DEWS2005 6B-o4

XMLデータの要約情報を用いた高速なXPath処理方法

江田　毅晴† 鬼塚　真† 山室　雅司†

† NTT サイバースペース研究所
E-mail: †{eda.takeharu,onizuka.makoto,yamamuro.masashi}@lab.ntt.co.jp

あらまし XML が普及したため, 大量の XML データを扱う XML データベースのニーズが高まっている. XML データにおけるノード間の先祖子孫関係を判定可能な範囲ラベルを利用した XPath 処理手法は, 大規模 XML データ検索の基盤手法として広く研究されている. 本稿では, 長い XPath 問合せを構造ジョインを用いて処理する際に, 構造ジョインが多段になり処理速度の低下を招く問題を回避する方法を提案する. 提案手法では, XML データの要約情報であるストロング DataGuide を利用することにより, 構造ジョインの回数を削減し, かつ一次記憶に読み出すノード数を削減することが出来る. シングルパス問合せ (XP(/,//,*)) は構造ジョインを行うことなく, 小枝パターン (XP(/,//,*,[])) の場合でも, 問合せを木構造表現した際に分岐となるノードと葉ノード間の構造ジョインのみで XPath を処理することができる. 評価実験の結果, 111MB の XML データに対して既存の構造ジョインを用いた最適化手法と比較し, 2 から数百倍の高速化を達成した.

キーワード XML, XPath, XML データベース

Fast XPath Processing with XML Summaries

Takeharu EDA†, Makoto ONIZUKA†, and Masashi YAMAMURO†

† NTT Cyber Space Laboratories
E-mail: †{eda.takeharu,onizuka.makoto,yamamuro.masashi}@lab.ntt.co.jp

Abstract A wide spread of XML has invoked a lot of needs to handle a large volume of XML data. For the purpose of XPath processing, structural join methods have been studied well, where nodes are labeled so that any ancestor-descendant relationship between nodes can be determined by comparing their labels. In this paper, we solve a problem of structural join methods; longer path expressions require many joins and cause the performance degradation. We preprocess XPath queries against *strong* DataGuides of XML trees, which keeps the accurate paths in the XML trees. In our method, single paths (XP(/,//,*)) can be processed without joins and twig patterns (XP(/,//,*,[])) can be processed with joins among branching nodes and leaves in queries. The cost is reduced from the two factors, the number of joins, and the number of nodes read from disk. Experimental results show that our method outperforms the structural join method from two to three hundreds times faster when the database size is 111MB.

Key words XML, XPath, XML Databases

1. はじめに

現在, XML はデータ交換フォーマットおよびデータ保存フォーマットとして広く利用されている. 例えば, 電子物流分野では, データ交換フォーマットとして PML(Physical Markup Language) [12] と呼ばれる XML ボキャブラリが提案されている. PML は, センサーによって取得される多種多様な情報を表現するための拡張機能を持つため, 実際の PML データインスタンスは非常に多様になると予想される. また電子行政分野ではデータ交換のための語彙の標準化作業が開始されており, その体系は, 大規模な XML の名前空間の階層構造となっている. これらのことから, 多様かつ大量の XML データを管理できる XML データベースのニーズが高まっていることが分かる.

RDBMS に対して SQL という問合せ言語が提供されているように, XML データベースに対しては, XPath や XQuery といった問合せ言語が提案されている. XML データはラベル付き木でモデル化されるため, XML の問合せ言語は木の構造指定と値に対する処理が基本になる. XQuery のサブセットである XPath では, 正規表現で構造指定が出来るため, 非常に柔軟に問合せを記述することが出来る. しかしながら, XPath 問合せ処理を XML データの木構造探索として処理しようとすると XML データの全検索が必要になり, 高速に処理できないという問題があった.

この問題を解決するため, 範囲ラベル付け手法に基づいた構造ジョインアルゴリズムが提案されている [2]. 構造ジョインを用いた XPath 処理手法は, 大規模 XML データに対して, `descendant` 軸を処理するのに適した手法である. 範囲ラベルを各 XML ノードに付与すると, 範囲ラベルの比較のみで任意のノード間の先祖子孫関係が判定可能になる. この性質を利用して, XML の木構造を分解・格納し, 問合せに必要なノードのみを記憶装置から取り出し, 問合せが指定する構造にマッチする部分木を結合する (構造ジョイン). これによって XML データの全検索を回避することができる. この手法では特別なスキーマ等の情報は利用しないため, 大規模かつ多様な XML データに対する XPath 処理の基礎手法と位置付けられている.

一方, データベース利用者の問合せとは多様なものであり, XML データを管理する中で, より正確にデータの位置を記述した, 長い XPath 問合せを書くような状況も当然考えられる. しかしながら, 上記構造ジョインを用いた XPath 処理手法は, XPath 問合せに現れるタグの個数に比例した回数の構造ジョインを行う必要があるため, XPath 問合せが長くなると処理に時間がかかるという問題がある. 実際, 構造ジョインの一つである `Stack-tree` ジョイン [2] を実装して, 4. 節に示す実験環境にて XPath 処理時間を計測したところ, 1 回の構造ジョインを実行する問合せ "`//parlist//listitem`" の処理には 3492 ミリ秒ですむところ, 8 回構造ジョインを行う問合せ "`//site//categories//category//description //parlist //listitem//text//bold//emph`" の処理には, 12539 ミリ秒もかかっている.

そこで, 本研究では XML データの要約情報を用いて, XPath 問合せが長い場合にも, 高速に構造ジョイン手法を用いて XPath を処理する手法を提案する. 提案手法では, あらかじめ XML データの要約情報としてストロング DataGuide を構築しておく. XPath 問合せを木構造表現した際に, 分岐となるノード単位でシングルパス列に分解し, シングルパスは DataGuide を用いて処理し, シングルパス間の分岐処理は, 構造ジョインを用いて処理する. これにより, シングルパスの XPath 問合せに関しては構造ジョインを省くことができ, 分岐を含む場合にも, 分岐ノードと葉ノード間の構造ジョインのみで XPath 処理を行うことができる. 提案手法を実装し, 分岐を含まないシングルパスの XPath 処理に関して評価実験を行った結果, 既存手法である構造ジョインを用いた XPath 処理手法に比べ, 111MB の XML データの場合, 2 倍から数百倍の高速化を達成した. また, 1.1MB から 111MB までの XML データに対して, 提案手法がスケーラビリティを持つことも確認した.

