差分によって記述されたXMLデータに対する効率的な検索方法

西村　雄介†　　田島　敬史††

† 北陸先端科学技術大学院大学情報科学研究科 〒923-1292 石川県能見市旭台 1-1
†† 京都大学情報学研究科 〒606-8501 京都市左京区吉田本町
E-mail: †y-nishi@jaist.ac.jp, ††tajima@i.kyoto-u.ac.jp

あらまし　本研究では，類似性の高い XML データが多数必要となるような応用において，これらのデータを共通部分の記述と差分の記述のみで表現し，データベースに格納する．格納されたデータは，検索するユーザーには差分を意識することのない検索を可能とし，更新するユーザーには共通部分と差分を明確に意識できる管理を可能とする．この機構により，データ量の縮小，共通部分の一括更新，類似データ間における違いが明確なことによる可読性の高さ等の利点が得られる．本研究で開発する機構では，ある XML 文書の一部として，他の XML の文書の全体または一部を取り込むことができ，さらに，その取り込むデータの一部を上書きすることができる．このような機構の実現において発生する問題の中で，本研究では，差分によって表現されているこれらのデータ群を，経路アプローチを用いて関係データベースに格納する方式について，三つの方式を提案し，比較検討を行った．

キーワード　XML，差分記述，XPath，問合せ処理

1. ま え が き

現在，XML フォーマットによるデータは様々な分野で用いられており，システム間のデータ交換用として使用されるだけでなく，用途によってはシステム内のデータフォーマットとしてもその利用が期待されている．とくに，従来の関係データベースなどによる定型的なデータ項目から成り立つデータではなく，データ項目が不定型なデータや，文書が混在するようなデータについて，XML はより有用である．

また，XML フォーマットが利用されている様々な応用の中には，扱うデータの中に極めて類似性の高いデータが多く含まれるような応用がある．製品仕様データを例にすると，一つの製品には「塗装色」だけが異なるようなバリエーションが多く存在し，それぞれ別の製品として認識されるが，そのほとんどのデータの内容は同じである．これを差分だけ，つまり関連する製品が記述されている部分を示すポインタと，異なる部分である「塗装色」を記述するだけで，バリエーションを登録することができれば，データ量を圧縮できるだけでなく，共通部分を一括に更新することができ，また，diff 等のツールを使用することなく類似データ間における違いが明確になり可読性が向上する

本研究では，このような差分を用いた表現を実現するための枠組みと，この枠組みで表現されたデータを管理するための機構を提案する．われわれの枠組みでは，差分の表現として，他の XML の文書の全体または一部を参照するだけでなく，その取り込むデータの一部を差分で上書きすることができる．本研究ではこのような表現を記述するための書式を定義し，この書式によって記述されたデータをを関係データベースに格納するための手法について提案する．関係データベースに格納されたデータは，検索するユーザーには差分を意識することのない検索が可能であり，更新するユーザーには共通部分と差分を明確に意識できる管理が可能になる．

関係データベースへの格納形式としては，スキーマ定義がない XML データにも対応が可能な経路ベースアプローチ [8] を採用する．ノードのラベリングには更新に強い dewey オーダーを使用する [7]．検索については XPath [2] に基づいたものを，更新については XUpdate [6] で可能な機能を考慮する

差分による記述を格納する手段については，いくつかの選択肢が考えられる．

まず，参照や差分を記述した XML データを格納する一方で，参照や差分を全て解釈した完全な XML データも同時に格納する方法が考えられる．この方式の場合，問合せは通常の XML 文書と同様に扱うことができる．しかし，参照する要素の子孫要素のデータ量が大きい場合は，データベース全体のデータ量が増加する．また，データの更新があった場合には，その箇所を参照しているような要素の全てを更新しなければならないため，更新時の負荷が高くなる．

また，逆に差分の記述を解釈せずに格納し，問合せ実行時に参照関係を解釈しながら問合せの評価を行う方法もある．この場合，更新は通常の XML 文章と同様に行うことができるが，問合せは文章間の参照関係をたどりながら行わなければならないため，効率が悪くなる．ただし，差分で記述した XML 文書以外には何も生成しないためデータ量は小さい．

そこで，本研究では，これら二つの方式に加えて，差分記述の解釈を途中まで行った中間結果を保存し，これを索引として使用する中間的な方法も併せて提案する．この中間結果とは参照と差分の関係のみを記述した木構造の索引である．各ノードは元の文書の要素とリンクするための値と，その要素に到達するまでのパス式を記述する．この索引は，参照と差分の関係のみを格納しているため，それ以外の一般の要素を変更を行ったとしても，参照されている要素でなければ影響を受けない．また，参照関係はすでに解釈され，存在しうるパターンは絞られ

