RDB上のXMLビューに対するDOMプログラム実行の効率化手法

小島 章[†,☆] 森嶋 厚行[††] 古宮 誠一[†]

インターネットにおけるデータ交換フォーマットの標準としてのXMLの地位が確立され，XMLデータを入力とするアプリケーションが増加している．その一方，依然としてデータ管理の重要なツールとしてリレーショナルデータベース（RDB）が広く利用されている．本論文では，既存のRDBに格納されているデータに対してXMLビューを構築し，DOM APIを利用したアプリケーションプログラムを適用するための方式と，プログラム実行の効率化手法について述べる．また，実験の結果を通じて，提案方式が実行の効率化に有効であることを示す．

A Method for Efficient Execution of DOM Programs on XML Views over RDBs

Akira Kozima,[†,☆] Atsuyuki Morishima[††] and Seiichi Komiya[†]

As XML has been recognized as the de-facto standard for data interchange through the Internet, more and more application programs to process XML data are being developed. On the other hand, relational databases are still widely used as important tools for the data management. This paper presents a method that allows XML application programs using DOM APIs to take as input relational data through their XML views and explains an optimization technique. Our experimental results show that the proposed technique is effective for the efficient execution of programs.

1. はじめに

インターネットにおけるデータ交換フォーマットの標準としてXMLが認知され，XMLデータを入力とするアプリケーションが増加している．その一方，依然としてリレーショナルデータベース（RDB）がデータ管理のための重要なツールとして広く利用されており，多量のデータがRDBに格納されている．そのため，RDBに格納されたデータ（もしくはその一部）をXMLデータと見なして扱うための“**RDB上のXMLビュー**”の構築とその処理に関する研究が行われてきた[4),7)~9)]．

これらの研究では，RDB上のXMLビューを定義するために**ビュー問合せ**を利用する．XMLビューはこのビュー問合せを実行することにより**実体化**される．しかし，ビューの一般的な利用方法では，すべての部分が一度に実体化されるのではなく，アプリケーションの要求に応じて，必要な部分だけが実体化されることが多い．この点を強調するために，このようなビューは**仮想的**であると呼ばれる．

これまで，RDB上の仮想的なXMLビューに関する研究では，XQuery等の宣言的XML問合せがXMLビューに対して行われた場合を想定し，要求された部分だけを実体化するために必要なSQL問合せを構築する手法の開発が行われてきた．

本論文では，それらと異なり，DOM APIを用いてXMLデータにアクセスするアプリケーションプログラム（以下，DOMプログラム）が，RDB上の仮想的なXMLビューを扱うための機構，および実行の効率化について提案する．DOM APIを用いれば，ソフトウェア開発者はプログラミング言語におけるオブジェクトを扱うのと同様の手続き的な方法でXMLデータを扱うことが可能であるため，DOM APIは広く利用されている．したがって，本研究で取り組む課題は重要であると考えられる．

† 芝浦工業大学大学院工学研究科
Graduate School of Engineering, Shibaura Institute of Technology
†† 筑波大学大学院図書館情報メディア研究科/知的コミュニティ基盤研究センター
Graduate School of Library, Information and Media Studies/Research Center for Knowledge Communities, University of Tsukuba
☆ 現在，NECソフト株式会社
Presently with NEC Soft, Ltd.

RDB 上の仮想的な XML ビューに対する DOM 操作は，XML ビューをすべて実体化し，通常の DOM ライブラリを用いて操作を行う場合と比べて次のような利点がある．第 1 に，必要な時点で必要なデータだけが計算されるため，巨大データへの対応が容易であることである．RDB 上の XML ビューをすべて実体化するとサイズが巨大になる可能性があるため，これは望ましい性質である．一方，通常の DOM ライブラリはすべてのデータを一度にメモリ上に展開するため，大きな記憶領域を必要とする．第 2 に，必要なときに最新のデータが計算されるため，対象となる RDB の更新頻度が高い場合でも，最新のデータベースの内容への対応が容易なことである．

本論文で提案するシステムのアーキテクチャを**図 1** に示す．ここでは概要だけを説明し，詳細は 3 章以降で説明する．ユーザがシステムに与えるものは 2 つある．まず，RDB に格納されたデータの XML ビューを定義するビュー問合せを与え，次に，XML ビューで定義された XML データを利用する DOM プロラムを与える．

与えられた DOM プログラムの実行は次のように行われる．DOM プログラムから DOM API による XML データの操作（以下，**DOM 操作**）が呼び出されると，その答えを計算するための SQL 問合せが**SQL 問合せ生成器**によって生成され，RDB に投入される．システムは SQL 問合せの結果を DOM 操作の結果に変換し，DOM プログラムに返す．

ここで，各 DOM 操作の適用時に毎回 SQL 問合せを生成・実行するのは，明らかに非効率である．また，いくつかの DOM 操作をまとめて 1 つの SQL 問合せを作成するのも簡単には行うことができない．なぜなら，DOM 操作は手続き型プログラムの中に埋め込まれており，それらの実行制御は DOM 操作のみに依存しているわけではないからである．したがって，通常の単一の宣言型言語に対する問合せ最適化手法をそのまま適用することは困難である．

そこで，我々は一度作成された DOM プログラムは何度も再利用されることが多いことに着目した．提案システムでは，DOM プログラムの実行を監視し，2 回目以降の同じプログラムを効率良く実行するための**SQL 生成ルール**を見つける．これは「DOM プログラム実行時のどのタイミングに，どのような SQL 問合せを生成・実行すればよいか」を記述したルールである．DOM 操作の実行ごとに SQL 問合せの生成・実行を 1 つずつ行う場合と比べて，少ない実行コストで同じ効果を持つ SQL 問合せを生成するために利用

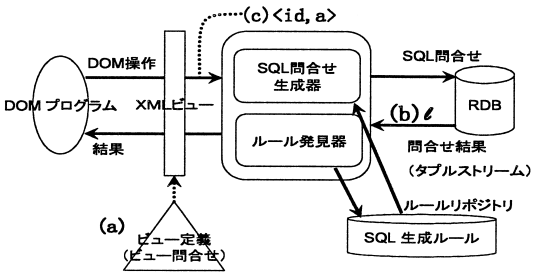


図 1 システムの全体像
Fig. 1 Architecture.

される．SQL 生成ルールは図 1 の**ルール発見器**により発見され，**ルールリポジトリ**へ格納される．

本論文の構成は次のとおりである．2 章で関連研究を説明する．3 章では，RDB 上の XML ビューについて説明する．4 章では，RDB 上の XML ビューで DOM プログラムを実行させるための基本的な枠組みを説明する．5 章で，DOM プログラム実行の効率化手法を提案する．具体的には，Sorted-Outer-Union プランを用いた SQL 生成ルールを提案する．6 章で実験結果を示し，5 章で提案した SQL 生成ルールが実行コストを大幅に削減する可能性があることを示す．7 章で，提案手法の改良について述べ，8 章の実験でその効果を示す．9 章は，本論文のまとめである．

2. 関 連 研 究

SilkRoute[4] と XPERANTO[9] は，RDB 上の XML ビューを実現するシステムとしてよく知られている．これらは，仮想 XML ビューに対する XQuery 問合せの実行を実現している．我々のシステムは，SilkRoute と同じ仕組みを用いて仮想 XML ビューの定義を行うが，宣言的な問合せ言語ではなく，DOM API を利用した手続型プログラムが実行可能な仕組みを提供する点が異なる．

我々の知る限り，これまでに RDB 上の仮想 XML ビューに対して DOM 操作を可能にした唯一のシステムは ROLEX[2] である．ただし，ROLEX は一般の RDBMS ではなく特定のメインメモリデータベースと組み合わせて利用されるシステムである．ROLEX 開発の主な動機は，メインメモリデータベースが提供する同時実行制御やリカバリ処理といったデータ管理機構を有効活用したいということである．一方，我々のシステムの目的は，数多く存在する既存の RDB に格納されているデータ対して，XML データを入力とするプログラムを簡単に適用可能にする仕組みを提供することであり，SQL インタフェースを持つ任意の RDBMS と組み合わせて利用される．ROLEX では

Supplier(suppkey, name, addr, nationkey)
Nation(nationkey, name, regionkey)
Region(regionkey, name)
PartSupp(partkey, suppkey, availqty)
Part(partkey, name, brand, size)

図 2 TPC-H database 構造の一部
Fig. 2 Part of TPC-H database schema.

