JAIST Repository
https://dspace.jaist.ac.jp/

Title	差分によって記述されたXMLデータの格納検索方式
Author(s)	西村，雄介
Citation	
Issue Date	2006-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1993
Rights	
Description	Supervisor:田島　敬史，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

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

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

西村　雄介

2006年3月

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

指導教官　田島 敬史 客員助教授

審査委員主査　田島 敬史 客員助教授
審査委員　日比野 靖 教授
審査委員　片山 卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

410095 西村　雄介

提出年月: 2006 年 2 月

Copyright © 2006 by Nishimura Yusuke

概 要

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できるような機構を提案する．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，経路ベースアプローチを用いて関係データベースに格納し，XPath による問合せを効率良く実行するための手法について検討する．

目 次

第1章　まえがき

1.1　背景

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．

具体的に，製品仕様に関するデータを例に説明する．製品仕様の項目は製品の種類，機能による違いで異なることがあり，また将来的にも項目が増加，あるいは逆に減少することが起きやすい．また，注釈などの文章を項目の中に併記するなど，属性や桁などを一律に合わせた固定的なスキーマ定義が困難な場合が多い．

固定されたスキーマを用いて製品ごとに仕様項目を属性とするようなテーブルを作成するとテーブル数が爆発的に増えてしまう．また，製品が新たな機能を持ち，仕様項目が増えるたびにテーブル定義を変更することは，あまり現実的ではない．固定的なスキーマをもたない XML フォーマットを使用することで，これらの製品仕様データをほぼそのままの形で表現でき，かつ，経路ベースアプローチなどスキーマ定義が必要でない関係データベースへの格納手法を採用することにより，効率良く検索することができる．

一方で，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．

差分による記述は，可変的な項目が必要なデータに使用可能なだけでなく，マニュアルなどの一般的な文書においても，類似性が高い文章を大量に生成しなければならない場合においては有用である．例えば，共通的に使用される部分と製品ごとに異なる部分を別に記述することができれば，無駄な重複が減り，マニュアルに欠陥があった場合の修正が容易になる．また，製品について選択が可能なオプションの組み合わせが極めて多い場合などにおいて，オプションごとに差分のみ記述することができれば，あらかじめすべての組

み合わせについて完全な XML 文書を作成する必要はなく，顧客が選択したオプションなどにより注文に応じたユーザーマニュアルを，共通する内容をもつ文書へのポインタと，オプションによって異なる部分を記述する部分だけで表現することができる．

1.2 目的

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，そ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．本論文では特に，これらのデータ群を経路ベースアプローチを用いて関係データベースに格納し，XPath による問合せを効率良く実行するための手法を提案する．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取り込み，かつ，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．関係データベースへの格納形式としては，スキーマ定義がない XML データを対象とするため，基本的には経路アプローチを採用する．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．

差分による記述の部分をどのように格納するかについては，いくつかの選択肢が考えられる．

まず，差分の取込みや上書きを全て解釈して，完全な通常の XML データとしてから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 文章と同様に扱えばよいが，多くのデータから取り込まれているデータがある場合には，同じデータのコピーが多数生成されるため，データ量が増加し，また，データの更新があった場合には，その部分を取り込んでいるようなデータ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．このとき，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，データ量を少なくすることが可能であり，先述した顧客ごとのマニュアルを作成するような例の場合は，XML 文章が注文件数に応じて大量に生成されるため極めて有用である．

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．この手法では，主に XPath による検索を考慮し，取込みによって生成される仮想的な（実際には存在しない）文書について，

本研究では，前述の二つの手法と提案手法の三つの手法を比較し，検証する．

1.3 構成

まず，第２章では本研究に類似する手法，および本研究で使用する関連研究についていくつか述べる．

第３章では，提案手法として，まず差分の記述記法を定義し，差分を記述した場合に仮想的に生成される要素についてパス式および Dewey オーダーがどのような形で構築しなければならないか述べる．なお，パス式および Dewey オーダーは本研究が採用する「経路ベースアプローチ」における XML の構造を示す値である．つぎに，それに基づいて三種類の格納手法のそれぞれについて関係スキーマと検索，更新の処理の内容について述べる．

第４章では，三種の格納手法について評価を行い，それぞれの格納手法における特徴をのべ，最後の章で本研究のまとめを述べる．

第2章　関連研究

2.1　XML文書の一部を参照する既存の記法

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，[5] 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．[5] の Validity Constraint によれば，IDREF 型の値は常に同一文章内の ID 型の値について参照していなければならず，XML 文書の解析段階で，対応する参照先が存在しない IDREF 型の値が存在している場合は，その XML 文書の妥当性のチェックでエラーとするように定められている．

次に，[5] 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．[5] の 2.8 Prolog and Document Type Declaration に示されている文書型宣言 (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，[5] 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，[5] の 3.3.3 Attribute-Value Normalization に示されている通り，解析対象実体の内容（置換テキスト）の中に別の解析対象実体が含まれていてもよく，その場合，再帰的に処理を行う必要がある．

また，2004/12/20 に勧告が出された XInclude という新しい規格 [6] 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，XPointer 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も可能である．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．また，参照先の文書に XInclude が存在した場合は再帰的に処理される．

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．

2.2 経路ベースアプローチと Dewey オーダー

2.2.1 経路ベースアプローチ

XML を関係データベースに格納する方式には，経路ベースアプローチ [4] を採用する．XML を関係データベースの表にマッピングする場合に，スキーマ定義に対応する表を生成する構造写像アプローチ [1] では，二次元表にマッピングすることが困難なスキーマを持たないデータを格納することに XML 文章を使用するという本研究の目的に合致しない．つまり，XML 文書に含まれるすべての要素について，要素間の親子関係や記述できる値，要素の省略ができるかどうかといった定義を参考にして，個々の XML 文書に応じたテーブルを関係データベースに作成するという方法では，そもそもスキーマ定義がなければ関係データベースに登録できないということになってしまう．

一方，スキーマではなく，文書，要素，属性，パス，など XML のデータモデルに従ってマッピングを行うアプローチをモデル写像アプローチ [4] という．これは，XML そのもののスキーマ情報を使用せず，上記の XML のデータモデルに応じたテーブルを関係データベースに作成する．これは XML のスキーマ定義を必要としないため，スキーマが可変的な文書でも使用する事が可能であり，本研究の目的に合致する．本研究では，モデル写像アプローチの中でも特に，XML 中の各要素について，根ノードからその要素までのパスを記録する方式である経路ベースアプローチを採用する．

[4] で提案されている手法を用いた場合に生成される関係データベースのテーブルを図 2.1 に示す．なお，説明欄は著者が説明のために独自に加えたものである．

図 2.1 からも明らかなように，経路ベースアプローチでは，XML 文書の要素の定義情報に応じたテーブルを生成するのではなく，XML そのもののデータモデルを使い上記の例で示したテーブルにすべての文書のデータを格納していく．

ただし，本研究では図 2.1 のような関係スキーマとは異なる定義を使用している．その最も大きな違いは，ノードラベリングの方法である．ノードラベリングとは，ノードの接続関係や順序など構造の情報を示すための情報の付与方法である．関係データベースの基本的な概念である関係モデルには格納する順番による順序づけという概念は存在せず，順序という概念が必要な場合は明示的に番号を付与する必要がある．よって，XML のように順序という概念を持ち，また XPath のようにその順序に基づいた問い合わせを行う場合は，このノードラベリングの方法を検討しなければならない．

図 2.1 では Element 表の start, end, index, reindex 列に相当するのがノードラベリングに基づいた情報である．start と end 列は，要素がどの要素を子要素とするかを示し，また index と reindex 列は要素がどのような順序で並んでいるかを示すためのものである．以下，その内容について詳しく説明する．

表名	列名	説明
Element	docID	ドキュメントに一意な値
	pathID	パスに一意な値
	start	<要素>の位置
	end	</要素名>によって要素が終了している位置
	index	同一レベルにおける要素の順序
	reindex	同一レベルにおける要素の降順の順序
Attribute	docID	ドキュメントに一意な値
	pathID	パスに一意な値
	start	属性の開始位置
	end	属性の終了位置
	value	属性の値
Text	docID	ドキュメントに一意な値
	pathID	パスに一意な値
	start	テキスト要素の開始位置
	end	テキスト要素の終了位置
	value	テキスト要素の値
Path	pathID	パスに一意な値
	pathexp	ルート要素から各要素までのパス上の要素を/で連結したパス式

図 2.1: 経路ベースアプローチを用いた場合の関係スキーマの例

start および end 列にはそれぞれ該当文書の先頭から開始要素および終了要素までのバイト数を示す整数を格納している．ある要素が別の要素を子要素としている場合は，親要素の start よりも後ろに子要素の start があり，親要素の end よりも前に子要素の end が存在するはずである．この関係を利用して，子要素を含む要素全体を取得することができる．これを範囲ラベリングとよぶ．しかし，この方法は元の文章が 1 バイトでも変更された場合に，変更箇所以降のノードラベルについてすべて変更しなければならないという問題点を持つ．

次に，図 2.1 における index と reindex は同じ兄弟関係，すなわち同一親を持つ要素の順序を示す番号が登録される．これを Local Ordering（または Sibling Ordering）という．図 2.1 では，index には昇順の順序が，reindex には逆順の順序を示す番号が格納されている．これにより XPath[7] の軸（Axis) として定義されている following-sibling や preceding-sibling の評価を容易に処理することができる．

本研究では，データを参照することにより実際には存在しない仮想的な要素が大量に生成される場合が多いため，このような変更箇所が大きく及ぶようなノードラベリングでは問題が多い．しかし，XPath による問い合わせを効率よく評価する上では上記のような接続関係や順序など構造の情報は必要である．

2.2.2 Dewey オーダー

そこで，変更があってもできる限り範囲箇所を限定するために提案されている Dewey オーダー [8] を，本研究では使用する．Dewey オーダーのノードラベルは次のように再帰的に与えられる [9].