ており，XPath による問い合わせを簡略化できる．よって，検索処理についても実用的な速度で可能になることが期待できる．

本研究では，この三つの手法を比較し，検証する．

2. 関連研究

XML 文書の一部を参照する記法はすでに w3c からいくつかの規格が提案されている．最も広く利用されているものは, [4] の 3.3.1 Attribute Types にあげられている ID 型と IDREF 型を用いる方法である．ID 型の属性に属性値として ID の値をもたせ，その ID の値を持つ要素の内容を参照できるようにする．これにより重複したデータの記述を防ぐことができる．

次に, [4] の 4.2.2 External Entities に示されている External Parsed Entity（外部解析対象実体）を使用する方法がある．文書型宣言 (DTD) に URI を記述し，特定の要素へ外部に存在している XML 文書の内容全体を取り込むというものである，これらは, [4] の 4.1 Character and Entity References に示されているように，本文からは Entity References（実体参照）という記法によって参照する．解析対象実体であるため，置換された文書で解析が行われ，妥当性のチェックが行われる．

また，2004/12/20 に勧告が出された XInclude という新しい規格 [5] がある．これは，外部にある XML 文書または一般のテキストを XML 文章中に取り込むことができるようにするためのものであり，XPointer [3] を用いて参照先の XML 文章の一部を取得することや，外部参照が存在しないときの代替テキストを記述することが可能である．XInclude は解析とは直交した概念であり，解析段階では外部の文書のマージが行われることはなく，置換された文書で妥当性のチェックが行われることはない．

上記のいずれの方法も，XML データの範囲を指定し，それをそのまま使用することによって，データを生成する．一方，本研究では取り込む内容の一部についてさらに差分を記述してその部分だけ上書きして取り込むことを可能とする．また，本研究では，スキーマ定義を行わない XML 文章を対象としているため，妥当性のチェックに関しては考慮していない．そのため，XInclude と同様に参照先が存在しない参照が存在した場合の代替文書を記述できるようにしている．

3. 提案手法

3.1 xi:include

xi:include 要素は，その要素が別の要素によって置き換えられることを示す．ただし，置き換えと呼ぶと後述の xi:overwrite と曖昧になるため，以下，この処理自体は「取込み」と呼ぶことにする．置き換えが行われる別の要素とは，XML として存在する特定の要素であり，この要素に定める特別な属性を記述することで，その要素を指し示す．この指し示された要素を参照先要素とよび，そのときに参照先要素を指し示している xi:include 要素を参照元要素とよぶ．

この要素には特別な属性として，xi:href 属性または xi:idref 属性が定められており，このいずれか，または両方を記述し，参照先の要素を指し示さなければならない．それ以外の属性を記述した場合は，取り込まれた要素の属性として扱われる．

xi:href は XML 文書に関連づけられている URI を記述する．省略を行った場合は自らの文書を参照しているとみなす．xi:idref 属性は上記 URI 文書の一部を示すために，xi:id 属性によって定められている該当文書の唯一の要素を指し示す．この要素により複数の箇所を参照し，兄弟要素が増加すると，この要素以降の兄弟要素の Dewey オーダーに影響を与えてしまう．よって，同じ ID の要素が XML 文書内に複数，存在してはならない．id 型をもった属性であればどのような属性名であっても，idref 属性の型を持った属性は指し示すことが可能であるが，型という概念が存在しない整形式の XML 文書では不可能である．よって，xi:idref 属性が指し示すことができる要素には namespace を使って特定の属性名を使うことにする必要がある．本論文では，xi:id という属性名で記述する．

参照先の要素名を変更する場合は xi:name 属性を指定する．指定のない場合は，参照先の要素名を xi:name 属性の値とする．また，xi:idref 属性の記述を省略した場合は文書全体，すなわちドキュメントノードを指し示しているとみなす．

xi:include 要素の子要素には以下に述べる xi:overwrite 要素または xi:failback 要素のみを記述することができる．参照先の要素の子要素に xi:include が含まれていた場合も取込処理が行われる．この場合は，先に参照先の xi:include を処理する．これを繰り返しているうちに，参照先がすでに参照した要素，またはその祖先要素にあたる場合は永久ループしてしまうため，これを認めない．

