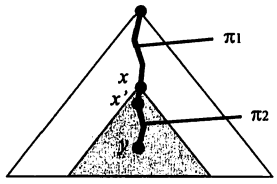


図 5 XML 木の位置 (x, y) におけるパスパターン $\pi_1[\pi_2]$ の生起

ると単純パスパターンになるが，この場合は，x' が存在せず，$x = y$ となる．) 図 5 に，パスパターン $\pi_1[\pi_2]$ とその生起位置 (x, y) との関係を図示した．

XML 木におけるパスパターン $Q_i = \pi_1[\pi_2]$ の生起位置は，以下のようにして求めることができる．パストライを対象に，単純パスパターン $\pi_1\pi_2$ の生起位置となる節点 y を探す．$\pi_2 \neq \varepsilon$ の場合には，得られた節点 y に対し，根から y へ向かうパス上において上述の条件を満たすような節点 x を求める．節点 x と y の深さの差 d を求め，節点 y に対 $\langle Q_i, d\rangle$ をもたせる．これにより，XML データ走査時において，上記の y に対応する XML 木の節点 $\hat{y}$ に出合った際に，パストライの節点 y のもつ情報 $\langle Q_i, d\rangle$ から，節点 $\hat{x}$ を $\hat{y}$ の d 世代上の先祖として得ることができる．こうして，XML 木における Q_i の生起位置 $(\hat{x}, \hat{y})$ をすべて求めることができる．

さらに，テキスト節点におけるキーワード照合を扱うため，以下の拡張を行なう．$e = e(w_1, \ldots, w_m)$ を，キーワード $w_1, \ldots, w_m$ の生起の有無に関する論理式とする．*XPattern* とは，パスパターン $\pi_1[\pi_2]$ と論理式 e の組をいい，$\pi_1[\pi_2 : e]$ と表わす．XPattern $\pi_1[\pi_2 : e]$ 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によって論理式 e が真になるときをいう．この定義より，パスパターン $\pi_1[\pi_2]$ は，XPattern $\pi_1[\pi_2 : \mathbf{true}]$ と見なすことができる．

XPattern は，単独では非常に制限された XPath 式であるが，複数の XPattern を組み合わせることで，複雑な XPath 式を扱うことができる．詳細については論じないが，以下でその具体例をいくつかあたえる．

論理演算: Q = /order//address[street or region] という質問式を考える．Q_1 = /order//address[street: true], Q_2 = /order//address[region: true] とおくと，質問式 Q の生起位置は Q_1, Q_2 の両方の生起位置として求められる．したがって，/order//address$[\varepsilon : Q_1 \vee Q_2]$ のように，XPattern の組み合わせとして表現できる．

述語の入れ子: Q = //address[//region[//country and zipcode]] という質問式を考える．Q_1 = //address//region[//country: ture], Q_2 = //address//region[zipcode: true] とおくと，Q は，//address$[$//region : $Q_1 \wedge Q_2]$ と表わせる．

パス途中の述語: 質問式 Q = /order[//address]//zipcode を考える．Q_1 = /order[//address: true] とおくと，式 Q_1//zipcode$[\varepsilon : true]$ の処理を遅延評価によって実現できる．

以上のようにすれば，//b[//d]//c[//c][d]//e のような XPath 式も扱うことができる．さらに，親軸・先祖軸などの後方軸 (Backward axes) を含む XPath 式についても，パストライを用いて前方軸 (Forward axes) のみを含む XPath 式に変換し，処理を行なうことが可能であるが，これについては稿を改めて述べることにする．

6. ま と め

本論文では，漸増的なパストライ構築に基づくストリーム型の XML フィルタリング手法を提案した．XAXEN で必要であった前処理を排しつつも，なお XMLTK と比較して高い性能をもつことを実験により示した．

文 献

[1] Aho, A. V. and Corasick, M.: Efficient string matching: An Aid to Bibliographic Search, *Comm. ACM*, Vol. 18, No. 6, pp. 333–340 (1975).

[2] Altinel, M. and Franklin, M.: Efficient Filtering of XML Documents for Selective Dissemination of Information, *VLDB'00*, pp. 53–64 (2000).

[3] Apweiler, R., Bairoch, A., Wu, C. H., Barker, W. C., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., Martin, M. J., Natale, D. A., O'Donovan, C., Redaschi, N. and Yeh, L.-S. L.: UniProt: the Universal Protein knowledgebase, Vol. 32, pp. D115–D119 (2004).

[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] Clark, J. and DeRose, S.: *XML Path Language (XPath) Version 1.0* (W3C Recommendation 16 Nov. 1999). `http://www.w3.org/TR/xpath`.

[6] Diao, Y., Altinel, M., Franklin, M., Zhang, H. and Fischer, P.: Path sharing and predicate evaluation for high-performance XML filtering, *ACM TODS*, Vol. 28, No. 4, pp. 467–516 (2003).

[7] Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, *VLDB'97*, pp. 436–445 (1997).

[8] Green, T. J., Gupta, A., Miklau, G., Onizuka, M. and Suciu, D.: Processing XML streams with deterministic automata and stream indexes, *ACM TODS*, Vol. 29, No. 4, pp. 752–788 (2004).

[9] H., L. and D, S.: XMill: an efficent compressor for XML data, *SIGMOD'00*, pp. 153–164 (2000).

[10] Ley, M.: DBLP Computer Science Bibliography, `http://dblp.uni-trier.de/`.

[11] Mitarai, S., Ishino, A. and Takeda, M.: Light-weight acceleration for streaming XML document filtering, *SWOD2007* (2007).

[12] Navarro, G. and Raffinot, M.: *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press (2002).

[13] Peng, F. and Chawathe, S. S.: XSQ: a streaming XPath engine, *ACM TODS*, Vol. 30, No. 2, pp. 577–623 (2005).

[14] SAX 2.0: Simple API for XML, `http://www.megginson.com/SAX/`.

[15] 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).

[16] YFilter: Filtering and Transformation for High-Volume XML Message Brokering, `http://yfilter.cs.berkeley.edu/code_release.htm`.

[17] 御手洗秀一，石野 明，竹田正幸：パストライを用いた高速・軽量な XML 文書フィルタリング，*DEWS2007* (2007).

[18] 石野 明，竹田正幸：パスプ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，*DBSJ Letters*, Vol. 4, No. 4 (2006).