NAIST-IS-MT0351086

修士論文

アクセスの偏りを考慮した XML データ処理の効率化に関する研究

中尾 伸章

2005年2月3日

奈良先端科学技術大学院大学
情報科学研究科 情報生命科学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士 (工学) 授与の要件として提出した修士論文である。

中尾 伸章

審査委員：

植村 俊亮 教授　（主指導教員）
関 浩之 教授　（指導教員）
松本 健一 教授　（指導教員）
宮崎 純 助教授　（委員）

アクセスの偏りを考慮した XML データ処理の効率化に関する研究*

中尾 伸章

内容梗概

本論文では，アクセスの偏りを考慮するにより XML データへの問合せ処理の効率をよくする二つの手法を提案する．XML フォーマットの利用が拡大し，様々な XML データが大量に生成されている．データサイズが数 MB から数 GB にもなる大規模な XML データもその一例である．大規模な XML データはそのサイズの大きさから処理コストが高くなる．しかし，問合せ処理時にアクセスが必要な部分は必ずしもデータ全体ではない．実際には，XML データ内の要素や属性毎のアクセスには偏りがある．そこで本論文では，アクセスの偏りを考慮する二つの手法を用いて効率のよい問合せ処理を実現する．一つは部分圧縮によるアプローチであり，アクセスの偏りを元に XML データを部分的に圧縮する手法である．これによりデータサイズを軽量化し，問合せ処理時に必要に応じて解凍処理を行うことで処理の効率をよくする．二つめは XML データの分割によるアプローチであり，アクセスの偏りを元に XML データを複数の XML データに分割する手法である．問合せ処理時には，必要なデータが記述されている XML データのみを処理することで問合せ処理を効率化できる．提案手法に基づいた実装を行い，その結果から，アクセスの偏りに応じて問合せ処理を効率化できることを確認できた．

キーワード

XML, 問合せ処理, アクセス頻度, データ圧縮, データ分割

* 奈良先端科学技術大学院大学 情報科学研究科 情報生命科学専攻 修士論文，NAIST-IS-MT0351086, 2005 年 2 月 3 日.

Studies on Efficient Query Processing for XML Data Considering Access Bias*

Nobuaki Nakao

Abstract

We propose two schemes for efficient query processing of XML data by taking locality of data access into account. By the spread of XML format, large-scale data described with XML are increasing. There are such XML data whose sizes are several GB. In large-scale XML data, the processing cost grows depending on the sizes. However, in general, there exists locality in XML data access, that is, we can find some portions that are accessed more frequently than elsewhere. In our schemes, for XML data being processed, we assume that typical queries and their frequencies are known in advance. Based on the information, we can know bias of the access frequencies of each portion.

Our scheme consists of two methods. One is a method of using partial compression, and the other is a method of using partitioning. We can reduce the size of the data processed by these two methods. This enables us to process XML queries efficiently by taking bias of access into account.

Keywords:

XML, query processing, access frequency, data compression, data partitioning

* Master's Thesis, Department of Bioinformatics and Genomics, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT0351086, February 3, 2005.

目 次

第 1 章　序論 **1**

1.1　XML データの拡大 1

1.2　XML データの処理方法 2

1.3　アクセスの偏りを考慮した問合せ処理の効率化 3

1.4　本論文の構成 5

第 2 章　基本事項 **7**

2.1　XML 7

2.1.1　XML 文書 8

2.1.2　XML データモデル 10

2.2　XPath 11

2.2.1　ロケーションパス 12

2.2.2　省略記法 14

2.2.3　文書順 14

2.3　XML データの処理 API 15

2.3.1　DOM 15

2.3.2　SAX 16

第 3 章　関連研究 **19**

3.1　XML データの圧縮に関する研究 19

3.2　XML データの分割に関する研究 21

3.3　XML データの問合せ処理に関する研究 21

第 4 章　アクセス頻度を考慮した XML データの部分圧縮手法 **23**

4.1　提案手法の概要 23

4.2 部分圧縮処理 27
4.2.1 アクセス頻度の算出 27
4.2.2 閾値の設定 29
4.2.3 データの圧縮 30
4.3 問合せ処理 32
4.4 評価実験 35
4.4.1 実験環境 35
4.4.2 実験内容 36
4.4.3 結果と考察 38
4.5 まとめ 42

第 5 章 アクセス頻度を考慮した XML データの分割手法 43
5.1 提案手法の概要 43
5.2 XML データの分割および構造概要の定義 45
5.3 分割処理 47
5.3.1 問合せ式による断片の規定 47
5.3.2 分割時の処理 55
5.4 問合せ処理 57
5.4.1 問合せの解析と変換 57
5.4.2 結果の統合 59
5.5 評価実験 61
5.5.1 実験環境 61
5.5.2 実験内容 61
5.5.3 結果と考察 63
5.6 まとめ 67

第 6 章 結論 69
6.1 まとめ 69
6.2 今後の課題 69

謝辞 71

参考文献 **73**

付録 **78**

A XMark データの DTD 78

図 目 次

2.1 XML 文書例 9
2.2 XML 文書を表す木構造の例 11
2.3 DOM 処理のイメージ図 16
2.4 SAX 処理のイメージ図 17

4.1 XML データ例 24
4.2 XML データ（図 4.1）を部分的に圧縮した例 25
4.3 XML データ（図 4.1）の木構造 26
4.4 XML 木（図 4.3）を部分的に圧縮した例 26
4.5 各ノードのアクセス頻度を求めるアルゴリズム 28
4.6 XML データの部分圧縮を行うアルゴリズム 30
4.7 部分圧縮された XML データの SAX での処理例 33
4.8 部分圧縮された XML データの DOM での処理例 34
4.9 XMark のスキーマ 36
4.10 圧縮率 38
4.11 A に基づいて部分圧縮したデータへの問合せ結果 40
4.12 B に基づいて部分圧縮したデータへの問合せ結果 41

5.1 分割前の XML 木 43
5.2 分割後の断片 (1) 44
5.3 分割後の断片 (2) 44
5.4 構造概要の例 46
5.5 欲張り法による併合アルゴリズム 53
5.6 格納先を記述した構造概要 56

5.7 問合せ処理の流れ 57
5.8 問合せの解析，変換アルゴリズム 58
5.9 分割前後の処理時間比（1 MB） 64
5.10 分割前後の処理時間比（5 MB） 65
5.11 分割前後の処理時間比（10 MB） 65

表 目 次

2.1 XPath の省略記法 14

4.1 典型的な問合せセットとその発行頻度の例 27
4.2 圧縮部分の構造情報 31
4.3 使用した計算機に関する情報 35
4.4 問合せセット A 37
4.5 問合せセット B 37
4.6 発行した問合せの特徴 39
4.7 解凍処理が必要な問合せの特徴 39

5.1 典型的な問合せセットとその発行頻度の例 47
5.2 問合せセットの各問合せ式に基づく断片の候補 49
5.3 断片の候補に関する情報 50
5.4 併合後の断片を規定する式 54
5.5 実験で用いた問合せセットとそれらの発行頻度 61
5.6 問合せセットに基づく断片の候補 62
5.7 併合結果 63