3.2 xi:overwrite

xi:overwrite 要素は，xi:include によって参照先の要素を取り込む際に，その取り込む内容の一部を変更するためのものである．以下，この処理は「上書き」と呼ぶことにする．

xi:overwrite 要素は必ず xi:include 要素の子要素でなければならず，一つの xi:include 要素につき 0 回以上の記述が可能である．xi:overwrite 要素には上書きする要素の名前を示す element 属性を記述しなければならない．参照先要素の子要素に xi:include が含まれていた場合は，先に子要素の xi:include の取り込みを行った後に，xi:element 属性の要素を取得し，上書きを行う．xi:element が示す要素が存在しない場合は，この xi:overwrite 要素による上書きそのものを行わない．この場合は処理可能な文書であると見なす．xi:element 属性による複数箇所の指定を認めるが, [4] にあるように，整形式の制約として同一名の属性名が認められないということが定められている．よって，属性を 2 回記述するのではなく，値をカンマで区切り，複数の要素名を記述できるような書式で実現する．

xi:overwrite 要素の子要素には上書きする内容にあたる XML 文書を記述する．ただし，ここに記述できるのは element 属性で指定された要素の子要素の内容である．これも include の場合と同様に，上書きを行うことによって，その上書きされた要素の兄弟要素が増加することによる Dewey オーダーへの影響を避けるという理由のためである．xi:include と同様に，上書きを行う要素に属性を追加したい場合は xi:overwrite の属性として記述すればよい．element 属性で示されている要素の要素名

自体を変更する場合は，name 属性に新しい要素名を記述する．name 属性を省略した場合は element 属性に記述されている属性名を使用するものとする．なお，xi:overwrite の子要素の中に xi:include を記述することも認める．また，後述する xi:failback による取り込みが行われた場合においても xi:overwrite が行われるものとする．

3.3 xi:failback

xi:failback 要素は必ず xi:include 要素の子要素でなければならない．一つの xi:include 要素につき 0 回または 1 回の記述が可能である．親要素の xi:include による参照先が特定できない場合はこの要素の子要素で置き換えが行われる．xi:failback が記述されていない場合は，改行コードを持つテキストノードが xi:failback として登録されていると見なす．

xi:failback 要素には新しい要素の名前である xi:name 属性を設定することができる．設定がない場合は，xi:include と同じ値を使用するものとする．

なお，xi:failback 要素の子要素に xi:include 要素を記述することも認める．

3.4 仮想的に生成される要素のパス構成

取込や上書きの処理によって仮想的に生成される要素がもつ，該当文書のルートノードからのパス式の構成を示す．パス式はルートノードから順番に該当ノードまでの要素名を羅列したものであり，要素名ごとの区切り文字に'/' を使用して，一つの文字列にしたものである．

仮想的に生成する前の段階の文書が図 1 に示す内容であったとする．このとき，A，B，C，D，Z の大文字で示した部分についてはパス式を示し，x，e，o，f の小文字による部分は単一の要素名を示す．パス式の部分は存在しない場合があり，その場合はそのパス式の前に付いている区切り文字も含めてそのパス式より除く．例えば，'/A/a/b' というパス式において A が存在しない場合は'//a/b' ではなく'/a/b' となる．

要素名またはパス式	例
xi:include の要素のパス式	/A/xi:include
xi:include の xi:name 属性の値	x
x の要素の子孫要素のパス式	/Z/x/B
上書きする要素のパス式	/A/xi:include/xi:overwrite/C
xi:overwrite の xi:name 属性の値	o
xi:failback の xi:name 属性の値	f
xi:failback の子孫要素のパス式	/A/xi:include/xi:failback/D

図 1 生成する前の文書の要素名，パス式

元データにおける要素名やパス式が図 1 のようであったとすると，仮想的に生成される要素のパス式は図 2 のようになる．

番号	生成される要素	パスの構成
1	参照だけを行い，上書きが行われない要素	/A/x/B
2	上書きを行う要素	/A/x/B/o/C
3	failback による要素	/A/f/D

図 2 仮想的に生成される要素のパス式

図 3 と図 4 の例では，Product alpha という製品の基本モデルと，青い外装のモデル，さらにそのエンジン違いのモデルの 3 つの製品を表している．xi:include による取込処理を行った後の，完全な形の XML 文書が図 18 である．この XML を木構造で表したものが図 5 である．この図では，xi:include は「取込」というノード名で表し，参照先要素を点線で示した．xi:overwrte はどの要素を上書きするのかをノード名で示した．

```
<?xml version="1.0">
<items>
  <item xi:id="100">
    <name>Product alpha</name>
    <model>normal</model>
    <exterior>
      <color>red</color>
    </exterior>
    <engine>
      <xi:include
        xi:href="engine.xml"
        xi:idref="300" />
    </engine>
  </item>
  <xi:include xi:idref="100" xi:id="101">
    <xi:overwrite xi:element="model">
      blue model
    </xi:overwrite>
    <xi:overwrite xi:element="color">
      <main>blue</main>
      <sub>black</sub>
    </xi:overwrite>
  </xi:include>
  <xi:include xi:idref="101" xi:id="102">
    <xi:overwrite xi:element="model">
      blue limited model
    </xi:overwrite>
    <xi:overwrite xi:element="engine">
      <xi:include
        xi:href="engine.xml"
        xi:idref="400" />
    </xi:overwrite>
  </xi:include>
</items>
```

