XMLDBにおけるデータ更新を考慮したモデル写像アプローチ

佐村　敏治† 　佐藤　隆士††

† 明石工業高等専門学校電気情報工学科
†† 大阪教育大学情報処理センター
E-mail: †samura@akashi.ac.jp, ††sato@cc.osaka-kyoiku.ac.jp

あらまし　XML はデータ記述およびデータ交換の新たな標準として利用されている．XML をデータベースに格納するのに多くの研究者は関係データベースを用いることを提案している．関係データベースへの写像アプローチが様々ある中で, モデル写像アプローチは, XML スキーマの必要がなく, データベーススキーマはどんな XML でも同じなので, 頻繁に変更の生じる XML には扱いが容易である．しかしこのアプローチは XML ラベリング手法を除き更新操作に対してあまり議論されていない．本研究では更新を考慮したモデル写像アプローチを提案する．XML データの構造の変化に対して更新を最小にするデータベーススキーマを定義し，他のモデル写像アプローチと同レベルの検索ができるように設計した．最後に実験結果についても説明する．
キーワード　XML, 関係データベース, 半構造データ, XML データベース

A model-mapping approach considering data update on XMLDB

Toshiharu SAMURA† and Takashi SATO††

† Electrical and Computer Engineering, Akashi National College of Technology
†† Information Processing Center, Osaka Kyoiku University
E-mail: †samura@akashi.ac.jp, ††sato@cc.osaka-kyoiku.ac.jp

Abstract　XML is emerging as a new major standard for description and exchange of data. More researches have proposed using relational database storing XML. In the various mapping approaches for storing XML, the model-mapping approaches do not require XML schema so that fixed database schema can be used. However, these approaches are not so discussed for update operation excluding XML labeling methods. In this paper, we present a model-mapping approach considering update operation. This method is to define the database schema to reduce the change to the update operations as much as possible, and is designed to retrieve the same level as other model-mapping approaches. The experimental results are also presented.
Key words　XML, Relational Database, Semi-Structured Data, XML Database

1. は じ め に

XML(Extensible Markup Language) は，データ記述およびデータ交換の基盤技術として登場した．XML を利用した応用技術の増加に伴い，XML データも増加するため，それらを効率的に格納・検索するシステムとしてデータベースが必要となる．XML を格納したり運用したりするデータベースとして多くの研究者は関係データベースを提案している．

XML を関係データベースに格納するモデルとして，構造写像モデルとモデル写像モデルに大きく分類される [1]．構造写像アプローチは XML スキーマに基づき，データベーススキーマを設計するために構造情報を利用する [2], [3], [4]．ただしこれらのモデルは，XML の構造が変化するとデータベーススキーマも変化しなければならないので，頻繁に構造が変わる XML データには適さない．一方，モデル写像アプローチは XML の構造が変化してもデータベーススキーマは変化しないように設計されている．よって XML を柔軟に扱うことができ，関係データベースの管理面においても優位である [1], [5], [6], [7], [8], [9].

ただどちらのモデルにしても更新 (ノードの追加，削除，移動) の操作についてはあまり議論されていない．ほとんどがラベリングの手法に関してである [10], [11], [12], [13].

本研究では XML を関係データベースで管理する場合のデータ更新を考慮したモデル写像アプローチを提案する．XML の構造の変化に対してデータベーススキーマの更新を最小になるようにモデルを構築する．特徴として，更新に強いデータベーススキーマを実現するために双方向リストの概念を利用す

る．また従来のモデルにおいて更新時に大きなコストとなっていた絶対ロケーションパスを関係表として持たずに，マテリアライズドビューを生成することで，データ更新における複雑なパスの変更を避けている．これらの特徴を生かすことにより，XUpdate [14] 等の更新言語から SQL への変換が従来のモデルより容易になることが期待される．そして更新についての頑健性を調べるために，Michigan Benchmark [15] を用いて性能評価を行った．

また検索についても他のモデル写像アプローチと同レベルの検索できることを要求する．本稿では XML データベースの検索でしばしば用いられる Bosak Shakespeare コレクション [16] を用いて実験を行い，モデルの有効性を示した．

本稿の構成は次のとおりである．まず 2 節では，本研究の関連研究について述べる．3 節において，更新を議論する場合の問題点と，我々の提案するモデルについて説明する．4 節で提案したモデルが更新過程から SQL にどのように変換するかについて議論する．5 節で更新と検索についての評価実験の結果を説明し，6 節でまとめと今後の課題について述べる．

2. 関 連 研 究

関係データベースを用いた XML データベースとして主要なアーキテクチャを図 1 に示す．XML から関係データベースへの格納部 (左) と，XQuery 等での問い合わせに対して SQL 言語に変換して結果を出す検索部 (右)，XUpdate 等の更新操作に対応する更新部 (下) の処理体系からなる．