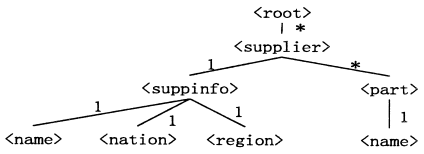


図 3 RDB 上の XML ビュースキーマ
Fig. 3 Schema of XML view over the RDB.

```
1. construct <root>
2.  { from Supplier $s
3.    construct
4.      <supplier> <suppinfo><name>$s.name</name>
5.        { from Nation $n
6.          where $s.nationkey = $n.nationkey
7.          construct
8.            <nation>$n.name</nation>
9.              { from Region $r
10.               where $n.regionkey = $r.regionkey
11.               construct <region>$r.name</region> }
12.       }</suppinfo>
13.       { from PartSupp $ps, Part $p
14.         where $s.suppkey = $ps.suppkey,
15.               $ps.partkey = $p.partkey
16.         construct
17.           <part> <name>$p.name</name> </part> }
18.     </supplier>
19. }
20. </root>
```

図 4 TPC-H database に対する RXL ビュー
Fig. 4 RXL view of TPC-H database.

処理の効率化のために，DOM プログラムが DOM 木の各辺をたどる確率を表現した navigational profile という情報を利用する．それに比べ，本システムが出力する SQL 生成ルールは DOM プログラムの実際の実行時の情報を利用して作成されるため，より細かな SQL 生成の制御を行うことが可能である．

3. RDB 上の XML ビュー

本章では，本システムが扱う RDB 上の XML ビューについて説明する．本システムでは SilkRoute の論文[3] と同じ枠組みを利用する．

3.1 例

まず RDB が TPC-H データベース[10] を格納しているとする．**図 2** はそのスキーマの一部である．このデータベースは，部品（part）や，部品を提供する会社（supplier）に関する情報等を格納している．我々は，このデータベース上に，**図 3** のスキーマ構造を持つ XML ビューを構築すると仮定する．各 supplier 要素は，1 つの suppinfo 要素と複数の part 要素を子要素として持つ．さらに，各 suppinfo 要素は supplier の name，nation，region を表す要素を持つ．また，各 part 要素はその name を持つ．

3.2 RXL によるビュー問合せ

XML ビューを定義するためのビュー問合せ(図 1 (a))の記述には，RXL[3]（Relational to XML transformation Language）を用いる．**図 4** は，図 2 の RDB を図 3 のスキーマに従う XML データにマッピングするためのビュー問合せを RXL で記述したものである．

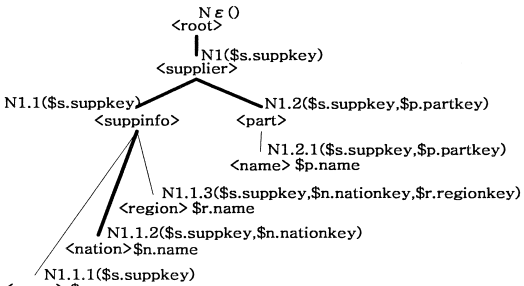
RXL は，SQL の from-where 節と XQuery の return 節（RXL では construct 節と呼ぶ）を組み合わせたものである．まず，`from` 節ではタプル変数を定義する．タプル変数はリレーション上の各タプルに順に束縛される．たとえば，タプル変数 `$s`（2 行目）はリレーション `Supplier` の各タプルに束縛される．RXL 問合せの動きを簡単にいえば，`from` 節で定義されているタプル変数をリレーションの各タプルに順に束縛し，それが `where` 節で指定された条件を満たすならば，`construct` 節に記述された XML 断片を出力する．`construct` 節は，“{ }” で囲まれた副問合せを持つことができる．たとえば 5 行目から始まる副問合せは，`where` 節の条件 `$s.nationkey=$n.nationkey` によって，`Supplier` と `Nation` を結合する．

XML 要素インスタンスの計算． ある RXL 問合せが与えられ，その `construct` 節に含まれている XML 要素（型）が 1 つ指定されたとする．そのとき，その要素インスタンスを計算するための SQL 問合せは RXL 問合せから簡単に求めることができる．具体的には，ある `construct` 節に含まれている XML 要素のインスタンスは，その `construct` 節を副問合せとして含む上位のすべての問合せの `from` 節のリレーションの結合演算を行い，その結果の各タプルに対して 1 つずつ生成される．たとえば，図 4 の問合せにおいて `<name>` および `<nation>` 要素のインスタンスは次の SQL 問合せの結果の各タプルに対して 1 つずつ生成される．

```
select $s.suppkey, $s.name from Supplier $s;

select $s.suppkey, $n.nationkey, $n.name
from Supplier $s, Nation $n
where $s.nationkey=$n.nationkey;
```

このような SQL 問合せの生成は，ビュー問合せを**ビュー木**（viewtree）[3] と呼ぶ木構造で表現することによって整理することができる．図 4 のビュー問合せ



q(N1) = Supplier $s
q(N1.1) = Supplier $s
q(N1.1.1) = Supplier $s
q(N1.1.2) = Supplier $s ⋈ Nation $n
q(N1.1.3) = Supplier $s ⋈ Nation $n ⋈ Region $n
q(N1.2) = Supplier $s ⋈ PartSupp $ps ⋈ Part $p
q(N1.2.1) = Supplier $s ⋈ PartSupp $ps ⋈ Part $p

図 5 ビュー木 $\mathcal{T}_1$（太線は $\mathcal{T}_1'$ を表す）
Fig. 5 Viewtree $\mathcal{T}_1$ (Bold lines represent $\mathcal{T}_1'$).

をビュー木として表現したものを**図 5** に示す．ビュー木は次のようなものである．

- RXL 問合せの `construct` 節に含まれる XML 要素間の親子関係を木構造で表している．
- Dewey Decimal Encoding[5] を利用して，ビュー木の各ノードに識別子〔以降，**VID**（Viewtree ID)〕を付ける．たとえば，VID として N1.x を持つノードは，N1 を持つノードの子ノードである．
- 各ノードは，そのノードが表す XML 要素のインスタンスを計算するための SQL 問合せを持つ．たとえば，q(N1.1.2) は `<nation>` 要素のインスタンスを計算する SQL 問合せであり，`Supplier $s` と `Nation $n` を結合した結果の各タプルから XML 要素インスタンスが生成されることを示す．
- 各ビュー木ノードには，要素インスタンスを表すタプル（SQL 問合せの計算結果のタプル）のキー属性を記述する．`<supplier>` 要素に対応する N1 ノードのキー属性は `$s.suppkey` である．
- いくつかのノードには，XML 要素が含む値を表すためのリレーショナル属性（たとえば `<name>` 要素の `$s.name`）が記述されている．

以降では，ビュー問合せによって計算される，各 XML 要素インスタンスのことを **XML ノード**☆と呼ぶ．形式的には，ビュー問合せによって計算される各 XML ノードは組 $(n, \bar{k})$ で表現される．ここで，n はこの XML ノードを計算する SQL 問合せを持つビュー木ノードの VID であり，$\bar{k}$ は n に付随するキー属性の値（の列）である．また，XML ノード $(n, \bar{k})$ のタグは，ビュー木ノード n が持つタグである．たとえば XML ノード (`N1, [$s.suppkey='#s1']`) は，ある `<supplier>` 要素インスタンスを表す．

```
1.  void main() {
2.      Document doc= new Document(v);
3.      Node root=doc.getDocumentElement()(id1);
4.      show(root)
5.  }
6.
7.  void show(Node node) {
8.      for (Node n=node.getFirstChild()(id2); n!=null;
9.                      n=n.getNextSibling()(id3)) {
10.       System.out.println(n.getNodeName()(id4));
11.       System.out.println(n.getNodevalue()(id5));
12.       show(n);
13.     }
14. }
```

図 6 DOM プログラム $\mathcal{P}_1$ の一部
Fig. 6 Fragment of DOM program $\mathcal{P}_1$.