図 3 car.xml

このとき，青いモデル (xi:id=”101”の item) の主な色を示す main 要素は，xi:include による取り込みが行われると，/items/item/exterior/color/main というパス式を持つことになる．このとき items が A,item が x，B は exterior，o は color（name 属性は存在していないが，存在していない場合のデフォルトは element 属性の値としているため），C は main

```
<?xml version="1.0">
<items>
  <item xi:id="300">
    <name>Engine alpha</name>
  </item>
  <item xi:id="400">
    <name>Engine beta</name>
    <turbo />
  </item>
</items>
```

図 4　engine.xml

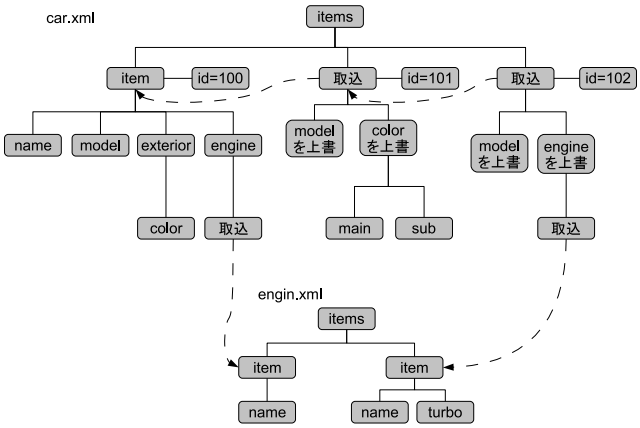

図 5　car.xml, engine.xml を木構造で表現した図

になる．

また，上書きが行われていない要素の例として，xi:id=”101” の item における name 要素について示す．この要素のパスは/items/item/name であるが，この場合は items が A,item が x, name が B となり，C は上書きを行わないので存在しない

次にネストした取込について考える．本研究におけるネストした取込とは，xi:include による取り込みが行われた結果の要素の中に xi:include が含まれていることをいう．よって，取り込み対象の参照先要素，上書きする要素である xi:overwrite 要素の子孫要素，xi:failback の要素の子孫要素の中に含まれていることが考えられる．

先述の xi:overwrite の記法の節で述べたように，取り込み対象の参照先要素に xi:include が含まれていた場合は，まず取り込み対象に存在する xi:include の取り込み処理を行ってから，xi:overwrite の上書きを行い取り込みを行う必要がある．図 3 の XML 文書において xi:id=”101” の item は xi:id=”100” の item 要素を参照している．この xi:id=”100” の要素の子孫要素には xi:include が含まれている．この場合は，まず xi:id=”100” の xi:include の取り込みを行ってから，xi:id=”101” を取り込まなければならない．

記法の定義で示したように，参照先の要素に存在する xi:include によって取り込まれた要素について xi:include による記述がある限りは何度も繰り返して，参照先に xi:include による要素がなくなるまで行う．よって，参照元から見ると常に，xi:include が存在しない要素が取込対象となるため，図 1 で示したところの B の一部と考えられる．実際には後ろに連結する形になるため，取り込む要素を R とした場合は B/R という形になる．

3.5　仮想的に生成される要素の dewey オーダー構成

次に，仮想的に生成される要素について，dewey オーダーは以下のような手順で生成することができる．

パス式の表記と同様に文字に置き換えて表現する．すなわち，「1.2」など，ピリオドによって分割された複数の順序から構成されるものを A,B,C のように大文字で，必ず存在する一つの順序を x のように小文字で表すとする．生成する前の文書の dewey オーダーを以下のように表現する．

要素	dewey オーダー
xi:include の要素	A
参照先要素	X
X の子孫要素	X.B
xi:overwrite の要素	A.y.C
xi:failback の要素	A.z.D

図 6　生成前の文書の dewey オーダー

図 6 について補足する．xi:overwrite の dewey オーダーにおける y は xi:include 内における xi:overwrite の記述の順序である．xi:overwrite は xi:include の 1 レベル下の要素でなければならないため，このような dewey オーダーになる．また，この順序は意味を持たないため，使用することはない．xi:failback の dewey オーダーにおける dewey オーダーも同様である．

図 6 によって示された dewey オーダーを使用すると，新たに生成される dewey オーダーは図 7 のように表現できる．

要素	dewey オーダー
上書きされない要素	A.B
上書きする要素	A.B.C
failback による要素	A.D

図 7　新たに生成される要素の dewey オーダー

図 3 の例を使って具体的に説明する．図 3 の dewey オーダーが図 8 であったとする．

番号	要素	dewey
1	`<xi:include xi:idref="100" xi:id="101">`	1.2
2	`<item xi:id="100">`	1.1
3	`<model>`	1.1.2
4	`blue model`	1.2.1.1
5	`<xi:include xi:href="engine.xml" xi:idref="400">`	1.1.4.1

