DEWS2004 7-A-02

実体化 XSLT ビューの漸進的更新

道上　龍介† FongYee Chan† 鬼塚　真† 芳西　崇†

† 日本電信電話株式会社 NTT サイバースペース研究所　〒 239–0847 神奈川県横須賀市光の丘 1–1
E-mail: †{michigami.ryusuke,fongyee.chan,onizuka.makoto,honishi.takashi}@lab.ntt.co.jp

あらまし　本論文では，クラス $XP^{\{[],*,//\}}$ の XPath のみを使用した XSLT 変換における実体化 XSLT ビューに対し，変換元の XML ソースデータに変更があった場合にその情報を用いて差分更新を行なう手法を提案する．本手法は二つの段階からなる．1）全変換を行ない，XML ソースデータに適用された XSLT テンプレートの呼び出し履歴を XT(XML Transformation) 木として記録する．2）XML ソースデータに加えられた変更操作に応じて XT 木の更新すべき部位を見つけ，その部位のみ部分更新を行なう．提案手法においては，差分更新のコストはソースデータに加えられた変更の大きさに依存するため，全変換と比べて飛躍的に性能を向上することが出来る．また，XT 木の空間コストは一般的に変換結果の XML データの大きさに応じたものであり，XML ソースデータの大きさには依存しない．評価実験の結果，全変換と比べて差分更新が有効であることを確認した．

キーワード　XML, 半構造データ, Web とインターネット, XSLT

Incremental Maintenance for Materialized XSLT Views

Ryusuke MICHIGAMI†, FongYee CHAN†, Makoto ONIZUKA†, and Takashi HONISHI†

† NTT Cyber Space Laboratories, NTT Corporation　1–1 Hikarinooka, Yokosuka–Shi, Kanagawa 239–0847, Japan
E-mail: †{michigami.ryusuke,fongyee.chan,onizuka.makoto,honishi.takashi}@lab.ntt.co.jp

Abstract　This paper proposes an incremental maintenance algorithm for materialized XSLT views defined with XPath expressions in the $XP^{\{[],*,//\}}$ class, so as to efficiently update the XML transformation result. The algorithm consists of two processes. 1) Storing a dynamic call tree of XSLT templates as XT (XML transformation) tree during the full XML transformation. 2) In response to XML source data updates, locating the affected portions of the XT tree and update them by re-evaluate the XSLT templates. Our algorithm outperforms the existing full transformation method whose performance generally depends on the whole size of the source data, since the cost of the incremental maintenance depends on the size of the updated portion of the source data. In addition, our algorithm requires a modest size of memory space since the space cost of our algorithm depends on the number of XT nodes that generally scales to the size of the transformed result. The evaluation result also shows that the incremental maintenance algorithm improves the performance over the full transformation by factors of 1 to 3.

1. はじめに

INTERNET の普及に伴い，バックエンドにファイルやデータベースを用いて XML データを管理し，それを変換して動的に web ページを生成するサイトが増加してきている（NEC Research Index や DBLP 等の論文情報, 空港での飛行機の発着状況, 電子番組表 EPG, 株価の情報の web サイトなど）．このようなサイトは XSLT 処理系を用いて XML を HTML へと変換するが，この XML 変換は web サーバの性能のボトルネックの一つであり，その高速化は重要な技術的課題である．

動的 web サイトにおけるデータの処理を分析してみると，一般的な特徴として 1）バックエンドのソースデータは比較的大量である，2）ソースデータに対する追加・更新・削除操作はデータ全体のごく一部を対象とする，ということが挙げられる．例えば，FlightArrivals.com [5] 等の空港での飛行機の発着情報を提供する web サイトでは，データ量は個々の空港の一日の発着便数というように比較的大量であり，データの更新操作は飛行機の状態（到着済み, 到着予定時刻, 搭乗開始, 出発済み等）が変化したときにその飛行機のデータのみを更新する．また電子番組表の例では，地上波だけでも十数チャンネル分の 1〜2 週間分の番組情報を蓄積する必要があり，データの更新操作はプロ野球の延長やニュース速報による割り込みなどで番組が変

更になる場合や，未登録の新しい一日分のデータが追加になるように，データ全体一部が対象である．このように更新対象がソースデータ全体の一部であるにも関わらず，現状の XSLT 処理技術（全変換）ではソースデータ全体に対し変換を再実行する以外に，更新を変換結果に反映する手段がないという問題がある．

本稿では，上述した特徴「更新操作対象がソースデータ全体の一部である」ことを踏まえ，あらかじめ実体化した XSLT の変換に関する結果を，更新差分情報を用いて漸進的更新（incremental maintenance）することで，更新を変換結果に反映するアルゴリズムを提案する．本論文における貢献を以下に整理する．

1.1 貢　　献

ソースデータの更新がどの XSLT テンプレートに影響するかを判定するには，変換結果だけの実体化では，全ソースデータに対して XSLT プログラムを再実行する必要が生じてしまう．このため本手法では，事前に XSLT テンプレートの呼び出し木（XT 木）の情報を実体化し，各 XT ノードでテンプレート ID とそのコンテキストの XML ノード ID を保持する．ソースデータが更新された際に，XML のルートノードから更新された対象ノードまでの XML ノード ID パス（更新対象パス）が，XT 木のどの箇所に変更影響を及ぼすかトップダウンに判定し，変更影響が及ぶ箇所について XSLT テンプレートを再実行する．この結果，本手法は従来の全変換の方法と比較して，XSLT テンプレートの再実行回数（XPath 式の評価回数）を削減できるため，性能を向上することが可能である．