本稿の構成を下記に示す. まず, 2. 節で基礎事項について説明する. 3. 節で提案手法について説明し, 4. 節で提案手法の実験結果について報告する. 5. 節で関連研究について述べ, 最後に 6. 節で本稿をまとめ今後の方向性について議論する.

2. 準　　備

本稿で使う基本的な語彙及び概念について説明する.

XML 木: XML データの ID, IDREF 属性は本稿では取り扱わない. つまり XML データは木であり, XML 木と呼ぶ(注1). XML 木を, $T = (V, root, E, \Sigma)$ で与える. 但し, V はノード集合, $root \in V$ は根ノード, E は枝集合, Σ はタグ (ノード名) の集合である. 属性は V に含まれていると考える. ノードは唯一に区別することができ, 以降 XML 木のノードを, xmlノードと呼ぶ(注2). E は, $E \subset V \times V$ であり, 一つの枝は, (v_{i1}, v_{i2}) (但し $(v_{i1}, v_{i2} \in V)$) の形で与えることができ, v_{i1} と v_{i2} が, 親子の関係にあることを示す. V 中の各 xmlノードは, $root$ を除いて E 中にそれぞれ一度だけ子 xmlノードとして登場するものとする. すなわち, どの xmlノードも枝を親へと辿ることにより, xmlノードから $root$ までのパスが一意に決定される(注3). このパスを**ノードパス**と呼び, ノードパスの集合を, P_n で表す. ノードパス上のノードのタグを連ねたものを**タグパス**と呼び, タグパスの集合を P_t で表す(注4). ノードパスは, xmlノードに対してユニークに決まる $root$ からの経路であるので, ノードパスと xmlノードは一対一の関係にあり, $|P_n(T)| = |V(T)|(= |E(T)| + 1)$ が成り立つ. ただし, $|A|$ は集合 A の濃度 (個数) を表す. 同一のタグパスを持つ xmlノードが複数存在する. 表 1 に図 1 の XML 木のノードパスとタグパスを示す. $|P_n(T)| \geqq |P_t(T)|$ が成り立つ. また, タグ集合の濃度も考えると, 結局, $|P_n(T)| \geqq P_t(T) \geqq |\Sigma(T)|$ が成り立つ. 表 2 に, いくつかの XML データの $|P_n(T)|$ と $|P_t(T)|$ および $|\Sigma(T)|$ の値を示す.

表 1　図 1 の XML 木のノードパスとタグパス.

ノードパス	タグパス
root	l_{root}
root.v_1	l_{root}."Book"
root.v_1.v_4	l_{root}."Book"."title"
root.v_1.v_5	l_{root}."Book"."author"
root.v_2	l_{root}."Book"
root.v_2.v_6	l_{root}."Book"."title"
root.v_2.v_7	l_{root}."Book"."comment"
root.v_3	l_{root}."Book"
root.v_3.v_8	l_{root}."Book"."isbn"
root.v_3.v_9	l_{root}."Book"."author"
$\|P_n(T)\| = 10$	$\|P_t(T)\| = 6$

表 2　XML データの各種情報.

	$\|P_n(T)\|$	$\|P_t(T)\|$	$\|\Sigma(T)\|$
dblp	11226036	159	35
XMark3 (100MB)	3024328	549	74
Gene Ontology	9996388	35	16
SwissProt	5432193	265	85

XPath: 本議論で扱う XPath 問合せの範囲は, XP(/,//,*,[]) [20] である. すなわち, タグを指定するノードテスト, 親子関係を指

(注1)：テキストノードも扱わないことにする. 提案する XPath 処理方法は容易にテキストノードに対する条件を含む場合に拡張できる.
(注2)：本議論では, ノードの順序は考慮しないが, 実際のシステムでは, 範囲ラベルを用いているためノードの順序も扱うことができる.
(注3)：$root$ のみ親を持たないため.
(注4)：以降, T の構成要素 N を $N(T)$ で表す. 但し, N は $root, E, \Sigma, TEXT, P_n, P_t$ のどれかである.

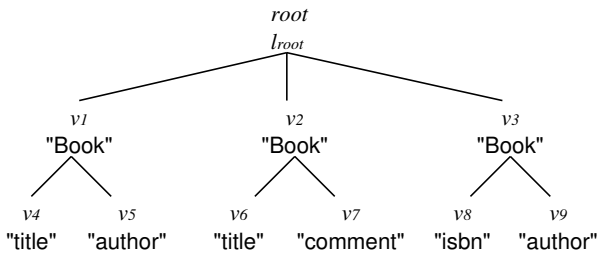

図 1　XML　木.

定する child 軸, 先祖子孫関係を指定する descendant 軸および, 述語を許可する.

XPath 問合せはラベル付けされた木とみなすことができ, 問合せ木と呼ぶことにする. (1)//Book//title および (2)//Book[./isbn]//author の問合せ木を図 2 に示す. 問合せ木中のノードは XPath におけるノードテストであり, 先

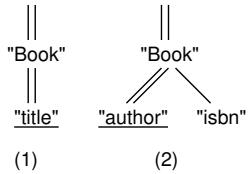

図 2　問合せ木.

祖子孫関係は 2 重線, 親子関係は 1 重線の枝でそれぞれ示される. 下線で示すノードがアウトプットノード (問合せ処理後に出力する xmlノード) である.

XP(/,//,*,[]) の範囲の XPath 問合せ処理は, 問合せ木中の全てのノードテストが xmlノードにマッチし, かつマッチした xmlノードが問合せ木中の枝が示すノードテスト間の関係を満たすような, XML 部分木列を返す問題である. マッチした部分木を

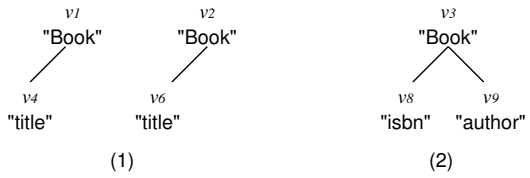

図 3　マッチ木の列.

マッチ木と呼ぶことにする. XML 木 T に対して XPath 問合せ q を評価する操作を, $xpathEval$ で表し, その結果をマッチ木の列とする. 図 3 に, 図 2 の問合せ木に対するマッチ木列を示す.

XML 木に対して問合せ木にマッチする部分木を素直に探そうとすると, 木構造の全探索が必要になる. XML 木が大きく一次記憶に格納できないような場合には, 二次記憶への入出力が発生し, 全検索は非常にコストが高くなる.

範囲ラベル付け手法と構造ジョイン: 上記で説明した XPath 処理における全検索を回避するために, 範囲ラベル付け手法に基づいた構造ジョインアルゴリズムが開発されている [2](注5). この手法の基本的アイディアは, ノード間の先祖子孫および親子関係を判定可能な範囲ラベルを xmlノードに付与することにより, XPath 問合せの軸処理を範囲ラベルの比較操作に置き換え, XML 木の枝の探索を回避するというものである.