- ルート要素のノードラベルは 1
- ある兄弟ノードにおいて i 番目のノードのノードラベルは，その親ノードのノードラベルを p とするとき'p.i'

つまり，先祖のすべての要素について先述した Local Ordering を格納した配列が Dewey オーダーである．これにより，子孫関係にある要素は同じ Local Ordering を保持しているため，2.1 における start 列と end 列にあたる情報を取得することができる．

Dewey オーダーは更新においても範囲ラベリングよりも少ない操作ですむ．XML の要素に含まれる値の更新では，総バイト数が変化した場合，それ以降のすべての要素のラベリングを修正する必要があるが，Dewey オーダーはバイト数とは何の関係もないため問題にならない．また要素自体の更新においても，Local Ordering で構成されていることから，あくまでも影響を及ぼすのは，更新される要素の順序が後ろになる兄弟要素と，その兄弟要素の順序を持つ子孫要素だけである．更新された要素の兄弟でない部分に対しては影響を及ぼさない．

本研究では，参照を記述する要素を解釈し，仮想的な要素を生成したとしても，参照を記述した要素の兄弟要素には必ず一つしか生成しないという制限を付けることで，参照を

記述していない部分に対するノードラベルに一切の影響を与えないようにしている．この制限については差分の記述記法について述べている．

2.2.3 逆経路索引

本研究では，経路ベースアプローチにおいて，逆経路索引 [2] を使用した索引を使用している．通常，上記の経路ベースアプローチにおいてパスを格納する場合はルートノードからの絶対パスを登録し，要素のテーブルと関連づけ，XPath 等の問い合わせにおいて，そのパス情報を使用して検索を行う．このとき，一般的にはパス表現の列に対して索引を設定することにより，レスポンスの向上を図るが，XML 文書はその性質から，前方一致検索を行うと同じ文字列が続きやすく，効率的に検索できない場合がある．また，属性名で検索を行うなど，問い合わせの条件によっては後方一致の検索が必要になり，索引が使用されない場合がある．これを解決するため，上記のパスの逆順，すなわちリーフノードからルートノードへのパスを記述し，後方一致であった検索を B-Tree の索引において効率的に処理できる前方一致検索に変更することができる．

第3章　提案手法

3.1　差分の記述記法

本研究では，差分の記述記法について以下のように定める．差分の取込みの機構に関する要素については，他の要素と区別するため，適切な名前空間を設定して，各要素に接頭辞を付ける必要がある．ここでは仮に接頭辞を xi とする．取込みの機構に関する要素としては xi:include, xi:failback, xi:overwrite の三つを定義する．

3.1.1　xi:include

xi:include 要素は該当する要素を参照先で置き換えることを示す．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．．また，xi:include に該当する要素を参照元要素とよぶ．

この要素には href 属性または idref 属性のいずれか，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，置き換えられた先の属性として扱われる．

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．idref 属性は上記 URI 文書の一部を示すために，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 の要素が対象となる XML 文書内に存在してはならない．なお，XML の定義 [5] より ID の値の一意性は Validity Costraint として定義されているが，本研究では整形式の XML についても扱うため，先述の条件をあえて記述している．同様に，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，idref 属性が指し示すことができる要素には id など，属性名で判断可能な要素でなければならない．idref 属性の記述を省略した場合は文書全体，すなわちルートノードを指し示しているとみなす．

include 要素の子要素には以下に述べる overwrite 要素または 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は再帰的に処理される．

3.1.2 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その展開を行った後に，element 属性の要素を取得し，上書きを行う．

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．element 属性による複数箇所の指定を認めるが，XML[5] の 3.1 Start-Tags, End-Tags, and Empty-Element Tags にある Well-formdness Constraint:Unique Att Spec にあるように，整形式レベルにおける制約において同一名の属性名が認められない．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，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．ただし，後の検索処理の簡略化のため，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 を記述し再帰的に include を処理することも認める．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．

3.1.3 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にもかかわらず，親要素の xi:include による参照先が特定できなくてもエラーにはならない．ただし，この場合は，xi:include の要素がそのまま要素として登録される．

xi:failback 要素には新しい要素の名前である name 属性が必ず存在しなければならない．これも xi:overwrite と同様に Dewey オーダーへの影響を避けるためであるが，xi:overwrite とは異なり，参照先にある元の要素名が存在しないため，省略ができないことによる．

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．

3.2 仮想的に生成される要素のパス式

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 3.1 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
参照先の要素名	x
x の要素の子孫要素のパス式	/Z/x/B
xi:overwrite の element 属性の値	e
上書きする要素のパス式	/A/xi:include/xi:overwrite/C
xi:overwrite の name 属性の値	o
xi:failback の name 属性の値	f
xi:failback の子孫要素のパス式	/A/xi:include/xi:failback/D

図 3.1: 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 3.1 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 3.2 のようになる．なお，再帰に関する詳しい説明は後述する．

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 3.2: 仮想的に生成される要素のパス式

図 3.3 の例では，Product A 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．これを完全に展開すると図 3.5 となる．

このとき，青いモデル (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

```
<?xml version="1.0">
<items>
  <item id="100">
    <name>Product A</name>
    <model>normal</model>
    <exterior>
      <color>red</color>
    </exterior>
    <engine>
      <xi:include
        href="engine.xml"
        idref="300" />
    </engine>
  </item>

  <xi:include idref="100" id="101">
    <xi:overwrite element="model">
      blue model
    </xi:overwrite>
    <xi:overwrite element="color">
      <main>blue</main>
      <sub>black</sub>
    </xi:overwrite>
  </xi:include>

  <xi:include idref="101" id="102">
    <xi:overwrite element="model">
      blue limited model
    </xi:overwrite>
    <xi:overwrite element="engine">
      <xi:include
        href="engine.xml"
        idref="400" />
    </xi:overwrite>
  </xi:include>
</items>
```

図 3.3: car.xml

```
<?xml version="1.0">
<items>
  <item id="300">
    <name>Engine A</name>
  </item>
  <item id="400">
    <name>Engine B</name>
    <turbo />
  </item>
</items>
```

図 3.4: engine.xml

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 になる．

また，上書きが行われていない要素の例として，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

次に再帰について考える．本研究における再帰または再帰的な処理とは，xi:include による取り込みが行われる要素の中に xi:include が記述されることをいう．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 3.3 の XML 文書において id=”101”の item は id=”100”の item 要素を参照している．この 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 id=”100”の xi:include の取り込みを行ってから，id=”101”の取り込まなければならない．

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再帰的な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．この後に xi:overwrite が含まれる xi:include による取り込みを行うが，この段階では通常の XML 文書と同様に扱うことができる．

つまり，取り込み対象の参照先要素に xi:include が存在し，それが取り込まれることによって新たな要素が生成された要素も，参照元から見ると図 3.1 で示したところの B の一部であると考えられる．同様に上書きする要素である xi:overwrite 要素の子要素に xi:include が存在しても図 3.1 で示したところの C の一部である．

図 3.3 の例であれば，エンジンの名前を示している name 要素について，id=”101”の要素

```
<?xml version="1.0">
<items>
  <item id="100">
    <name>Product A</name>
    <model>normal</model>
    <exterior>
      <color>red</color>
    </exterior>
    <engine>
      <item id="300">
        <name>Engine A</name>
      </item>
    </engine>
  </item>
  <item id="101">
    <name>Product A</name>
    <model>blue model</model>
    <exterior>
      <color>
        <main>blue</main>
        <sub>black</sub>
      </color>
    </exterior>
    <engine>
      <item id="300">
        <name>Engine A</name>
      </item>
    </engine>
  </item>
(つづく)
```

図 3.5: 展開後の car.xml

```
  <item id="102">
    <name>Product A</name>
    <model>blue limited model</model>
    <exterior>
      <color>
        <main>blue</main>
        <sub>black</sub>
      </color>
    </exterior>
    <engine>
      <item id="400">
        <name>Engine B</name>
        <turbo />
      </item>
    </engine>
  </item>
</items>
```

図 3.6: 展開後の car.xml（続き）

において生成されるパス式は/items/item/engine/item/nameとなる．このとき，A=/items, x=/item, B=/engine/item/nameとなる．ここで，再帰的に取り込まれた要素をダッシュ付きで表すとすると，x’は/item，B’=/nameとなる．

id=”102”はそのエンジンそのものを上書きし，別のエンジンを使用している．このときのエンジンの名前を示しているname要素も/items/item/engine/item/name であり，構成もid=”101”のときと変わらない

id=”101”の場合でエンジンそのものでなく，エンジンのnameについてさらに上書きが行われ，name2という名前に置き換えられたとすると，/B=/engine/itemとなり/o=/name2となる．