またこのトップダウン法による XT 木の漸進的更新アルゴリズムが特に有効であるのは，XPath のクラスが変数参照を含む $XP^{\{[],*,//\}}$ [7] である（そのクラスにおけるアルゴリズムを 3. 章で詳述する）．$XP^{\{[],*,//\}}$ は，ワイルドカードを含むノードテスト，軸は self，child，decendant の子孫方向の軸のみ，および述語を利用できる XPath のクラスである．このクラスの XPath が有効な理由は以下のとおりである．

（1） XT 木の変更影響箇所の判定の際に，子孫方向にのみ XT 木を辿れば良い．

（2） XT 木の変更影響箇所の判定では，XSLT で使用される XPath 式について a） 述語以外の部分は，オートマトンによる XPath 式の評価手法 [6] 等と同様に，XPath 式のノードテスト・軸を更新対象パスと文字列レベルで比較することで処理可能，b） 述語部分に関する部分は，XT 木に保持されている XT ノードの存在が述語が真として評価されたことに相当するため，XT ノードの保持するノード ID と更新対象パスのノード ID とを比較することで処理可能．

（3） XT 木の変更影響箇所判定後の XSLT テンプレート再実行は，子孫方向の軸にのみ限定されているため，再実行のコストは一般的に XML ソースデータの更新サイズに依存する．

$XP^{\{[],*,//\}}$ のクラスでは following や ancestor 等の軸が利用できないが，次の理由により十分実用上有効なクラスである．1） 応用で多く使われている XML データ（MPEG7 や TVAnytime などのコンテンツ流通や電子カルテなど）は混在内容や要素の順序を意識しない半構造データモデル的に利用されている．このため，要素の順序を指定する following, preceding 関連の軸は使われない．2） 先祖方向の軸を利用する XSLT プログラムは，変数宣言を用いることで，子孫方向の軸と変数参照を利用する等価な XSLT プログラムに書き換え可能である．

一般的な XPath 仕様を許す場合，トップダウン法による XT 木の漸進的更新アルゴリズムが有効にはならない理由を以下に述べる．データソースの更新により影響を受ける XT ノードを特定する場合，上述したように XSLT プログラムの XPath 式群と更新対象パスとのマッチングを判定する必要がある．XSLT プログラムの XPath 式で parent, ancesor, following, preceding の軸が利用される場合，

（1） XML 木において上位方向の参照が可能であるため，XSLT プログラムの任意の XPath 式がソースデータの更新の影響を受ける可能性が生じる．つまり全て XT ノードを辿る必要が生じてしまう．

（2） これらの軸の評価は XPath 式の文字列レベルの比較では不十分であり，且つ XML データを参照することなく XT 木への変更影響箇所を判定するためには，ロケーションステップ毎に XPath 式の評価結果を実体化しておかなければならない．この結果，実体化結果サイズがソースデータサイズに依存して膨大になってしまう(注1)．

（3） XT 木の変更影響箇所判定後の XSLT テンプレート再実行では，任意の軸を利用可能であるため，XML ソースデータ全体をアクセスする可能性がある．

また本手法の空間・計算コストに関する概要は以下のとおりである．

空間コスト　空間コストは XT 木のノード数，つまり呼び出される XSLT テンプレート数は，一般に変換結果サイズに依存する．web サーバなどの HTML 生成での変換の場合は，変換結果サイズは十分小さいことが多いため本手法は問題なく適用ができる．

計算コスト　計算コストは，更新対象 XT ノードの検索コストと，XT ノードの更新コストとの和である．変換結果 XML ビュー中の更新箇所数を m とすると，どちらのコストも m に依存する．

更に実験を行い，構造保存するような変換やソートを行う XSLT 変換において，全変換時と比較して 1〜3 桁性能が向上したことを確認する．

本稿の構成は次のとおりである．まず，2. 節では提案手法で取り扱う対象を定義する．そして， 3. 節で提案手法の手順を説明し，4. 節で 提案手法のコストを検討する．5. 節では評価実験とその結果を述べ，6. 節で関連する研究について述べる．7. 節ではまとめと今後の課題を述べる．

2. 仕様と課題の定義

本節では，提案手法で使用する XPath，XSLT，XUpdate を定義し，また提案手法の対象とする課題を定義する．

(注1)：この詳細については本論文の範囲を超えるため省略する．

2.1 XPath

1. 節で述べたように半構造データ的な XML 応用を想定し，XML は混在内容を許さず且つノード間の順序を意識しないデータモデルとする．この結果 XPath の仕様を以下のように簡略化したものと定義する．これは文献 [7] において $XP^{\{[],*,//\}}$ と定義されているものと同様である．

XPath 式　2つの XPath 式を | で接続した表現，もしくは絶対パス式か相対パス式
絶対パス式　'/' 相対パス式
相対パス式　LocationStep の並び.
LocationStep　軸, ノードテスト, 省略可能な述語の繰り返し，の 3 つ組.
軸　self, child, descendant のいずれか.
ノードテスト　要素名, @属性名, *, @*, text() のいずれか.
述語　XPath 式によるノードの存在判断述語，もしくは基本型（整数, 小数, 文字列）に対する演算とそのパラメータによる比較演算述語．但し，パラメータは XPath 式により参照される属性もしくはテキストノードの値または定数値のいずれかである．
また W3C で定義されている XPath の関数（ノードセット, 文字列，ブール，数値上の関数）のうち，ノード間の順序により定義される position() はノード間の順序を意識しない前提に反するため使用できないこととする．