第1章　序論

1.1 XML データの拡大

WWW (World Wide Web) の爆発的な普及により，多様で膨大なデータがインターネット上に氾濫している．その代表的なものの一つに電子化された文書がある．当初，電子化された文書は，WWW の登場とともに急速に普及した HTML (Hyper Text Markup Language)[19] 形式で記述されたものが主流であった．しかし，HTML は電子的な文書管理を目的として開発された言語である SGML (Standard Generalized Markup Language)[34] をベースに開発された言語であったため，WWW 上でのデータ交換という利用領域においてその拡張性や柔軟性等の面で限界が生じてきた．

こうした状況を踏まえて W3C (WWW Consortium) が推奨したメタ言語が XML (Extensible Markup Language)[23] である．XML は，SGML のサブセットとして簡明な表現が行えるよう言語仕様が規定されるとともに，電子的なデータ交換の役割も担えるよう設計が行われた．このため，XML は文書の表現形式のみにとどまらず，より広い情報交換の形式として急速に普及しつつある．

XML は構造化文書であり，内部に記述するタグとその入れ子構造によって木構造で表現することができる．また，タグ内の要素や属性が名前を持つことから，自己記述的な表現が可能であり，様々な種類のデータが一つの XML データに記述されることも多い．データ交換への対応に加え，これらの柔軟な表現力により，様々な領域のデータが XML で記述されている．数百 MB から数 GB のデータサイズを持つ大規模な XML データもその一例であり，観測記録やサーバのアクセスログ，オントロジなど利用される機会が広がっている．

1.2 XML データの処理方法

XML で記述したデータ（以後 XML データと呼ぶ）は，XML 専用の API (Application Programming Interface) で処理することが多い．XML データの処理では DOM (Document Object Model)[18] と SAX (Simple API for XML)[14] の二つの API を主として使用する．DOM は W3C が推奨した，XML および HTML 形式のデータのための API である．データ交換におけるデータを処理対象としていることから，言語やプラットフォームに中立であるよう設計されており，現在多くの環境で使用できるようになっている．DOM は XML データをメモリ上にオブジェクトとして展開した上で処理を行う．XML データは木構造のオブジェクトとしてメモリ上に展開され，要素や属性もオブジェクトとして扱われる．メモリ上に展開されたデータに対して処理を行うことから，データに対して対話的な操作が可能で複雑な編集処理に向いているが，その反面メモリの消費が大きい．SAX は XML 開発者のメーリングリストにおける議論の中から誕生し，XML の簡単な処理を実現することを目的に開発された API である．DOM とは異なり国際的な標準化団体によって勧告された仕様ではないが，現在はデファクト・スタンダードとなっており，多くの環境で使用できるようになっている．SAX は，XML データをパースしながら逐次的な処理を行う．パーサが要素の開始や終了といったイベントを通知し，アプリケーションが各イベントを受取った際に処理が行われる．データの先頭から最後までパースによって発生するイベントに合せて逐次的な処理を行うため，複雑な編集処理ではコードが複雑化し不向きとされるが，DOM に比べて高速な処理を少ないメモリ消費で行うことができる．

これら二つの API で XML データを処理する場合，その処理コストは，XML データ全体の大きさに依存するといえる．DOM では XML データ全体をパースする時間と，パースしたデータを展開できるだけのメモリ領域が必要であり，SAX においても XML データ全体をパースする時間が必要となる．このため，大規模な XML データでは，その巨大なデータサイズとそれに伴うノード数の増加によって処理コストが必然的に高くなる．たとえば，一要素あたりのデータサイズが小さい title 要素と，一要素あたりのデータサイズが大きい body 要素の集合からなる大規模 XML データがあるとする．この XML データに対して，title 要

素の内容のみを問い合わせる処理を考える．DOM では，XML データ全体をメモリ上に展開する必要がある．よって，title 要素と body 要素の全てがメモリ上に展開される．このため，問合せの評価に必要な title 要素のデータサイズが小さくてもよりサイズが大きい body 要素の影響でメモリの消費は高くなり，処理コストが高くなる．さらに，title 要素が小さくても，全体のデータサイズが巨大であればメモリ上に展開できず，処理が不可能な事態も起こりうる．SAX の場合，XML 文書の先頭から終了まで全体をパースしてイベントを発生させる．よって，要素の名前に関係無く，title 要素でも body 要素でも開始や終了のたびにイベントが発生する．また，一般に文字列データで発生するイベントはその大きさによっていくつかに分割されて発生するため，データサイズの大きな body 要素では発生するイベントがより多くなる．よって，問合せの評価に必要な title 要素に比べ，body 要素にかかる処理時間がよりかかってしまう．このように問合せの評価に必要な部分木のサイズに関係なく，XML データ全体のサイズが処理コストを大きく左右する．

以上のように，問合せ処理に必要な部分が局所的である場合や，さらに問合せの発行頻度等によって部分木毎のアクセスの偏りが周知の状況においても，既存の処理系では，XML データの処理コストはデータ全体のサイズに左右されてしまう．

1.3 アクセスの偏りを考慮した問合せ処理の効率化

本論文では，アクセスの偏りを考慮することで XML データへの問合せ処理を効率化する二つの手法を提案する．前述した，問合せに必要な部分の局所性や，問合せが発行される頻度から得られるアクセス部分の偏りが周知であれば，それに基づいて XML データを加工しておくことで問合せ処理を効率化できると考えられる．さらに，問合せ処理時に問合せを解析することによって，加工後の部分木から処理に必要なものを限定すれば，より処理するデータサイズを減少でき，処理コストの効率化が可能である．本論文では，これらの点に注目し，アクセスの偏りを考慮した，次の二つの手法を提案する．

(1) アクセス頻度を考慮した XML データの部分圧縮手法 [31]

アクセス頻度を考慮して XML データ内の部分木を圧縮する．アクセス頻度の低い部分木を圧縮することで，XML データのサイズが軽量化される．このため，アクセス頻度が高く，圧縮されなかった部分木への問合せ処理が高速化できる．また，アクセス頻度が低く，圧縮された部分木への問合せの場合でも，問合せを解析することで必要な部分のみを解凍して処理することができる．

(2) アクセス頻度を考慮した XML データの分割手法 [30]

アクセス頻度を考慮して XML データを複数の XML データに分割する．アクセス頻度や構造に応じて部分木を抽出し，XML データを生成する．任意の数に分割する場合，アクセス頻度と処理時間から処理コストが最も低くなるような分割を判断することができる．また，DataGuide[6] に基づく構造概要を生成し，分割時の記述先の判断や，問合せの解析に利用することで，より問合せ処理を効率化できる．

本論文では，アクセスの偏りに関する情報を得るために，XML データへの典型的な問合せセットとそれらの発行頻度が既知であることを仮定する．これらはデータへ発行された問合せの記録を蓄積することで抽出することが可能であり，対象となる XML データのサイズや構造に関係がないため，比較的容易に収集できると考えられる．問合せセットを元に，各手法の特徴を考慮して部分木毎のアクセスの偏りを求め，利用する．

提案手法について，それぞれの性能と特徴を評価した．実験の結果により，(1) の手法では，圧縮によるデータサイズの軽量化により，アクセス頻度が高く圧縮されなかったデータへの問合せ処理が高速化できたほか，圧縮されたデータであってもその解凍処理にかかるコストによって問合せ処理の高速化がみられた．(2) の手法においても，アクセスの局所性と頻度に基づいて問合せ処理が高速化できた．また，分割後の問合せ処理時に必要な問合せの解析・変換を，構造概要の利用によって実現し，同じスキーマに基づく XML データにおいてはデータサイズが大きいほど有効であるという知見が得られた．