図 8　図 3 の dewey オーダーの例

このとき，青いモデル（xi:id=”101”）の model の名前を示す blue model というテキスト要素の Dewey オーダーを求め

る．blue model というテキスト要素は,「上書きをする要素」に相当するため，図 7 より A.B.C という Dewey オーダーの構成になっている．まず，A は図 6 にあるとおり，xi:include が記述されている要素の dewey オーダーの値に相当する．これは，図 3 の 1 番で示されている要素であるので，この値より A=1.2 となる．同様に，X も該当する箇所のオーダーをそのまま使用するため図 3 の 2 番の値になるから，X=1.1 となる．B は，xi:overwrite の対象となっている要素の dewey オーダーから求める．この場合は，上書きは<model>という要素に対して行われているため，図 3 の 3 番の dewey オーダーを使用する．この要素は図 6 より X.B であるため，X の順序以降を取得すればよい．よって，B=2 となる．C の取得は xi:overwrite の子要素，すなわち上書きする内容である blue model というテキスト要素を使用する．これは 8 の 4 番の dewey オーダーに相当する．この要素の Dewey オーダーは図 7 より A.y.C という構成になっている．A=1.2 であることはわかっている．y は次の 1 レベルなので，y=1 であり，C は最後の順序を取得することになり，C=1 ということになる．

以上，最終的に生成される dewey オーダーは A.B.C であるから 1.2.2.1 となる．

ネストした取込が存在する文書はパス式と同様に考えられる．つまり，それぞれ B, C, D の順序について後ろに連結する．

3.6 格納方法

紙面の都合上，各方式について概略を述べる．各方式ごとに使用する関係スキーマや問い合わせを行う SQL の例，更新手順については [9] を参照のこと．

3.6.1 方式 1：完全な XML 文書化を行い格納する方法

まず，一番単純な方法はデータベースに登録する際に参照を含む文書をそのまま格納する一方で，参照による取り込みによって生成された完全な XML 文書を別に生成し，データベースに登録する方法である．この完全な XML 文書を索引文書とよぶ．この方法は，問合せについては何も考慮を行う必要がなく，データ量が増加するものの，他の方法のように何度もテーブルを問合せする必要がないため，最も高速に処理することが期待できる．しかし，元の XML 文章に一部修正を加えた場合は，特別な機構を持っていなければ，索引文書を完全に登録し直す必要がある．再計算のコストは一般に大きいと考えられるため，更新のための特別な機構について考える必要がある．

そこで，差分の文書の要素と索引文書との要素のリンクを考える．このとき，索引文書内の取り込みによって生成された要素と，差分の文書である取り込みの指示が記述された要素では，Dewey オーダーが一致するため，容易に取得できる．しかし，取り込み時に参照を行った要素に対してはリンクを持っていないため，参照先が変化した場合に再度，検索を行わなければならない．そこで，参照先の要素と取り込みによって生成された要素について関連づけを行う．

また，索引文書には XML の要素だけが必要であり，テキスト値はパス式と同様に差分文書と共有することができる．よって，テキストノードの値は差分文書の値を使用するようにすることにより，データ量を削減することができる．

3.6.2 方式 2：問合せ時に参照を行う方式

つぎに xi:include を含む文章をそのまま登録し，問合せ時に xi:include 以下の要素を参照して問合せする方法である．更新に関しては，登録時に何の解釈も行っていないため，負荷は最小に抑えられる．データ量についても小さく抑えられるが，後述するように，XPath 式による問い合わせについて，それを経路ベースアプローチによるパス式のテーブルと単純に比較できず，取り込みや上書きを考慮に入れながら検索を行わなければならない．このための検索パターンが増加する問合せに必要な手続きが多いため，データ量が少ないにもかかわらず速度が低下する可能性が高い．

例えば，ネストした取込および failback を考慮せず，XPath の問い合わせが/a/b/c であった場合，図 2 に適用すると，図 9 の六種類が考えられる．

生成する要素	A	x	B	o	C
上書きしない要素		a	b/c		
	a	b	c		
	a/b	c			
上書きする要素		a	b	c	
		a		b	c
	a	b		c	

図 9 /a/b/c という問い合わせについて考えられるパターン

よって，xi:include を全く解釈せずに格納する方式では，問合せ時に，これら六種類のパターン全てについて問合わせを実行する必要がある．このように XPath の問い合わせ式において前提となるレベルが n レベルの場合，一般に $\sum_{k=1}^{n} k$ 個のパターンが考えられる．