2.2 XSLT

W3C で定義される XSLT1.0 の仕様において，下記の主要な変換命令でない命令群は考慮しないこととする．

外部ファイル参照命令群
xsl:apply-import, xsl:import, xsl:include

変換出力の書式指定命令群
xsl:output, xsl:preserve-space, xsl:processing-instruction, xsl:strip-space, xsl:decimal-format, xsl:number

その他命令群
xsl:fallback, xsl:message, xsl:namespace-alias, xsl:attribute-set

また下記の XSLT1.0 の命令群については，他の命令で書き換え可能であるため考慮しないこととする．

xsl:call-template, xsl:for-each
xsl:apply-template に書き換え可能.

xsl:when, xsl:otherwise, xsl:choose
xsl:if に書き換え可能.

xsl:attribute-set
xsl:attribute に書き換え可能.

xsl:key
XPath の述語（キー値を等価条件で指定する）に書き換え可能.

以上の議論と文献 [2] で定義されている XSLT サブセット $XSLT_0$ を考慮し，サブセット XSLT の文法を以下のように定義する．但し XSLT プログラムで使用される XPath 式は，上記の $XP^{\{[],*,//\}}$ の仕様に加えて xsl:variable, xsl:param により定義される変数値の参照を許す.

XSLT の定義
XSLT プログラムはテンプレート宣言の集合であり，テンプレート宣言は（起動のためのマッチパターン, 省略可能なモード名, パラメータ群）を入力とし，変換結果の生成・他のテンプレートを呼び出す機能を有する．具体的には下記の文法に従うものとする．

テンプレート宣言:
```
<xsl:template match="パターン" mode="モード名">
  （入力パラメータ宣言部）*
  （変数定義部）*
  （if 条件部 | 変換結果生成部 | テンプレート起動部）*
</xsl:template>
```

パターン:
```
'/' | ノードテスト （'|' ノードテスト）*
```

if 条件部:
```
<xsl:if test="述語">
  （変換結果生成部 | テンプレート起動部）
</xsl:if>
```

変換結果生成部:
```
（要素生成部 | 属性生成 | テキスト値生成）
```

要素生成部:
```
<xsl:element name="要素名">
  （変換結果生成部 | テンプレート起動部）*
<xsl:element>
```

テンプレート起動部:
```
<apply-templates select="XPath 式" mode="モード名">
  <param-with select="XPath 式"/>*
  <xsl:sort select="XPath 式" オプション属性/>*
</apply-templates>
```

2.3 XUpdate

本論文では，XUpdate 命令 を XML ソースデータの更新箇所を表現するものとして定義する．具体的には，更新操作がデータソースに対して実行されたる場合，個々の更新箇所を特定する表現として XUpdate 命令を用いる．

XUpdate の定義
XUpdate 命令は部分木挿入，属性値またはテキスト値の更新，部分木の削除の 3 種類があり，それぞれ以下のように表現することとする．

insert(p, r)　新規に挿入される部分木のルートノードまでの更新対象パス式 p と挿入部分木 r からなる．
update(p, v_o, v_n)　更新される値までの更新対象パス式 p, 更新前値 v_o，更新後値 v_n からなる．
delete(p)　削除される部分木のルートノードまでの更新対象パス式 p からなる．
但し，更新対象パス式は XML ソースデータにおいて XML ノードを一意に特定する表現であり，その文法を以下のように定義する．
更新対象パス式　更新 LocationStep の並び.
更新 LocationStep　child 軸とノード表現の組.
ノード表現　ノード ID とノード名の組.
ノード ID とは XML ソースデータにおいて一意に XML ノードを特定する値である．

例えば，文献 XML データにおいてある本にある著者を追加する XUpdate 命令は図 1 のように表現される．

2.4 課 題 定 義

XML ソースデータ XML に対し，XSLT による XML 変換プログラム T が用意されているとする．そして XML に対

```
insert((1,bib)/(5,book)/(8,author),
       "<author role="3rd">
          <name>Serge Abiteboul</name>
        </author>"
```

図 1　XUpdate 命令の例

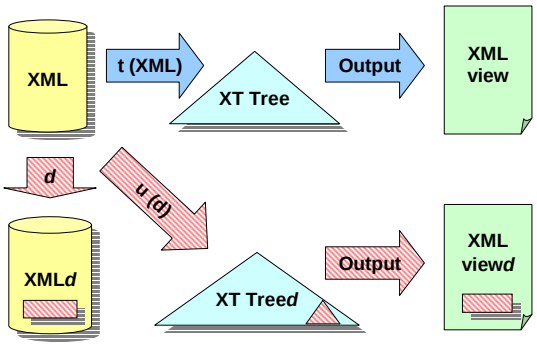

図 2　差分更新処理の概要

し更新操作 $u(d)$ が行われた結果を XML_d とする．本論文で解決する課題は，全変換 $t(XML, T)$ の実行の際に何らかの実体化結果 M を保存・利用することで，全変換 $t(XML_d, T)$ と等価な変換結果を高速に生成する，ことである．