範囲ラベルを各 xmlノードに対して, XML 木の一回の深さ優先探索により前置順と後置順それぞれの狭義単調増加数列の対で与える (図 4). 前置順と後置順の値をそのまま範囲ラベルとして与えずに間隔を空ける理由は, XML データの更新操作に対応させるためである. 間隔は更新操作の特徴に応じて決定することが出来る. 間隔がつまった場合には, 小規模に間隔をメンテナンスする方法が提案されている [3], [26]. 範囲ラベルを XML 木

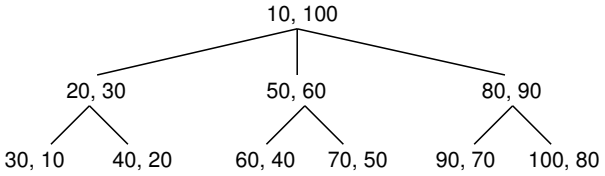

図 4　範囲ラベル.

中の全 xmlノードに付与すると, 任意の xmlノード間の先祖子孫関係が範囲ラベルの比較のみで判定できる. さらに根 xmlノードからの深さもラベルとして与えると, 親子関係も判定可能である.

範囲ラベルを xmlノードに付与し xmlノード単位で分解格納し, さらに XML 木のタグ名に索引を張ることにより, XPath 問合せは, 下記の二段階で処理することができる.

（1） タグ上の索引を利用して XPath 問合せ木に登場するノードテストにマッチする xmlノードを二次記憶装置から取り出す.

（2） 取り出した xmlノードリストから, 問合せ木のそれぞれの枝が示す関係を持つマッチ木を得る.

この方法では, タグ名に索引を張っているため, XML データの全検索を回避することが可能になる. (2) の処理は構造ジョインと呼ばれ, 範囲ラベルの特徴を利用することで, 既存の RDBMS で実現されているジョインアルゴリズムよりも高速なアルゴリズムが各種研究されている [2], [15]. 大規模 XML データ処理においてはディスク I/O が性能上のボトルネックとなるため, データの全検索を回避する上記手法は, 高速な XPath 処理手法の基礎であるとみなされている [15].

XPath を処理する際の構造ジョインのコスト: 文献 [24] によると構造ジョインを用いた XPath 処理のコストは, 次の三つの物理操作のコストからなる.

（1） 索引アクセス $f_I \times n$

（2） ソートオペレーション $nlogn \times f_s$

（3） 構造ジョイン
(Stack-tree Desc)　$2 \times |AB| \times f_{IO}$
(Stack-tree Anc)　$2 \times |AB| \times f_{IO} + 2 \times |A| \times f_{st}$
(これらは Stack-tree ジョインのスタック操作の回数および入出力のための I/O コストである)

但し, f_I, f_s, f_{IO}, f_{st} はそれぞれシステム依存のコスト正規化のための定数であり, n はアイテム数, $|A|, |B|$ は 2 項構造ジョイ

(注5)：範囲ラベルは, 各 xmlノードにあらかじめ付与されているラベル (ノード名, タグ) とは違う.

ンを行う xmlノードの濃度, $|AB|$ は結果の濃度である. 文献 [24] では, これらの情報を利用し, 構造ジョイン実行順序最適化を行っている.

3. 提 案 手 法

提案手法の基本的アイディアは, 次の二つである.

（1） 問合せ木中に含まれるシングルパスの処理は, XML データのタグパス情報を用いて, 構造ジョインをスキップして処理する.

（2） 二次記憶からの xmlノード選択処理を, タグ単位ではなくパス単位で行うことによって, 一度に取り出す xmlノード数を削減する.

図 5 に提案手法の処理手順と, 既存手法である構造ジョインのみで XPath を処理する手法との比較を示す. 各物理操作はそ

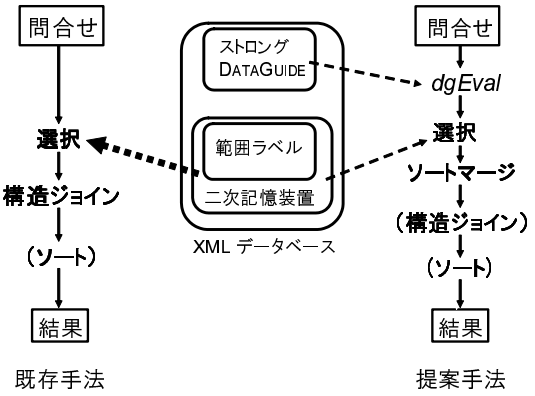

図 5 構造ジョインによる XPath 処理方法と提案手法の比較.

れぞれ,

- **選択** · · · 二次記憶装置からの xmlノードの選択
- **構造ジョイン** · · · xmlノードの構造ジョイン
- **ソート** · · · 前置順での xmlノード列のソート
- *dgEval* · · · DataGuide に対する XPath 処理
- **ソートマージ** · · · ソートされた xmlノードリスト群のソートマージ

を示す. 図 5 左側が既存手法のフローであり, 右側が提案手法である. 真中は本手法で利用する XML データベースにて利用可能な情報を示している.

既存手法では, 最初に問合せ木中にあらわれるノードテストにマッチする xmlノードを二次記憶装置から**選択**する. 二次記憶装置に対する最初の 選択では, それぞれの xmlノード列は前置順にソートされている. これらに対して**構造ジョイン**を実行し, 必要であれば**ソート**を行って結果を出力する. **ソート**は, 構造ジョインの結果である xmlアウトプットノード列が前置順にソートされていないときに必要である.

これに対して提案手法では, まず始めにストロング DataGuide に対して XPath 問合せ処理 *dgEval* を行う. 次に, *dgEval* の結果に対して, 必要な箇所だけ二次記憶装置に対して**選択**を行い, さらにその結果を**ソートマージ**する. 問合せに分岐がある場合は, **構造ジョイン**を行い, 結果を出力する.

提案手法は, 構造ジョイン回数および二次記憶装置から取り出す xmlノード数を減らすことができるため, XPath 処理を高速化することが出来る. 詳細については 3.3 節にて説明する.

3.1 ストロング DataGuide

DataGuide とは, 半構造データベースの要約情報であり [7], 問合せ書換えや, データベースの中身の可視化のために考案されたものである. 本手法では, 特にストロング DataGuide と呼ばれるクラスを利用する(注6).