いずれの場合も，先に取込先のXML要素中にxi:includeを含まないようにしてから，取り込みを行う．よって，図3.1であるところのB,C,Dにおいてその後ろに再帰によって生成される部分が存在する可能性がある，ということを考慮しなければならない．

3.3 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が記述されている要素	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3.7: 生成前の文書のdeweyオーダー

図3.7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3.7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3.8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 3.8: 新たに生成される要素の dewey オーダー

図 3.3 の例を使って具体的に説明する．図 3.3 の dewey オーダーが図 3.9 であったとする．

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

図 3.9: 図 3.3 の dewey オーダーの例

このとき，青いモデル（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 3.8 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 3.7 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 3.3 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 3.3 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 3.3 の 3 番の dewey オーダーを使用する．この要素は図 3.7 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 3.9 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 3.8 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．

次に，再帰が存在する場合を考える．再帰は，参照する要素の子要素に存在する場合と，xi:overwrite 内に含まれる場合，xi:failback のパターンが考えられるが，いずれの場合もパス式と同様に考えられる．つまり，それぞれ B, C, D の順序について再帰によって

得られる部分が追加される．

再帰された文書に対する dewey オーダーの生成を図 3.3 と図 3.4 の例を使って説明する．ここでは青いモデル（id=”101”）の engine の名前を示す Engine A というテキスト要素の Dewey オーダーを求めることにする．このとき，該当のテキスト要素を構成するノードが図 3.9 および図 3.10 のような dewey オーダーであったとする．

番号	要素	dewey
1	<item id="300">	1.1
2	Engine A	1.1.1.1

図 3.10: 図 3.4 のの dewey オーダーの例

id=”101”は id=”100”を参照先要素として取り込んでいるが，この参照先要素の中に，求めたい Engine A というテキスト要素は存在しない．この要素は，この中に含まれている<engine>要素の子要素にある xi:include による取り込みによって得られる先に存在している．

まずは，id=”100”における Engine A の Dewey オーダーを求める．これは,「上書きされない要素」であるから A.B を求めればよい．A は xi:include が記述されている部分であるから図 3.9 の 5 番目の要素の Dewey オーダーである 1.1.4.1 を使用する．同様に，参照開始要素である図 3.10 の 1 番目の要素から X=1.1，Engine A のテキスト要素である図 3.10 の 2 番目の要素から B=1.1 を求めることができる．よって，この部分の Dewey オーダーは A.B=1.1.4.1.1.1 となる．

つぎに，この要素を参照先要素として取り込んでいる id=”101”の要素として考える．このとき，求めるテキスト要素は「上書きされない要素」であるから A.B を求めればよいことになる．A は id=”101”による xi:include の要素に相当するため，図 3.9 の番号 1 にあたる 1.2 が A となる．つぎに，参照開始要素は図 3.3 の番号 2 の item 要素なので，1.1 が X となる．B は先述した値である 1.1.4.1.1.1 から求めればよいので B=4.1.1.1 となる．この B の値は，再帰を処理する前の B の値である 4.1 と 3.10 の再帰部分から得られた 1.1 から成り立っている，ともいえる．

以上から，id=”101”の Engine A という要素の Dewey オーダーは 1.2.4.1.1.1 となる．

結論として，新しい dewey オーダーの生成には図 3.7 でいうところの A,B と存在すれば C，場合により D も必要であることがわかる．また，B を生成するためには X も必要であることがわかる．再帰している場合は，再帰によって得られる A 以外の要素が必要であり，それらは再帰が記述されている場所に応じて B または C または D の後ろに付く．

3.4 格納方法

この章では，先述した差分によって記述されたXML文書をどのようにして格納するのかを，いくつかの方法を用いて検討する．

3.4.1 共通仕様

関係スキーマ

関係スキーマは図2.1に示したものを，Deweyオーダーであること，およびxi:includeを格納する上で必要と思われる点を考慮して修正し，図3.11のような形で定義を行った．このスキーマは各方式の基本となるもので，各方式はこのスキーマをさらに修正して使用する．

表名	列名	説明
Document	docID	ドキュメントに一意な値
	URL	ドキュメント名．URLで記述する
Element	eID	要素に一意な値
	docID	ドキュメントに一意な値
	pathID	パスに一意な値
	Dewey	要素のDeweyオーダー
	kind	要素の種類
Value	eID	要素に一意の値
	value	属性の値
Path	pathID	パスに一意な値
	pathexp	ルート要素から子要素までの要素を/で表現したパス式

図 3.11: 本研究で使用した基本的な関係スキーマ

図2.1との違いはDeweyオーダーを使用した点以外に，以下のような点がある．

- 複数の文書を扱えるようにするためにDocumentテーブルを新設した．
- AttributeやTextテーブルを廃止し，属性やテキスト要素もElementテーブルに登録する．これらの区別はkind列で行えるようにする．kind列の内容は，図3.12に記述した．この列の内容を取得するだけで，xi:includeに関する要素なのかどうかを判別することができる．この考え方は[3]による実装例を参考にした．
- 要素の値については別のテーブルにした．これは，value列が非常に大きな値になる可能性を考慮している．

kind の値	要素の種類
0	通常の要素
1	属性
2	テキスト要素
10	xi:include の要素およびその子孫要素
11	10 の要素に存在する属性
12	xi:include の子孫要素に存在するテキスト要素

図 3.12: kind 列の内容と要素の種類の一覧

検索

XPath 式を与えると，要素を子孫要素も含めて返すような問い合わせを考える．このとき，結果の中に，要素・属性の名前と値，パス式，dewey オーダーを返すことにより，XML 文書を復元することが可能である．

[4] では XPath 式から SQL への変換に関してのアルゴリズムが示されている．しかし，本研究では再帰の回数に制限を設けていないため，任意の深さの再帰的な取込みを含む文書に対しての SQL への変換は難しい．このような部分も含まれているため，SQL の結果を取得して，その内容からさらに SQL を実行するようなアプリケーションによる検索も考慮する．

更新

更新については，XUpdate[10] によって更新式そのものを XML 文書として定義する書式が提案されている．Xindice[11] などの一部の XML ネイティブデータベースで使用されている．普及が進んでいる仕様とはいえないが，基本的にこの仕様によって可能な操作について考慮しておく必要がある．主に可能な操作を図 3.13 で示す（Xindice に関する解説記事 [12] を参考に作成した）．

機能	説明
ノードの挿入	あるノードの前後，および親と子を指定し， その子の前に追加することができる
ノードの更新	テキストノードおよび属性の値を更新する ノード自体を置き換える操作は削除と追加で行う
ノードの削除	指定したノードを削除する
ノード名の変更	指定したノードの名前を変更する

図 3.13: 更新において考慮する操作

このうち，ノードの名の変更は頻繁に用いられる処理ではないと考えられることと，ノードの追加および削除で対応できることから，対象には含めないこととする．

本研究では，さらに，xi:include の参照関係に関する図 3.14 の場合を特別に考慮する必要がある．この中で，参照先要素に関するものと，上書き対象要素に関する者は一般の要素に対する変更にも影響する．それ以外は，xi:include 要素の子孫要素内の変更に対する物である．

機能	説明
参照先要素の変更	参照先要素の子孫要素について 図 3.13 の操作を正しく反映する． とくに参照先要素が完全に削除または追加された場合は xi:failback を参照または参照しないように変更しなければならない
上書き対象要素の変更	上書き対象の要素が変化した場合に正しく反映する．
参照要素の追加	参照先の要素が追加，または id,idref,href などの要素が 変更された場合に正しく要素を参照できるようにする
上書き要素の変更	xi:overwrite 要素が追加・削除された場合，または その子孫要素が変更された場合に正しく反映する
failback 要素の変更	xi:failback 要素が追加・削除された場合，または その子孫要素が変更された場合に正しく反映する

図 3.14: xi:include に関して更新において考慮する操作

以上の処理を行う場合，各格納方式でどのような SQL による問い合わせと更新操作が必要なのかを見積もり，評価を行う．

3.4.2 各格納方式の概要

まず，一番単純な方法はデータベースに登録する際に xi:include を含む文章をそのまま格納する一方で，xi:include による取り込みによって生成された完全な XML 文章を別に生成し，データベースに登録する方法である．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，解釈後の結果をすべて再計算する必要がある．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．

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がある．更新に関しては，登録時に何の解釈も行っていないため，負荷はすべての方式の中で最小に抑えられる．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，問合せに必要な手続きが多く，SQL の発行回数が増加する．また，複

雑な SQL により効率的な検索ができないなど，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．

最後に，上記二つの中間をとる方式であり，途中までの結果を索引としてテーブルに格納する方式である．この方式では，xi:include に関する検索を行う場合に，xi:include によって参照される要素（およびその子孫要素）と，xi:overwrite の子孫要素だけを検索する．これらを可能にするため，それぞれそれらの要素に至る経路（パス式）のパターンを格納する索引を提供し，それらを併せて検索する．これにより，検索の回数は減少し，かつ更新は xi:inlcude に関しない部分において影響を及ぼさない．ただし，図 3.14 に関する操作は先述の索引を更新する必要があり，影響が大きくなる．

以下，三つの方式について順に詳しく説明していく．

3.4.3 完全な XML 文章化を行い格納する方法

更新を考える必要がなければ，最も単純に解決することができる方法である．しかし，更新を考える以上，取り込み処理に関する要素の関係を記述する必要がある．更新時に xi:include を検索し，更新を行っていく方法もあるが，この場合は後述する検索時に参照する方式と同様の段階を踏む必要があり，要素同士の参照関係を結んでいく方が，より簡単な更新処理になる