3.　実体化 XSLT ビューの漸進的更新

本節では，提案手法について説明する．まず処理の概要，次いで XT 木の生成法を述べ，最後に XT 木の漸進的更新方式について述べる．

3.1　提案手法の概要

提案手法による処理の概要を示す（図 2）．処理は大きく二つのステップに分けられる．

1）全変換実施/XT 木の生成

まず，処理 $t(XML)$ は XML データと XSLT 変換プログラムを入力として，変換全体を XT（XML Transformation）木に記録する．次いで，XT 木を利用して XML view を生成する．

2）差分更新

XML データに変更 d が生じると，処理 $u(d)$ により，変更 d と変換プログラムに基づいて XT 木を更新する．この時 XT 木は，XML データの変更の影響を受ける部位のみが処理され，それ以外の部位には手をつけない．そして，更新された XT 木から XML view を生成する．

3.2　XT 木の生成

本節では，変換過程を記録する XT 木について述べる．まず XT 木を構成する XT ノードを説明し，それから XT 木の生成アルゴリズムについて述べる．

3.2.1　XT ノード

XT 木は次の二種類の情報を保持する．

1. xsl:template の動的な呼び出しグラフ．
2. 個々の xsl:template の変換結果の XML 部分木．

これらの情報を，以下の XT ノードを組み合わせて表現する．

XT-template

XML ノードに対する XSLT テンプレートの適用を表す．XSLT テンプレートの ID と，テンプレートの適用対象の XML ノードのノード ID，およびテンプレートによる変換結果の XT-view ノードの組を保持する．

XT-set

xsl:apply-templates 命令を表し，XT-template ノードのリストを保持する．各 XT-template ノードは，select 属性で指定された XML ノードにマッチする XSLT テンプレートから生成されたものである．

XT-sortedset

ソートを指定された xsl:apply-templates 命令を表し，XT-template ノードとソートキー値の組のリストを保持する．各 XT-template ノードは，select 属性で指定された XML ノードにマッチする XSLT テンプレートから生成されたものであり，ソートキー値は当該 XML ノードにおけるキー値である．なお，XT-set との違いはソートに関する機能のみである．以下では，それ以外の共通する機能について述べるときには両者をまとめて XT-(sorted)set と表記する．

XT-if

xsl:if 命令を表す．条件が真なら apply-templates に応じた XT-(sorted)set ノードを保持する．

XT-view

変換結果の XML ビューノードを格納する．

検索起点リスト

select 属性値が絶対ロケーションパスを含む XT-(sorted)set ノードのリストを保持する．

3.2.2　XT 木の生成

XT 木の生成は，XML ソースデータと，XSLT プログラムに基づいて行なう．XSLT 変換を行ないながら，その過程で XML ソースデータに適用された XSLT テンプレートの呼び出しとテンプレートの変換結果を XT 木に記録していく．

まず，XML のルートノードを対象とした XSLT テンプレートから XT-template ノードを作成し，それを XT 木のルートとする．また，XT 木のルートノードを検索起点リストに追加する．

テンプレートに含まれる XSLT 命令に対し，表 1 にしたがって XT 木を生成していく．その際 XT ノードのうち，絶対ロケーションパスを含む select や test 属性値を持つものについては，検索起点リストに追加する．テンプレートを適用できる XML ソースデータのノードがなくなったら生成は完了である．

XSLT プログラムと XML データ，および生成した XT 木の例を示す（図 3，図 4，図 5）．

XSLT プログラムのテンプレート t3 と XML ソースデータ 3 行目の article 要素から，XT-template ノード T3 が生成される．T3 は XSLT プログラム 19 行の xsl:element 命令による XML ビュー V3 と 20 行の xsl:apply-templates 命令に対応する XT-set ノード S3 を保持する．S3 は xsl:apply-templates

表 1　XSLT 命令ごとの XT 木生成処理

xsl:element	指定された名前の XML ノードを生成し，XT-view ノードに追加する．
xsl:copy-of	指定された XML ノード配下の部分木をコピーし，XT-view ノードに追加する．
xsl:copy	現在の XML ノードをコピーし，XT-view ノードに追加する．
xsl:apply-templates	sort を指定されているなら XT-sortedset ノードを，そうでなければ XT-set ノードを生成し，現在の XT ノードの子として追加する．select 属性で指定される XML ノード集合の XML ノードそれぞれに対し，適用可能な XSLT テンプレートを探して XT-template ノードを生成し，XT-(sorted)set の子として追加する．このとき，ソートを指定されているなら，ソートキー値の順序となるように追加する．
xsl:if	test 属性で指定される条件を評価し，その結果を保持する．結果が真であれば，if 節内の apply-templates から XT-(sorted)set を生成し，保持する．

```
...
 1: <!-- テンプレート t1 -->
 2: <xsl:template match="/">
 3:   <xsl:element name="html">
 4:     <xsl:element name="table">
 5:       <xsl:apply-templates select="dblp" mode="dblp"/>
 6:     </xsl:element>
 7:   </xsl:element>
 8: </xsl:template>
 9:
10: <!-- テンプレート t2 -->
11: <xsl:template match="dblp" mode="dblp">
12:   <xsl:apply-templates select="*">
13:     <xsl:sort select="title" order="descending"/>
14:   </xsl:apply-templates>
15: </xsl:template>
16:
17: <!-- テンプレート t3 -->
18: <xsl:template match="article">
19:   <xsl:element name="tr">
20:     <xsl:apply-templates select="title" mode="title"/>
21:     <xsl:apply-templates select="author" mode="author"/>
22:   </xsl:element>
23: </xsl:template>
24:
25: <!-- テンプレート t4 -->
26: <xsl:template match="title" mode="title">
27:   <xsl:element name="td">
28:     <xsl:value-of select="text()"/>
29:   </xsl:element>
30: </xsl:template>
...
```