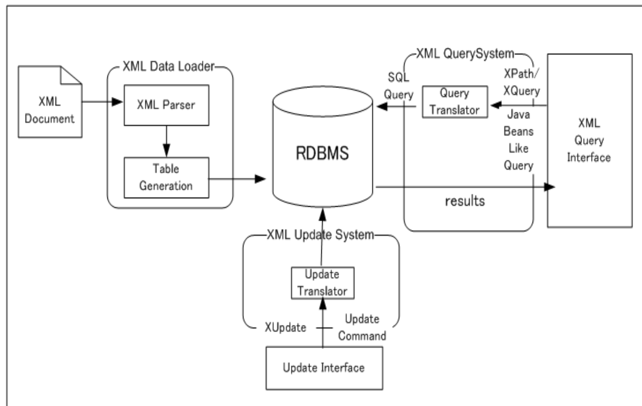

図 1　アーキテクチャ

モデル写像アプローチでデータベーススキーマの顕著なモデルとして，Ege [5]，XRel [1]，XParent [6]，Saxophone [8], IN-ode [7], 油井と森嶋の方法 [9] 等がある．本稿では XParent [6] を例に説明する．

XParent は 4 つのテーブルから構成される．

$Path(PathID, Len, PathExp)$

$DataPath(Pid, Cid)$

$Element(PathID, Ordinal, Did)$

$Data(PathID, Did, Ordinal, Value)$

図 2 の XML データ [17] を例に説明する．図 3 はこのデータを木構造で表したものである．図中の数字については次節で説明する．XParent によりテーブルに格納した結果を表 1 から

```
<addresses>
  <address id="1" date="2005/12/20">
    <name>
      <first>John</first>
      <last>Smith</last>
    </name>
    <city>Houston</city>
    <state>Texas</state>
    <country>United States</country>
    <phone type="home">333-300-0300</phone>
    <phone type="work">333-500-9080</phone>
    </address>
</addresses>
```

図 2　XML　例

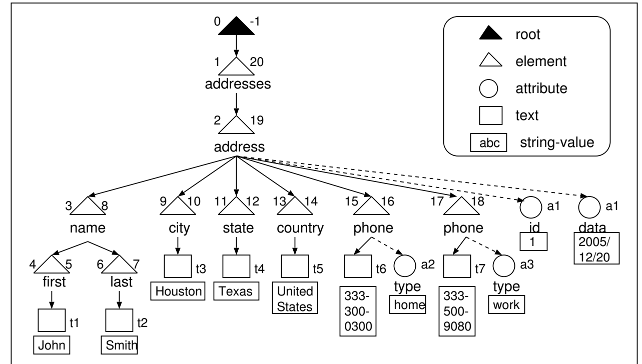

図 3　図 2 の構造を表す木構造

表 1　XParent の Path テーブル

PathID	Len	PathExp
1	1	/addresses
2	2	/addresses/address
3	3	/addresses/address/@id
4	3	/addresses/address/@date
5	3	/addresses/address/name
6	4	/addresses/address/name/first
7	4	/addresses/address/name/last
8	3	/addresses/address/city
9	3	/addresses/address/state
10	3	/addresses/address/country
11	3	/addresses/address/phone
12	4	/addresses/address/phone/@type

表 3 に示す．

Path テーブル (表 1) は，PathExp で絶対ロケーションパスを示し，対応する PathID で他のテーブルと結合する．Path テーブルは XPath による高速検索に用いられる．Len は階層の深さを表す．Element テーブル (表 2 左) は，要素が出現した時，Did の番号で割り当てられる．また同じ絶対ロケーションパス (PathID) を持つ兄弟ノードの順序を Ordinal 属性で表す．DataPath テーブル (表 2 右) は，Element テーブル Did の親子関係を表す．Pid が親の Did を表し，Cid がその子要素の Did を表す．このテーブルを用いて先祖子孫関係の問い合わせを行う．検索を高速に行うため，Ancestor テーブルと呼ばれる全ての先祖子孫関係を格納したテーブルを使うことも提案している．Data テーブル (表 3) は，テキストノード及び属性値

表 2　XParent の Element テーブル (左) と DataPath テーブル (右)

PathID	Ordinal	Did
1	1	1
2	1	2
5	1	3
6	1	4
7	1	5
8	1	6
9	1	7
10	1	8
11	1	9
11	2	10

Pid	Cid
0	1
1	2
2	3
3	4
3	5
2	6
2	7
2	8
2	9
2	10

表 3　XParent の Data テーブル

PathID	Did	Ordinal	Value
3	2	1	1
4	2	1	2005/12/20
6	4	1	John
7	5	1	Smith
8	6	1	Houston
9	7	1	Texas
10	8	1	United States
12	9	1	home
11	9	1	333-300-0300
12	10	1	work
11	10	1	333-500-9080