関係スキーマ

図 3.15 にこの手法を使用した場合の関係スキーマを示す．強調が行われている行は新たに追加，修正を行った部分を示す．

veid 列は，document type が 2 の文書，つまり include を解釈して，参照元の要素で置き換えた文書について，設定する属性である．document type が 1 の文書の内容を解釈して生成する文書が type2 の文書であるが，更新処理を軽減するため，value テーブルのデータは type2 の要素に対応した値を生成しない．そのため，type2 の文書は対応する type1 の文書の eid を veid 列に持ち，value テーブルのデータを共有する．xi:include や xi:overwrite も同様に，type2 の要素は，それぞれ，参照元，および上書き内容の要素への値を設定する．

refeid 列は，再帰要素が設定されていた場合に設定する属性である．veid 列に値を設定するために xi:include や xi:overwrite が存在した場合は，参照元や上書き内容の要素を取得し，その eid を取得して，veid を設定する．しかし，再帰が行われていた場合，すなわちその参照先や上書き内容の要素の中に xi:include が含まれていた場合は，その eid を取得することはできない．

この場合は，その再帰している xi:include の要素について document type が 2 の文書である取り込み済みの要素を取得する．再帰している要素と dewey オーダーが同じ要素を取得し，その要素に設定されている veid を生成しようとしている要素の veid に設定する．このとき同時に，取得した eid を refeid 列に格納する．

表名	列名	説明
Document	docID	ドキュメントに一意な値
	URL	ドキュメント名．URL で記述する
	kind	*include* を含む文書は *1*，*include* を解釈した文書は *2*，それ以外は *0*
Element	eID	要素に一意な値
	docID	ドキュメントに一意な値
	pathID	パスに一意な値
	Dewey	要素の Dewey オーダー
	kind	要素の種類．*10* 以上の値も通常の値として登録する
	veID	*value* テーブルを参照する場合に使用する要素 *ID*
	refeID	再帰要素について *veid* を取得した要素 *ID*．設定方法は後述する
Value	eID	要素に一意の値
	value	属性の値
Path	pathID	パスに一意な値
	pathexp	ルート要素から子要素までの要素を/で表現したパス式

図 3.15: 完全な XML 文書化を行う方式の関係スキーマ

つまり，再帰要素についてはすでに document type2 の文書が存在している必要がある．もし，存在していない場合は，その xi:include に関する処理は保留して，生成されるまで待つ，といった処理が必要である．

検索

基本的には完全な XML になっているので通常と同じ方法で検索を行う．ただし，以下の点を考慮する．

1. 取り込み済み（展開済み）の XML で検索を行う場合は，document type が 0 または 2 の文書の要素のみを取得する
2. document type が 2 の文書について要素の値を取得する時に，value テーブルを検索する場合は veid 列を使用する

例として，図 3.3 のような文書に対して/items/item/engine/name という Xpath による問い合わせを行った場合は，図 3.16 のような SQL を生成する．

ノードの挿入

ノードの挿入は，ノードの親要素を指定して追加するか，その前後の要素を指定して追加する．挿入は以下の手順で行う．ただし，xi:include 要素自体の追加，および xi:overwrite

```
select e2.eid, e2.docid, e2.dewey, v1.value, p1.path
from document d1, element e1, path p1, element e2, value v2
where d1.docid = e1.docid
and p1.pid = e1.pid
and p1.exp = '/items/item/engine/name'
and e1.docid = e2.docid
and e2.dewey like e1.dewey || '%'
and ((d1.kind = 0 and e2.eid = v2.eid) or
     (d1.kind = 2 and e2.veid = v2.eid))
order by e2.docid, e2.dewey
```

図 3.16: 検索例

要素自体の追加は後述する．

1. 対象の要素を document type が 0 または 1 の文書に追加する
2. 同様に対象の要素を document type 2 の文書に追加する．このとき，eid は新しく採番し，veid は document type 1 のものを使用する．
3. xi:include が新しくその要素を参照していないかチェックし，存在していた場合は，挿入した要素を該当の xi:inlcude と同じ dewey オーダーで type2 の文書に追加する．failback 要素による要素が存在していた場合は削除してから，追加を行う
4. 追加した要素の祖先要素が xi:include で参照されている場合（具体的には，祖先要素の type1 文書の eid が type2 文書の veid に存在し，かつ dewey オーダーが違う場合）は，参照先の要素にも追加を行う．ただし，このとき，追加する要素名が xi:overwrite の element 要素と一致する場合は，xi:overwrite の子要素を追加する．また，祖先要素が element 要素と一致する場合は，追加を行ってはならない．
5. 追加した要素の順序が後ろに当たる兄弟要素とその子孫要素について存在する dewey オーダーを更新する．更新は document type0 または 1 の要素について，まず更新を行い，その後に更新した eid について同じ値を veid 列にもつ document type2 の要素をすべて新しい dewey オーダーでふり直す．

xi:include が追加された場合は，以下のような手順で行う

1. 対象の要素を document type1 の文書に追加する．対象の文書が document type 0 であれば，type1 に変更する
2. href と idref 要素を用いて，参照先の要素が存在するかどうかチェックする．存在した場合は，その参照先の要素で document type2 の文書に追加する．存在しない場合

はfaiback要素を参照元として追加する．failback要素も存在しない場合はxi:include をそのまま登録する．追加する場合はxi:overwriteによる上書きも考慮に入れる．

3. 参照先の要素にxi:includeが存在する場合は，その要素に対応するtype2 の文書から要素を取得して，type2の要素を新たに生成する．詳しい説明は本節の「関係スキーマ」におけるrefEid列の設定に関する部分で述べている．

4. 追加した要素について，一般の要素の追加手順3と同様の処理を行う．

5. 追加した要素がxi:includeで参照されている要素の時は参照先のtype2の文書にも追加を行う．処理の内容は一般要素の追加手順4と同じである．

6. 一般要素の追加手順5と同様の処理を行う．

xi:overwriteが追加された場合は，以下のような手順で行う．xi:overwriteの子孫要素が追加された場合は，以下の手順の3を省略する．

1. 対象の要素をdocument type1の文書に追加する．

2. deweyオーダーを使用し，type2の文書についてelement属性の値に合致する要素が存在するかチェックする．存在した場合は追加した要素で該当要素を置き換える

3. xi:overwriteの子孫要素内にxi:includeが含まれている場合は，xi:includeにおける手順3と同様の処理を行う．

4. 追加した要素について，一般の要素の追加手順3と同様の処理を行う．

5. xi:includeにおける手順5と同様の処理を行う．

6. 一般要素の追加手順5と同様の処理を行う．

xi:failbackが追加された場合，以下のような手順で行う．xi:failbackの子孫要素が追加された場合は，以下の手順3を省略する．

1. 対象の要素をdocument type1の文書に追加する

2. deweyオーダーを使用し，type2の文書にxi:includeの要素がそのまま使用されている場合は，それを置き換える．

3. xi:failbackの子孫要素内にxi:includeが含まれている場合は，xi:includeにおける手順3と同様の処理を行う．

4. xi:includeにおける手順5と同様の処理を行う．

5. 一般要素の追加手順5と同様の処理を行う．

ノードの更新

先述したように，本研究ではノードの更新はテキストノードの変更のみを考慮している．このとき，変更する要素は，document type 0 または 1 の文書から取得し，対応する value テーブルの値を更新するだけでよい．

なお，テキストノードだけでなく，通常のノードの更新，とくに，xi:include に関する href 属性などの値の変更を仮に許可したとしても，参照する要素が変化するため，ノードそのものを入れ替える必要がある．よって，ノードそのものを追加・削除する場合とほとんど変わらないと考えられる．

ノードの削除

通常の要素の削除は以下のようにして行う

1. 対象の要素を document type が 0 または 1 から eid を取得する
2. document type 2 の文書から，先に取得した eid の値を veid に持つ値をすべて取得する．
3. xi:include によって新たに生成された要素全体を削除する場合（具体的には 2 の手順で得られた要素の dewey オーダーと 1 の手順で得られた dewey オーダーが違い，かつ，type1 の文書の要素の dewey オーダーが xi:include の要素の場合）は，xi:faiback による置き換えを行う．xi:failback の置き換えについてはノードの追加を参照.
4. xi:include によって新たに生成された要素の子孫要素を削除する場合（手順 3 について，対応する type1 の文書の要素が xi:include でない場合）は，削除対象の要素名が overwrite の element 要素に一致していないかどうかチェックする．一致していた場合は，該当の xi:overwrite に対応する type2 の要素を取得する．また，対象の xi:overwrite について該当要素の eid を veid にもつ要素をすべて取得する．
5. 手順 1,2,4 で得られた要素について後ろに当たる兄弟要素とその子孫要素について存在する dewey オーダーを更新する．
6. 手順 1 で得られた要素の子要素について．それぞれ上記の手順を繰り返し，子要素が存在しなくなるまで続ける．ただし，xi:include の要素，およびその子孫要素については，別項を参照する
7. 上記で取得されたデータをすべて削除する

xi:include の要素が削除された場合は以下の手順で実行する

1. 対象の要素を document type=1 から取得する.（他のタイプには存在していないはずであるが）

2. １で取得した要素について同じ dewey オーダーをもつ，document type 2 の要素を子孫要素を含め取得する

3. document type 2 の要素について，該当の eid を refeid に持つ要素を取得する．さらに，その取得した eid について refeid に持つ要素を取得し，これを繰り返す．