図 3　XSLT プログラムの例

命令の結果，すなわち article 要素の子である title 要素（XML ソースデータ 5 行目）とテンプレート t4 から生成された XT-template ノード T4 を保持する．もし title 要素が複数あればその数だけ XT-template ノードが生成され，S3 に保持される．なおこの例では，XML ソースデータ 3 行目の article 要素は author 要素を子として持たないため，XSLT プログラム 21 行目の apply-templates 命令は実行されず，したがってこれに対応する XT-set ノードも生成されない．

3.3　XT 木の漸進的更新

XML ソースデータに変更が加えられた時に，一貫性を保つように XT 木を更新する．更新処理は，変更後の XML ソースデータ，XT 木，XML ソースデータに対する変更 XUpdate 命令，の三つの情報に基づいて行なう．

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <dblp>
 3:   <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
 4:     <editor>Paul R. McJones</editor>
 5:     <title>The 1995 SQL Reunion: People, Project, and Politics ...</title>
 6:   </article>
 7:   <article mdate="2002-01-03" key="tr/gte/TR-0263-08-94-165">
 8:     <author>Frank Manola</author>
 9:     <title>An Evaluation of Object-Oriented DBMS Developments:
10: 1994 Edition.</title>
11:   </article>
12: ...
13: </dblp>
```

図 4　XML ソースデータの例

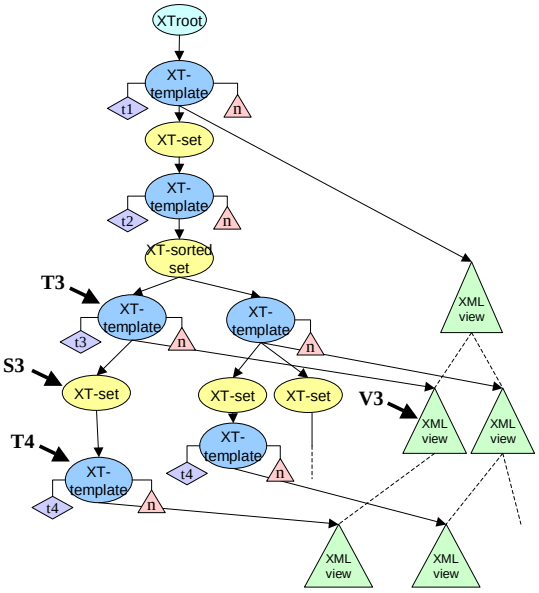

図 5　XT 木の例

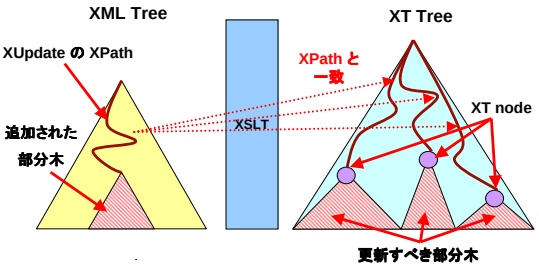

図 6　XT 木更新処理の概要

XT 木更新処理は大きく二つの段階に分かれる．

1）XT ノードの検索：XT 木の更新すべき部位を見つける．

2）XT ノードの更新：見つけた XT ノードのみを更新する．

insert 命令を例として XT 木更新処理の概要を示す（図 6）．まず，XT 木のルートノードから，各 XT ノードが保持する XML ノードと XUpdate 命令の更新対象パス式を比較しながら XT 木を辿り，追加 XML 部分木を参照している XT ノードを探し出す．そして，探し出した XT ノードを更新する．

3.3.1　XT ノードの検索

XT ノードの select 属性値（以下 $xpath_{\mathrm{sel}}$ で表す）は，子の XT ノードのテンプレートがどの XML ノードに適用されたものであるかを表している．検索起点リストの XT ノードか

ら，ある XT ノード n までの経路上の XT ノード列を考えると，各 XT ノードの $xpath_{\mathrm{sel}}$ を結合したものは，XT ノード n がどの XML ノードに適用されたものであるかを表す XPath（以下 $xpath_{\mathrm{XT}}$ で表す）になる．

更新対象の XT ノードは，変更された XML ノードを変換対象とする XT ノードである．したがって，更新対象パスと一致する $xpath_{\mathrm{XT}}$ を持つ XT ノードが更新対象となる．

検索は，XT 木のルートから開始して子孫方向へ辿りつつ行なう．辿り着いた XT ノードでは、$xpath_{\mathrm{XT}}$ までは更新対象パスと一致しているので、そこから続けて XT ノードの $xpath_{\mathrm{sel}}$ と更新対象パスの比較を行なう．$xpath_{\mathrm{sel}}$ が更新対象パスの残り部分と一致しない場合は変更の影響を受けないため，その XT ノード配下を検索対象から除外することにより，検索コストを軽減している．

・$xpath_{\mathrm{sel}}$ と更新対象パスの比較

$xpath_{\mathrm{sel}}$ と更新対象パスの比較について詳しく説明する．$xpath_{\mathrm{sel}}$ と，更新対象パスをロケーションステップ単位で比較し，一致する間ロケーションステップを進める．$xpath_{\mathrm{sel}}$ の末尾に達し，かつ更新対象パスの末尾には達しない場合，子の XT ノードで同様の比較を続ける．

逆に更新対象パスの末尾に達した場合は，XUpdate 命令の種類によって異なる．update/delete 命令ならば，その XT ノードは影響を受けるノードであり，見つけ出した XT ノードを更新対象リストに追加する．

insert 命令の場合は，XT ノードの追加に備える必要があるため，挿入された XML 部分木のノードとの比較をさらに続ける．今度は $xpath_{\mathrm{sel}}$ に対し，XML 部分木を子孫方向へ辿っていく．$xpath_{\mathrm{sel}}$ の末尾まで一致したら，その時点での XML ノードがテンプレートの適用対象 XML ノードであり、現在の XT ノードが新たな XT ノードを追加すべきノードとなる．XML ノードの情報と XT ノードを組にして更新対象リストに追加する．