を Value に格納する．

更新に関しては ID などのラベリング手法に対して多くの研究がされている．これらは先祖子孫関係の検索も含めて議論されて，予めノード間にスペースを空ける方法 [12] や挿入するノードを浮動小数点で表現する方法 [13], VLEI コードと呼ばれる挿入制限のない値をノードにつける方法 [11] などのラベリング手法が提案されている．

3. 更新を考慮したモデル写像アプローチ

3.1 更新操作における考察

写像アプローチを XML データベースとして構築するために更新操作は不可欠である．更新にはノードの挿入，削除，移動の 3 種類の操作が挙げられる．

挿入に対して XParent を例にとると次の変更が必要になる．

（1） 兄弟ノードの順序を示す Ordinal 属性の再構成

（2） 親子関係を示す DataPath 属性の再構成

（3） 絶対ロケーションパスに関する Path テーブルの変更

図 4 のようなノードの挿入では Path テーブルは一行の追加ですむが，図 5 では子孫ノードに対しても変更が必要となる．従って XParent では汎用的な更新操作を記述することは困難であることが分かる．

また削除は，ノードを検索しノードを削除する操作を合わせたものとなる．多くのコストがノード検索に占められることから，削除は検索の性能に依存すると考えられる [18]．移動は挿入と削除の組み合わせになる．

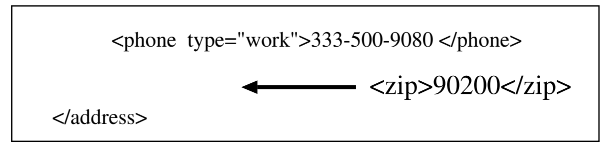

図 4　ノードの挿入 1

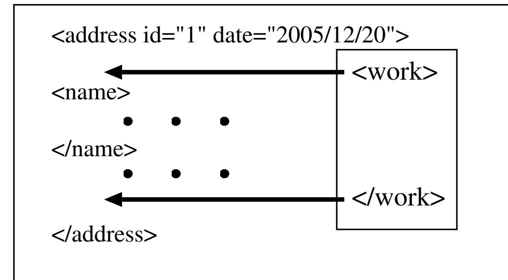

図 5　ノードの挿入 2

3.2 データベーススキーマとその特徴

本研究の目的は，更新に対してできるだけ変更が少なく，他のモデル写像アプローチと同等な検索ができるようなデータベーススキーマを設計することである．

この目的を実現するため，我々は次の 3 つのテーブルと 1 つの関数，そして 1 つのマテリアライズドビュー (materialized view) で構成する新しいモデル写像アプローチを提案する．

Table

$Element(seq, name, sID, snID, eID, enID, aID, tID, pID)$

$Text(tID, Value)$

$Attribute(name, aID, Value)$

Function

function $absPath(sid\text{: } integer) : text;$

Materialized View

$PathMV(sid, path)$

このときの 3 つのテーブル属性とその説明を表 4 に示す．また図 2 のデータを格納したテーブルを表 5 , 6 に示す．

表 4　テーブル属性の説明

テーブル名	属性	説明
Element テーブル	seq(sequence)	要素の出現順序
	name	要素名
	sID(start element ID)	開始要素 ID
	snID	開始要素の次 ID
	eID(end ID)	終了要素 ID
	enID	終了要素の次 ID
	aID(attribute ID)	属性 ID
	tID(text ID)	テキストノード ID
	pID(parent ID)	親要素の sID
Text テーブル	tID(text ID)	テキストノード ID
	Value	テキストノード値
Attribute テーブル	name	属性名
	aID(attribute ID)	属性 ID
	Value	属性値

表 5　OurModel の Element テーブル

seq	name	sID	snID	eID	enID	aID	tID	pID
10	addresses	1	2	20	-1	0	0	0
20	address	2	3	19	20	1	0	1
30	name	3	4	8	9	0	0	2
40	first	4	5	5	6	0	1	3
60	last	6	7	7	8	0	2	3
90	city	9	10	10	11	0	3	2
110	state	11	12	12	13	0	4	2
130	country	13	14	14	15	0	5	2
150	phone	15	16	16	17	2	6	2
170	phone	17	18	18	19	3	7	2

表 6　OurModel の Text テーブル (左) と Attribute テーブル (右)

tID	Value
1	John
2	Smith
3	Houston
4	Texas
5	United States
6	333-300-0300
7	333-500-9080

name	aID	Value
id	1	1
date	1	2005/12/20
type	2	home
type	3	work

このスキーマの特徴について述べる．

（1）　更新に強いデータベーススキーマを実現するため，双方向連結リストの概念を用いる (図 6)．

element(要素) テーブルの属性として親要素 ID へのポインタと次要素へのポインタを導入する．このことにより要素の追加に対しては，前後要素の親要素および次要素 ID にのみ注目すればよい．親要素 ID へのポインタは親子関係の検索のために使用する．次要素 ID へのポインタは，高速に XML を構成するため [9] と，更新を行うときに自身の ID を変更しなくてよいために使用する．格納時は，自身の ID を出現順序通りに割り振っているが，それを維持する必要はないく，実際に次節で述べるように要素の挿入を考えると，ID の順序は前後する．