1.4 本論文の構成

本論文の構成は以下の通りである．まず第 2 章では，関連事項として，XML とその周辺技術について述べる．第 3 章では，本論文に関連する研究を紹介し，本論文の立場を述べる．第 4 章では，提案手法の一つである，アクセス頻度を考慮した XML データの部分圧縮方式について述べ，実験結果を通して評価および考察を行う．また，第 5 章では，もう一つの提案手法である，アクセス頻度を考慮した XML データの分割方式について述べ，実験結果を通して評価および考察を行う．最後に，第 6 章で，本論文についてまとめる．

第2章　基本事項

2.1 XML

XML (Extensible Markup Language; 拡張可能なマーク付け言語)[23] は，W3C (World Wide Web Consortium) の中に設立された XML 作業グループによって設計され，1998 年に勧告となった．XML は構造化文書の国際規格である SGML (Standard Generalized Markup Language)[34] のサブセットとして設計されたメタ言語である．しかし，柔軟性に富んだ表現力や電子的なデータ交換の役割を担えるような設計等，その構文や機能には SGML には無い，拡張された部分をも含んでおり，機能を限定したサブセットという意味ではない．XML は，仕様の簡明さ，表現の簡明さと柔軟性，インターネット標準であること等を目標として規格の設計がなされた．以下は，XML 作業グループが実際に挙げた XML の設計における目標である．

(1) XML は，インターネット上でそのまま使用できる

(2) XML は，広範囲のアプリケーションを支援する

(3) XML は，SGML と互換性をもつ

(4) XML 文書を処理するプログラムは書き易い

(5) XML では，オプションの機能はできるだけ少なくし，理想的には一つも存在しない

(6) XML 文書は，人間にとって読みやすく，十分に理解しやすい

(7) XML の設計は，すみやかに行う

(8) XML の設計は，厳密で，しかも簡潔なものとする

(9) XML 文書は，容易に作成できる

(10) XML では，マーク付けの数を減らすことは重要ではない

XML は SGML の後継言語として，WWW (World Wide Web) 上で流通する文書の表現形式として開発されたが，データ交換のフォーマットとしての有効性が注目され，文書のみにとどまらず様々なデータの記述フォーマットとして採用された．XML で記述されたデータ（XML データ[1] ）は増加の一途をたどり，それに伴い様々な要求が提言されてきた．XML で記述されたデータへの問合せ処理の効率化もその主たる一つである．

2.1.1 XML 文書

XML 文書は大きく分けると，XML 宣言，文書型定義，XML インスタンス[2] の三つの部分から構成される．なお，XML 宣言および文書型定義は必須ではない．図 2.1 は XML 文書の例である．

XML 宣言では，XML のバージョン，文字の符号化方式等を指定する．図 2.1 では 1 行目が XML 宣言である．この場合，XML のバージョンが 1.0，符号化方式が iso-2022-jp であることを示している．

文書型定義 (Document Type Definition; DTD)[17] はどのような XML インスタンスを許容するかを表すスキーマであり，どのような要素や属性が使われるかを定義したものである．図 2.1 では 2 行目が文書型定義を指す．この場合，外部ファイルである book.dtd によって XML インスタンスの表現が定義されることを示しているが，具体的な内容については本論文では省略する．なお，DTD は SGML から引き継がれて利用されているスキーマ言語であるが，XML Schema[22], RELAX[12] など XML 専用のスキーマ言語もいくつか存在する．

[1] XML データという用語は仕様書には表れないが，XML で記述されるデータが文書としての利用にとどまらないことからこの用語を用いる．

[2] XML インスタンスという用語は仕様書には表れないが，本論文では XML 宣言と文書型宣言を除く文書データという意味でこの用語を用いる．

```
<?xml version="1.0" encoding="iso-2022-jp"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book bkID="01">
  <title>XML Processing</title>
  <authors>
    <author>Tanaka Taro</author>
    <author>Suzuki Hanako</author>
  </authors>
  <price>2800</price>
</book>
```

図 2.1 XML 文書例

XML インスタンスは要素の階層構造であり，要素は開始タグと終了タグの対によって表現される．開始タグと終了タグに挟まれた文字列が要素の内容である．開始タグと終了タグの対はいくらでも入れ子にすることができ, タグの入れ子が XML インスタンスを表現している．開始タグは，< と > で要素名を囲んだ文字列であり，終了タグは < / と > で要素名を囲んだ文字列である．開始タグにおいて，要素名と > の間には属性名と属性値を = でつなげたものを幾つか記述することができる．たとえば，図 2.1 の 3 行目 `<book bkID="01">` は，要素名 book，属性名 bkID，その属性値 01 を含んだ開始タグである．なお，この開始タグに対応する終了タグは 10 行目の `</book>` である．

開始タグと終了タグの対応がとれており，親子関係にある要素のタグが正しい入れ子になっているなど，XML で規定されたタグ付け規則に従っている XML 文書を整形式 (well-formed) の XML 文書という．特に，文書型定義と XML インスタンスの両方を持ち，XML インスタンスの表現が文書型定義に適合している XML 文書を妥当 (valid) な XML 文書という．

本論文において対象とする XML 文書は，全て整形式または妥当であることが保証されているものとする．

2.1.2 XML データモデル

XML 文書は，XML インスタンスの階層構造に基づいて木構造で表現することが可能であり，その木構造は XML 木と呼ばれる．本論文で扱う XML 木はラベル付き順序木とし，木構造におけるノード（Node ; 節点） は次に示す根ノード，要素ノード，属性ノード，文字列ノードの四つとする．

(1) 根ノード (Root Node) : 木構造の根となるノード

(2) 要素ノード (Element Node) : 子となるノード（子ノード）をいくつか持つことができ，子ノードになり得る型は要素ノード，属性ノード，文字列ノードのうちいずれか一つである．要素ノードのラベルは要素名を示す．

(3) 属性ノード (Attribute Node) : 子ノードを持たない．ラベルは属性名，属性値を示す．なお，複数の属性が存在する場合，XML では属性間の順序は区別しない．