XML ノード間の親子関係は次のように定義される．XML ノード $(n_1, \bar{k}_1)$ が $(n_0, \bar{k}_0)$ の子要素となる条件は，ビュー木において n_1 が n_0 の子であり，かつ $\bar{k}_1$ が，$\bar{k}_0$ と同じキー属性に関して，同一の値を持つことである．たとえば，XML ノード (`N1.1.2, [$s.suppkey='#s1', $n.nationkey='#UK']`) は (`N1.1, [$s.suppkey='#s1']`) ノードの子である．

4. XML ビューに対する DOM 操作の実現と問題

DOM（Document Object Model）[11] は，HTML や XML データを操作するための API を提供する．XML データのインスタンスは，要素の親子構造を反映させたオブジェクト木（**DOM 木**）として表現する．

4.1 単純な SQL 問合せ生成方式

図 6 は DOM API を利用した Java プログラム（DOM プログラム）の例である☆☆．このプログラム $\mathcal{P}_1$ は，入力として XML データ（ファイル名 v で指定される）をとる（2 行目）．そして，この XML データの木構造に従って XML ノードを前順走査し（4, 7–14 行目），各 XML ノードのタグと内容を出力する（10–11 行目）．本論文で扱う DOM 操作は，1 つの XML ノードを入力とし，1 つもしくは複数の XML ノードを結果として返すものである．$\mathcal{P}_1$ の，`getFirstChild` メソッド（8 行目）は，与えられた XML ノードの最初の子ノードを返す．`getFirstChild` メソッドが入力として XML ノード (`N1.1, [suppkey='#s2']`) を受け取ったときには，本システムは次のように SQL 問合せを生成する．すなわち，ビュー木上で N1.1 の子である N1.1.1 の SQL 問合せに，選択条件 “`$s.suppkey='#s2'`” を

☆ ビュー木ノードとは異なることに注意．

☆☆ 右肩の文字については後述．

```
(1) select $s.suppkey, $s.name
    from Supplier $s where $s.suppkey='#s2'
(2) select $s.suppkey, $n.nationkey, $n.name
    from Supplier $s, Nation $n
    where $s.nationkey=$n.nationkey and
          $s.suppkey='#s2'
(3) select $s.suppkey, $p.partkey,
    from Supplier $s, PartSupp $ps, Part $p
    where $s.suppkey=$ps.suppkey and
          $ps.partkey=$p.partkey and
          $s.suppkey='#s2'
   order by $s.suppkey, $p.partkey
```

図 7 実行される SQL 問合せの例
Fig. 7 SQL query examples.

追加し，結果を計算する SQL 問合せ（**図 7** (1)）を導出する．

`getNextSibling` メソッド（9 行目）は，与えられた XML ノードの（DOM 木において）隣接する弟ノードを返す．前の例に示した `getFirstChild` メソッドの実行結果が（`N1.1.1`，`[suppkey='#s2']`）であり，その結果に `getNextSibling` メソッドを適用したとする．本システムでは，現在のビュー木ノード N1.1.1 に隣接する弟ノードの SQL 問合せ q(N1.1.2) に選択条件を追加した SQL 問合せ（図 7 (2)）を実行する．同じ `getNextSibling` メソッドの適用であっても，XML ノード（`N1.2`，`[suppkey='#s2'`，`partkey='#p3']`）に適用した場合は異なる処理が行われる．なぜなら，`<supplier>` 要素は複数の `<part>` 要素を持つからである（図 3）．このような場合には，兄弟ノードは同じ SQL 問合せ（図 7 (3)）によって計算される．システムは，この問合せを実行し，`$s.partkey='#p3'` を持つタプルまでスキップし，「次の」タプルから結果の XML ノードを計算する．この SQL 問合せは `order by` 節を用いて，キー値順にタプルの順序を固定している．

このように SQL 問合せを作成することにより，DOM API 操作のうち，特定の要素型を指定したノードの取得と，各ノードのポインタをたどったナビゲーションが可能である☆（新たなノードの追加や更新に関係する操作，および任意の XPath 式を直接評価する操作等は不可能である）．ただし，以上のように各 DOM 操作の実行ごとに SQL 問合せを実行する方法は単純ではあるが非効率である．本論文ではこれを**単純な SQL 問合せ生成方式**と呼ぶ．

```
select * from (
select 1 as L1, $s.suppkey, N as L2, N, N as L3, N, N    // N1
from Supplier $s
union
select 1 as L1, $s.suppkey, 1 as L2, N, N as L3, N, N    // N1.1
from Supplier $s
union
select 1 as L1, $s.suppkey, 1 as L2, N,
       2 as L3, $n.nationkey, $n.name                     // N1.1.2
from Supplier $s, Nation $n
where $s.nationkey=$n.nationkey
union
select 1 as L1, $s.suppkey, 2 as L2, $p.partkey,
N as L3, N, N                                             // N1.2
from Supplier $s, $PartSupp $ps, Part $p
where $s.suppkey=$ps.suppkey, $ps.partkey=$p.partkey,
)
order by L1, $suppkey, L2, $p.partkey, L3, $n.nationkey
```

図 8 $\mathcal{T}_1'$ のための Sorted outer union plan
Fig. 8 Sorted outer union plan for $\mathcal{T}_1'$.

4.2 Sorted-Outer-Union プランと ADO ラベル

実は，DOM プログラムがある特定の順序で DOM 操作を実行するならば，より効率の良い SQL 問合せを利用できる．分かりやすい例としては，DOM 木を前順に走査するプログラムがあげられる．この場合，**Sorted-Outer-Union プラン**[9]（以下，**SOU プラン**）を利用すれば，1 度の SQL 問合せの実行だけですむ．直感的には，SOU プランとは SQL 問合せの一種であり，DOM 木の木構造を Unnest したものに相当するリレーションを問合せ結果として返すものである．SOU プランにはいくつかの変種が存在するが，結果の各タプルは，DOM 木の 1 つのノード，もしくは根ノードから葉ノードへのパスに対応する．

SOU プランを利用すると，DOM 木を前順走査する DOM プログラムが消費する順に DOM 木の各ノード（タプル）が得られるため，XML データをすべて実体化して通常の DOM ライブラリを利用する場合とは異なり，すべての DOM 木ノードを同時にメモリに保持する必要はない．

本システムで利用する SOU プランを説明するために，$\mathcal{T}_1$（図 5）の部分木であるビュー木 $\mathcal{T}_1'$（太線）を使い説明する．$\mathcal{T}'_1$ で計算される XML ビューを前順に走査するための SOU プランは**図 8** である．SQL 問合せ中の `N` は null である☆☆．図 8 の SQL 問合せ結果の各タプルは，$\mathcal{T}_1'$ の XML ビューで計算されるいずれかの XML ノード 1 つに相当する．その XML ノードを $(n, \bar{k})$ とすると，その情報はタプル内で次のように表現されている．(1) VID である $n = \mathtt{N}\, l_1.l_2.\ldots.l_n$ は分解され，属性 $L1$, $L2$, …, Ln の値として格納さ

☆ 現時点のシステムで実装されている DOM 操作は付録 A.1 に示す．

☆☆ これは単純な SOU プランである．より洗練された SOU プランの実現については文献 4)，9) 等で議論されている．

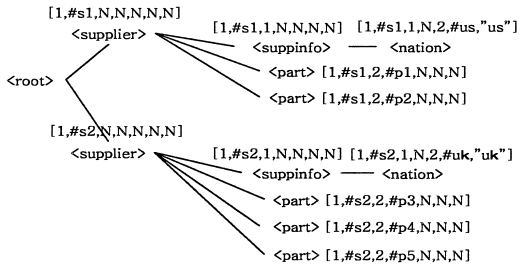


図 9 XML インスタンスの構造とその ADO ラベル
Fig. 9 Structure of an XML instance and ADO labels.