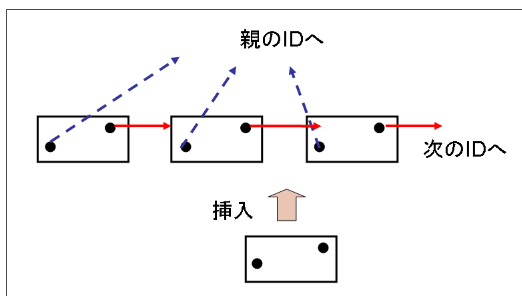

図 6　双方向連結リストの概念を用いたモデル

（2）　開始要素 ID と終了要素 ID を導入する．

要素挿入が 図 4 のタイプか図 5 のタイプかを区別するには開始要素 ID と終了要素 ID との両方を導入する必要がある．

本稿ではルート要素は pID= 0，enID= −1 に割り当てている．属性及びテキストが存在しない場合は aID= 0，pID= 0 とする．図 3 で，要素記号の右の数字が開始要素 ID(sID) を表し，左の数字が終了要素 ID(eID) を表す．混合内容では eID= 0 と enID= 0 になる．

（3）　ノードの種類ごとにそれぞれテーブルを用意する．

ノードの種類ごとにそれぞれテーブルを持つことで XML 構造が明確になる．例えば属性ノードやテキストノードがどの要素にぶら下がっているかがすぐ分かるようになる．本稿では，element テーブルに属性 ID とテーブル ID へのポイントするためのテーブル属性 (aID，tID) を導入した．XML の属性は要素内であれば，順序を持たないので，element テーブルに aID 属性を 1 つだけ持てばよい．

（4）　element テーブルに seq 属性を導入する．

seq 属性は，XParent における Ordinal 属性を廃止したことにより導入された属性である．3.1 節で述べたように Ordinal 属性は更新により再番号づけが必要になってくるため廃止した．しかし，これを廃止すると”/PLAY/ACT[2]” のようなノードの順番を表す指定ができなくなる．そこで seq 属性の順番でソートする．

本稿では要素間にスペースを空ける最も簡単な方法を採用した．ただし，この属性の実装には様々な方法が考えられる．例えば現在研究されているラベリング手法を適応すると先祖子孫関係の検索も高速にできることが期待される．

（5）　絶対ロケーションパスを関係表として持たずに，マテリアライズドビューとして生成する．

我々のモデルの大きな特徴は Path テーブルを持たないことである．3.1 節で説明したように Path テーブルは更新操作では変更が面倒な作業となるからである．しかし XPath による検索では絶対ロケーションパスの利用は非常に便利である．そこで我々は Element テーブルから絶対ロケーションパスを生成する関数を定義する．これにより，要素の挿入により絶対ロケーションパスのどこを変更すれば良いかを考える必要がなくなり，ただ関数を呼び出せば動的に変更される．*absPath* のアルゴリズムを図 7 に示す．ただし SQL 文については PostgreSQL における PL/pgSQL 表現に従う．

```
function absPath(sid: integer): text;
var id: integer;
    path: text;
    data: record;
begin
  id := sid; path :="";
  repeat
    SELECT INTO data *
    FROM element
    WHERE sid = id;
    path :=" /" || data.name || path;
    id := data.pid;
  until id = 0;
  return path
end;
```

図 7　絶対ロケーションパス生成アルゴリズム：*absPath*

この関数を使って生成されるマテリアライズドビュー PathMV を表 7 に示す．PathMV の Path 属性は，関数 $absPath(sid)$ を呼び出して格納する．

表 7　OurModel における PathMV マテリアライズドビュー

sID	path
1	/addresses
2	/addresses/address
3	/addresses/address/name
4	/addresses/address/name/first
6	/addresses/address/name/last
9	/addresses/address/city
11	/addresses/address/state
13	/addresses/address/country
15	/addresses/address/phone
17	/addresses/address/phone

PathMV は更新が生じるときに作りなおすようにする．これにより図 4 のような挿入でも，図 5 のような挿入でも意識することなく絶対ロケーションパスの更新を行うことができる．

この操作は XParent の Path テーブルと比較してあまり違いがないように思われる．しかし次節で述べるように XParent による Path テーブルの更新作業は非常に面倒で，アルゴリズムが複雑になることが分かる．我々のモデルでは，テーブルの再構成を一つの操作手順で行うことができる．また Path テーブルを再構成させる仕組みが XParent には備わっていない．