3.3.2 XT ノードの更新

更新対象リストに含まれる XT ノードを，表 2 にしたがってそれぞれ再評価する．

表 2　XT ノード更新処理

XUpdate 命令	1. insert 命令 2. XPath の述語が真になるような変更
新たに変換対象となった XML ノードから，表 1 にしたがって XT 部分木を生成し，XT-(sorted)set ノードに追加する． 1. の場合，参照している XML ノードに対して適用可能な XSLT テンプレートを探して，両者から XT-template ノードおよびその配下の XT 部分木を生成し，XT-(sorted)set ノードに追加する．XT-sortedset の場合，バイナリサーチによりソート順を保つように追加を行なう． 2. の場合，保持しておいた変更前の述語の状態が偽であれば 1. と同じ処理を行なう．	
XUpdate 命令	1. delete 命令 2. XPath の述語が偽になるような変更
XT-(sorted)set から XT-template ノードを削除する． 1. の場合，XT-template ノードを削除する． 2. の場合，保持しておいた変更前の述語の状態が真であれば 1. と同じ処理を行なう．	
XUpdate 命令	ソートキー値が変わるような変更
XT-sortedset ノードの保持する当該 XT-template ノードを，ソート順序を保つように位置を入れ替える．	
XUpdate 命令	condition が真になるような変更
XT-if ノードに保持している変更前の述語の状態が偽であれば，参照している XML ノードに対して適用可能な XSLT テンプレートを探して，両者から XT-template ノードを生成し，現 XT ノードに追加する．	
XUpdate 命令	condition が偽になるような変更
XT-if ノードに保持している変更前の述語の状態が真であれば，子の XT-template ノードを削除する．	

4. 議　　論

本節では，提案手法における XT 木の空間コスト，差分更新の計算コストの二点について検討する．

4.1 XT 木の空間コスト

XT 木は，個々の XSLT テンプレートによる変換結果である XML ビュー部分木を，テンプレート呼び出しグラフによって結びつけた構造をしている．

前者のみを抽出し結合したものが，変換結果の XML ビューとなる．したがって，全ての XML ビュー部分木のノード数の和を n_v とおくと，XML ビューのノード数も n_v となる．

後者は，主に XT-template ノードと XT-(sorted)set ノードからなる．それぞれのノード数を n_t，n_s とおくと，呼び出しグラフを表す XT ノード数 n_g は $n_g = n_t + n_s$ となる．

変換結果の XML ビュー部分木の平均的なノード数を k とおくと，n_t の値は $n_t = n_v/k$ と表すことが出来る．また，XML ビュー部分木はそれぞれ一つの XT-template ノードと結びついており，XML ビュー部分木の数も n_t となる．

XT-(sorted)set ノードは，子ノードを持つものと持たないものに分けられる．前者は配下に XML ビュー部分木を持っており n_t と関係がある．前者の平均的な子ノードの数を $\bar{a}$ とおくと，前者のノード数は $n_t/\bar{a}$ で表すことが出来る．一方，後者は変換結果に現れないため n_v とは無関係である．後者のノード数を b とおくと，n_s は $n_s = n_v/(\bar{a}\,k) + b$ と表せる．

XT 木の全ノード数 n_{VT} は n_g と n_v の和であり、$n_{\mathrm{VT}} = n_v(1/k + 1/(\bar{a}k) + 1) + b$ と表わすことが出来る．

ここで k，$\bar{a}$ はそれぞれ 1 以上であるから，n_v の係数は 3 以下となる．一方 b は，一致する XML ノードがソースデータに現れない apply-templates 命令を XSLT プログラムで記述すればそのぶん値が増えるので n_{VT} の値に限度はない．しかし，子を持たない XT ノードは XT 木中のリーフにしか存在しないことから，一般の XSLT プログラムであれば，XT 木の全ノード数に対する b の割合はあまり大きなものとはならないと考えられる．

よって，空間コストは変換結果 XML ビューと同程度から数倍程度となり，さほど大きくはないといえる．

4.2 差分更新の計算コスト

差分更新の計算コストは，XT ノード検索コストと XT ノー

ド更新コストの和である．それぞれについて詳細に述べる．

1）XT ノード検索コスト

更新対象の XT ノードは，変更された XML ノードを変換対象とする XT ノードである．更新対象パスと一致する $xpath_{\mathrm{XT}}$ を持つ XT ノードが更新対象となる．このとき，更新対象パスの段数と同じだけの比較を行う必要がある．

更新対象の XT ノードがひとつしかない場合の検索コストを c_l とし，XML ソースデータの変更に応じて変換結果 XML ビュー中の更新される箇所数を m とおくと，更新対象の XT ノードを全て抽出するコストは，最大で $m \times c_l$ となる．ただし最大となるのは，各更新対象 XT ノードまでの $xpath_{\mathrm{XT}}$ がそれぞれ全く異なる場合のみである．