ネストした取込が存在する場合は，先述したように B または C の後ろにつく形で存在する可能性がある．図 9 を例にすれば，B=b/c のパターンについて後ろの c がネストによって取り込まれた c の可能性がある．ただし，ネストによる対象が B 全体，および C 全体の場合は，何回ネストしているのか上限がわからないため，すべてのパターンで検索し尽くすことは難しい．よって，この方式を採用する場合はネストを認めないか，あるいは B 全体，C 全体のネストを認めないようにする必要がある．

xi:failback が存在する場合は，図 9 の x を f, B を D に変更したパターンを追加する．よって，存在する場合を考慮するとパターンは 2 倍多くなることになる．

3.6.3 方式 3：途中結果を格納する方式

最後に，上記二つの中間をとる方式がある．この方式では，参照関係によって生じる仮想的な要素に関する検索を行う場合に，その参照関係を問い合わせ時に辿ることはせずに，参照関係を木構造で表した索引を該当の文書の格納時に作成する．この索引は，参照に関する要素，すなわち，xi:include と xi:overwrite に関する要素のみを持つ．この要素には，その参照要素に至るパス式とその先を示す xi:include の参照先，または xi:overwrite の上書き内容の要素と被上書き内容の要素の id が記述されている．これによって，XPath による問い合わせが

「索引に含まれるパス＋その子孫要素」のパターンによる検索でよいため，検索の回数は減少し，かつ更新は参照関係に関しない部分において影響を及ぼさない．

例として，図 10 に，図 3 の xi:id=”102” の item に関する索引を示した．各ノードは xi:include または xi:overwrite に対応しており，ノード名にその要素までのパスと，その要素が指し示している要素の内容を記述した．xi:overwrite の場合は，上書きする要素（つまり，これは xi:overwrite 自身）と，上書きされる要素のそれぞれについて記述した．

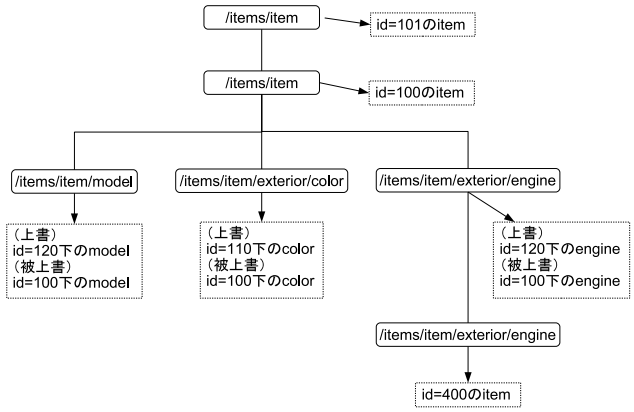

図 10　car.xml の id=102 に関する方式 3 の索引を木で表した図

4. 評　　価

前章までで述べた問合わせ手法を用いて，問合せ時に参照を解決する方式と，途中結果を索引として格納しておく本研究の提案方式での，問合せの処理効率の比較実験を行った．

なお，使用した SQL や更新処理の詳細については [9] を参照のこと．

4.1 実験環境

実験用データとしては，XMark Version0.96 [1] を用いて生成したデータに対して，以下のような実験を行った．なお，データ量を表すスケールは 0.02 であり，ノード数は，100,905 である．

なお，実行環境は以下の通りである．

- SunBlade1500 (UltraSPARCIII 1.5GHz メモリ 1GB)
- Soralis 9
- PostgreSQL Version 7.4.3

4.1.1 実　験　1

XMark により生成された文書には，オークションの商品一覧を示す item 要素を `/site/regions/(region)` の子要素として記載している(注1)．item には商品名や支払い方法のほかに，どの国のものなのかを示す location 要素を子要素として持っている．一方，オークションの情報は `/site/open_auctions/open_auction` にあり，開始価格や現在の価格，商品などをの子要素として持つ．ただし，商品の要素には重複記載を避けるため，itemref 要素を使用して参照を行っている．

この `/site/open_auctions/open_auction/itemref` 要素に換えて，item 要素を参照する xi:include を記述し，item 要素の子要素がオークション情報に含まれているかのように検索できるようにする．このとき，`/site/open_auctions/open_auction/item/location` という XPath による問い合わせを行った場合の時間，およびデータ量を比較する．itemref 要素は 435 カ所存在しているため，xi:include は 435 要素存在していることになる．

結果を図 11 に示す．方式 1 は，途中結果を格納する場合と比較して，キャッシュがない場合でも 3 倍程度高速に検索できた．一方，方式 2 は，15 パターンの検索について，途中に経由する参照も含めすべて検索を行っているため，20 秒程度時間がかかっている．方式 3 は，問い合わせ時に参照を行う方式よりも格段に高速に処理できているが，完全に XML の展開を行う方式には及ばない．なお，iPath とは方式 3 で提案した索引であり，このデータ量を示している．

方式	検索時間 (s)	cashless 検索時間 (s)	Element(M)	iPath(M)
方式 1	0.023	0.652	88.984	0.000
方式 2	19.187	20.578	36.184	0.000
方式 3	0.660	1.783	36.184	0.360