れている．(2) $\bar{k}$ で示されるキー値は，それぞれ対応するキー属性の値として格納されている．

これらのタプルは，`order by` 節により次のソートキーで並べ替えられる．まず，$L1$，次にビュー木ノード N1 のキー属性（この場合 `$s.suppkey`．一般には $k_{(1,i)}$ と表記する），続いて $L2$，N1 の 1 段下のビュー木ノード N1.x で初めて出現するキー属性（この場合 `$p.partkey`．$k_{(2,i)}$），... といった順である．実はこのタプルの順序は，XML ビューの計算結果である DOM 木を前順に走査した場合の順序と同じになる（**図 9**．各ノードの [...] は対応するタプル）．したがって，SQL 問合せを実行し，結果のタプルを順に読み込むだけで DOM 木の走査を行えることになる．これを別の言葉でいい換えると次のようになる．

ADO ラベルと DOM 木構造の関係． 各 XML ノードにラベル “L1, $k_{(1,1)}$, . . . , $k_{(1,n_1)}$, L2, $k_{(2,1)}$, . . . , $k_{(2,n_2)}$, L3, $k_{(3,1)}$, . . .” を割り当て，これらのラベル間の辞書式順序を定義すると，この順序は XML ビュー（DOM 木）の木構造を表現している．

我々は，このラベル付けの枠組みを **Augmented Dewey Order Encoding** と呼び，各 XML ノードを表すラベルを **ADO ラベル**と呼ぶ．定義より，ある XML ノードを表す ADO ラベルが与えられたとき，そのノードを計算するために利用されたビュー木ノードの VID $l_1, l_2, \ldots l_n$ は ADO ラベルから簡単に抽出できる．この性質は 5.3 節で説明するアルゴリズムで利用される．

この SOU プランは $\mathcal{P}_1$ に対して単純な SQL 問合せ生成方式を行うのに比べて効率的であると考えられる．なぜなら SQL 問合せは一度しか実行されず，かつすべてのタプルは無駄に捨てられることなく DOM 操作の結果として消費されるからである．一般的に，SOU プランが効果を発揮するのは，DOM プログラムがこのような深さ優先順走査もしくはそれに類する処理（詳細は 5 章で説明する）を行う場合である．

```
1. void show(Node node) {
2.   if (node!=null) {
3.      System.out.println(node.getNodeName());
4.      System.out.println(node.getNodevalue());
5.   }
6.   Node n1=node.getFirstChild();
7.   show(n1);
8.   Node n2 node=n1.getNextSibling();
9.   show(n2);
10.  }
11. }
```

図 10 二分木走査用 DOM プログラム
Fig. 10 DOM program for binary tree traversal.

```
void show(Node node) {
    for (Node n=node.getFirstChild()(id3); n!=null;
                  n=n.getNextSibling()(id4)) {
       System.out.println(n.getNodeName()(id5));
       val= n.getNodevalue()(id6);
       if (!val.equals(‘‘....’’))
          System.out.println(n.getNodevalue()(id7));
       show(n);
    }
}
```

図 11 DOM プログラム $\mathcal{P}_1'$ の `show` メソッド
Fig. 11 The `show` method of $\mathcal{P}_1'$.

4.3 問　　題

問題は，DOM プログラムのソースコードから上のような「効果的なパターン」を抽出することが困難なことである．同じ機能を実現する DOM プログラムの実装は一般に数多くある．たとえば，DOM 木が二分木の場合，図 6 と**図 10** のプログラムはどちらも同じ結果を出力する．極端な DOM プログラムを考えれば，ある DOM 木を走査するために，繰返しや再帰構造はいっさい使わずに，その DOM 木に含まれる XML ノードの数だけ DOM 操作をハードコーディングしているようなものも考えられる．また，DOM プログラムは必ずしも $\mathcal{P}_1$ の深さ優先走査のような「完全なパターン」を実装しているとは限らない．たとえば，基本的には深さ優先走査をするのだが，いくつかの数少ない XML ノードはスキップするような DOM プログラム（たとえば，**図 11** の `show` メソッドを持つプログラム $\mathcal{P}_1'$）も，同じ SOU プランによって効率的に実行できると考えられる．

5. DOM プログラム実行の効率化

4.3 節で述べた理由から，我々は，ソースコードを解析したりあらかじめ用意したパターンを利用したりするというアプローチではなく，DOM プログラムの実行を監視して効率的な SQL 問合せの生成ルールを見つけるというアプローチをとることにした．発見された SQL 問合せ生成ルールは，DOM プログラムの 2 度目以降の実行時に利用される．

5.1 概　　要

まず，ある DOM プログラムを初めて実行するとき

は，4 章で説明した単純な SQL 生成方式で SQL 問合せを生成，実行する．このとき，RDB から DOM 操作の結果として返されるタプルを時間順に並べたものをタプルストリームと呼ぶ．ルール発見器には，タプルストリームの各タプル l が順次入力される（図 1 (b)）（DOM プログラムを初めて実行するときにも ADO ラベルを求める事が可能になるように，単純な SQL 生成方式で作られる SQL 問合せにも，$L1$，$L2$ といった VID を表すための属性を追加しておく）．

厳密にいうとルール発見器は，結果の各タプルを入力として受け取るだけでなく，実行した DOM 操作に関する情報も受け取る（図 1 (c)）．形式的には，ルール発見器への入力は，タプルストリームの各タプルを表す ADO ラベル l に，プログラム中の各 DOM 操作に結び付けられた ID（図 6 の各 DOM 操作の右肩）id☆と，その DOM 操作実行時に渡されたパラメータの集合 a を組み合わせた $(\langle id, a\rangle, l)$ となる．つまり，DOM プログラムに含まれる DOM 操作（id を持つ）がパラメータの a を用いて実行され，その結果，SQL 問合せが実行されて結果としてタプル（ADO ラベル）l が返されたとする．このとき，$(\langle id, a\rangle, l)$ がルール発見器の入力として渡される．ルール発見器はこの入力の並びを監視し，効率の良い SQL 生成規則を発見し，ルールリポジトリに格納する．

2 回目以降の DOM プログラム実行時には，ルールリポジトリに格納された SQL 問合せ生成規則を利用して，SQL 問合せ生成器（図 1）が SQL 問合せを生成する．同時に，2 回目以降の実行においても，ルール発見器は新しい規則を発見するための監視を行う．

5.2 SQL 生成ルールの詳細

$\mathcal{T}$ をあるビュー木とし，$\mathcal{P}$ をある DOM プログラムとする．このとき，**$\mathcal{T}$ と $\mathcal{P}$ 上の SQL 生成ルール**（以下，単に**ルール**）は，$p \to t$ と表される．ここで，p は組 $\langle id_i, a_i\rangle$ の列である．id_i は $\mathcal{P}$ に含まれる各 DOM 操作につけられた操作 ID，a_i は，その DOM 操作を実行したときのパラメータ値の集合である．ルールの右辺に現れる t は，$\mathcal{T}$ もしくは $\mathcal{T}$ の**縮約**（contraction）[6] である．一般に，木 T の縮約とは，T のいくつかの辺をそれぞれ 1 点に縮める（両端の頂点を同一と見なす）ことによってできた木である．T の縮約は T の部分木とは異なり，T では隣接してなかった頂点どうしが辺で直接結び付くことができる．次のルール R_1 は，図 5 の $\mathcal{T}_1$ と図 6 の $\mathcal{P}_1$ 上のルールの例である．

$$R_1 : [\langle id1, \{obj = \mathrm{doc}\}\rangle, \langle id2, \{obj = \mathrm{N}\varepsilon\}\rangle] \to \mathcal{T}_1$$

ここで obj は，DOM 操作（メソッド）が適用されたオブジェクトを表す特別なパラメータ名，`doc` は DOM 木のドキュメントノード（ルートノードの親となるノード）のための特別な VID である．R_1 の意味は「DOM 操作 $id1$（図 6 の 3 行目）の実行後，DOM 操作 $id2$（図 6 の 8 行目）が実行されたら，$\mathcal{T}_1$ に対応する SOU プランである SQL 問合せを生成し，実行せよ」である．実行結果のタプルストリームは，これ以降の DOM 操作の結果として利用される．

ルール中の a_i のパラメータ値としては，XML ノード $(n, \bar{k})$ ではなく，VID である n（たとえば $\mathrm{N}\varepsilon$）のみが利用される．このように設計した理由は次のとおりである．すなわち，XML ノード（インスタンス）があるパターンにマッチすべきという条件はあまりにも厳しく，RDB のデータの値が少し変わってしまっただけでもパターンにマッチしなくなるからである．

次のルールは，右辺が $\mathcal{T}$ そのものではないという意味で，R_1 よりもより一般的なルールの例である．$\mathcal{P}_2$ をある DOM プログラムとし，これが $id8$ と $id9$ で表される DOM 操作を含んでいるとする．$id8$ の DOM 操作は `a.getElementsByTagName(x)` とする．このとき，次の例は $\mathcal{T}_1$ と $\mathcal{P}_2$ 上のルールである．

$$R_2 : [\langle id8, \{obj = \mathrm{doc}, par1 = \text{“nation”}\}\rangle \langle id9, \mathrm{N1.1.2}\rangle,] \to \mathcal{T}_2$$

ここで，$\mathcal{T}_2$（**図 12** に示す）は，$\mathcal{T}_1$ の縮約であり，“nation” は DOM 操作 $id8$ を実行するときに `x` の値として利用された値である．

5.3 ルール生成アルゴリズム

先に述べたように，提案手法の特徴の 1 つは，単純で効率良く SQL 問合せ生成ルールを発見するためのアルゴリズムである．基本的なアイデアは，ADO ラベルの性質を用いて，(1) ADO ラベルのストリームを監視して SOU プランでまとめて計算できる部分を発見し，(2) その部分をまとめて計算する SQL 問合せを生成するルールを構築する，ということである．SOU プランでまとめて計算できるかどうかの判定は，

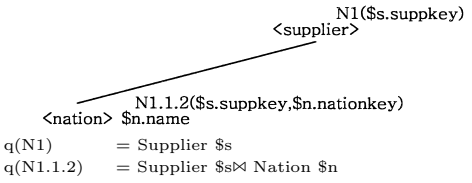


図 12 $\mathcal{T}_1$ の縮約 $\mathcal{T}_2$
Fig. 12 A contraction $\mathcal{T}_2$ of $\mathcal{T}_1$.