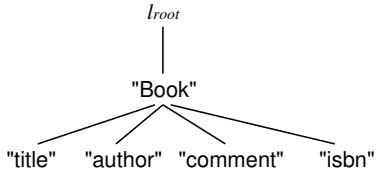

図 6 ストロング DataGuide.

XML 木のストロング DataGuide [7] とは, XML 木のタグパスを, 根 xmlノードから葉 xmlノードへ向かって共通のタグに関して集約し, 木構造表現したものである. 図 6 に図 1 の XML 木のストロング DataGuide を示す. XML 木 T のストロング DataGuide $DG(T)$ は, $(V_{DG}, root, E_{DG}, \Sigma)$ と表すことが出来る. 但し, V_{DG}, E_{DG} は, それぞれストロング DataGuide 中のノード集合および枝集合である. 以降では, DataGuide 中のノードは dgノードと呼ぶことにする.

タグパスの集合 P_t と V_{DG} との間には一対一対応があり, 一つのタグパスがストロング DataGuide 中の一つの dgノードに対応する. よって $|V_{DG}(T)| = |P_t(T)|$ であり, ストロング DataGuide の大きさは, タグパスの濃度 $|P_t|$ に比例する. 表 2 の $|P_t|$ の値をみると, 実際の XML データのストロング DataGuide の大きさは, 非常に小さいことが分かる. また, タ

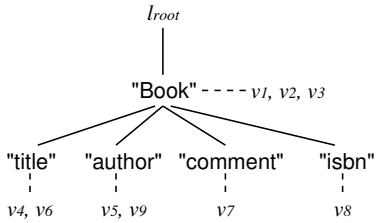

図 7 xmlノード集合のストロング DataGuide によるクラスタ化.

グパス集合を用いて xmlノード集合は分割できるので, ある dgノードに対応するタグパスを持つような xmlノード群を図 7 のようにクラスタ化することができる. このクラスタを本稿では, dgクラスタと呼ぶ.

dgクラスタとタグによるクラスタの関係: タグクラスタ (タグ毎に xmlノードをクラスタ化したもの) は, dgクラスタのアッパークラスタになっている. すなわち, 任意のタグクラスタは, ある dgクラスタ集合によってクラスタ化される. 今, タグ t の生成するタグクラスタを C_t とすると, C_t はタグパスが $*.t$ で

(注6)：以降, DataGuide と言ったときにもストロング DataGuide を指すものとする.

あるような, dgクラスタ群で構成される. この性質により, 提案手法での二次記憶からの xmlノードの取り出しは, より細かい粒度で行うことができる. 詳細は, 3.3 節にて説明する.

ストロング DataGuide の構築方法: ストロング DataGuide を定義通りに XML 木から構築しようとすると [1], XML 木を何度も探索する必要があり, 空間効率が悪く構築時間もかかる. そこで, XML データの SAX イベントシーケンスから, XML 木構造を構成することなく DataGuide を構築できるアルゴリズムを構築した. 図 8 にアルゴリズムを示す. 簡単のため属性処理

input: SAX イベントシーケンス:$\{e_i\}_{i=1}^{m}$
output: ストロング DataGuide : DG
1 DG を初期化. 現在 dgノード n を DG の $root$ dgノードとする.
2 **for** $(i = 1; i < m + 1; i = i + 1)$
3 **if** (e_i が開始タグ)
4 **if** (e_i を名前とする n の子 dgノードが存在しない)
5 n の子供に e_i をタグ名とする子 dgノードを追加.
6 **endif**
7 n を e_i をタグ名とする子 dgノードに移動.
8 **else if** (e_i が終了タグ)
9 n を n の親 dgノードに移動.
10 **endif**
11 **endfor**
12 DG を出力する.

図 8 DataGuide 構築アルゴリズム.

は省略してあるが, 属性も同様に処理できる. このアルゴリズムは, SAX イベントシーケンスの一回の走査で DataGuide を構築することが出来る. また, 範囲ラベルも SAX イベントシーケンスの一回の走査で付与可能であるので, データ投入の際に両方の処理を並行して行うことができるため, 効率が良い.

3.2 DataGuide を用いた XPath 処理アルゴリズム

ストロング DataGuide はラベル付き木であるので, それを XML 木とみなして XPath 処理を行うことができ, この処理を $dgEval$ と呼ぶ. XML 木 T のストロング DataGuide に対して問合せ q を評価する操作を,

$$dgEval(T, q)$$

で定義し, $dgEval$ のマッチ木を, dgマッチ木と呼ぶことにする. $dgEval$ の処理には各種既存の XPath 処理エンジンを利用することができる.

DataGuide を用いた XPath 処理アルゴリズムを, (1) シングルパス (XP(/,//,*)), (2) 小枝パターン (XP(/,//,*,[])) の順番で説明する.

(1) シングルパス: シングルパスの XPath 処理アルゴリズムを図 9 に示す. まず, (1 行目) 問合せを DataGuide に対して処理し, マッチ木列を取り出す. 次に, (2 行目) それぞれのマッチ木列に対して, (3 行目) マッチ木のアウトプット dgノードを取り出し, そのアウトプット dgノードにクラスタ化された xmlノード列を二次記憶装置から取り出す. (5 行目) それぞれの xmlノード列をソートマージして結果を得る.

(2) 小枝パターン: 図 10 に, 小枝パターンの XPath 処理アルゴリズムを示す. まず, (1 行目) 小枝パターンをシングルパスの列に分解する. そして, (2 行目) それぞれのシングルパスに対して,

Algorithm: $singleProcess$
input: XML 木 T, 問合せ q
output: 結果 xmlノード列 $\{v_i\}$
1 $(t_1, t_2 \cdots, t_m)$ = $dgEval(T, q)$
2 **for**$(i = 1; i < m + 1; i + +)$
3 xmlノード列 x_i = $select(T, getOut(t_i))$
4 **endfor**
5 **return** $merge((x_1, x_2, \cdots, x_m))$
method:
$getOut$(マッチ木 t)
マッチ木 t のアウトプット dgノードを返す.
$select$(XML 木 T, dgノード o)
dgノード o のタグパスを持つ xmlノード列を返す.
$merge$(xmlノードリスト列 $(x_1, x_2, \cdots, x_m)$)
全ての xmlノードリストをマージした結果を返す.