図 11　実 行 結 果

4.1.2 実　験　2

実験 1 で使用した文書に，person 要素という商品を参照する xi:include を 1 カ所加える．具体的には，`/site/open_auctions/open_auction/annotation` の子要素に person 要素への xi:include を記述し，`/site/open_auctions/open_auction/annotation/author` という XPath による問い合わせを行う．

結果を図 12 に示す．方式 3 が 9 倍程度高速に検索できるようになったが，一方で方式 1 はキャッシュの影響でさらに高速に処理できている．

方式	検索時間 (s)	cashless 検索時間 (s)
方式 1	0.020	0.149
方式 2	11.846	13.045
方式 3	0.0077	0.728

図 12　実 行 結 果

4.1.3 実　験　3

実験 1 について，xi:include に関係しない要素への検索を行う．具体的には `/site/regions/europe/item/mailbox/mail/to` という問い合わせを実行した．

結果を図 13 に示す．方式 2 はロケーションステップに応じて検索パターンが増えるため，比例して時間が掛かるが，方式 3 は方式 3 で使用している索引に該当するデータが存在しないため，高速に処理できた．

(注1)：(region) の部分には africa, asia など地域の名称が入る

方式	検索時間 (s)	cashless 検索時間 (s)
方式 1	0.015	0.164
方式 2	26.989	29.240
方式 3	0.200	0.274

図 13　実 行 結 果

4.1.4 実　験　4

上書きに関する検索を行う．具体的には図 14 のような XML を作成し，実験 1 と同様の問い合わせを実行する．

```
<?xml version="1.0">
<xi:include href="testdata.xml">
  <xi:overwrite element="from,  to" >
    All E-Mail adresses are masked.
  </xi:overwrite>
</xi:include>
```

図 14　メールアドレスのマスクをするための XML

なお，from 要素および to 要素は併せて 900 箇所に存在している．この XML 文書は実験 1 の XML 文書全体を参照しているため，参照先に実験 1 で追加した xi:include が存在していることになる．よってネストした取込を 1 度行わなければならない．図 15 に実験結果を示す．方式 2 は今回の実験ではネストした検索の対応ができなかったため，計測ができなかった．方式 1 は，実験 1 とほとんど検索時間に違いが見られなかった．方式 3 の場合，overwrite の項目が増加したために，iPath 表の大きさに比例して検索時間が増加した．ただし，データ量は差分を記述した文書と iPath 表のみ持てばよいため，大幅に圧縮できた．

方式	検索時間 (s)	Element(M)	iPath(M)
方式 1	0.170	122.072	0.000
方式 2	計測不可	-	-
方式 3	1.232	36.192	0.464

図 15　実 行 結 果

4.1.5 実　験　5

name 要素にも上書きを行うように変更するため，実験 2 の xml の内容を図 16 に変更した場合の処理時間を比較する．処理時間は，更新に必要な SQL の処理時間の総和とする．

処理時間を総和した値を図 17 の表にまとめた．方式 1 は，追加した XML 文書がそれほど大きなものではなかったにもかかわらず，方式 3 と比較しても 3 倍以上時間がかかっている．これは，上書きを行うためには，すでに存在する要素を削除し，新しい文書を追加する必要があるためである．方式 2 は，差分文書への追加を行ったのみであり，dewey オーダーの変更もなかったため，極めて高速に処理できた．方式 3 は上書きする要素の数ではなく，被上書き要素の箇所の数と同じだけの追加処理が必要なため，方式 2 よりも大幅に時間を消費した．

```
<?xml version="1.0">
<xi:include href="testdata.xml">
  <xi:overwrite element="from,  to" >
    All E-Mail adresses are masked.
  </xi:overwrite>
  <xi:overwrite element="name">
    Mr.X
  </xi:overwrite>
</xi:include>
```

図 16　overwrite を追加した XML

方式	処理の積算時間 (s)
方式 1	66.492
方式 2	0.079
方式 3	16.132

図 17　実 行 結 果

5. ま と め

本論文では，類似する XML データを大量に扱うような応用において，共通部分と差分の記述を用いて，そのようなデータを記述できる枠組みについて提案し，また，そのような形で記述されたデータをデータベースに格納する手法について提案し，それぞれを検索と更新の両面から評価を行った．

実験の結果，方式 1 は検索は最速であるが，一方で，データ量が大きくなり，かつ更新面における速度は，極めて遅いという欠点がある．一方で，テキストノードの値の変更については，差分で記述された文書と索引文書が同じ値を共有しているため，方式 2 や方式 3 と同じ処理手続きでよい．よって，更新でも，テキストノードの変更がほとんどを占めるような応用に適しているといえる．