(4) 文字列ノード (Text Node) : 子ノードを持たない．ラベルは XML で規定された文字列が連続したものを示す．

図 2.2 に，図 2.1 の XML 文書を表現する木構造を示す．

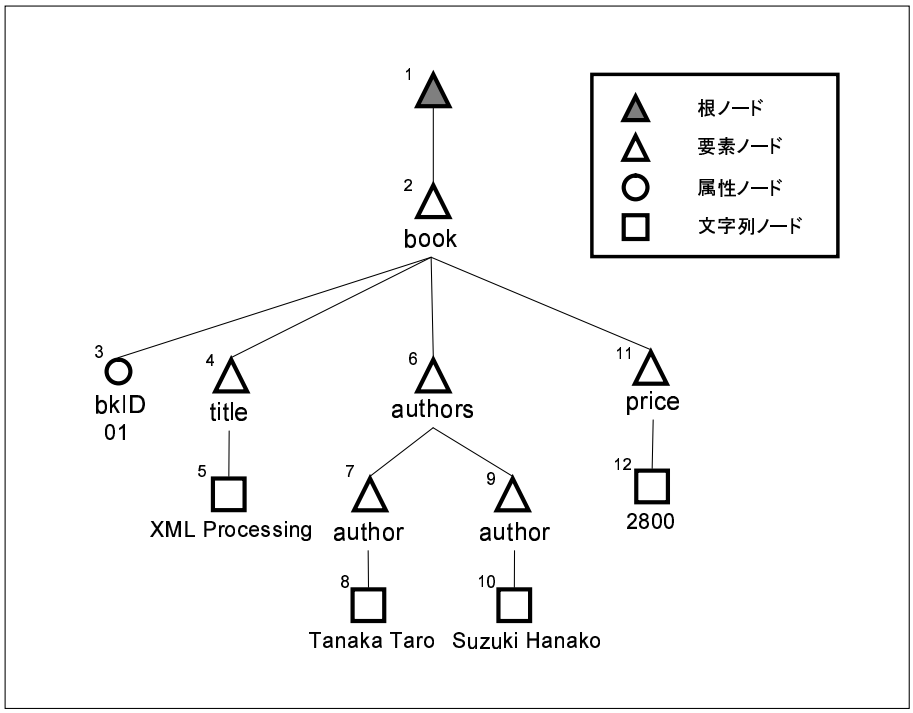

図 2.2 XML 文書を表す木構造の例

以後，本論文で扱う XML データのモデルを以下のように定義する．

【定義】XML データ

D は三つ組 $D = (V, E, r)$ で定義される．ただし，V はノード集合，E はエッジ集合，r は根ノードであり $r \in V$ である．

2.2 XPath

XPath (XML Path Language) 1.0 [27] は，XML 文書の特定の部分を指定 (addressing) するために考案された言語である．XPath は W3C によって策定されており，現在 XPath 1.0 が勧告，XPath 2.0 が作業草案 (Working Draft) の段階となっている．本来，XPath は XML 文書の変換のための言語である XSLT (XSL Transformations)[20] と，XML 文書の内部構造への指定をサポートする言語である XPointer[21] との間で共有する機能に、共通の構文 (Syntax) と意味論

(Semantics) を盛り込もうという試みから生まれた言語であり，問合せ言語ではない．しかし，その設計には XML 文書への問合せ言語である XQL (XML Query Language)[3] [13] の成果が取り入れられており，XML 文書への問合せ（検索）に関する技術，研究においてその表現は広く用いられている．また，XPath は W3C の勧告であることから，データ形式，文法ともに広く整備されており，その実装も多数存在する [25, 10].

XPath では，2.2 節で示したように XML を木構造のモデルで扱う．ノードには，要素，属性，文字列といった型があり，XPath はそれぞれの型のノードについて，その文字列値を計算する方法を定義する．

2.2.1 ロケーションパス

XPath は，ロケーションパスと呼ばれる経路式表現によって，XML 木をたどりながら目的のノードを指定する．たとえば，図 2.2 において,（根ノードの子である）要素ノード book の子である要素ノード title を指定する場合，次のようなロケーションパスで表現される．

/child::book/child::title

このように，XML 木内のノードを順次指定しながらたどることで最終的な目的であるノードを指定する．

ロケーションパスは一つ以上の，ノードの指定を行う単位の連続で表現されており，その単位はロケーションステップと呼ばれる．一つのロケーションステップは，次の三つの指定項目から構成される．

(1) 軸 (axis)：次のノードを探す方向

(2) ノードテスト：ノードの型とノードの名前（ラベル）

(3) 述語 (predicate)：[ ] 内部に記述される「式」による絞り込み

[3] XQL は，XPath の表現とは異なり，XML 文書の形式で問合せ内容を記述する．

軸は先祖，子孫，兄弟の方向に対して 13 種類が用意されている．また，(3) の述語はロケーションステップにおいて必須ではなく，また一つのロケーションステップに複数あってもよい．

たとえば，ロケーションステップ child::item[position()=2] を三つの指定項目に分解すると各項目とその意味は次のようになる．その他，軸，ノードテスト，述語のより具体的な内容については本論文では省略する．

(1) 軸 : 直下の子方向 (child 軸)

(2) ノードテスト : item という名前を持つ要素

(3) 述語 : 2 番目の item 要素

ここで，以後本論文で扱う，XML データへの問合せ式を次に示す．

【定義】問合せ式

使用する問合せ式の表現は， XPath のサブセットで記述されるものとし，述部を含まないロケーションステップと，`child` (“/”) 軸，および `/descendant-or-self::node()/` (“//”) からなるロケーションパスで記述されるものとする．

2.2.2 省略記法

XPath には，軸およびノードの種類の記述に対して次の表 2.1 に示す省略記法が用意されている．

表 2.1 XPath の省略記法

問合せ番号	省略記法
child::	(なにも書かない)
attribute::	@
/descendant-or-self::node()/	//
self::node()	.
parent::node()	..

省略記法により，次の二つのロケーションパスは等価であることがわかる．

/child::publications/descendant-or-self::node()/child::book[child::title=”XML”]/attribute::isbn

/publications//book[title=”XML”]/@isbn

2.2.3 文書順

XPath では，木構造をたどる順序が一意に決定されており，その順序は文書順 (document order)[4] と呼ばれる．文書順とは，深さの方向を優先する前順たどり (pre order) であって，要素ノードにおいては，XML 文書のテキスト表現において開始タグが現れる順番と等価になっている。また，要素ノードに従属する属性ノードは，文書順においてその要素の子ノードよりも先に位置するものとみなされる[5]．図 2.1 の XML 文書の開始タグの位置から，図 2.2 の木構造における文書順は，各ノードの左上に記述した数の通りの順になる．

[4] 文書順の逆でノードをカウントする順序は逆文書順 (reverse document order) と呼ばれる．

[5] 名前空間ノードがある場合は属性ノードよりもさらに先に位置するものとみなされる．

文書順は，XPath による問合せ結果の順序や，問合せ自身に深く関与する．たとえば，ロケーションステップの述語で用いられるコア関数 position() は，軸およびノードテストによって指定されたノード集合内における文書順を条件として指定するノードの絞り込みを行う関数である．

2.3 XML データの処理 API

問合せ処理をはじめとする XML データの処理においては，XML データを XML として構文解析（パース）し，処理を行う専用の API (Application Programming Interface) がいくつか標準化されている．その中で，現在主として使用されている DOM と SAX について，そのモデル及び処理の特徴について説明する．

2.3.1 DOM

DOM (Document Object Model)[18] は W3C によって勧告された，XML 文書および HTML 文書のための API である．DOM には 1 から 3 までレベルがあり，個々のレベルはさらにいくつかの個別の仕様に分かれている．現在，DOM Level 3 の仕様の一部までが勧告となっている．レベルが高くなるほど高機能ではあるが，処理コストが高くなるため，使用者が任意にレベルを選択して用いることができる．DOM は言語やプラットフォームに対して中立であるよう設計されており，任意の言語やプラットフォームで使用できる．

DOM は，名前が示す通り，XML データを木構造のオブジェクト（DOM 木と呼ばれる）で表現して扱う．XML 木における各ノードは，ほぼそのまま DOM 木上のオブジェクトに対応する．DOM 木は，XML データをパースすることでメモリ上に展開され，アプリケーションは DOM 木の各オブジェクトに対してアクセス・操作を行う．DOM では，XML データが全てメモリ上に展開されているため，対話的な操作が可能で，複雑な編集や更新を行う処理に向くとされる．反面，全てのデータをメモリ上に展開するため，データサイズの大きな XML データを扱う場合は多くのメモリ領域が必要となる．DOM による処理のイメージを

図 2.3 に示す.