2）XT ノード更新コスト

XML ビューにおいて更新が必要な箇所それぞれについて，再度変換を行なうため，m に依存した更新コストがかかる．

1），2）より，差分更新の時間コストは変換結果 XML ビュー中の更新箇所数 m に依存する．そのような更新箇所は，XML データ一箇所の変更に対しては一般にそれほど多くなく，本手法の対象として想定した XML データの一部のみが更新されるような用途においては十分適用可能である．

5. 評価実験

提案手法の性能を確認するためプロトタイプを作成した．

まず，XT 木生成時間と Saxon による変換時間とを測定した．この測定は，XT 木生成処理において全変換と同時に行なっている変換過程記録処理のオーバーヘッドが著しく大きくはないことを確認するために行なった．

また，差分更新による効果を確認するために，XT 木生成時間と，XT 木更新時間を測定した．

表 3　実験環境

CPU	Intel Pentium4 2.53GHz
RAM	2048MB
OS	Windows 2000
Java	Sun Java SDK 1.3.1_07
XML Parser	Xerces-J 2.3.0
XSLT Processor	Saxon 7.8
XPath Processor	Jaxen 1.0

5.1 実験条件

表 4　XML ソースデータ条件

サイズ（KB）	項目数
525	1,156
2,267	4,011
22,670	40,110
67,389	143,378

表 5　XSLT 変換プログラムの種類

XSLT-1	XML ソースデータの構造を保存するもの
XSLT-2	XML ソースデータをソートするもの

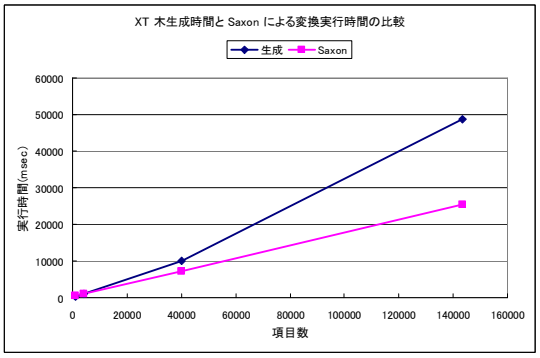

図 7　XT 木生成時間と Saxon による変換実行時間との比較

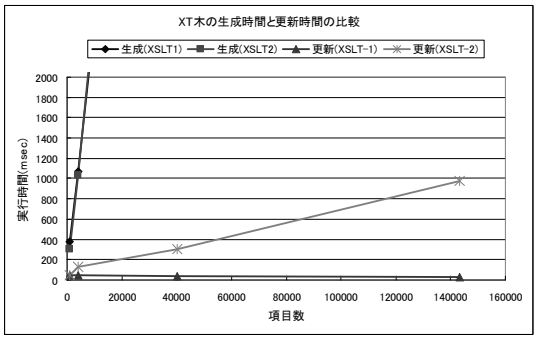

図 8　XT 木生成時間と XT 木更新時間の比較

実験環境，XML ソースデータのサイズ，使用した XSLT 変換プログラムの種類を示す（表 3，表 4，表 5）．XML ソースデータとしては DBLP [3] のサブセットを使用した．また、表 4 の “項目数” は “/dblp/*” の件数を表す．

XML データ変更命令は XML ソースデータに 1 項目分の XML 部分木を追加するものとした．

5.2 実験結果

XT 木生成時間と，Saxon で変換を行った時間を比較したグラフを示す（図 7）．変換プログラムとしては XSLT-1 を使用した．グラフの x 軸は XML ソースデータの項目数，y 軸は実行時間（ミリ秒単位）であり，項目数を変えたときの性能の変化を表している．

Saxon は項目数にほぼ比例した実行時間となっている．XT 木生成では項目数が大きくなるとグラフの傾きが大きくなっているが，項目数 140000 件時でも Saxon の約二倍の実行時間である．

XT 木生成時間と，XT 木更新時間を比較したグラフを示す（図 8）．XT 木更新時間としては XML データ変更命令を 200 回実行するのにかかる時間を測定した．グラフの x 軸は XML ソースデータの項目数を，y 軸は実行時間（ミリ秒単位）を表す．

XT 木生成時間は XSLT-1 と XSLT-2 とでは後者のほうが大きいが，グラフでは重なり合っている．また，XT 木生成時間よりも XT 木更新時間のほうがあきらかに短い．生成処理に対する更新処理の速度は，XSLT-1 で 7〜1500 倍、XSLT-2 で 6〜55 倍速い結果となった．

XT 木生成時間が XML ソースデータの項目数に応じて増加

しているのに対し，XSLT-1 による XT 木更新では XML ソースデータの項目数に関わらずほぼ一定の処理時間となっており，処理時間が更新サイズによって決まることがわかる．

XSLT-2 による XT 木更新では，XML ソースデータの項目数が大きくなるほど処理時間が長くなっている．XSLT-2 はソート条件が指定されているので，新規に追加された XML データに対して生成される XT ノードを XT 木に追加する際はソート順を保つような位置に追加する必要がある．追加位置を探すためにはバイナリサーチを使用しているが，サーチ時間はソート対象の XT ノード数が大きくなるほど長くなる．ソート対象の XT ノード数はソースデータの項目数に比例するため，このようにソースデータの項目数に応じて更新時間が長くなると考えられる．

6. 関 連 研 究

これまでの主な関連研究として，XSLT ビューの漸進的更新手法 [9]，半構造データベースの実体化ビューの漸進的更新手法 [1], [8]，そして XQuery の実体化ビューの漸進的更新手法 [4] とが挙げられる．

文献 [9] で提案される手法 incXSLT は，XSL テンプレートの動的呼び出し関係 execution flow を実体化し，それをトップダウンにトラバースして漸進的更新を行う．incXSLT では xsl:if をサポートしないが execution flow は XT 木とほぼ等価である．しかし，大きく我々の方法と比較して 3 つの相違がある．1) GUI からの XML 文書の更新を想定しているため，部分木ではなくノード単位での挿入/削除しか実現していない．2) ノードセットを返却する XPath 式には descendant 軸を使用できない．3) XPath 式の評価を差分実行をしないため，例えば DBLP データへの book 要素の追加は高価になってしまう．