しかし PathMV を全て作りなおそうとすると時間的コストは高い．マテリアライズドビューの作りなおしをどのタイミングで行うか，また全体で行うか，更新された部分について適用すべきかについては別の機会に議論したい．

4. 更新過程

本節では，提案したデータベーススキーマを基に更新過程がどのように SQL に変換されるかについて議論する．以下，図 2 の XML データを例として考える．まず，ノードの挿入に対して新しい ID を取得するため図 8 の関数を定義する．

```
CREATE FUNCTION max_eID() RETURNS INTEGER
   AS' SELECT max(eID) from Element ' LANGUAGE
'SQL';
CREATE FUNCTION max_tID() RETURNS INTEGER
   AS' SELECT max(tID) from Element ' LANGUAGE
'SQL';
CREATE FUNCTION max_aID() RETURNS INTEGER
   AS' SELECT max(aID) from Element ' LANGUAGE
'SQL';
```

図 8　ID 決定のための関数定義

Example1　address 要素の id 属性が "1" のとき，address 要素の子要素集合の最後に zip 要素を挿入する (図 4).

```
– Define temporary view and table for results of query
CREATE view ex03_temp01(seq) as
  SELECT max(e0.seq)
  FROM Element e0, Element e1, PathMV p1, Attribute a1
  WHERE e1.sID = p1.sID
    AND e1.aID = a1.aID
    AND p1.path ='/addresses/address'
    AND a1.name ='id'
    AND a1.value ='1'
    AND e0.pID = e1.sID;
CREATE TABLE ex03_temp02(seq integer, sID inte-
ger,prevenID
    integer, tID integer, pID integer);
  INSERT INTO ex03_temp02
    SELECT (e0.seq+1), (max_eID()+1), e0.enID,
           (max_tID()+1), e0.pID
    FROM Element e0
    WHERE e0.seq = (SELECT seq FROM ex03_temp01);

– Update process for Element table
UPDATE Element SET enID =
    (SELECT sID FROM ex03_temp02)
  WHERE enID = (SELECT prevenID FROM ex03_temp02);

– Insert process for Element table
INSERT INTO Element (seq, name, sID, snID,
    eID, enID, aID, tID, pID)
  SELECT seq, 'zip', sID, (sID+1), (sID+1), prevenID,
    0, tID, pID
    FROM ex03_temp02;

– Insert process for Text table
INSERT INTO text (tID, text)
  SELECT tID, '90200' FROM ex03_temp02;

– Drop temporary view and table
DROP table ex03_temp02;
DROP VIEW ex03_temp01;
```

図 9　**Example1** における挿入過程の SQL 文例

結果例を図 9 に示す．大部分をノード検索による処理が占める．本稿ではデータを一時的に格納するため，一時的なビューとテーブルを定義して用いた．そして参照しているノードは，挿入したい要素の前の要素と親の要素だけであり，変更しているノードもそれだけである．更新操作のみに注目すると，1 つのテーブル更新作業と 2 つのテーブル挿入の命令しか必要ない．図 4 のタイプの挿入操作はほとんどがこの結果と同様に記述すればよい．ただし本モデルでは最後にマテリアライズドビューの作りなおしを行う必要がある．

XParent で同様の挿入を行うとどれくらいの変更・追加箇所があるか示す．

（1） Path テーブルに新しいロケーションパスの追加

（2） Element テーブルの追加，Did 属性の番号再構成 (順番通りに並ぶ必要がある)，Ordinal 属性の確認と再構成する必要があれば実施

（3） DataPath テーブルの追加

（4） Elemnet テーブルの Did に対応する Data テーブルの追加

ノードの更新場所により，変更・追加条件が変わるので汎用的なアルゴリズムを示すのは困難である．

Example2 address 要素の id 属性が "1" のときの address 要素の子要素として work 要素 (仕事用) を挿入する (図 5).

```
– Define temporary table for results of query
CREATE TABLE ex02_temp(seq integer, sID integer,
  prevsID integer,snID integer, eID integer, prevenID integer);
INSERT INTO ex02_temp
  SELECT (e0.seq+1), (max_eID()+1),
    e0.sID, e0.snID, (max_eID()+2),e0.eID
    FROM Element e0, pathmv p0, attribute a0
    WHERE e0.sID = p0.sID
    AND e0.aID = a0.aID
    AND p0.path ='/addresses/address'
    AND a0.name ='id'
    AND a0.value ='1';

–Insert process
UPDATE element SET snID =
    (SELECT sID FROM ex02_temp)
  WHERE sID = (SELECT prevsID FROM ex02_temp);
UPDATE element SET enID =
    (SELECT eID FROM ex02_temp)
WHERE enID = (SELECT prevenID FROM ex02_temp);
UPDATE element SET pID =
    (SELECT sID FROM ex02_temp)
  WHERE pID = (SELECT prevsID FROM ex02_temp);
INSERT INTO element (seq, name, sID, snID, eID, enID,
    aID, tID, pID)
  SELECT seq, 'work', sID, snID, eID, prevenID, 0, 0, prevsID
  FROM ex02_temp;

– Drop temporary table
DROP TABLE ex02_temp;
```