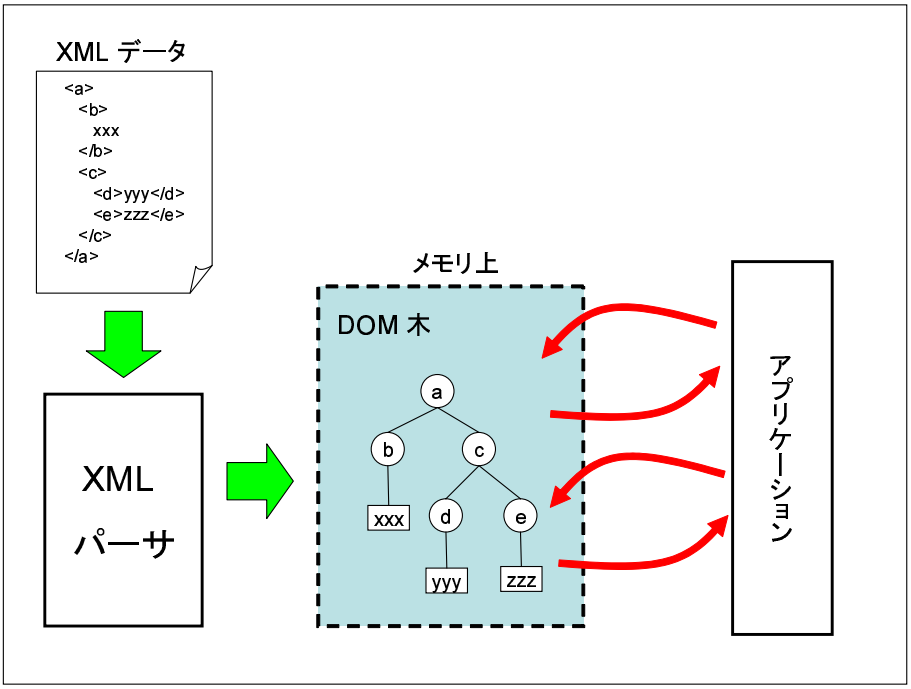

図 2.3 DOM 処理のイメージ図

図 2.3 のように，まず XML データ全体を XML パーサ（DOM パーサ）によってパースし，メモリ上に木構造のオブジェクト（DOM 木）として展開する．アプリケーションは，メモリ上の DOM 木をたどりながら要素，属性といったオブジェクトを操作することができる．

2.3.2 SAX

SAX (Simple API for XML)[14] は，David Megginson を中心とする XML 開発者のメーリングリスト XML-DEV[6] の議論の中から誕生し、国際的な標準化団体による仕様ではないにも関わらず、デファクト・スタンダード（事実上の標準、業界標準）となっている API である．SAX の最新バージョンは 2.0 である．バージョン 1.0 と 2.0 ではインタフェースの構成などが大きく異なり，バージョ

[6] http://www.xml.org/xml/xmldev.shtml

ン 2.0 が主流となっている．現在，SAX は代表的なほとんどの XML パーサや，多くの言語でサポートされている．

SAX は，イベント駆動型の API で，パースの進行に併せて文書の開始，要素の開始などのイベントが通知され，アプリケーションはイベントを受け取るハンドラによって対応する処理を行う．イベントの通知を元に逐次的な処理を行うため，DOM よりもメモリの消費量・処理速度の面で優れている．反面，構造に対する複雑な編集・更新や，複数の XML データ間をまたぐような処理ではコードが複雑化し，不向きとされる．SAX による処理のイメージを図 2.4 に示す．

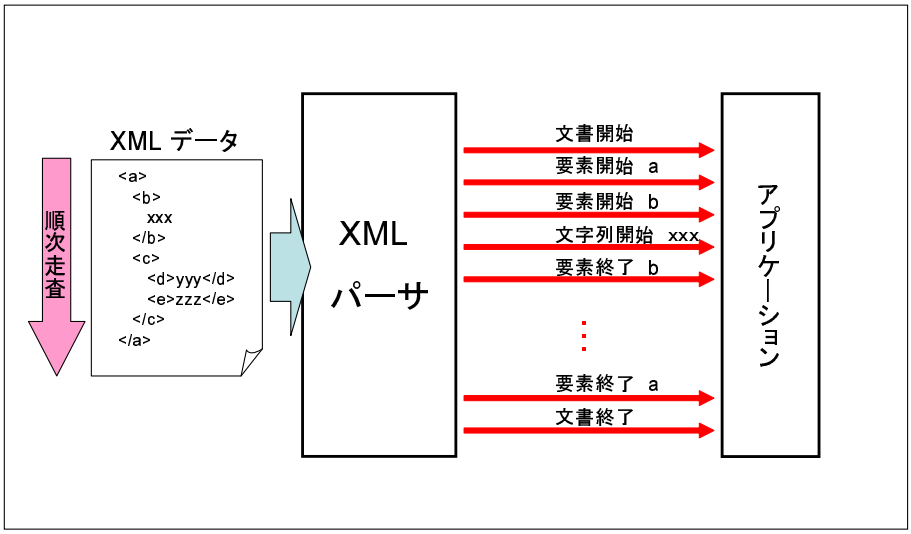

図 2.4 SAX 処理のイメージ図

図 2.4 のように，XML パーサ（SAX）パーサが XML データを先頭から順次走査しながらイベントを発生させる．イベントは文書や要素の開始・終了や属性，文字列データなどによって生成される．アプリケーションは，受取るイベントに応じて処理を取り決めておくことで，イベントの通知によって逐次的に処理を行う．

第3章　関連研究

XML はその登場から数年が経過し，多くの研究が行われてきた．本章では，本研究の関連研究を，XML データの圧縮，分割，問合せ処理の効率化に関する研究の三つに分類して紹介する．

3.1 XML データの圧縮に関する研究

XML はテキストデータとしてデータが記述されるため，gzip[7]，bzip2[5] など，既存のテキスト圧縮ツールでも比較的高い圧縮率を得ることができる．しかし，XML データの構造に着目してより高い圧縮率を実現する，XML データ専用の圧縮手法が数多く研究されている．これまでになされてきた XML データ専用の圧縮手法は簡単に次の二種類に分類できる．

(1) 圧縮処理後の出力がバイナリデータとなる手法

(2) 圧縮処理後の出力が XML データのままの手法

(1) の代表的な研究として，XMill[8] がある．XMill は，XML データを，構造に関する部分とテキストノード内のデータ部分に分けて圧縮を行う．各要素への経路を元にデータを分類し，同じまたは類似した経路を辿る要素のデータを同じ container と呼ばれる単位にまとめて圧縮を行う．圧縮には一般のテキスト圧縮ツールを用い，各 container ごとに最適な圧縮ツールを選択すればより高い圧縮率を実現することができる．圧縮ツールの選択は，使用者が指定できるほか，自動判定の機能も有している．しかし，XMill は，圧縮後のデータが元のデータに対応する構造を保持していない，圧縮したデータに問合せが出来ない，という特徴が欠点として挙げられる．