図 9 XPath 処理アルゴリズム: シングルパス.

input: XML 木 T, 問合せ q
output: 結果 xmlノード列 $\{v_i\}$
1 シングルパス列 $(p_1, p_2, \cdots, p_m) = decompose(q)$
2 **for**$(i = 1; i < m + 1; i = i + 1)$
3 xmlノード列 $V_i = singleProcess(T, p_i)$
4 **endfor**
5 **return** $getOutSeq(branchJoin(q, (V_1, V_2, \cdots, V_m)))$
method:
$decompose$(問合せ q)
問合せ q をシングルパス列に分解する.
$branchJoin$(問合せ q, xmlノード列 $\{V_i\}$)
問合せ中の分岐ノードと葉ノード間で構造ジョインを行い xml マッチ木列を返す.
$getOutSeq$(xmlマッチ木列 t)
xmlマッチ木列 t のアウトプット xmlノード列を返す.

図 10 XPath 処理アルゴリズム: 小枝パターン.

(3 行目) シングルパス処理を行う. 最後に, (5 行目) それぞれのシングルパスの結果 xmlノード列間で構造ジョインを行う. 小枝パターンを分解するアイディアは, 文献 [25] の方法と類似のものである.

ストロング DataGuide は, タグパスを正確に保持しているためシングルパスの処理は, DataGuide の情報のみで行うことができる. 一方, 小枝パターンの処理は, アルゴリズムが示すように, シングルパス処理が基本となる. 問合せのマッチ木の候補であるタグパスの分岐情報は, DataGuide には保持されていないため, 構造ジョインを行うことによりタグパスにマッチする各ノードパスに分岐が存在し, 小枝パターンにマッチすることを確認する必要がある.

3.3 議　　論

3.3.1 提案手法のコスト

図 5 に示した提案手法のコストについて説明する.

***dgEval* の計算コスト:**
文献 [8] より, *Core* XPath と呼ばれる XPath1.0 のサブセットは, $O(|T| * |Q|)$ で処理可能であることが証明されている. 本稿で議論している XP(/,//,*,[]) は, *Core* XPath のサブセットであるので, $dgEval$ の計算量は,

$$O(|DG(T)| * |Q|)$$

である.

選択とソートマージのコスト: 提案手法は**選択**のコストを, 二

つの点で既存手法に対して削減している.

（1） dgクラスタに対する xmlノード選択による, 選択 xmlノード数の削減.

既存手法の**選択**が, タグ単位, すなわちタグクラスタ単位で行われるのに対して, 提案手法では, $dgEval$ を先に行うことにより, タグパス単位で**選択**を行う. dgクラスタとタグクラスタの関係から, 取り出す xmlノード数は, 少なくなる.

（2） ストロング DataGuide の利用によるシングルパスの中間ノードのスキップによる選択 xmlノード数の削減.

既存手法では, 問合せ中のノードテスト毎に**選択**を実行していたが, 提案手法では $dgEval$ により, **選択**をしなければならないノードテストが問合せ木中の分岐ノードと葉ノードに限定されている. それ以外のシングルパスの中間ノードの分の**選択**は実行する必要がない.

しかしながら, 一回の $dgEval$ の評価結果が複数の dgクラスタにまたがっている場合は, 複数回**選択**を行う必要がある. この**選択**の呼出し回数だけ**ソートマージ**のコストが増加する. つまり, **選択**で呼び出した回数分の xmlノードリストの**ソートマージ**を行う.

複数回の**選択**と**ソートマージ**のコストは, 4. 節の評価実験で実際に観察することができる.

構造ジョイン のコスト:

XPath 処理での, 構造ジョインのコストは構造ジョインの回数と, それぞれのジョイン結果の xmlノード数で見積もることができる. 提案手法では, シングルパスの中間ノードに関する構造ジョインをスキップしているので, 構造ジョイン回数を削減している. また, 分岐処理のために行う構造ジョインに関しても, シングルパス処理の結果と, 既存手法で処理した結果は同じであるので, 結果 xmlノード数が増えることはないので, コストが増加することはない.

3.3.2 処理可能な XPath クラスの拡大

現実には, XP(/,//,*,[]) のクラスだけではなく,

```
/Booklist/Book[./authors/person = 'Tanaka']
```

や,

```
//Book[./isbn = '1-111-111-1']//person
```

のような, テキスト値に対する条件指定を含む XPath 問合せを書くことが多い. そこで提案手法を用いてこれらの XPath 問合せを処理する方法を図 11 に示す. 図 11 には, 値条件を切り離し, 構造部分を先に DataGuide に対して適用してから値のフィルタ条件を適用する, 非常に直感的な手法を示した. しかし, 実際は構造部よりも, フィルタ条件を先に適用した方が高速に実行できる場合も考えられる. この, 値処理と構造部処理の組み合わせの最適化方法は, テキストノードの二次記憶への格納方法とともに今後の課題である.

1 XPath 式を $XP(/, //, *, [])$ の範囲の構造部分と, 値に対するフィルタ条件とに分離する.
2 構造部分をストロング DataGuide と, 範囲ラベルを用いて処理し, 結果を得る.
3 値に対するフィルタ条件を結果に対して適用し, フィルタ処理を行う.

図 11 テキスト値に対する条件を含む XPath の処理手順.

3.3.3 小枝ジョインとの統合

`Stack-tree` ジョインや $\epsilon\epsilon$ ジョイン [16] が二項ジョインであるのに対して, 小枝ジョインは多項ジョインである. 小枝ジョインでは中間生成物を生成する必要がないため, 最適な構造ジョイン方法であるとされている. しかしながら, 小枝ジョインアルゴリズムおよびその改良手法 [11], [14] では, ジョインされる xmlノードのタグと範囲ラベルのみの情報で**選択**を行っている. そのため, 中間生成物を生成しないものの, 問合せ木中の全ノードテストが指定する xmlノードにアクセスすることに変わりはない. 上記述べたアルゴリズムと同様に, 小枝ジョインアルゴリズムに対してストロング DataGuide の情報を利用してアクセスする xmlノードを減らすことができる.

3.4 具 体 例

アルゴリズムの具体例を示す. XML 木 T は図 1 とし, 図 4 のように範囲ラベルが付与されているものとする. ストロング DataGuide は, 図 6 であり, 各 dgノードはタグパスを用いて表現している.

例 1: シングルパス (*singleProcess*)

$Q =$“`//Book//title`”
(1 行目) $dgEval$ により, dgマッチ木列

$$((\text{“}l_{root}\texttt{.Book}\text{”}, \text{“}l_{root}\texttt{.Book.title}\text{”}))$$

を得る.
(2, 3, 4 行目) それぞれのマッチ木に対して, アウトプットノードを取り出し, *select* を行う.

$$\begin{aligned} i = 1 \ : \quad & getOut((\text{“}l_{root}\texttt{.Book}\text{”}, \text{“}l_{root}\texttt{.Book.title}\text{”})) \\ & = \text{“}l_{root}\texttt{.Book.title}\text{”} \\ x_1 \ = \ & select(T, (\text{“}l_{root}\texttt{.Book.title}\text{”})) = (v_4, v_6) \end{aligned}$$

今回は, $m = 1$ であるから, マージはそのままの結果 (v_4, v_6) を返す.