☆ 現実装では，id は各 DOM 操作関数の引数として明示的に与える．これらの id は，DOM プログラムをプリプロセサにかけることにより，追加することを想定している．

```
 1. t={};
 2. inOrder=false; // true when the input follows the order
 3. <id ', a '>=<nil, [ ]>; l '=infinite;
 4. for each (<id, a>, l) in the Input Stream {
 5.    if (l' < l) {
 6.       if (!inOrder) {
 7.          inOrder=true;
 8.          t.add(l '. getViewTreeNodeID());
 9.          <id_2, a_2>=<id ', a '>;
10.       }
11.       t.add(l.getViewTreeNodeID());
12.    } else {
13.       if (inOrder) {
14.          inOrder=false;
15.          outputRule( [<id_1, a_1>, <id_2,a_2>]->t );
16.          t={};
17.       } else <id_1, a_1>=<id ', a '>;
18.    }
19.    <id ', a '>=<id, a>; l '=l;
20. }
```

図 13 アルゴリズム **RuleFind**
Fig. 13 Algorihtm for finding rules.

ADO ラベルのストリームが辞書順で並んでいるかどうかによって簡単に判定できる．また，ADO ラベルから VID が容易に抽出できることを利用して，SQL 問合せ生成ルールの構築が効率良く行われる．

図 13 は，ルール生成器の内部で利用されるルール生成アルゴリズム **RuleFind** である．**RuleFind** が発見するルール $p \rightarrow t$ の p の長さは 2 である．すなわち，発見されるルールは 5.2 節で例にあげたルール R_1 や R_2 のように，$[\langle id_1, a_1\rangle, \langle id_2, a_2\rangle] \rightarrow t$ という形式を持つ．

RuleFind では，ストリーム中で連続する 2 つの DOM 操作とその結果（タプル）に関する情報が $(\langle id', a'\rangle, l')$ と $(\langle id, a\rangle, l)$ に格納される（4，19 行目）．変数 `t`（1，8，11，15 行目）は，ルール $p \rightarrow t$ の t（$\mathcal{T}$ の縮約）を表現する．具体的には，`t` は縮約に含まれる VID の集合を持つ．たとえば，`t={N1, N1.1.2}` は，ノードとして `N1` と `N1.1.2` を含む縮約を表す（図 12）．

RuleFind が行うことは単純である．2 つの連続した DOM 操作の結果（ADO ラベル）l' と l を毎回比較し（5 行目），その順序が辞書順に従っている間は，l から抽出された VID を `t` に追加する（8，11 行目）．$\langle id', a'\rangle$ の値は，ルールの左辺 p を構築するために利用される（9，15，17 行目）．$(\langle id', a'\rangle, l')$ の初期値は $(\langle nil, \{\}\rangle, infinite)$ である（3 行目）．ここで，$infinte$ はどの ADO ラベルと比較しても大きい特別なラベルである．

タプルストリームで ADO ラベルが辞書順に並んでいないことを発見すると，**RuleFind** はその後の DOM 操作の結果は別の SQL 問合せで計算されるべきと判断し，これまで構築した `t` を持つルールを出力し（15 行目），新しいルールを構築するために `t` を初期化する（16 行目）．

RuleFind は効率的であり，S をストリームのサイズ，V をビュー木のノード数としたときに，$O(S)$ の時間と $O(V)$ の領域しか必要としない．

5.4 ルールの利用

ルール $p \rightarrow t$（ここで $p = [\langle id_1, a_1\rangle, \langle id_2, a_2\rangle]$）がルールリポジトリに格納されているとき，DOM プログラムの 2 度目以降の実行では，このルールは次のように利用される．最初は，単純な SQL 生成方式を用いて DOM プログラムの実行を開始し，システムに順次渡される $\langle id, a\rangle$（図 1 (c)）を監視する．これらが連続して $\langle id_1, a_1\rangle$，$\langle id_2, a_2\rangle$ にマッチするのを発見すると，SQL 生成器は t に対応する SOU プランである SQL 問合せ q を生成し，さらに不必要なタプルを除去するための選択条件を q に追加する．たとえば，id_2 が `getFirstChild` のときには，3 章で説明した `getFirstChild` の場合と同じように選択条件が追加される．

DOM 操作の結果として必要なタプルが問合せ結果に含まれない場合は，その問合せ結果の利用を打ち切って単純な SQL 生成方式を利用する状態に戻り，再び実行の監視を始める．また，データの更新の結果，2 度目の実行時には，最初の実行時のルールが有効でなくなる場合がある．この場合にも，その問合せ結果の利用を打ち切って単純な SQL 生成方式を利用する状態に戻る．しかし，データ更新の影響が部分的に限定される場合には，その実行中に他のルールにマッチすれば，対応する SOU プランが生成・利用される．したがって，必ずしもデータ更新のたびに完全に単純な SQL 生成方式に戻るわけではない．

6. 実 験 1

SOU プランを利用する効果を検証するための実験を行った．実験で用いた PC は，CPU が Pentium-4 1.5 GHz，メモリが 256 MB，OS が Linux RedHat8.0 である．この PC 上で PostgreSQL7.3.2 を動作させ，様々なサイズの TPC-H データベースを格納した．また，同じ PC に，提案手法を組み込んだ DOM API 互換のライブラリを Java（JDK1.4）で実装した．実験では，TPC-H データベース上に図 3 のスキーマに従う XML ビューを構築するため，XML ビュー問合せとして $\mathcal{T}_1$（図 5）を定義し利用した．データベースのサイズが増大するに従い，`supplier` と `part` 要素の数が増加するが，木の高さは変わらない．

実験にあたっては，本手法が有効と考えられる状況

```
1.   void main() {
2.       Document doc= new Document(v);
3.       NodeList a=doc.getElementsByTagName("suppinfo")^(id8);
4.       Node b=null;
5.       for (int i=0; (b=a.item(i)^(id9))!=null; i++) { show(b); }
6.   }
```

図 15 $\mathcal{P}_3$ のメイン関数（show 関数は，図 6 と同じ）
Fig. 15 The main program of $\mathcal{P}_3$ (show function is given in Fig. 6).

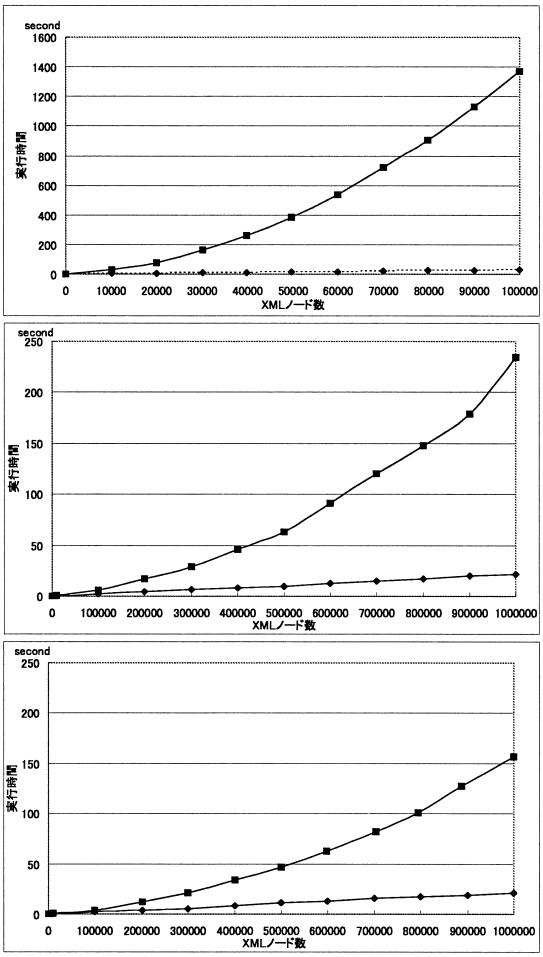


図 14 実験結果 1
Fig. 14 Experimental results 1.