また，圧縮後のデータに対する問合せを可能にした手法としては Tolani 等の XGrind[15] がある．XGrind は，圧縮データに対して直接問い合わせが可能な手法としては最初に提案された物であり，基本的に，もとの XML データの木構造を残したまま，各要素中のテキストデータの部分を個別に圧縮する．データの木構造が圧縮形式中にほぼそのまま残っているため，圧縮したデータを先頭から解凍しながら問合せを実行していく処理が比較的簡単に実現できるが，問合せを実行するためには，データ全体を（必ずしも解凍はしなくても）ディスクから一度読み出すことになるため，圧縮データが非常に大きい場合は，I/O コストが大きくなる．東原等は [32] において，圧縮データへの問合せ処理時に必要の無い部分木をスキップするための索引を埋め込み，その索引を，XML データのストリーム処理のための索引構造である Stream Index (SIX) 上で管理している．これにより，パース処理にかかるコストを削減し，問合せ処理を効率化している．

これら (1) に分類される手法は，圧縮率の向上に重点を置いているものが多い．XML データ全体を圧縮処理し，出力がバイナリデータとなるのもこのためである．そのため，問合せや解凍において専用の処理機構が必要なものが多い．

(2) の代表的な研究として XML CSV[36] について述べる．XML CSV は，あらかじめ処理される要素が既知の状況において，処理に必要の無い要素の内容を CSV 形式のようにまとめ，一つの要素内に列挙して記述する．XML データではタグを用いて自己記述的な表現が可能であるが，反面タグによってデータが冗長になる．XML CSV はこのようなタグ部分の冗長性を解消することで処理を効率化するための研究である．なお，圧縮後のデータが XML 形式であることから，圧縮と同時に元の XML データへ復元（変換）するための XSLT データを生成することで，容易に復元が行えるよう工夫されている．しかし，属性値やテキストノード内のデータサイズが大きいような XML データではタグによる冗長性が少ないため，圧縮を行ってもデータサイズがほとんど軽量化されない．

出力が XML 形式であるというだけでなく，問合せ処理の効率化を考慮している点や，部分圧縮に近いことから，XML CSV は本研究に近いといえるが，タグによる冗長性のみの考慮である点や，圧縮部分の選定に言及していない点等で本研究と異なる．

3.2 XML データの分割に関する研究

XML 文書の分割に関する研究として Bremer 等の研究 [4] がある．[4] は，分散環境下における XML データ処理の効率化を目的とした研究であり，分散した XML データについての索引を提案し，それを Repository Guide と呼ばれる構造概要で管理している．XML データの分割を，XPath 式を用いることで関係表における垂直・水平分割に対応させて定義している．分割された XML データの各ノードの位置や出現単語に関する三つの索引を生成し，Repository Guide と関連付けて分散環境での問合せ処理を効率化している．また，Dewey Order[11] に基づく [3] の手法を用いることで，ノードの位置を表す索引を二進数表現を利用して軽量化し，索引全体のデータサイズの巨大化を抑える工夫がなされている．しかし，分散環境下での処理に焦点を当てた研究であるため，分割する部分（を指す XPath 式）を具体的に決定する方法や問合せの変換については議論されていない点で本研究とは異なっている．

3.3 XML データの問合せ処理に関する研究

XML データの問合せ処理に関する研究は数多く存在する．ここでは，本研究と同じく，問合せの評価に必要な部分木（ノード集合）の局所性を利用して問合せ処理の効率化を行う研究について紹介する．

中島等は，DOM のメモリ消費の高さに着目し，処理に必要な部分のみを主記憶上に展開して処理できる SPlitDOM[35, 33] を提案している．Marian 等は，問合せ処理時に XQuery[24] 問合せを静的に解析し，問合せの評価に必要な部分を射影して処理する手法を提案している [9]．横山等は，SAX を用いた処理において，発生イベントとそれに付随するデータの保存先をアクセス頻度に応じて切り分けることで処理時間を短縮する手法を提案している [29]．また，天笠等は関係データベースに格納された大規模 XML データの処理において，問合せ処理に必要な部分のみを動的にマッピングして使用することで高速化する手法を提案している [28].

本研究は，問合せの評価に必要な部分の局所性に加え，その頻度を利用して

XML データそのものを加工し，問合せ処理を効率化するという点でこれらの研究とは異なっている．

第4章　アクセス頻度を考慮した XML データの部分圧縮手法

4.1 提案手法の概要

一章で述べたように，XML データの処理コストはデータ全体のサイズに大きく左右される．ゆえに本手法では，XML データを部分的に圧縮し，処理に不必要なデータのサイズを縮小させることによって処理を効率化する．また，部分的な圧縮を行った後もデータは XML フォーマットを維持し，使用者は圧縮を意識せずにデータを扱うことができる．処理に必要なデータが圧縮されている場合，解凍によってデータを復元する必要があるため処理コストが余分にかかってしまう．このため，本手法では部分木毎のアクセス頻度によって圧縮するかどうかを判別することで使用頻度に応じて処理を効率化する．アクセス頻度は，XML データへの問合せセットとその発行頻度が既知であることを仮定し，それを元に部分木毎に算出する．アクセス頻度の値から圧縮の可否を判断し，値が低い部分木を圧縮し，値が高い部分木を元の状態で残しておく．また，どの部分木が圧縮されたかを管理する情報を圧縮時に生成し，処理時に利用する．これにより，圧縮されていないデータの処理が高速化されるだけでなく，圧縮されたデータの処理においても，必要な圧縮部分のみを判別して解凍して処理できるため，解凍する圧縮部分のデータサイズやその数によっては処理の高速化が期待できる．

例として，図 4.1 で示す XML データを部分的に圧縮することを考える．図 4.1 で示される XML データの各要素のアクセス頻度が既知であり，たとえば book 要素の子である body 要素と，magazine 要素のアクセス頻度が一定の値よりも低ければ，その部分木を圧縮して図 4.2 のような XML データへと加工する．ただし，圧縮したデータに記述されている属性 Cid は，圧縮部分を識別するための

識別子であり，詳細は後で説明する．これにより，データサイズの軽量化により，アクセス頻度がより高かった他の要素に対する問合せ処理の高速化が期待できる．

```
<books>
  <book>
    <title>XML 概説</title>
    <authors>
      <author>田中 太郎</author>
    </authors>
    <body>
      XML はインターネットの標準として W3C より勧告されたメタ言語である。メタ言
      語とは、言語を作る言語という意味である。つまり、ただ単に XML を使うだけで
      情報を記述することは出来ない。まず、情報を記述するための言語を XML を用い
      て作成し、それを用いて情報を記述することになる。
      XML の … (中略) … XML の強力さとカバー範囲の広さを示すものである。
    </body>
  </book>
  <book>
    <title>XML 詳説</title>
    <authors>
      <author>山田 次郎</author>
      <author>中村 三郎</author>
    </authors>
    <body>
      近年、XML は、Web 上でデータ交換の標準フォーマットとして欠かせない存在とな
      り、電子商取引やマルチメディアコンテンツなど幅広い分野で利用されています。
      例えば、インターネットを利用した映画や音楽の配信に利用されるなど、XML は非
      常に高い注目を浴びています。また、Visual C# .NET では、データ形式として XML
      を利用した … (中略) … XML 文書の作成方法、XML の活用方法を説明する。
    </body>
  </book>
  <magazine>
    <title>XML World</title>
    <body>
      今号では、blog 文化の拡大とともに注目を集めている RSS について特集をお送りし
      ます。RSS とは Rich Site Summary または RDF Site Summary の略で、コンピュ
      ータ用のデータ形式の一種です。ニュースサイトやウェブログなど、どんなサイトで
      も RSS 配信がされていさえすれば、様々なソフトでその情報を簡単に活用することがで
      きるのです。実際多くのウェブサイトですでに RSS の配信がはじまっています。RSS を
      利用するためには RSS リーダーというソフトを使います。インターネット上には様々な
      RSS … (中略) … 最新の記事の見出しを一覧で表示することができるのです。
    </body>
  </magazine>
</books>
```