図 10 **Example2** における挿入過程の SQL 文例

更新結果の例を図 10 に示す．また更新後の Element テーブルを表 8 に示す．"seq=21" が新しく挿入された行である．要素の挿入により，挿入の前の要素と親の要素以外の情報しか必要ないことが分かる．

XParent で同様の挿入操作を行うとする．その子孫要素は深

表 8 更新後の Element テーブル

seq	name	sID	snID	eID	enID	aID	tID	pID
10	addresses	1	2	20	-1	0	0	0
20	address	2	21	19	20	1	0	1
21	work	21	3	22	19	0	0	2
30	name	3	4	8	9	0	0	21
40	first	4	5	5	6	0	1	3
60	last	6	7	7	8	0	2	3
90	city	9	10	10	11	0	3	21
110	state	11	12	12	13	0	4	21
130	country	13	14	14	15	0	5	21
150	phone	15	16	16	17	2	6	21
170	phone	17	18	18	22	3	7	21

さが 1 つ増えるが，他の兄弟要素も深さが 1 つ増えるかどうかは設計者に依存する．そしてモデル写像アプローチでこの点について指摘している研究はほとんどない．

（1） Path テーブルに新しいロケーションパスの追加と再構成．子孫要素にも及ぶ

（2） Element テーブルの追加，PathID の再構成 (子孫要素にも及ぶ)，Did 属性の番号再構成 (順番通りに並ぶ必要がある)，Ordinal 属性の確認と再構成する必要があれば実施

（3） DataPath テーブルの追加 (子孫要素にも及ぶ)

（4） Elemnet テーブルの Did に対応する Data テーブルの追加と Path テーブルで変更した PathID 属性の再構成 (子孫要素にも及ぶ)

以上の複雑な行うための汎用的なアルゴリズムを考えることは非常に困難で，XParent で更新を扱うのは不向きであることが分かる．

最後に削除について例を示す．

Example3 name 要素を削除し，全ての子孫要素を親要素に結合させる．

結果例を図 11 に示す．削除要素の前要素と子要素の親要素の変更だけでよいことがわかる．しかしこれも XParent のようなモデルを用いると，Path テーブルの子孫要素にまで関わるロケーションパスの変更を強いられる．また Element テーブルにおける変更も子孫要素にまで生じて複雑になる．

5. 実験による性能評価

本モデルの有効性を調べるため性能実験を行う．

実験環境は，Pentium4 1.7GHz の CPU，1024MB の RAM，70GB のハードディスクを用いる．OS は Vine Linux 3.2，使用した DBMS は PostgreSQL 7.4.8 である．時間計測には PostgreSQL の対話型インタフェース psql の"\ timing" を用いた．

PathMV のマテリアライズドビューは，PostgreSQL の機能として備わっていないので，実際の表を作成し，“Truncate” コマンドとトリガの組合せでマテリアライズドビューを実現した．また制御を要する関数定義には PL/pgSQL を用いている．

```
– Define temporary table for results of query
CREATE TABLE ex03_temp(sID integer,
    snID integer, eID integer, enID integer, pID integer);
INSERT INTO ex03_temp
  SELECT e0.sID, e0.snID, e0.eID, e0.enID, e0.pID
    FROM Element e0, pathmv p0
    WHERE e0.sID = p0.sID
    AND p0.path ='/addresses/address/name';

–Delete process
UPDATE element SET sID =
    (SELECT sID FROM ex03_temp)
  WHERE sID = (SELECT snID FROM ex03_temp);
UPDATE element SET sID =
    (SELECT eID FROM ex03_temp)
  WHERE sID = SELECT enID FROM ex03_temp);
UPDATE element SET pID =
    (SELECT pID FROM ex03_temp)
  WHERE pID = (SELECT sID FROM ex03_temp);
DELETE FROM element where sID =
    (SELECT sID FROM ex03_temp);

– Drop temporary table
DROP TABLE ex03_temp;
```

図 11 **Example3** における削除過程の SQL 文例

5.1 更新による性能実験

更新についての頑健性を調べるために，Michigan Benchmark [15] を用いて性能評価を行った．ds0.1x(ファイルサイズ 45MB，要素ノード数 67696) のデータを用いる．本ベンチマークから生成される XML データは，2 つの要素 (eNest:66655 nodes, eOccasional:1041 nodes) しかなく，主要となる eNest 要素には 6 つの属性 (aUnique1, aUnique2, aLevel, aFour, aSixteen, aSitxyFour) とテキストノードからなる．特徴として，ほとんどの要素を"eNest" 要素が占め，最大深さ 16 までのネスト構造である．深さについては属性の"aLevel" で表されている．