方式 2 は，全く逆の性質を持ち，データ量，更新速度ともに良好だが，検索処理が現実的に使用可能な範囲を超えてしまっている．これは，今回の差分記述の自由度の高さが問題になっているとも考えられ，より制限した差分記述にすれば実用的な速度が期待できる可能性がある．

方式 3 では，データ量を大幅に圧縮することが可能であり，とくに，索引による絞り込みがきく場合と，索引を経由しない検索については比較的高速な速度が期待できる．また，取込に関わる要素である，取り込み指示要素，被取り込み要素，上書き指示要素，被上書き要素 への変更がなければ，手法 2 の処理と同じく，ほとんど更新の負荷が掛からない．しかし，検索処理では方式 1 と比べ 3 倍以上時間がかかっている．とくに，問い合わせが方式 1 よりも複雑なため，キャッシュが存在しても速度が向上しにくい．データ量の圧縮を行いたい場合，および更新を頻繁に行うような文書の格納に適しているといえるだろう．

今後の課題として，実験においては，より大規模な文書や，より複雑な構成になっている XML 文書についての検証を行

う必要がある．例えば，XPath における問い合わせについても，述語 (prediaction) を使用した問い合わせや多様な軸 (axis) を使用した問い合わせついて，検証を行う必要がある．dewey オーダーが構成できることを示したため，問い合わせの多くは可能であると考えられるがまだ明らかでない．同時に，複雑なネストによる取込が XML 文書に存在した場合の更新処理への影響についても検証を行わなければならない．とくに，方式１は再帰的な記述が行われていた場合に，連鎖的な更新が必要であり，速度への影響が懸念される．

機能面においては，テキスト要素の内容に関する差分を正規表現で記述できるような枠組みがあれば，使用用途は増加すると思われる．方式３であれば，正規表現と置換後の結果を索引に登録し，差分を表現することができる．

文法面においては，いくつかの拡張が考えられる．たとえば，上書きだけでなく削除が必要な応用が多く考えられるため，このような記述ができるように文法を考慮すべきである．現在は実験５で示しているような形で実現し，削除による他の要素への dewey オーダーへの影響を避けているが，dewey オーダーの順序の間に抜けを認め，XPath による一部の問い合わせを考慮しなければ，このような機能も考えられる．また，上書きの指定について element による要素名でなく XPath による指定があれば仕様用途が広がると考えられる．しかし，方式２による問い合わせでこれを解釈することは難しい．

最後に，どのような差分にすれば効率がよいかという議論がある．この議論については今回の論文では全くふれることができなかったが，多くの応用で検討しなければならない課題だろう．

文　　献

[1] *XMark An XML Benchmark Project.* http://monetdb.cwi.nl/xml/index.html.

[2] World Wide Web Consortium. *XML Path Language(XPath) version 1.0.* 1999. http://www.w3.org/TR/xpath/.

[3] World Wide Web Consortium. *XPointer Framework.* 2003. http://www.w3.org/TR/2003/REC-xptr-framework-20030325/.

[4] World Wide Web Consortium. *Extensible Markup Language (XML) 1.1.* 2004. http://www.w3.org/TR/xml11/.

[5] World Wide Web Consortium. *XML Inclusions (XInclude) Version 1.0.* 2004. http://www.w3.org/TR/xinclude/.

[6] The XML:DB Initiative. *XUpdate Working Draft.* http://xmldb-org.sourceforge.net/xupate-wd.html.

[7] K.S.Beyer J.Shanmugasndaram I.Taminov, S.Viglas. *Storing and querying ordered XML using a relational database system.* 2002.

[8] Toshiyuki Amagasa Masatoshi Yoshikawa. *XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Realational Database.* 2001.

[9] 西村雄介（修士論文）. 差分によって記述された XML データの格納検索方式. 2006. http://www.jaist.ac.jp/library/thesisdb_html/simppdf.html.


```
<?xml version="1.0">
<items>
  <item xi:id="100">
    <name>Product alpha</name>
    <model>normal</model>
    <exterior>
      <color>red</color>
    </exterior>
    <engine>
      <item xi:id="300">
        <name>Engine alpha</name>
      </item>
    </engine>
  </item>
  <item xi:id="101">
    <name>Product alpha</name>
    <model>blue model</model>
    <exterior>
      <color>
        <main>blue</main>
        <sub>black</sub>
      </color>
    </exterior>
    <engine>
      <item xi:id="300">
        <name>Engine alpha</name>
      </item>
    </engine>
  </item>
  <item xi:id="102">
    <name>Product alpha</name>
    <model>blue limited model</model>
    <exterior>
      <color>
        <main>blue</main>
        <sub>black</sub>
      </color>
    </exterior>
    <engine>
      <item xi:id="400">
        <name>Engine beta</name>
        <turbo />
      </item>
    </engine>
  </item>
</items>
```

図 18　取り込み後の完全な形の car.xml