4. 評 価 実 験

上記説明したアルゴリズムのうち, シングルパスの処理に関して実装し, 評価実験を行った. 実験環境の概要を図 12 に示す.

実験環境は Pentium 4 2.53GHz, 2GB メモリ 上の Linux 2.4.27 であり, データベースとして PostgreSQL 7.4 を用いた. 文献 [10] と同様のテーブルを構築し, データへのアクセス手段は PostgreSQL の *SPI*(注7) を用いている. 構造ジョインは, C++で `Stack-tree` ジョインを, PostgreSQL のモジュールとして組み込んでいる(注8). データの選択は *SPI* のカーソルを用いておこない, また PostgreSQL が誤って遅いプランを選択するのを回避するため, パラメータを調節して, 常にインデックスを用いて高速にデータにアクセスするようにしている. パフォーマンスの比較対象は二項構造ジョインである `Stack-tree` ジョインを組み合

(注7)：*Server Programming Interface.*
(注8)：gcc V3.4, -O2 オプション使用.

例 2: 小枝パターン

$Q =$ “//Book[./isbn]//author”
(1 行目) *decompose* により, シングルパス列

(“//Book”, “//Book/isbn”, “//Book//author”)

を得る.
(2, 3, 4 行目) それぞれのシングルパスに対して, シングルパス処理を行う.

$i = 1$:
$V_1 = singleProcess(T, \text{“//Book”}) = (v_1, v_2, v_3)$
$i = 2$:
$V_2 = singleProcess(T, \text{“//Book/isbn”}) = (v_8)$
$i = 3$:
$V_3 = singleProcess(T, \text{“//Book//author”}) = (v_5, v_9)$

(5 行目) 分岐ノードと葉ノード間で構造ジョインを行い, アウトプットノード author を出力する.

$$branchJoin(\text{“//Book/[.isbn]//author”}, (V_1, V_2, V_3)) = (((\text{“Book”}, v_3), (\text{“isbn”}, v_8), (\text{“author”}, v_9)))$$
$$getOutSeq(((\text{“Book”}, v_3), (\text{“isbn”}, v_8), (\text{“author”}, v_9))) = (v_9)$$

わせた *Fully-Pipelined plan* [24] である. *Fully-Pipelined plan* は, 構造ジョインに適した実行計画であり, 8 割方最適な実行計画となる.

ストロング DataGuide は jdk 5.0 で実装し, $dgEval$ は, java で実装されたストリーム XPath エンジンである XSQ [18] を用いて実現している. ストリーム型の XPath 処理エンジンは各種開発されており, XML データの一回のスキャンで XPath 処理を行うことができる(注9). ストロング DataGuide は一次記憶に保

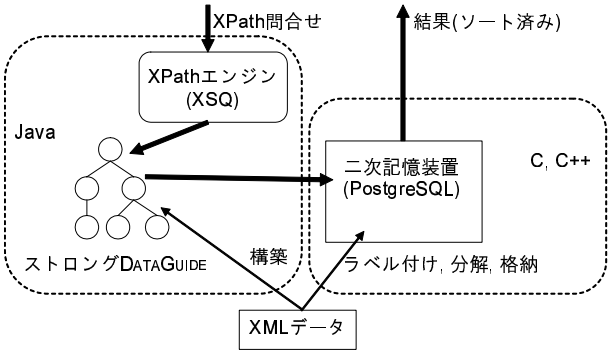

図 12　実験環境.

持し, そのイベントシーケンスを生成することにより, XSQ を用いて XPath 処理を行う. ストロング DataGuide 中でマッチした dgノードを利用して, 二次記憶装置から xmlノードを取り出しソートを行い, 結果を得ている.

以下の実験では XPath 問合せパース開始からアウトプットノードがソート終了するまでの時間を計測しており, アウトプットノードを根とする部分木構築の時間は含まれていない. 各問合せに対して, それぞれ 20 回同じ問合せ処理を連続して行い, そのうちの 11 回から 20 回めまでの平均値を計測値としている.

(注9)：各ソフトウェアによって処理可能な XPath クラスは異なる [6], [9].

4.1　データセット

XMark ベンチマーク [19] のデータジェネレータを用いて実験データを生成した. 表 3 にベンチマークデータの特徴を示す. 表 3 は, DataGuide の大きさも示している. ストリーム型の XPath エンジンは, XML 木の一回のスキャンで XPath を処理可能であるので, DataGuide の大きさは, 問合せ処理速度に大きな影響を与える. 表から分かるように, データベースサイズが増加しても, DataGuide の大きさは増加せず小さい. つまり, ストリーム型処理器による DataGuide 処理は非常に高速に処理できると予測される.

4.2　問合せセット

表 4 に, 今回の実験で計測した問合せセットを示す. $|sj|$ は, 既存手法で構造ジョインのみを用いて XPath 問合せを処理した際に実行する構造ジョインの回数を示している. $|A_{\text{dg}}|$ は, 問合せが DataGuide 中でマッチした dgノードの濃度を示している(注10). 例えば, (問合せ 2)`//location` の場合は,

```
site.regions.africa.item.location
site.regions.asia.item.location
site.regions.australia.item.location
site.regions.europe.item.location
site.regions.namerica.item.location
site.regions.samerica.item.location
```

の 6 個のタグパスにマッチする. 提案手法では, これらの dgノードに対応する xmlノードリストをそれぞれ独立に取り出すので, 最終的にソートマージを行わなければならない. $|A_1|, |A_2|, |A_3|$ は, それぞれ XMark1, XMark2, XMark3 における問合せの結果 xmlノード数を表す.

4.3　実験結果および考察

図 13-15 は, それぞれのデータに対する処理時間を示している. s-join method が既存手法である構造ジョインによる XPath 処理手法であり, DG-based method が DataGuide を用いた提案手法の処理時間である. 横軸の番号は図 4 の問合せの番号を示し, 縦軸は, ミリ秒での処理時間を示している. 3 つのグラフは, 縦軸のスケールが異なる点に注意されたい.

全ての図からわかる, 問合せによる処理時間の傾向は,

（1） 構造ジョインによる XPath 処理手法は, 問合せの長さが長くなるほど処理時間が増加している.