ベンチマークによる問い合わせとして，選択 (QS), 結合 (QJ), 集約 (QA)，更新 (QU) が用意されていて，本稿では更新問い合わせ (QU) について評価を行った．各問い合わせを SQL に変換して更新を行った．操作において注目した点について説明する．

QU1:Point Insert 図 5 の場合に相当する．“eNest” 要素は深さ 14 の箇所に挿入され，この要素の先祖子孫関係だけ深さ 17 に変更される．挿入過程は前節の Example2 と同様に扱える．選択するノードの情報を一時的なテーブルに格納し，処理が終了した時点で削除する

QU2:Point Delete 選択要素の子要素の親 ID(pID) を対応する親 ID に変更して，要素を削除する

QU3:Bulk Insert 選択された要素は図 4 と図 5 の両方存在するので，場合分けして挿入する

QU4:Bulk Delete 対応する要素 ID を親要素として持つ要素がなければ葉要素として処理する

QU5:Bulk Load，QU6:Bulk Reconstrunction 本モデルではファイルの読み込み (QU5) は Java 等の言語を通じて行うため省略する．また部分木ノードの読み込み (QU6) は次要素 ID を辿っていけばよいので，本モデルでは更新操作ではないことから省略する

QU7:Restructuring ノードの入れ換え操作であるが，親 ID(pID) の変更だけで更新できる

実験計測結果を表 9 に示す．時間的コストが発生するのは検索およびその結果を一時的な表やビューに格納する操作である．XParent において本ベンチマークで SQL 文を作成することは，テーブルに依存した形で生成することは可能であっても汎用的に生成することは非常に困難である．

表 9 Michigan Benchmark を用いた更新時間

Query	Query Description	Time(second)
QU1	Point Insert	0.63
QU2	Point Delete	0.44
QU3	Bulk Insert	10.68
QU4	Bulk Delete	2.03
QU7	Restructuring	2.01

表 9 の結果には PathMV のマテリアライズドビューの作りなおしの時間は考慮されていない．もし最初から作りなおしを行うと，平均 70.56 秒程度の時間がかかり，更新時間のほとんどを占めてしまうことが分かる．対策としてトリガ時に更新要素とその子孫ノードのみを作りなおすようにすれば短時間による更新も可能だと考えられる．

5.2 検索による性能実験

検索について他のモデルでも比較される Bosak Shakespeare コレクションを用いる [16](全ファイルサイズ：7.5MB)．検索項目として [1] による XPath による問い合わせを用い，SQL に変換した結果で時間を計測した (図 12)．

```
Q1  /PLAY/ACT
Q2  /PLAY/ACT/SCENE/LINE/STAGEDIR
Q3  //SCENE/TITLE
Q4  //ACT//TITLE
Q5  /PLAY/ACT[2]
Q6  (/PLAY/ACT)[2]/TITLE
Q7  /PLAY/ACT/SCENE/SPEECH[SPEAKER='CURIO']
Q8  /PLAY/ACT/SCENE[//SPEAKER='Steward']/TITLE
```

図 12 Bosak Shakespeare コレクションを用いた検索

本モデルと XParent とを比較した実験結果を図 13 に示す．ただし XParent は Ancestor テーブルを用いていない．単純な検索 (Q1，Q2) についてはそれほどの差は見られないが，複雑な検索については差が生じているものもある．全体的には検索における性能劣化はないものと判断できる．

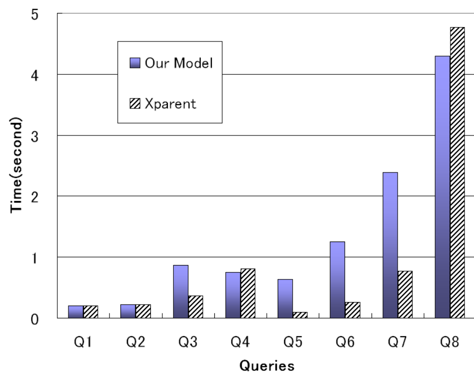

図 13　我々のモデルと XParent との Bosak Shakespeare コレクションを用いた検索時間

ただし，どうしても検索時間では，XParent に及ばない項目もある．例えば順序を決定する検索 (Q5，Q6) については明らかに XParent の方が優位である．これは XParent は Ordinal 属性を持ち，その数値を直接参照しているのに対し，本モデルでは，兄弟要素を並べて，seq 属性でソートさせてから順序を決定するのでコストはかかってしまう．