と DOM プログラムの典型的な利用形態をともに考慮し，次の問合せセットを用意した．(1) XML データ全体を前順走査するプログラム $\mathcal{P}_1$ および途中でいくつかのノードをスキップするプログラム $\mathcal{P}_1'$，(2) XML データから指定されたタグ名を持つ部分を抽出し，その後抽出した部分を走査するプログラム，(3) DOM 木のルートからナビゲーショナルに順次アクセスを行い，特定のオブジェクトにアクセスするプログラム．

(1) DOM プログラム $\mathcal{P}_1$ の実行

まず，様々なサイズのデータベースに対して $\mathcal{P}_1$（図 6）を実行した．$\mathcal{P}_1$ は XML ビューに含まれるすべての XML ノードを前順に走査するプログラムである．**RuleFind** はこれらの XML ノードをすべて計算する 1 つの SQL 問合せを作成する．本手法にとって理想的な状況であるといえる．結果を**図 14**（上）に示す．横軸はデータベースのサイズであり，縦軸は時間を表す．データベースサイズは，XML ビューに含まれる XML ノードの数で表されている．実線は 1 回目の実行にかかった時間，点線は 2 回目以降の実行時間である．1 回目の実行時は単純な SQL 問合せ生成によって実行が行われるためノード数と同じ数の SQL 問合せが発行されるが，2 回目以降は 1 つの SQL ですべての XML ノードを前順に走査するための SOU プランが実行される．結果から分かるように，SOU プランは実行時間を大幅に短縮できる．

なお，$\mathcal{P}_1$ は素直に木の走査を実装しているが，4.3 節で述べたように，同じ効果を持つプログラムは多様な方法で実装可能である．また，完全な走査が現れることはまれである．我々は $\mathcal{P}_1$ の 11 行目に if 文が入り，ある条件を満たした場合にはノードがスキップされるような場合（図 11 の $\mathcal{P}_1'$）についても実験を行った．この場合も同じ SQL 生成ルールが出力され，したがって出力された SOU プランも $\mathcal{P}_1$ とほぼ同じ実行時間であった．本論文ではグラフは省略する．

(2) より現実的なプログラムの実行

次に，より現実的なプログラム $\mathcal{P}_3$（**図 15**）と $\mathcal{P}_4$（付録 A.2 の図 23）の実行を行った．これらのプログラムは，XML ビュー全体を前順に走査するのではない．$\mathcal{P}_3$ は，XML ビューの中からいくつかの XML ノードを取り出し，取り出された各 XML ノードを頂点とする DOM 木の部分木それぞれに対して前順走査を行う．$\mathcal{P}_4$ は，ナビゲーショナルなアクセスを行う問合せである．具体的には，XML ビューのルートから，/supplier/suppinfo/name というパスにある特定の要素の値を取得する．この $\mathcal{P}_3$ と $\mathcal{P}_4$ は，実際の DOM プログラムで見ることができる典型的なパターンである．どちらのプログラムも，1 回目の実行時には，ノード数と同じ数の SQL 問合せが発行される．

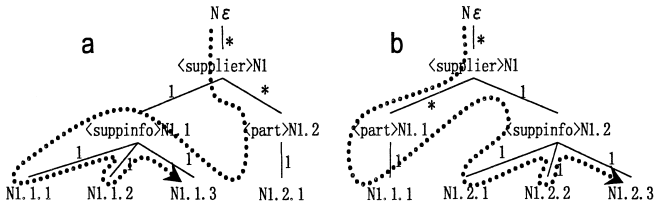


図 16 VID とアクセス順序
Fig. 16 VIDs and different access orders

興味深いことに，本システムはこの $\mathcal{P}_3$ と $\mathcal{P}_4$ に対しても，2 回目の実行では 1 つの SQL 問合せしか必要としない．その理由は，結果はすべて ADO ラベルの辞書順に従って並んでいるからである．$\mathcal{P}_3$ の実行結果を図 14 (中)，$\mathcal{P}_4$ の実行結果を図 14 (下) に示す．これらのプログラムは XML ビュー全体を取得するのではなく，ごく限られた部分木だけを取得するため，$\mathcal{P}_1$ より速い実行時間となっている．しかし，SOU プランを利用すると，実行時間を短縮できることが分かる．また，1,000,000 ノードを持つ XML ビューのデータベースサイズは約 405 MB であるが，本手法では通常の DOM ライブラリを利用する場合と違い，XML ビューのサイズにかかわらずアプリケーションプログラムは一定かつわずかなメモリ領域しか必要としない．

7. 動的な VID の割当て

これまでは，ビュー木ノードの VID は，そのビュー木が与えられたときに決定していた．5.3 節で説明したように，アルゴリズム **RuleFind** では，入力される ADO ラベルのストリーム（タプルストリーム）の一部が特定の条件を満たしていれば，その部分をまとめて計算するような SQL 問合せを生成する．つまり，条件に従わない部分が多くなれば実行時に必要な SQL 問合せの数が増えてしまう．実は，実行時に動的に VID を割り当てることにより，タプルストリームが条件を満たす可能性を高め，効率の良い SQL 問合せを生成できる場合がある．

たとえば，図 5 のビュー木で表される XML ビューの DOM 木を，単純に前順（左側の部分木を右より優先して）にアクセスするのではなく，**図 16** (a) に示した順序でアクセスする DOM プログラム $\mathcal{P}_5$（コードは付録 A.2 の図 24）を考える．すなわち，各 supplier 要素についてまずすべての part 要素とその子要素にアクセスし，次に suppinfo 以下をアクセスする．この場合，**RuleFind** ではタプルストリームの中で，ADO ラベルの順序に関して成立すべき条件が，同じ supplier の part から suppinfo に移動するときに満たされない．したがって，各 supplier のこの部分でストリームが分断されるため，supplier の数を n とすると $n+1$ 回の SQL 問合せ実行が必要になる．

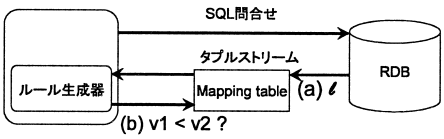


図 17 動的な VID の割当て
Fig. 17 Dynamic allocation of VIDs.

VID	VID'
1	1
1.1	1.2
1.1.1	1.2.1
1.1.2	1.2.2
1.1.3	1.2.3
1.2	1.1
1.2.1	1.1.1

図 18 対応表
Fig. 18 Mapping table.

ここでのポイントは，このプログラムは，これまで例で利用してきた図 5 のビュー木の part と supplier を左右入れ替えたビュー木 (図 16 (b)) を単純に前順走査するのと同等であることである．この場合，タプルストリームの順序が崩れることがないため，**RuleFind** をそのまま用いてもタプルストリーム全体を計算する 1 つの SOU プランを作成するような SQL 生成規則が導出される．したがって，SQL 問合せの実行は 1 回ですむ．

本章で説明する手法は，以上のアイデアを実現したものである．次のように，**RuleFind** の単純な拡張で実現できる．まず，**RuleFind** の実行前には，ビュー木ノードにデフォルトの VID を割り当てておく．これは通常の左の部分木を優先した Dewey Decimal Encoding である．しかし，タプルストリームが流れはじめると，その順序を監視し（**図 17** (a))，ビュー木ノードの VID を動的に VID' に書き換える．書き換え結果は**対応表**（Mapping Table）に格納する．**図 18** はその例である．書き換えられた表は，ADO ラベル間の順序を比較する際（図 13，5 行目）に参照され，

```
 1. Class MappingTable {
 2.
 3. t =  set of (VID vid, VID vid', boolean Fixed);
 4.
 5. Void update_mapping(ADO_Label l) {
 6.     VID v=l.getViewTreeNodeID();
 7.     for (i=1; i<= v.length(); i++) {
 8.        VID v2=v.prefix(i);
 9.        row r=t.getNewByOriginalVID(v2);
10.       if (!r.isFixed()) {
11.           row r2= t.getParent(r);
12.           m=t.LastNumberOfFixedVID_in_Siblings(v2);
13.           r.vid'= r2.getNewVID().concat(m+1);
14.           r.fixed=true;
15.           for each r in t s.t. r.fixed=false {
16.              r.get(v_i).vid'=consistent_VID(v2, v_i);
17.           }
18.       }
19.    }
20. }
21.
22. boolean isSmaller(VID v1, VID v2) {
23.     VID new_1= t.getNewByOriginalVID(v1);
24.     VID new_2= t.getNewByOriginalVID(v2);
25.     return  (new_1<new_2);
26. }
27. }
```

図 19 対応表関連手続き
Fig. 19 Methods for the mapping table.