（2） 提案手法は, $|A_{dg}|$ の値が大きいとき (問合せ 2, 3, 4, 5) には, 処理時間が増加しているが, 問合せの長さが長くなっても処理時間は増加しない.

表 3　データセット.

XML データ			DataGuide	
名前	ファイルサイズ	xmlノード数	容量	dgノード数
XMark1	1.1MB	52,138	7kB	455
XMark2	22MB	1,023,421	9kB	549
XMark3	111MB	5,072,510	9kB	549

(注10)：$|A_{\text{dg}}|$ の値はデータに応じて異なるが, DataGuide の大きさと同様, 大きくは異ならない. 表の数値は, XMark2 のものである.

表 4　問 合 せ.

	XPath 問合せ	$\|sj\|$	$\|A_{\mathrm{dg}}\|$	$\|A_1\|$	$\|A_2\|$	$\|A_3\|$
1	`//categories`	0	1	1	1	1
2	`//location`	0	6	217	4350	21750
3	`//parlist//listitem`	1	9	576	12496	60481
4	`//parlist/listitem`	1	9	576	12496	60481
5	`//parlist//listitem//parlist`	2	9	77	1615	7677
6	`//site//categories//category//description//parlist`	4	1	3	94	449
7	`/site/categories/category/description/parlist`	4	1	2	63	285
8	`//site//categories//category//description//···//bold//emph`	8	2	1	8	30
9	`//site/categories/category/description/···/text/bold/emph`	10	1	0	6	26

図 13 の場合, 構造ジョイン回数が少ない場合には, 既存手法の方が高速である. しかしながら, 構造ジョイン回数が増えた場合には, 提案手法が上回る. 図 14 になると, 構造ジョインを要求しない問合せ 1, 2 以外は, 提案手法の方が高速である. 問合せが長くなればなるほど差はひろがっている. 構造ジョインが要求されない場合で, つまり $|A_{dg}| = 1$ である問合せ 1 の場合, 既存手法では, 単純な二次記憶装置からの xmlノードの取り出し時間になるはずなので, 問合せ 1 の差は, XSQ の $dgEval$ の処理時間と考えられる. 図 15 になるとその差は歴然となり, 構造ジョインの必要な問合せ 3, 4, 5, 6, 7, 8, 9 では, 2 倍から数百倍提案手法は高速に XPath を処理している.

処理時間の内訳: 図 16 は, XMark3 データセットの際の, XSQ による DataGuide 処理時間を示している. これにより, XSQ 処理は非常に高速に実行されていることが分かる. XSQ 処理以外の時間は, xmlノードの取り出しとソートにかかる時間である. 問合せ 2, 3, 4, 5 では, 処理時間が大きいが, これは $|A_{dg}|$ の増加に起因するものである. さらに $|A_{dg}|$ が等しい問合せ 4, 5 で処理時間が異なるのは, 図 4 に示す xmlノードの選択率の違いによるものである.

スケーラビリティ: 図 13-15 でも, 提案手法がスケーラビリティを示していることは分かるが, より詳細に図 17 に横軸をファイルサイズとして, 問合せ 6 の場合の処理時間を示す. データベースサイズの増加に対して, 構造ジョインの処理時間はほぼ線形に増加している. 一方, 提案手法では, ほとんど処理時間は増加していない. この傾向は, 実験を行った全ての問合せに対して同じであった.

小枝パターン処理に対する考察: 小枝パターン処理アルゴリズムはシングルパス処理が基本になり, 分岐ノードのみ構造ジョインを行う. シングルパス処理結果である xmlノード列は前置順でソートしてあるため, そのまま分岐ノードにおける小枝パターンの引数 xmlノードとして与えることができる. すなわち, 小枝パターンの処理速度は, 特に長いシングルパスを含む場合は, 上記シングルパス処理の結果に準ずると予測される.

以上により, データベースが大きくかつ問合せの長さが長い場合には, DataGuide を用いた提案手法は, 構造ジョインによる XPath 処理手法を大きく上回ることが分かった. 特に, 111MB の XMark の場合, 既存手法で構造ジョインが必要な問合せに関

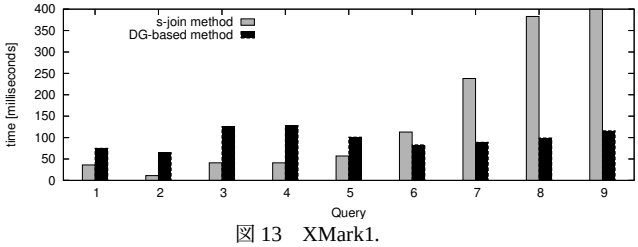

図 13　XMark1.

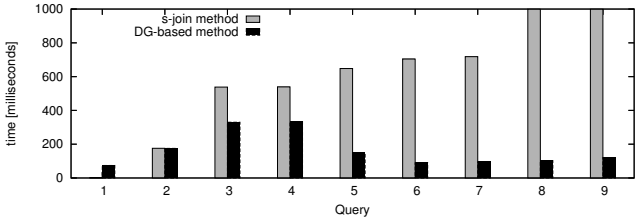

図 14　XMark2.

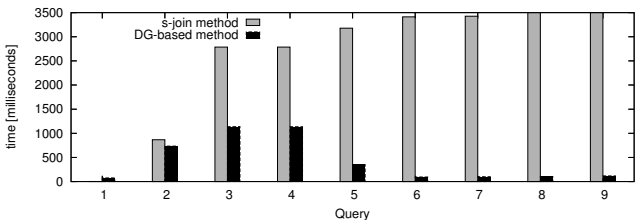

図 15　XMark3.

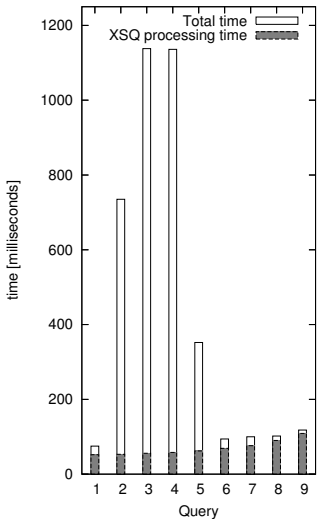

図 16　処理時間全体における XSQ の処理時間の割合.