文献 [1] は，Lorel を拡張した問い合わせ言語を用いてビューを定義し，その実体化結果を漸進的に更新するための更新操作を生成する手法を提案している．提案手法では，実体化結果を生成する際に参照した全てのオブジェクトの OID を保持し（RelevantOids と呼ぶ），ソースデータが更新された際に更新されたオブジェクトの OID が RelevantOids に含まれるか否かを判定することで，変更影響が実体化ビューに影響するか否かを判定する．Lorel により定義されるビューはソースデータの構造を保存する変換に限られるため，ソースデータに対する更新操作を比較的容易に実体化ビューへの更新操作へと変換することが可能である．これとは異なり XSLT により定義されるビューでは，ソースデータの構造を変更する変換が記述可能であるため，XSLT プログラムの解釈履歴を保存しそれを漸進的に更新することが必要である．また，Lorel は OQL の拡張言語であるため，XPath の child 軸に相当するトラバース処理しかサポートしない．WHERE 句による選択演算を XPath の述語処理に相当すると考えると，可能である XPath 式のクラスは $XP^{\{\}}$ に制限されている．

文献 [8] は，web 上の半構造データを対象に実体化されたビューを漸進的に更新するための更新操作を生成することを目的とし，複数のサイトへのアクセス数を軽減するビュー定義の分割手法を提案している．提案手法によるビューの表現能力は $XP^{\{[],*,//\}}$ クラスの 1 つの XPath 式と等価であるため，文献 [1] の手法と同様に定義されるビューがソースデータの構造を保存する変換に限らており，また XSLT のように階層的に XPath 式が適用される例には適用できない．

文献 [4] は，代数的手法を用いて XQuery により定義・実体化されたビューを漸進的に更新する手法について述べている．提案手法は，実体化した問い合わせ結果に対し，本論文の手法と同様にトップダウンに XPath 式の再評価することで変更影響箇所を判定し，XQuery を部分再評価する．しかし提案手法は，Lorel 同様に $XP^{\{[]\}}$ 相当の XPath クラスに制限されているため，ノードテストのワイルドカードと decendant 軸の処理ができない．また WHERE 句については常に再評価するため非効率である．更に XSLT は XQuery と異なり，XPath 式の呼び出し関係を静的に決定することができないため，この提案手法をそのまま XSLT に適用はできない．

7. ま と め

XML データの変換は XML データのサイズが大きい場合や，変換が複雑である場合，特にコストが高くなる．本稿では，あらかじめ XML データの変換を行なっておき，XML データに変更が生じた際，影響を受ける部位のみを変換することで，コストを削減する手法について検討した．

評価実験により，XT 木生成時間は一般的な XSLT 処理系と比べてそれほど遅くはならないこと，XT 木更新を XT 木生成よりはるかに短い時間で行なうことができることなど，提案手法の有効性を確認した．

今後の課題として，更新規模に応じた差分変換と全変換の切替や，サポートする XSLT 命令の拡充などが挙げられる．

文　　献

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 38–49, 24–27 1998.

[2] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[3] DBLP Computer Science Bibliography. http://dblp.uni-trier.de/.

[4] M. El-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An algebraic approach for incremental maintenance of materialized XQuery views. In *Computer Science Technical report Series*. Worcester Polytechnic Institute, 2003.

[5] FlightArrivals.com. http://flightarrivals.com/.

[6] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT*, pages 173–189, 2003.

[7] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of PODS*, pages 65–76, 2002.

[8] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *Proceedings of VLDB*, pages 227–238, 1996.

[9] L. Villard and N. Layaida. An incremental XSLT transformation processor for XML document manipulation. In *Proceedings of WWW*, pages 474–485, 2002.