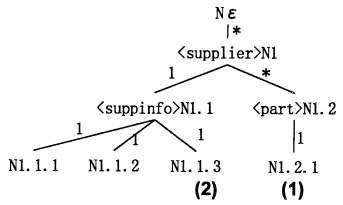


図 20 VID の動的な割当ての例
Fig. 20 Example of dynamic allocation of VIDs.

新しい VID' を用いた比較が行われる．2 回目以降の実行の際には，この VID' を利用して SQL 生成や処理が行われる．

対応表の具体的なアルゴリズム（Class MappingTable）を**図 19** に示す．概要は次のとおりである．まず，タプルストリームの各タプル（ADO ラベル）l を読むごと（図 17 (a)）に，対応表メソッド `update_mapping(ADO_Label l)` が呼ばれる．これにより，対応付けが更新される．また，**RuleFind**（図 13）の 5 行目では，VID の大小関係は通常の VID の比較ではなく MappiingTable クラスの `isSmaller(VID v1, VID v2)` を呼び出す（図 17 (b)）ことにより計算される．これは，$v1$ と $v2$ を直接比較するのではなく，対応表に書き込まれた新たな VID' 間の比較を行う．

update_mapping メソッド． 簡単な例を用いて説明する．まず，**図 20** のビュー木が与えられ図中に表される VID があらかじめ割り当てられているとする．あるDOM プログラムをこのビュー木上で実行したとき，タプルストリームには ADO ラベル l_1，l_2 の順に現れたとし，l_1 に含まれる VID は N1.2.1（図 20 (1)），l_2 の VID は N1.1.3（図 20 (2)）であったとする．このとき，`update_mapping`(l_1)，`update_mapping`(l_2) が順に呼ばれる．対応表の書き換えは次のように行われる．

(a)

VID	VID'	Fixed
1	1	True
1.1	1.2	false
1.1.1	1.2.1	false
1.1.2	1.2.2	false
1.1.3	1.2.3	false
1.2	1.1	True
1.2.1	1.1.1	True

(b)

VID	VID'	Fixed
1	1	True
1.1	1.2	True
1.1.1	1.2.2	false
1.1.2	1.2.3	false
1.1.3	1.2.1	True
1.2	1.1	True
1.2.1	1.1.1	True

図 21 対応表の書き換え
Fig. 21 Updating mapping table.

(1) `update_mapping`(l_1) が呼ばれたとき：このメソッドの実行後，対応表は最終的に**図 21** (a) になる．Fixed 欄は書き換えの結果が決定されたことを表すフラグであり，初期値は false である．メソッドの実行は次のように行われる．まず，l_1 が表す XML ノードに対応する VID は 1.2.1 であることを知る（6 行目）．次に，これらの prefix である 1，1.2，1.2.1 のそれぞれを順に $v2$ に格納し（8 行目），7–19 行目を繰り返す．ここでは，$v2$=1 の場合について説明する．対応表から，元の VID 欄の値が 1 である行を取り出し（9 行目），Fixed 欄が true でないことを調べる．Fixed 欄が true でなければ，旧 VID が 1 のノードに対して，次に説明する新しい VID を割り当て Fixed 欄を true にする（10–14 行目）．ここで新しい VID' は，DOM 木において旧 VID が $v2$（ここでは 1）のノードの親の VID' の後ろに，次に説明する m を追加したものである．m は，$v2$ の兄弟ノードで Fixed 欄が true である新しい VID' のうち，最大の VID' の最後尾の値（存在しなければ 0）に 1 を加えたものである．$v2 = 1$ の場合は親ノードの VID は ε（空）であり，また兄弟ノードも存在しないため空 VID の後ろに 1（=0+1）を追加した “1” が VID' 欄に追加され，Fixed 欄に true がセットされる．最後に，Fixed 欄が true になった VID' に合わせて対応表中の決定されていない VID' 欄を変更する（15–17 行）．これらの Fixed 欄は false のままである．この操作を $v2 = 1.2$，$v2 = 1.2.1$ の場合に関しても繰り返すと図 21 (a) が得られる．

(2) `update_mapping`(l_2) が呼ばれたとき：まず，l_2

が表す XML ノードに対応する VID は 1.1.3 であることを知る（6 行目）．次に，これらの prefix である 1，1.1，1.1.3 のそれぞれを順に $v2$ に格納し（8 行目），7–19 行目を繰り返す．結果として新たに 1.1→1.2 と 1.1.3→1.2.1 の書き換えが行われる．その結果は図 21 (b) になる．

is_smaller メソッド． `is_smaller(VID` $v1$`, VID` $v2$`)` は，$v1$ と $v2$ の大小関係を直接比較するのではなく，それぞれのタプルの VID' 欄に格納されている値を比較した結果を返す．たとえば，`is_smaller(1.2, 1.1.3)` は，1.2 < 1.1.3 ではなく 1.1 < 1.2.1 を評価し，true を返す．

以上の仕組みにより，**RuleFind** はビュー木に新しい VID' が付与されたものとして実行される．その結果，たとえば $\mathcal{P}_5$ のプログラムを実行するときには，図 16 (a) ではなく図 16 (b) のビュー木に対して左の子優先の走査を行っていると見なされる．

8. 実　験　2

動的な VID 割当ての効果を調べるための実験を行った．実験環境は 6 章と同じである．図 16 (a) の順序でアクセスする DOM プログラム $\mathcal{P}_5$ を実行し，VID 固定の場合と，対応表を利用して動的に VID を割り当てた場合の比較を行った．**図 22**（上）は，VID 固定でプログラム $\mathcal{P}_5$ を実行した結果である．実線は 1 回目の実行時間，点線は 2 回目の実行時間を表す．図 22（下）は，動的に割り当てた VID を利用した結果である．

7 章で述べたように，VID 固定の場合は，同一の supplier の part 要素から suppinfo 要素に移動するとき，タプルストリームの順序が条件を満たさなくなり，SQL が分断されてしまう．したがって，2 回目でも $n+1$ 回の SQL 問合せの実行が必要になる．一方，対応表を利用した場合，プログラムは図 16 (b) のビュー木から計算される DOM 木を，単純に走査することになる．したがって，2 回目の実行時にはタプルストリーム全体を計算する 1 つの SOU プランが作成，利用される．これらの 2 つを比べると，1 回目の実行では対応表作成のオーバヘッドがあるため多少 VID を動的に割り当てた方が遅くなるものの，2 回目の結果は大幅に速くなる．この実験では VID 固定の場合に 2 回目の実行時間がそれほど短縮されていないが，その理由は，SQL 実行回数が多いことに加え，2 回目に実行する SQL 問合せが DOM 操作の結果に必要な以外のタプルも数多く出力してしまい，その計算とタプルのスキップに時間がかかっていることも一因になっている．

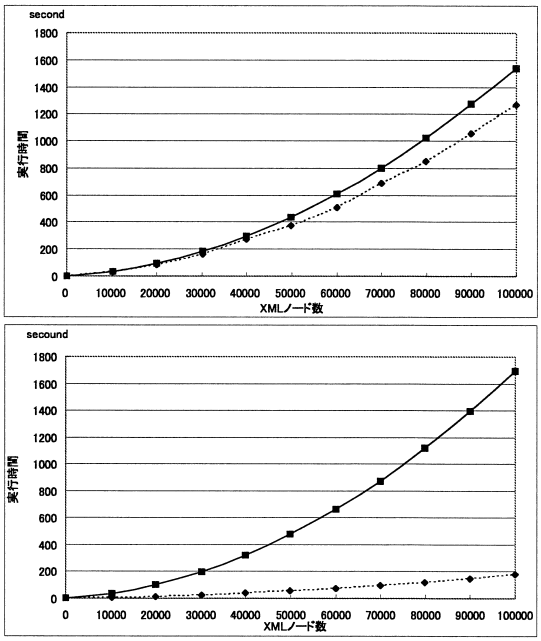


図 22 実験結果 2
Fig. 22 Experimental results 2.