図 4.1 XML データ例

```
<books>
  <book>
    <title>XML 概説</title>
    <authors>
      <author>田中 太郎</author>
    </authors>
    <body Cid="1">
      H4sIAAAAAAAAAHVTW47bMAz89ymIHKBOgu2mH14fohco
      aJm2ieq1lJTUPX0pZdFtEPTH4kua4ZDuBmNDovkHFpM5
    </body>
  </book>
  <book>
    <title>XML 詳説</title>
    <authors>
      <author>山田 次郎</author>
      <author>中村 三郎</author>
    </authors>
    <body Cid="2">
      H4sIAAAAAAAAAHVFU7XKcMAz8f0+hyQOUu7STHSWYcfv
      OgILcDA2seyjF5Om7vkvz0UzsFeV/avWz8uFWy7LkmSg
    </body>
  </book>
  <magazine Cid="3">
    H4sIAAAAAAAAAAI1UW27bMBD89ykWOUAVu22AAIo+e42CI
    lcWY4rLLEm7yuk7lIOkblCgP6KX+5rdGXrX2yCZ3U9TbfE
    SqxZ4tPd9dwf7jr4xrp+dn3bXL7wojxRO5/u2ne/3xJve
  </magazine>
</books>
```

図 4.2 XML データ（図 4.1）を部分的に圧縮した例

また，図 4.1 の XML データを木構造で表現した図 4.3 の場合，その部分的な圧縮は図 4.4 のようになる．ただし，データサイズを表現するため，テキストノードは内容を表示せず，圧縮されていないものは黒塗りの三角形で，圧縮されたものは灰色の三角形で表し，その大きさがデータサイズを表現しているものとする．

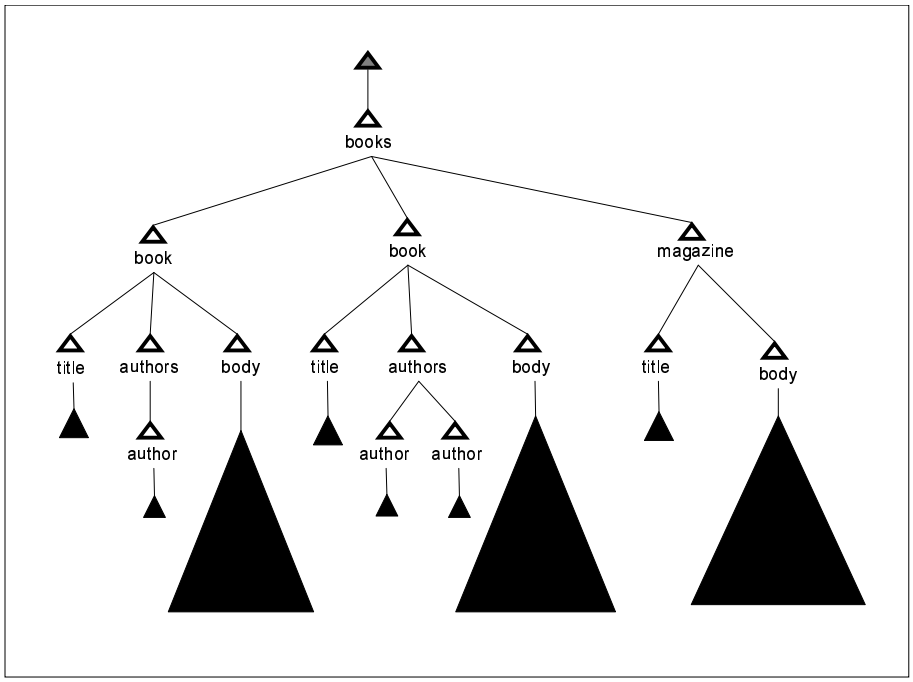

図 4.3 XML データ（図 4.1）の木構造

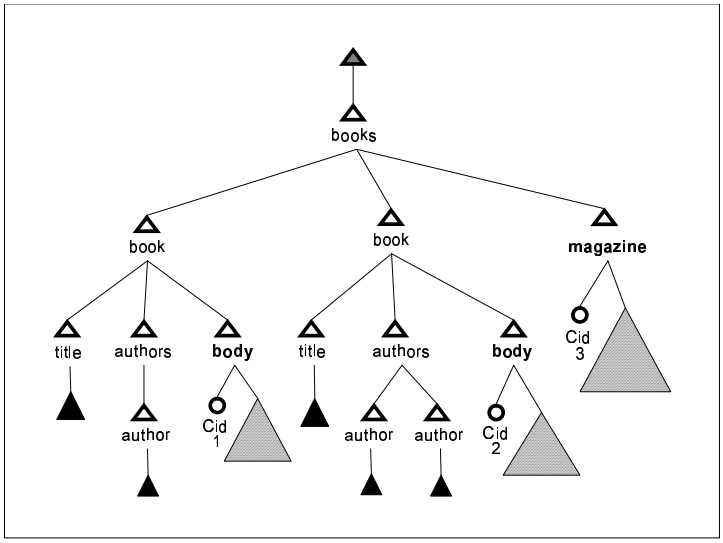

図 4.4 XML 木（図 4.3）を部分的に圧縮した例

4.2 部分圧縮処理

4.2.1 アクセス頻度の算出

本手法では，部分木毎にアクセス頻度を算出し，圧縮するかどうかの判断に用いる．XML データに対する典型的な問合せセットとその発行頻度が既知であることを仮定し，それを元に部分木毎のアクセス頻度を算出する．問合せの発行履歴を元に，発行回数の多い上位 n 件を典型的な問合せセットとし，統計から算出した確率の値を各問合せの発行頻度として用いる．図 4.1 の XML データに対する典型的な問合せセットとその発行頻度の例を表 4.1 に示す．このような情報を元にアクセス頻度を算出する．

表 4.1 典型的な問合せセットとその発行頻度の例

問合せ	XPath 式	発行頻度
q_1	/books/book/title	0.5
q_2	//author	0.3
q_3	/books/book	0.1

アクセス頻度は部分木を圧縮するかどうかを判断する指標となる．圧縮によってどの程度データが縮小されるかは，圧縮するデータのサイズに強く依存する．よって本手法では，問合せの発行頻度と部分木のデータサイズの両方を考慮してアクセス頻度を算出する．