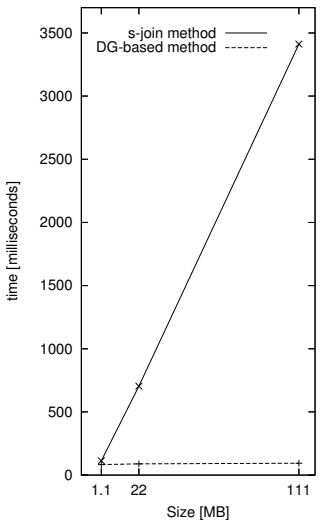

図 17　スケーラビリティ.

しては, 提案手法は 2 から数百倍の高速化を達成している.

5. 関 連 研 究

XP(/,//,*) のクラスの XPath 処理は, ストロング DataGuide の情報のみで処理可能である. 文献 [13] では, 小枝パターンも全て処理できる要約情報として, F&B インデックス [1] を利用している. しかしながら, F&B インデックスは小枝パターンも構造ジョインなしで処理できる一方, 容量が大きいため一次記憶内での処理時間が増加するという問題がある. また構築するには XML 木を何度も探索する必要があるため, 高速に構築することができない. 一方, ストロング DataGuide の処理時間は, DataGuide の大きさに比例するが, 一般的にストロング DataGuide は小さいため高速に処理可能であり, また既に述べたように高速に構築することも可能である. ストロング DataGuide と F&B インデックスの空間容量と処理速度のトレードオフの計測は今後の課題である. また, 統語構造をマークアップした Treebank のような不規則なデータの場合, ストロング DataGuide も非常に大きくなってしまうという問題がある. この問題を回避するために, データマイニングアルゴリズムに基づいた要約情報が, 文献 [4] にて提案されている. また, 文献 [17] では, DataGuide を反転させ (*reversed*), 枝刈りを行って容量を削減する方法を提案している.

先祖子孫関係を判定可能な xmlノードラベル付け手法は, 多数提案されている. スキーマを用いない方法では, 大きく分けると, (a) 範囲を用いるもの [16], [25], (b) プレフィックスを用いるもの [21], (c) 多分木を用いるもの [22], (d) 素数を用いるもの [23] がある. 文献 [5] では, (a), (b) のラベル付け手法に関して, ラベルの付け直しをせずに更新操作を行った際に, ラベルを表現するのに必要なビット数に関して理論的な分析を行った. 文献 [23] では, 素数の分布の特徴と中国式剰余定理を利用して現実的な XML データに対して有効なラベル付け手法を提案している.

RDBMS への XML 格納および XPath 処理手法である XRel [25] が利用しているパステーブルは, XML 木中のタグパスの集合である. 本研究は, RDBMS に有効であったパスを用いた XPath 処理方法が, 構造ジョインによる XPath 処理においても有効であることを示したと言える. 文献 [27] では, 本研究と目的は異なるが, ストロング DataGuide を用いて, RDBMS に格納する XML データの情報を削減する方法を提案している.

6. ま と め

本稿では, XML データの要約情報としてストロング DataGuide を用いて構造ジョインの回数を削減すると同時に一次記憶に読み出す xmlノード数を削減する手法を提案した. 提案手法は, シングルパス問合せ (XP(/,//,*)) は構造ジョインを行うことなく, 小枝パターン (XP(/,//,*,[])) の場合でも, 分岐ノードと葉ノード間の構造ジョインのみで XPath を処理する. 評価実験の結果, シングルパスの問合せに対して, 100MB 以上の XML データの場合, 既存の構造ジョインを用いた最適化手法と比較し, 2 から数百倍の高速化を達成した. また, 100MB 以下の XML データに関して, 提案手法はスケーラビリティを持つ手法であることを示した. さらに提案手法の処理コストに対して考察した結果, 取り出しノード数の削減が, 高速な XPath 処理に有効であることを示した. 今後の研究課題としては, XML データの要約情報の, 処理能力とパフォーマンスのトレードオフ評価を挙げる.

文 献

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[2] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, 2002.

[3] Yi Chen, George Mihaila, Sriram Padmanabhan, and Ragesh Bordawekar. L-Tree: a Dynamic Labeling Structure for Ordered XML Data. In *Proc. DataX*, 2004.

[4] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Sim. APEX: An Adaptive Path Index for XML data. In *Proc. SIGMOD*, 2002.

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

[6] Yanlei Diao, Mehmet Altinel, Michael J.Franklin, Hao Zhang, and Peter Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. Technical report, EECS, University of California, Berkeley and IBM Almaden Research Center, San Jose, 2002.

[7] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB*, 1997.

[8] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. VLDB*, Hong Kong, 2002.

[9] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. In *Proc. ICDT*, 2003.

[10] Torsten Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD*, 2002.

[11] Jiang Haifeng, Hongjun Lu, Wang Wei, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins . In *Proc. ICDE*, 2003.

[12] http://www.epcglobalinc.org/. EPCglobal Inc.

[13] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering Indexes for Branching Path Queries. In *Proc. SIGMOD*, 2002.

[14] Franky Lam, William M. Shui, Damien K. Fisher, and Raymond K. Wong. Skipping Strategies for Efficient Structural Joins. In *Proc. DASFAA*, 2004.

[15] Hanyu Li, Mong Li Lee, Wynne Hsu, and Chao Chen. An Evaluation of XML Indexes for Structural Join. In *SIGMOD Record*, Vol. 33, September 2004.

[16] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *The VLDB Journal*, 2001.

[17] Hartmut Liefke and Dan Suciu. XMill: an Efficient Compressor for XML Data. In *Proc. SIGMOD*, 1999.

[18] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In *Proc. SIGMOD*, 2003.

[19] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J.Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, 2002.

[20] Thomas Schwentick. XPath Query Containment. In *ACM SIGMOD Record*, 2004.

[21] Igor Tatarinov, S.D.Viglas, K.Beyer, J.Shanmugasundaram, E.Shekita, and C.Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. SIGMOD*, 2002.

[22] Wang Wei, Jiang Haifeng, Hongjun Lu, and Jeffrey Xu Yu. PBi-Tree Coding and Efficient Processing of Containment Join. In *Proc. ICDE*, 2003.

[23] Xiaodong Wu, Mong Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proc. ICDE*,

2004.

[24] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proc. ICDE*, 2003.

[25] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel:A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. In *ACM Transactions on Internet Technology*, Vol. 1, 2001.

[26] 江田毅晴, 櫻井保志, 天笠俊之, 吉川正俊, 植村俊亮. XML 木のための動的範囲ラベル付け手法. 情報処理学会論文誌: データベース, 第 45 巻, 2004.

[27] 天笠俊之, 植村俊亮. リージョンディレクトリを用いた関係データベースによる大規模 XML データ処理. In *DBSJ Letters*, No. 2 in Vol. 3, 2004.