9. ま　と　め

本論文では，RDB に格納されているデータに対して仮想 XML ビューを構築し，DOM プログラムを適用可能なシステムを提案した．鍵となるのは，手続き型プログラムに埋め込まれている DOM 操作群を効率良く実行する技術である．本システムでは，最初の実行時に効率の良い実行のための SQL 問合せ生成ルールを発見し，2 回目以降の実行時にはそれを利用する．生成ルールを基に作成される SQL 問合せは Sorted-Outer-Union プランに基づいている．本論文では，実験により，Sorted-Outer-Union プランを利用できる状況においては，単純な SQL 生成方式にくらべ大幅に実行時間を短縮できることを示した．また，この仕組みをより広い範囲で適用可能にするための方式を開発し，実験によってその効果を示した．今後の課題としては，SQL 生成ルール発見アルゴリズムの効率の良さを利用し，ルールが発見されるとプログラム実行中にでもただちにそのルールを利用可能にする「Just-in-time」方式の開発があげられる．

参　考　文　献

1) 小島　章，森嶋厚行：RDB 上の XML ビューに対する DOM 操作実現機構，情報処理学会第 65

回全国大会講演論文集，No.3, pp.5–6 (2003).
2) Bohannon, P., Ganguly, S., Korth, H.F., Narayan, P.P.S. and Shenoy, P.: Optimizing View Queries in ROLEX to Support Navigable Result Trees, *Proc. VLDB 2002*, pp.119–130 (2002).
3) Fernandez, M., Morishima, A. and Suciu D.: Efficient Evaluation of XML Middleware Queries, *Proc. SIGMOD Conference 2001* (2001).
4) Fernandez, M.F., Kadiyska, Y., Suciu, D., Morishima, A. and Tan, W.C.: SilkRoute: A Framework for Publishing Relational Data in XML, *ACM Trans. Database Syst.*, Vol.27, No.4, pp.438–493 (2002).
5) Online Computer Library Center: Introduction to Dewey Decimal Classification. http://www.oclc.org/
6) Atallah, M.(Ed): *Algorithms and Theory of Computation Handbook*, CRC Press (1998).
7) 森嶋厚行：XML 出版の技術，吉川正俊編 XML データベース特集，日本オペレーションズ・リサーチ学会誌，Vol.50, No.6, pp.379–384 (2005).
8) Shanmugasundaram, J., Kiernan, J., Shekita, E., Fan, C. and Funderburk J.: Querying XML Views of Relational Data, *Proc. VLDB 2001*, pp.261–270 (2001)
9) Shanmugasundaram, J., Shekita, E., Barr, R., Carey, M., Lindsay, B., Pirahesh, H. and Reinwald, B.: Efficiently publishing relational data as XML documents, *VLDB Journal*, Vol.10, No.2-3, pp.133–154 (2001).
10) Transaction Processing Performance Council: TPC-H (Decision Support for Ad Hoc Queries). http://www.tpc.org
11) W3C: Document Object Model (DOM). http://www.w3.org/DOM/

付　　録

A.1 実装した DOM 関数

- Node インタフェース
  - getFirstChild() ノードの最初の子ノードを取得.
  - getNextSibling() ノードの直下のノードを取得.
  - getNodeName() ノードの名前を取得.
  - getNodeValue() ノードの値を取得.
  - getChildNodes() ノードの子をすべて含む NodeList を取得.
- Document インタフェース
  - Document() コンストラクタ.

```
1.  void main() {
2.      Document doc = new Document(v);
3.      NodeList suppList
4.           = root.getElementsByTagName("supplier")(id1);
5.      int x=0;
6.      Node supplier=suppList.item(i)(id2);
7.      while(supplier!=null){
8.        Node suppinfo = supplier.getFirstChild()(id3);
9.        while(suppinfo!=null){
10.         if(suppinfo.getNodeName()(id4).equals("suppinfo")){
11.           break;
12.         }
13.         suppinfo=suppinfo.getNextSibling()(id5);
14.       }
15.       if(suppinfo!=null){
16.         Node name = suppinfo.getFirstChild()(id6);
17.         while(name!=null){
18.           if(name.getNodeName()(id7).equals("name")){
19.             break;
20.           }
21.           name = name.getNextSibling()(id8);
22.         }
23.         if(name!=null){
24.           show(name);
25.         }else{ break; }
26.       }else{ break; }
27.       x++;
28.       supplier=suppList.item(x)(id9);
29.     }
30. }
31.
32. void show(Node node) {
33.     for (Node n=node.getFirstChild()(id10); n!=null;
34.                  n=n.getNextSibling()(id11)) {
35.       System.out.println(n.getNodeName()(id12));
36.       System.out.println(n.getNodevalue()(id13));
37.       show(n);
38.     }
39. }
```

図 23 DOM プログラム $\mathcal{P}_4$
Fig. 23 DOM program $\mathcal{P}_4$.

```
1.  void main() {
2.      Document doc = new Document(v);
3.      NodeList suppList
4.           = root.getElementsByTagName("supplier")(id1);
5.      int i=0;
6.      Node supplier=suppList.item(i)(id2);
7.      while(supplier!=null){
8.        NodeList childList
9.                  = supplier.getChildNodes()(id3);
10.       Node part = childList.item(1)(id4);
11.       while(part!=null){
12.         show(part);
13.         part = part.getNextSibling()(id5);
14.       }
15.       Node suppinfo = childList.item(0)(id6);
16.       show(suppinfo);
17.       i++;
18.       supplier = supplierList.getNextsibling(i)(id7);
19.     }
20. }
21.
22. void show(Node node) {
23.     for (Node n=node.getFirstChild()(id8); n!=null;
24.                  n=n.getNextSibling()(id9)) {
25.       System.out.println(n.getNodeName()(id10));
26.       System.out.println(n.getNodevalue()(id11));
27.       show(n);
28.     }
29. }
```

図 24 DOM プログラム $\mathcal{P}_5$
Fig. 24 DOM program $\mathcal{P}_5$.

- getDocumentElement() 文書のルート要素になっている子ノードへの直接アクセスを可能にする．
- getElementsbyTagName(String s) 所定のタグ名とともに，すべての Elements の NodeList を返す．

- NodeList インタフェース
  - getLength() リスト内のノード数を返す．
  - item(int index) 集合内の index 番目の項目を返す．

A.2 論文中で利用した DOM プログラム

本論文中で利用した DOM プログラムを，**図 23**，**図 24** に示す．

(平成 17 年 6 月 21 日受付)
(平成 17 年 8 月 10 日採録)

(担当編集委員 天笠 俊之)

小島 章（正会員）

2003 年芝浦工業大学工業経営学科卒業．2005 年同大学大学院工学研究科電気工学専攻修士課程後期課程修了．現在，NEC ソフト株式会社勤務．

森嶋 厚行（正会員）

1993 年筑波大学第三学群情報学類卒業．1998 年同大学大学院工学研究科修了．博士（工学）．1998〜2001 年日本学術振興会特別研究員．1999〜2000 年 AT&T Labs-Research 客員研究員．2001 年芝浦工業大学工学部情報工学科講師．現在，筑波大学大学院図書館情報メディア研究科/知的コミュニティ基盤研究センター助教授．情報統合，XML/WWW データベース等に興味を持つ．ACM，IEEE-CS，電子情報通信学会，日本データベース学会各会員．

古宮 誠一（正会員）

1969 年埼玉大学理工学部数学科卒業．博士（工学）．1970 年（株）日立製作所入社．IPA 技術センター特別研究員，IPA 新ソフトウェア構造化モデル研究本部長付等を経て 2001 年より芝浦工業大学教授．2003 年より同大学専門職大学院（MOT）教授を兼務．1992〜1993 年/1994〜1995 年/1996〜1997 年知能ソフトウェア工学研究会幹事/副委員長/委員長．1996〜1997 年電子情報通信学会情報・システムソサエティ運営委員．1994〜1997，1998〜1999 年電子情報通信学会論文誌編集委員．1998〜1999 年電子情報通信学会論文誌編集委員会幹事．