6. ま と め

本研究では，関係データベースを用いた XML データベースとして，新しいモデル写像アプローチを提案した．特にデータ構造が頻繁に変更される XML に対してデータベーススキーマの変更を少なくするように設計を行った．また検索についても性能劣化が生じないように，データベーススキーマを定義した．そして本モデルを用いると XUpdate などを用いて，要素の挿入操作や削除操作が可能になることを実例で示した．

特徴として挙げられるのは，更新に対して他のモデル写像アプローチでは必要な絶対ロケーションパスの関係表を廃止し，element テーブルの兄弟要素の順序 (”Ordinal”) 属性を廃止したことである．また，ノードの種類ごとにテーブルを作成しているので，XML の構造に近いモデルを実現している．そして，開始要素 ID と終了要素 ID を導入し，次要素 ID へポイントする属性を用意した．また seq 属性のソーティングにより”Ordinal” 属性の役割を補った．

また実験により本モデルの有効性を示した．ただし更新におけるマテリアライズドビュー作りなおしの議論や，いくつかの検索では他モデルより性能が悪い結果も得られ，今後の課題として残った．

本モデルはどんな XML データでも格納でき，更新に対して頑強なデータベースの構築が目的であるが，新たな機能を追加したり，非常に困難な検索を行ったりすると，現在のスキーマを変更する必要性が出てくるかもしれない．しかし今回議論した概念については引き継がれていくものと考える．今後様々なベンチマークテストおよび他のモデルとの比較を行い，本モデルの適用範囲についても議論していく必要がある．

今後，更新においては XUpdate などを例にとり実装を行っていきたい．また，seq 属性については現在議論されているラベリング手法などを使って seq 属性だけで先祖子孫関係を検索できるモデルを提案する価値があるように思われる [19]．

文　献

[1] M. Yoshikawa, T. Amagasa, T.Shimura, and S. Uemura. XRel: A Path-Based Approach to Strage and Retrival of XML Decuments Using Relational Databases. ACM Transsactions on Internet Technology, Vol. 1, No. 1, pp. 110-141, 2001.

[2] J. Dhanmugasundaram, K. Tufte, C. Zhang, H. Gang, D.J. DeWitt, and J.F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In Proceedings of the 25th International Conference on Very Large Data Bases, 1999.

[3] G. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschitzegger. X-Ray - Towards Integrating XML and Relational Database Systems. Proceedings of the 19th International Conference on Conceptual Modeling, Springer, pp. 339-353, 2000.

[4] D. Lee, W.W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. ER 2000. pp. 323-338, 2000.

[5] D. Florescu, and D. Kossmann. A perfomance evaluation of alternative mapping schemes for storing XML data in a relational database. INRIA, p.31, 1999.

[6] H.Jiang, H. Lu, W. Wang and J.X. Yu. Path materialization revisited: an efficient storage model for XML data, in Proceedings of the 13th Australasian Conference on Database Technologies, Australian Computer Society, Inc., pp.85-94, 2002.

[7] H.K. Lau, and V. Ng. INODE An Enumeration Scheme for Efficient Storage of XML Data. Cooperative Internet Computing, Kluwer Academic Publisher, pp.165-184, 2003.

[8] 横山 昌平, 太田 学, 片山 薫, 石川 博. XML パーサを考慮した応用向き関係データベース構成法, 第 13 回データ工学ワークショップ (DEWS2002), 2002.

[9] 油井 誠, 森嶋 厚行. PostreSQL を用いた多機能な XML データベース環境の構築. 情報処理学会論文誌, 情報処理学会, Vol.44, No.SIG12(TOD 19),pp. 11-22, 2003.

[10] P. F. Dietz. Maintaining order in a linked list. In Proceedings of the 14th annual ACM symposium on Theory of computing, pp. 122-127, 1982.

[11] 小林 一仁, 小林 大, 横田 治夫. 挿入制限のない XML 範囲ラベリング用コード. 信学技報, 電子情報通信学会, DE2003-13, pp.31-36, 2003.

[12] 江田 毅晴, 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. DBSJ Letters, No. 1 in Vol.1, 2002.

[13] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. 19th International Conference on Data Engineering (ICDE 2003), pp. 705-707, 2003.

[14] XML : DB Initiative. XUpdate -XML Update Language, 2006.http://xmldb-org.sourceforge.net/xupdate/

[15] The Michigan Benchmark, 2006. http://www.eecs.umich.edu/db/mbench/

[16] The Bosak Shakespeare collection, 2006. http://metalab.unc.edu/bosak/xml/eg/shaks200.zip

[17] K. Staken. XML:DB XUpdate Use Cases,2006. http://www.xmldatabases.org/projects/XUpdate-UseCases/

[18] L.H. Kit, V. Ng. Enumerating XML Data for Dynamic Updating. ADC 2005, pp. 75-84, 2005.

[19] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller. Nigel Westbury: ORDPATHs: Insert-Friendly XML Node Labels, SIGMOD Conference 2004, pp.903-908, 2004