4. 2で取得した要素のうち，削除を指定した要素以外について，対応する要素がxi:includeと一致していた場合は，xi:failback と入れ替える

5. 手順 1,2 で得られた要素について，後ろに当たる兄弟要素とその子孫要素について存在する dewey オーダーを更新する．

6. 該当の文書に xi:include が存在しなくなった場合は document type を 0 に変更する

xi:overwrite を削除する場合は，上記の手順４を除く手順を行う．

xi:failback の場合は手順２において，dewey オーダーの一致だけでなく eid と veid の一致をチェックし，一致した場合は，手順３以降の処理を行う．一致しない場合は該当の document type1 の要素のみ削除して終了する

3.4.4 問合せ時に取り込みを行う方式

更新には最も有利な方法である．しかし，与えられた XPath 式について，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ため，あまり現実的な方式とは言えない．

関係スキーマ

図 3.17 に，この手法を使用した場合の関係スキーマを表す．強調が行われている行は新たに追加修正を行った行を表す．

xiValue 列には，以下の属性についての値を登録する．これは，xi:include で使用する値であり，索引を設定することによる検索の高速化が必要とされるために新設した．以下の場合，この列に value と同じ値を設定する．

- xi:include に設定されている href 属性および idref 属性の値
- xi:overwrite に設定されている element 属性，name 属性の値
- xi:failback に設定されている name 属性
- すべての要素に設定されている id 属性

表名	列名	説明
Document	docID	ドキュメントに一意な値
	URL	ドキュメント名．URL で記述する
Element	eID	要素に一意な値
	docID	ドキュメントに一意な値
	pathID	パスに一意な値
	Dewey	要素の Dewey オーダー
	kind	要素の種類
	xiValue	***xinclude* に関する値を格納する．設定内容は後述する**
Value	eID	要素に一意の値
	value	属性の値
Path	pathID	パスに一意な値
	pathexp	ルート要素から子要素までの要素を/で表現したパス式
	pathrexp	***pathexp* の逆経路索引によるパス式**

図 3.17: 問い合わせ時に取り込みを行う場合の関係スキーマ

また，登録を行う際に検索を行いやすくするため，図 3.18 で示した内容を格納する．これらの内容は検索を行いやすくするためのものであるので，xiValue 列に値を格納するだけでよく，Value テーブルには値を設定しないことで，他のオリジナルな要素と区別する（ただし，図 3.18 の 3 番目に示した element 属性の展開に関しては，例外とする）．また，この処理により更新時に若干の処理が必要になる．

検索

経路ベースアプローチで格納されているデータベースに XPath による検索を行う場合は，Path 表に格納されているパス式を用いて検索する．取り込みを考慮した場合，パス式は図 3.2 のようになっている．よって，XPath による問い合わせに合致するパターンを，すべて検索することになる．

なお，以下に示す検索は xi:include による取り込みが発生する要素のみを対象としている．通常の要素については別に検索する必要がある．

例えば，再帰および failback を考慮せず，XPath の問い合わせが/a/b/c であった場合，図 3.2 に適用すると，図 3.19 の六種類が考えられる．

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．

この方式では問い合わせは以下の手順で行う．なおここで問合せ対象になっているのは，取り込みに関する部分のみである．xi:include 以外の要素は別に検索する必要がある．

- ルート要素に id 属性がない場合は登録時に生成する．この属性の値は他の要素の id 属性では使用できない値でなければならない
- xi:include の href と idref 属性がどちらかしか存在しない場合は，デフォルト値で，どちらの値も存在するように設定しておく
- xi:overwrite の element 属性がカンマで区切られていた場合は，それぞれを別の要素として格納する．このときは xiValue にそれぞれの値を格納し，Value テーブルにはオリジナルの Value への参照を設定しておく
- xi:overwrite の name 属性が設定されていない場合は，element 属性と同じ値で name 属性の値を登録する．

図 3.18: 検索を簡単にするための格納形式の変更

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 3.19: /a/b/c という問い合わせについて考えられるパターン

図 3.19 において，以下のような問い合わせを行う．なお，ここで示している A,B,x,o,C はそれぞれ図 3.19 で示したパス名で置き換える．

上書きを行わないパターンは以下のような手順で検索する

1. /A/xi:include/@href および/A/xi:include/@idref のパス式を持った要素を検索する
2. @href の要素の xivalue 列に格納されている URL から document 表を検索し docID を取得する
3. @idref の要素の xivalue 列を取得し，path 表の逆経路索引から@id/x で始まるパス式の xiValue 列の値に同じものを取得する．このとき 2 で得た did の要素を持つ必要がある．
4. 3 で得た要素から dewey オーダーでその子孫要素全体を取得する
5. 子孫要素のパス式が B と一致する要素を取得する

6. B のなかに，xi:overwrite の element 要素の xiValue 列の値（要素名）が含まれていた場合は，対象から除く．

上書きを行うパターンは以下のような手順で検索する

1. 上書きを行わないパターンの 1 から 4 までの検索を行い，参照先要素を取得する．このとき，参照開始要素/Z/x を取得しておく

2. /A/xi:include/xi:overwrite/@element の xiValue 列を取得し，1 で得た要素の中で/Z/x/B/(xiValue 列の値) のようなパス式を持つ要素が存在するかどうかチェックする

3. /A/xi:include/xi:overwrite/@name 属性を取得し，o であるかどうかチェックする

4. /A/xi:include/xi:overwrite/C の要素を取得する

これらを考慮に入れて生成する SQL は図 3.20 のようになる．このとき，各パターンの式を A, B, C, X, O とする

再帰が存在する場合は，B または C の後ろにつく形で存在する可能性がある．図 3.19 を例にすれば，B=b/c のパターンについて後ろの c が再帰による c の可能性がある．ただし，再帰による対象が B 全体，および C 全体の場合は，何回再帰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．

xi:failback が存在する場合は，図 3.19 の x を f, B を D に変更したパターンを追加すればよい．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．

更新

基本的には特別な処理は必要ない．ただし，図 3.18 で示したように，登録時に行った，検索を行いやすくするために行った処理に関連して，以下のような処理が必要になる場合がある．

- xi:overwrite の element 属性の値が変更された場合，その xi:overwrite に設定されている name 属性を取得する．その属性に対応する Value テーブルの値が存在しない場合は，デフォルト値が入っているだけなので，name 属性の値を新しい element 属性の値で置き換える

3.4.5 中間結果を格納する方式

上記二つの中間をとる方式であり，考えられるパス式の内容を途中まで列挙し，検索する内容を，xi:include による参照先要素と xi:overwrite の要素にとどめる方式である．

途中までの結果がすでに格納されているため，検索の高速化が期待できる．

関係スキーマ

図 3.17 の内容に加え，図 3.21 のテーブルを加える．

IPath 表は，再帰などを考慮した上で，xi:include による参照先要素または xi:overwrite の要素に至るパス式を格納する．xi:include による参照先要素または xi:overwrite の要素の eid は，exp に続く要素を指し示している．pipid および ppipid 列は exp で示したパス式の構成内容を示すための列である．

再帰を含めたパス式を生成した場合は，参照元を親，参照先を子という関係で表すと，図 3.24 のような木構造で表すことができる（木についての詳細は後述する）．xi:include を取り込んで完全な XML データを生成した場合に生成される木構造と比較すると，ここで示した木は xi:include と xi:overwrite に関する部分のみで構成された木である．これは，3.2 節で示したように，仮想的に生成される要素のパス式は xi:include, xi:overwrite, xi:failback を含めた参照先要素 id で生成できるからである．参照先要素 id は xi:include と一対一の関係にあるため，xi:include と xi:overwrite による木を構成すればよい．なお，pipid 列はこのうち，親のノードへの参照，ppipid はルートノードへの参照に相当する．

Include 表は，xi:include の要素ごとに登録される．failid は xi:overwrite に設定されれている xi:faiback 要素である．failid と irid が一致している場合は，該当する xi:include は xi:failback を実際に指していることになる．

Overwrite 表は xi:overwrite の要素ごとではなく，xi:overwrite によって上書きする要素ごとに生成する．xi:overwrite による上書き処理は複数の要素に対して行われるため，これらを区別して先述の木のノードとして登録しなければならない．なお，iid で示されている xi:include の eid が IPath 表の pipid と一致するとは限らない．これは，参照先要素の子孫要素を上書きするのではなく，参照先要素の子孫要素にある xi:include が参照する先の要素を上書きするすることがある．この場合は，子孫要素の xi:include が IPath の exp 列の内容に影響するため，IPath 上では，子孫要素の xi:include の子ノードとして xi:overwrite が登録される場合もあるからである．

図 3.3 を例にして，説明する．この xml を格納した場合の eid の値が図 3.22 であったとする．

図 3.3 の xml について，xi:include とその子要素の xi:overwrite の関係，および参照先要素との関係を示したものが図 3.23 である．この図では，各要素をノードで示し，ノード名を eid で表している．このうち，xi:include と xi:overwrite の要素はノード名をイタリック体で，とくに xi:overwrite の要素にはさらに下線を付している．また，xml 上の親子関係を実線の有向辺で，xi:include による参照関係は，参照元から参照先への点線の有向辺で示している．

ただし，このグラフでは，xi:overwrite による上書きを考慮していない．xi:overwrite は，xi:include による参照関係の先でも有効になる．この例でいえば，xi:overwrite の eid が 160 である要素の上書き対象は，eid が 120 の要素だけでなく，eid が 100 の要素にも遷移的に影響を及ぼす．一方で，100, 120, 150 のノード名をルートとした木はそれぞれ XML の文書である．つまり，100 から始まる木を参照する場合でも，eid が 160 のデータの部分木と

して参照する場合は，xi:overwrite として，130, 140, 160, 170 の要素を考慮する必要がある．一方，eid が 120 のデータの部分木として参照する場合は，xi:overwrite として，130, 140 の要素を考慮しなければらない．

よって，これらすべてを単一の木で表現するのは難しいため，IPath 表ではそれぞれを別々の木として参照する．図 3.3 について，IPath 表を登録した場合は，図 3.24 のような木構造になる．

図 3.23 と同様に，各要素をノードで示し，ノード名を eid で表している[1]．イタリック体と下線の定義も同じであるが，有向辺の意味が全く異なる．ここで実線の有向辺を示しているのは，IPath 表における親子関係である．点線による有向辺は，xi:include 要素が指し示している参照先要素を示している．

IPath 表の説明で示したように，IPath 表の木はすべて xi:include と xi:overwrite のみで構成された木である．一方，ノード間の xi:include と xi:overwrite 以外の要素は，xi:include によって参照されている要素の内容で示されている．つまり，eid が 120 の要素の子要素である eid が 130 や 140 の要素は，eid が 100 の要素の子孫要素またはそれ自身の要素に含まれている値を上書きし，新しい仮想的な要素を生成していることになる．

なお，eid が 150 の要素をルートとする木において eid が 130 の要素は，上書きの対象が 160 とかさなっているために，取り消されたものである．これは，再帰要素があった場合は，その再帰による新しい要素を生成した後に，上書きを行うためである．図 3.23 で示したように eid が 130 の要素は eid が 160 の要素によって参照した先に記述されている xi:overwrite のために，上書きした後に，さらに上書きがされたとみなされる．

なお，IPath 表, Include 表，Overwrite 表はそれぞれ図 3.25，図 3.26，図 3.27 のように登録されることになる．

検索

仮に/a/b/c の問い合わせがあった場合，xi:include による取り込み要素に関連する要素について問い合わせるためには，以下のパターン考慮するだけでよい．

- 途中までのパス表現=/a, それ以降の要素=/b/c
- 途中までのパス表現=/a/b, それ以降の要素=/c

ここで，それ以降の要素とは xi:include の参照先要素（xi:failback 要素を含む）または xi:overwrite 要素である．

よって，長さ n の XPath 式による問合せの場合，$n-1$ 個の問合せパターンでよいことになる．ただし，Overwrite と Include の表について検索を行うので，$2(n-1)$ 個の問い合わせパターンが必要である．

[1] この図ではわかりやすくするために，ノード名に xi:overwrite の eid を使用している．xi:overwrite に関する要素を IPath に登録する場合は Overwrite 表の ID を使用する．

上記の例のうち，途中までのパス表現を A, それ以降の要素を B とした場合，SQL による検索は図 3.28 のようになる．

この後，子孫要素を含めた全体の要素を取得する場合は，Dewey オーダーで子孫要素を取得し，overwrite の適用，および子孫要素に xi:include が存在した場合は，それを取り込み処理を行わなければならない．ただし，その内容は IPath 表から取得できる．

また，完全な Dewey オーダーが必要な場合は，IPath 表から pipid 列を順に取得して Dewey オーダーを構築する．ただし，順に取得するためには，数回の SQL が必要なため，IPath 表上に Dewey オーダーなどを設定することも考慮すべきである．

なお，XPath 式において，//（/descendant-or-self::node() の省略形）の場合は//に当たる部分に%を付けて範囲検索する．

ノードの挿入

通常のノードを追加した場合は，以下の二つの処理を行う必要がある．

- id 属性が付いたノードを追加した場合は，xi:include が追加した要素を参照していないかチェックする．参照していた場合は，以下の処理を行う
  - xi:failback を参照していた，Include 表の irid 列（参照先要素 eid）を追加したノードの eid に書き換える
  - 該当の xi:include を参照していたパスを，IPath 表から取得し，追加したノードの要素名で exp の値を置き換える
  - xi:faiback に xi:include を含んでいた場合は，該当の IPath 表のデータをすべて削除する
- 追加したノードが Include の参照先要素の子孫要素の場合は，Overwrite 要素の element 属性の要素名と一致すると，追加したノードは上書き対象となる．（具体的には，Include 表の irid 要素の子要素でかつ，その Include 表に対応している IPath 表の子ノードとして存在している Overwrite 要素の element 要素の内容が一致している場合）対象となった場合は，以下の処理を行う
  - 対象となった xi:include の下位要素として，IPath 表を追加する．
  - Overwrite 要素を追加する．このとき，追加したノードは該当 Overwrite 要素の eraseeid 要素として登録する

xi:include を追加した場合は，以下の処理を行う

- Include 表に値を追加する．
- IPath 表にルートノードとして追加する

- 参照先要素の中にIncludeが存在する場合はIPath表における子ノードを生成する．該当の子ノードについてIPath表に生成している木が存在する場合は木全体をコピーして新たに生成する
- 追加した要素が，他のxi:includeから参照される場合は，再帰対象の要素を追加したことになるため，以下の処理を行う
  - 再帰対象になったxi:includeのデータに対して，IPath表上の子ノードとして追加する．自ノードについて木を生成した場合はその木全体を子ノードとして追加する

xi:overwriteを追加した場合は，以下の処理を行う

- 上書き対象の要素が存在すればOverwrite表にデータを登録し，IPathデータを登録する．このとき，対象となる要素は，IPath上における子孫要素全体のxi:includeが参照する要素である．また，上書き対象の要素が競合している場合（上書き対象の要素の親要素に上書きしている場合など）は，より上位にあるincludeが持っているoverwriteの方が有効になる．ただし，xi:overwriteのname属性によって要素名が変わる場合は，この限りではない．

ノードの更新

テキストノードの変更についてはIPath表，Include表，およびOverwrite表のいずれも何の影響もない

ノードの削除

通常のノードを削除した場合，以下の二つの処理を行う必要がある．ほぼノードの追加の逆処理になる．

- id属性が存在する要素を削除した場合は，Includeによって参照されているかどうか確認する．参照されていた場合は，xi:faibackを参照するように変更する
- 削除したノードのeidがOverwrite表のeraseeid列に登録されていた場合は，該当のovewrite表を削除し，IPath表に登録されていた，Overwriteのパスをすべて削除する

xi:includeを削除した場合は，以下の処理を行う

- IPath表に存在している該当のノード，および子孫要素をすべて削除する
- Include表の該当値を削除する．

xi:overwrite を削除した場合は，以下の処理を行う

- IPath 上に存在するすべての xi:overwrite に関するノード，および子孫要素をすべて削除する
- それまで eraseeid で削除されていた部分について，他の overwrite から上書き可能かどうかチェックする．可能な場合は，上書き可能な通常の要素を追加した場合と同様に処理する

```
select e6.eid
from (path p1 natural join element e1),
     (path p2 natural join element e2),
     (path p3 natural join element e3),
     (path p4 natural join element e4),
     document d1,
     (path p5 natural join element e5),
     (path p6 natural join element e6),
     (path p7 natural join element e7)
where
    p1.pathexp = '/A/xi:include/@idref'
and p2.pathexp = '/A/xi:include/@href'
and e1.docid = e2.docid
and e1.dewey = e2.dewey
and e2.xivalue = d1.uri
and p3.pathrexp like '@id/X/%'
and e3.docid = d1.docid
and e3.xivalue = e1.xivalue
and e4.docid = e3.docid
and e4.dewey like e3.dewey || '%'
and p4.pathexp = substring(p3.exp, '[^@]+') || '/B/' || e5.value
and e5.docid = e1.docid
and e5.dewey like e1.dewey || '%'
and p5.pathexp = '/A/xi:include/xi:overwrite/@element'
and e6.docid = e5.docid
and e6.dewey = e5.dewey
and p6.pathexp = '/A/xi:include/xi:overwrite/@name'
and e6.xivalue = 'O'
and e7.docid = e6.docid
and e7.dewey like e6.dewey || '%'
and p7.pathexp = '/A/xi:include/xi:overwrite/C'
```

図 3.20: 問合せ時に取り込みを行う場合の SQL 例

表名	列名	説明
IPath	ipid	この表の主キー
	exp	該当の要素に到達するまでのパス式
	ieid	xi:include の場合は Include 要素の eid
	ovrid	xi:overwrite の場合は Overwrite 表の id
	pipid	exp を生成するもとになった ipid
	ppipid	exp の開始部分に相当する ipid
Include	ieid	xi:include が記述されている要素の eid
	irid	参照先要素の eid
	failid	xi:failback 要素の eid
Overwrite	ovrid	xi:overwrite 表の主キー
	oeid	xi:overwrite 要素の eid
	iid	xi:overwrite の親要素である include 要素の eid
	eraseeid	xi:overwrite の場合は上書きされる要素の eid

図 3.21: 途中結果を格納する場合の関係スキーマ

要素	eid
`<item id="100">`	100
`<model>`	101
`<color>`	102
`<engine>`	103
`<xi:include href="engine.xml" idref="300" />`	110
`<xi:include idref="100" id="101">`	120
`<xi:overwrite element="model">`	130
`<xi:overwrite element="color">`	140
`<xi:include idref="101" id="102">`	150
`<xi:overwrite element="model">`	160
`<xi:overwrite element="engine">`	170
`<xi:include href="engine.xml" idref="400">`	180
`<item id="300">`	500
`<item id="400">`	510

図 3.22: xi:include に関する要素についての eid の例

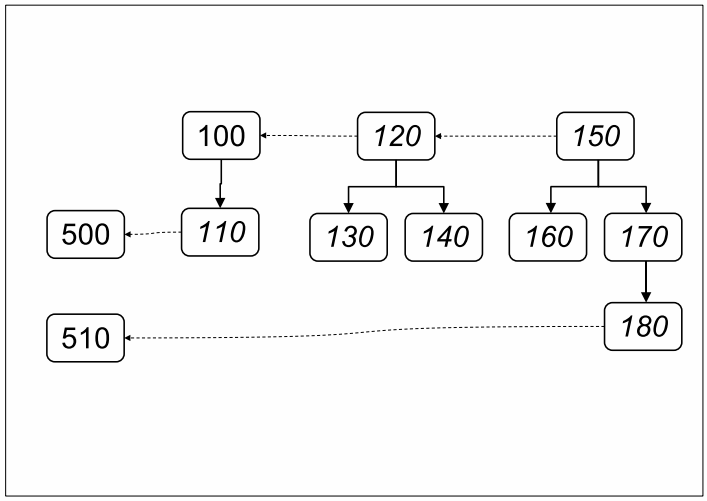

図 3.23: 図 3.3 の XML における xi:include と xi:overwrite の関係

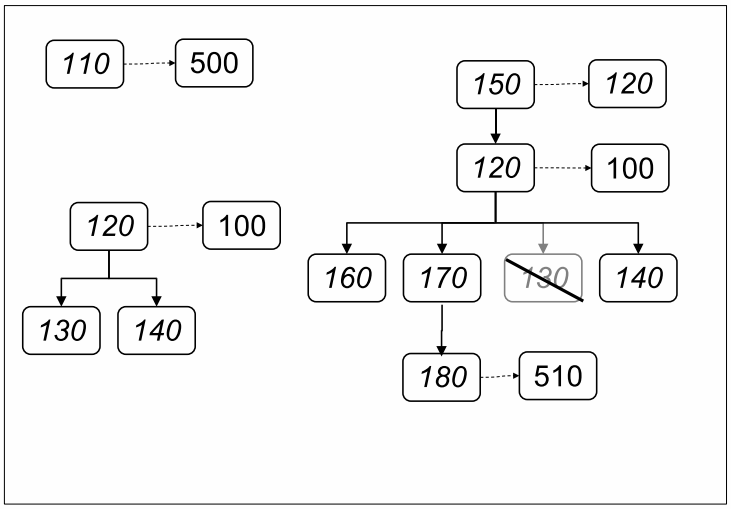

図 3.24: 図 3.3 を格納した場合の IPath 表の構造

ipid	exp	ieid	ovrid	pipid	ppipid
10	/items/item/engine/item	110		10	10
11	/items/item	120		11	11
12	/items/item/model		2000	11	11
13	/items/item/exterior/color		2001	11	11
14	/items	150		14	14
15	/items/item	120		14	14
16	/items/item/model		2002	15	14
17	/items/item/engine		2003	15	14
18	/items/item/engine/item	180		17	14
19	/items/item/exterior/color		2001	15	14

図 3.25: IPath 表に登録されるデータの例

ieid	irid	failid
110	500	
120	100	
150	120	
180	510	

図 3.26: Include 表に登録されるデータの例

ovrid	oeid	iid	eraseid
2000	130	120	101
2001	140	120	102
2002	160	150	101
2003	170	150	103

図 3.27: Overwrite 表に登録されるデータの例

```
select e2.eid
from ipath i1, include in1, element e1, path p1, element e2, path p2
where i1.exp = '/A'
and i1.ieid = in1.ieid
and in1.irid = e1.eid
and e1.pid = p1.pid
and e1.docid = e2.docid
and e2.dewey like e1.dewey || '%'
and e2.pid = p2.pid
and p2.exp = p1.exp || '/B'
and not exists (
  select * from overwrite o2, ipath i2, element e4
  where i2.ppipid = i1.ipid
  and i2.ovrid = o2.ovrid
  and o2.eraseeid = e4.eid
  and e4.docid = e2.docid
  and e2.dewey like e4.dewey || '%')
union all
select e2.eid
from ipath i1, overwrite o1, element e1, path p1, element e2, path p2
where i1.exp = '/A'
and i1.ovrid = o1.ovrid
and e1.eid = o1.oeid
and e1.pid = p1.pid
and e1.docid = e2.docid
and e2.dewey like e1.dewey || '%'
and e2.pid = p2.pid
and p2.exp = p1.exp || '/B'
and not exists (
  select * from overwrite o2, ipath i2, element e4
  where i2.ppipid = i1.ipid
  and i2.ovrid = o2.ovrid
  and o2.eraseeid = e4.eid
  and e4.docid = e2.docid
  and e2.dewey like e4.dewey || '%')
```

図 3.28: 検索例

第4章　評価

前章までで述べた問合わせ手法を用いて，完全なXML文書を登録する方式，および問合せ時に参照を解決する方式，途中結果を索引として格納しておく方式での，問合せの処理効率，およびデータ量の比較実験を行った．以下に，これまでに得られている結果について簡単に説明する．

なお，実行環境は以下の通りである．

- SunBlade1500 (UltraSPARCIII 1.5GHzx1 メモリ 1GB)
- Soralis 9
- PostgreSQL Version 7.4.3

postgresql.conf の内容の一部を以下に示す

- max_connections=3
- shared_buffers=50

共有メモリ領域がかなり小さいため，キャッシュが行われにくく，効率の悪い検索が行われると極端に処理が重たくなる[1]．また，大量の更新を行うと，すぐにバッファが溢れてしまい，何度も二次記憶に書き込まなくてはならない．よって，非常に性能差が出やすい環境であるといえる．

4.1　実験内容

実験用データとしては，XMark Version0.96 を用いてテスト用のXMLデータを生成した．このときに，XMarkのパラメータである，生成したXML文書の大きさを示すデータスケールは0.02である．

この実験では以下の点を前提条件とする．

- データ量は，データベースに格納したテーブルに割り当てられているページ数より算出しているため，8kバイト未満の誤差が生じている

[1]ただし，実際はOSによるディスクキャッシュがあるため，400kbyteしかキャッシュがない，ということではない

- データ量は索引を含む
- 更新は，数回の SQL に分かれているため，それぞれの SQL の実行時間の総和を実行時間とする

4.1.1 検索

実験 1 は仮想的に生成された要素への検索，実験 2 は上書きを大量に行った場合に，他の要素への検索の影響を調査する．

実験 1

元の XML 文書は/site/open_auctions/open_auction/itemref に，item 要素に設定されている id が格納され，item への参照が記述されている．

この itemref 要素に換えて xi:include を記述し，itemref ではなく，実際の item 要素が仮想的に存在するように記述する．

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．

実験 2

元の XML 文書に存在するメールアドレスに関する内容をすべてマスクするため，図 4.1 のような XML を記述する

```
<?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 4.1: メールアドレスのマスクをするための XML

この XML 文書に対し，実験１と同様の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．

4.1.2 更新

実験 3 は参照先要素の要素の追加，実験 4 は overwrite 要素の追加した場合の処理時間を比較する

実験 3

実験 2 の環境下で，XMark で生成した文書に item を追加する．図 4.2 の場合の処理時間を比較する

```
<item id="superitem>
  <name />
</item>
```

図 4.2: 追加処理を行う値

実験 4

name 要素へ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 4.3 に変更した場合の処理時間を比較する

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

図 4.3: overwrite を追加した XML

4.2 実験結果

4.2.1 実験 1

図 4.4 に実験結果を示す．完全に XML の展開を行った場合は，途中結果を格納する場合と比較しても 3 倍程度高速に検索できた．一方，問い合わせ時に参照を行う方式は，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，極めて検索処理が重い．途中結果を格納する方式は，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．

データ量については，完全に XML の展開を行う方式は途中結果を格納する方式の ipath 表を含めた値よりもさらに 2 倍以上大きくなっている．

方式	検索時間 (s)	Element(M)	iPath(M)
完全に XML の展開を行う方式	0.163	88.984	0.000
問合せ時に参照を行う方式	7.433	36.184	0.000
途中結果を格納する方式	0.581	36.184	0.232

図 4.4: 実行結果

4.2.2 実験 2

図 4.5 に実験結果を示す．この実験では再帰的な処理を使用しているため，問い合わせ時に参照を行う方式では計測できない．完全に XML の展開を行う方式はほとんど検索時間に違いが見られなかった．途中結果を格納する方式の場合，overwrite の項目が増加したために，iPath 表の大きさに従って若干値が増えている．ただし，データ量は差分しか持つ必要がないので，大幅に圧縮できていることがわかる．

方式	検索時間 (s)	Element(M)	iPath(M)	Value (M)
完全に XML の展開を行う方式	0.170	122.072	0.000	9.552
問合せ時に参照を行う方式	計測不可	-	-	-
途中結果を格納する方式	0.653	36.192	0.464	9.552

図 4.5: 実行結果

なお，ここで Value データの大きさを示したのは，完全に XML の展開を行った場合においても，提案手法では Value テーブルの複製を行わず，オリジナルな XML の値を参照するように変更しているため,「完全に XML の展開を行う方式」と「途中結果を格納する方式」では値が変化しない．仮に，実験 2 で登録した XML 文書を全く別の文書として登録した場合は，Value 表の値が 2 倍になる．

4.2.3 実験 3

各手法について，手順ごとに時間を示す.

完全に XML の展開を行う方式

処理時間は図 4.6 に示した．手順は，3.4.3 節で示した挿入手順の番号に対応している．完全にバッファをクリアした状態から処理を開始しため，最初の SQL はかなり処理が重くなっている．とくに，手順 1 の要素追加および手順 3 の要素追加は，キャッシュがかかっている場合は非常に早く処理される.

手順 4(3) は，手順 4 で追加した要素について，被参照チェックを行った結果である.

手順	処理内容	sql 実行回数	処理時間 (s)
1	追加要素の pid の取得	2	0.022
	要素の追加	4	0.398
2	要素の追加	3	0.042
3	要素の被参照チェック	1	0.505
	要素の追加	0	0.000
4	祖先要素の id の値取得	1	0.019
	要素の被参照チェック	1	0.051
	要素の overwrite チェック	1	0.056
	要素の追加	3	0.032
4(3)	要素の被参照チェック	0	0.000
5	dewey オーダーの変更	3	2.224

図 4.6: 実行結果

問合せ時に参照を行う方式

処理時間の一覧を図 4.7 に示した．この方式では，通常の XML と同様に扱えばよいため，完全に XML の展開を行う方式の一部を実行するだけでよい．番号は完全に XML の展開を行う方式に合わせている.

途中結果を格納する方式

処理時間の一覧を図 4.8 に示した．ipath 表には dewey オーダーを持っていないため，比較的早く処理が可能である.

手順	処理内容	sql 実行回数	処理時間 (s)
1	追加要素の pid の取得	2	0.022
	要素の追加	4	0.398
5	dewey オーダーの変更	1	0.817

図 4.7: 実行結果

手順	処理内容	sql 実行回数	処理時間 (s)
1	追加要素の pid の取得	2	0.022
	要素の追加	4	0.398
3	要素の被参照チェック	1	0.505
	要素の追加	0	0.000
4	祖先要素の id の値取得	1	0.019
	要素の被参照チェック	1	0.051
	要素の overwrite チェック	1	0.056
	要素の追加	0	0.000
5	dewey オーダーの変更	1	0.817

図 4.8: 実行結果

実験 3 のまとめ

処理時間を総和した値を図 4.9 の表にまとめた．今回の内容では要素の追加はあまり多くなく，dewey オーダーの変更がほとんどの要因をしめていることがわかる．

要素の追加はバッファのみの書き込みで終了していたため，比較的高速で終わったが，大規模な登録や，他のユーザーの使用によって，極端に処理効率が落ちる可能性もある．

方式	処理時間 (s)
完全に XML の展開を行う方式	3.349
問合せ時に参照を行う方式	1.237
途中結果を格納する方式	1.868

図 4.9: 実行結果

4.2.4 実験 4

各手法について，手順ごとに時間を示す．

完全に XML の展開を行う方式

処理時間の一覧を図 4.10 に示した．上書きを行うため，旧要素を削除し，新要素を追加したため，処理がかなりかかっていることがわかる．

手順	処理内容	sql 実行回数	処理時間 (s)
1	追加要素の pid の取得	1	0.018
	要素の追加	2	0.055
2	上書き先の検索	6	2.211
	旧要素の削除処理	1930	36.670
	新要素の追加処理	1930	27.020
3	検索処理	1	0.007
	追加処理	0	0.000
4	要素の被参照チェック	1	0.505
	要素の追加	0	0.000
5	要素の追加	0	0.000
6	dewey オーダーの変更	1	0.006

図 4.10: 実行結果

問合せ時に参照を行う方式

処理時間の一覧を図 4.11 に示した．この方式では，通常の XML と同様に扱えばよいため，完全に XML の展開を行う方式の一部を実行するだけでよい．番号は完全に XML の展開を行う方式に合わせている．

手順	処理内容	sql 実行回数	処理時間 (s)
1	追加要素の pid の取得	1	0.018
	要素の追加	2	0.055
6	dewey オーダーの変更	1	0.006

図 4.11: 実行結果

途中結果を格納する方式

処理時間の一覧を図 4.12 に示した．element 表に対する更新は発生しないものの，overwrite が大量に発生しているため，その件数だけ sql の発行回数が増えている．

手順	処理内容	sql 実行回数	処理時間 (s)
1	追加要素の pid の取得	1	0.018
	要素の追加	2	0.055
2	上書き先の検索	6	2.211
	IPath の追加処理	1930	13.510
3	検索処理	1	0.007
	追加処理	0	0.000
4	要素の被参照チェック	1	0.505
	要素の追加	0	0.000
5	要素の追加	0	0.000
6	dewey オーダーの変更	1	0.006

図 4.12: 実行結果

実験 4 のまとめ

処理時間を総和した値を図 4.13 の表にまとめた．完全に XML の展開を行う方式は，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，途中結果を格納する方式と比較しても 3 倍以上時間が掛かっている．

途中結果を格納する方式は上書きする要素の数ではなく，上書きされる要素の箇所の数に比例して追加処理が発生するため，比較的不利な実験かと思われたが，処理時間を大きく圧縮することができた．

方式	処理時間 (s)
完全に XML の展開を行う方式	66.492
問合せ時に参照を行う方式	0.079
途中結果を格納する方式	16.132

図 4.13: 実行結果

第5章　まとめ

本論文では，似たような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して効率良く検索するための手法について提案した．

本論文では，この枠組みについて二種類の対照的な方法と，より中間的な方法を提案し，データ量，検索速度，更新速度の面から比較を行った．

この結果から，検索の処理時間を重視する場合は,「完全にXMLの展開を行った」XMLを生成し，このXMLについて検索を行う手法を採用すべきであることがわかった．ただし，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．更新は，テキストデータの更新程度でとどめることができる場合は特に有用であると言える．

問い合わせ時に参照を行う方式は，差分で記述されたXML以外のデータをまったく持っていないため，更新面における速度は良好である．しかし，XPathによる問い合わせがあった場合に，検索を行う条件パターンが増加するため，速度面で不利であった．また，取り込み要素の中に取り込み要素が存在するような再帰的な検索についてすべてのパターンを列挙することは，不可能であり，あまり実用的な方法とは言えない．

そこで，中間的な方式として，xi:includeやxi:overwriteなど他の要素を参照または上書きを行うような要素についての関係を記述した木を構築し，この木に基づいて検索する方式を提案した．この方式では，データ量を大幅に圧縮することが可能であり，かつ更新処理における時間も完全に展開する方式と比較して４分の１程度に抑えることができたが，検索処理では逆に３倍以上時間がかかっている．とくに，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に適しているといえよう．

今後の課題として，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やXPathにおける問い合わせでも，述語(prediaction)の付いた問い合わせや多様な軸(axis)が付いた問い合わせついて，検証を行う必要がある．

参考文献

[1] Relational databases for querying XML Documents : Limitations and opportunities J.Shanumugasundaram, K.Tufte, C.Zhang, G.He, D.J.DeWitt, and J.F. Naughton In Proc VLDB 1999, pp.302-314, 19999

[2] XML 文書のための逆経路索引とその応用 山本 陽平，絹谷 弘子，吉川正俊，植村俊亮Conference Proceedings of Markup Technologies '99, GCA, pp. 127-135

[3] PostgreSQL を用いた多機能な XML データベース環境の構築 油井 誠, 森嶋 厚行 情報学会論文誌 Vol.44, No.SIG12(TOD19)

[4] XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Realational Database Masatoshi Yoshikawa, Toshiyuki Amagasa ACM Transactions on Internet Technology, Vol.1, No.1, August 2001

[5] Extensible Markup Language (XML) 1.1 World Wide Web Consortium http://www.w3.org/TR/xml11/, 04 February 2004

[6] XML Inclusions (XInclude) Version 1.0 World Wide Web Consortium http://www.w3.org/TR/xinclude/, 20 December 2004

[7] XML Path Language(XPath) version 1.0 World Wide Web Consortium http://www.w3.org/TR/xpath/, 16 November 1999

[8] Storing and querying ordered XML using a relational database system I.Taminov, S.Viglas, K.S.Beyer, J.Shanmugasndaram SIGMOD 2002, pp.204-215

[9] XML データベース技術解説 天笠 俊之, 吉川 正俊 オペレーションズリサーチ，第 50 巻, 第 6 号 pp365-372

[10] The XML:DB Initiative http://xmldb-org.sourceforge.new/xupate

[11] The Apache Software Foundation http://xml.apache.org/xindice

[12] 野畠 英明 XIndice:無料で使える XML データベース XML データの更新と削除 http://www.atmarkit.co.jp/fxml/tanpatsu/18xindice/xindice08.xml