問合せ q_i $(i = 1, 2, \ldots, n)$ の発行頻度を $F(q_i)$ $(0 \leq F(q_i) \leq 1)$ で表す．q_n の問合せ結果となる部分木に含まれる各ノード v_k $(k = 1, 2, \ldots, m)$ のアクセス頻度 $F(v_k)$ には $F(q_n)$ が加算される．ただし，“//” を含む問合せのように一つの問合せから同じノードが重複して呼び出される場合，その問合せの発行頻度は一度のみ加算されるものとする．また，問合せ結果となる部分木に含まれるノードだけではなく，その部分木の先祖ノードにも，データサイズを考慮してアクセス頻度を加算していく．具体的には，問合せの発行頻度と，問合せ結果である部分木のデータサイズが各先祖ノードを根とする部分木のデータサイズに対して占める

割合の積を各先祖ノードに加算する．よって，q_n の結果となる部分木の根を v，v の親ノードを v_{parent} とし，それぞれのデータサイズを $s(v)$，$s(v_{parent})$ とすると，$F(v_{parent})$ には $\frac{s(v)}{s(v_{parent})} \cdot F(q_n)$ が加算される．

以上のように問合せセットとその発行頻度を用いて各ノードのアクセス頻度を求めるアルゴリズムを図 4.5 に示す．ただし，$EvalQuery(d, q)$ および $EvalQueryAnc(d, q)$ は，XML データ d に問合せ式 q を適用し，その結果に含まれる全てのノード及びその全ての先祖ノードを重複無しで返す関数とし，また $getChild(v)$ はノード v に子ノードが存在する場合は子ノードを返し，子ノードが無い場合は偽を返す関数であるとする．

```
アルゴリズム assignFrequencies(d : XML データ, Q : 問合せセット)
begin
  foreach v in V
    foreach q in Q
      if (v ∈ EvalQuery(d, q))
        F(v) = F(v) + F(q)
      endif
      if (v ∈ EvalQueryAnc(d, q))
        while (v_child = getChild(v))
          F(v) = F(v) + (s(v_child) / s(v)) · F(q)
        end
      endif
    end
  end
  return V
end
```

図 4.5 各ノードのアクセス頻度を求めるアルゴリズム

$F(v)$ の値が高いノードほどアクセスされやすく，圧縮に不向きなノードであるといえる．

4.2.2 閾値の設定

ノード毎に算出したアクセス頻度を元に圧縮する部分木を決定し，XML データの部分圧縮処理を行う．部分木毎の圧縮の可否は，XML データの根ノードから子ノード方向へとトップダウンに判別していく．ただし，アクセス頻度にはデータサイズが考慮されているため，トップダウン処理の際に，アクセス頻度が高くデータサイズがごく小さいデータは圧縮されてしまう可能性がある．本手法ではそのような部分をホットスポットと定義し，圧縮部分に含まれるホットスポットを圧縮後もそのまま保持させることでアクセス頻度の高いデータへの問合せ処理が遅くならないよう考慮する．

ノードの圧縮の可否はアクセス頻度に対する閾値 α，ホットスポットの判定には閾値 β を設定して判断する．アクセス頻度が閾値 α 未満のノードを根とする部分木を圧縮し，β よりも高い圧縮部分内の部分木をホットスポットとする．閾値を元に XML データの部分圧縮を行うアルゴリズムを図 4.6 に示す．ただし，*addHotspotFlag(v,u)* は，ノード u に対してノード v をフラグ付けする関数とする．*compressionData(u)* は，ノード u において，*addHotspotFlag(v, u)* でフラグ付けされたノードを保持した状態で u 内の全ノードを圧縮する関数とする．また，$H = H \cup \{v\}$ において，*H.add(v)* はノード v を H に加える関数とする．

閾値 α を高く設定するほど部分圧縮は起こりやすく，圧縮によるデータサイズの軽量化が期待できる．反面，圧縮される部分木の増加は処理時間の増加に直結する．ただし，部分木のデータサイズが小さい場合，圧縮時に付加されるヘッダ情報や，圧縮率の低さによってかえってデータサイズが増大してしまうことがある[1]．この場合，圧縮されたデータは解凍処理にかかる時間が増大するだけでなく，XML データ全体のサイズの増加をも引き起こしてしまう．本手法では，簡単のためデータサイズが一定の値よりも小さい部分木については，アクセス頻度が閾値 α よりも低い場合であっても圧縮を行わない方針を取り，圧縮によりかえってデータサイズが増大してしまわないよう配慮する．

[1] 一般に，圧縮によるデータサイズの減少量はデータサイズではなくデータの連続性や繰り返しパターンの頻度に依存する．ここでの圧縮によるデータサイズの増加は，圧縮後に付加されるヘッダ等によって圧縮後のデータが圧縮前よりも増大する場合を指す．

```
アルゴリズム compressionXML(d : XML データ)
begin
  compressionXMLRecursive(d, r)
end
return d

アルゴリズム compressionXMLRecursive(d : XML データ, u : ノード)
begin
  if F(u) < α
    return compressionNode(u)
  else
    forsech v ∈ u の子ノード
      compressionXMLRecursive(d, v)
    end
  endif
end

アルゴリズム compressionNode (u : ノード)
begin
  forsech v ∈ getHotspots(u)
    addHotspotFlag(v, u)
  end
  compressionData(u)
end

アルゴリズム getHotspots(u : ノード)
begin
  H = ∅
  forsech v ∈ u の子ノード
    if F(v) > β
      H.add(v)
    else
      getHotspots(v)
    endif
  return H
end
```

図 4.6 XML データの部分圧縮を行うアルゴリズム

4.2.3 データの圧縮

本手法では，圧縮時に用いる符号化アルゴリズムについては規定せず，所与の符号化アルゴリズムを用いるものとする．ただし，圧縮後の出力結果がバイナリデータとなってしまう符号化アルゴリズムを用いた場合，そのままではテキスト

形式である XML フォーマットを維持できないため，base64[2] や uuencode[16] といったアルゴリズムによってデータのテキスト化を行う．

圧縮したデータの記述

部分木を圧縮し，その出力として得られるデータは，必要に応じてテキスト化を行った上で一つのテキストノードとして要素の中に記述する．圧縮しその出力として得られるデータは部分木内の構造を持たないため，解凍しなければ内部にどのようなノードが含まれているかが分からない．しかし，圧縮データの内部に評価すべきノードが存在する可能性のある問合せを処理する場合，その全ての圧縮データを解凍するのは現実的ではない．従って，問合せ処理のために解凍が必要な圧縮データを限定できるよう，圧縮した部分木の構造に関する情報を圧縮処理時に作成しておく．具体的には，圧縮した部分木毎に識別子を付加しておき，圧縮データを記述する要素に属性値や子要素の内容として保持させておく．また，圧縮データ内に含まれるノードの経路式とそのノードを含む圧縮データの識別子の一覧を保持した情報を圧縮部分の構造情報として，部分圧縮処理と同時に生成しておく．これにより，圧縮データ内に存在するノードがどこに存在するかをあらかじめ判断することで不必要な解凍処理を軽減することができる．図 4.2 のように部分圧縮された XML データ の場合，圧縮部分の構造情報は表 4.2 のようになる．ただし，圧縮した部分木の判定に用いる識別子を記述する場合は名前空間を用いて名前の競合を防ぐ必要があるが，簡単のため例では省略してある．

表 4.2 圧縮部分の構造情報

経路式	圧縮部分の識別子
/books/book/body	1,2
/books/magazine	3
/books/magazine/title	3
/books/magazine/body	3

4.3 問合せ処理

部分圧縮された XML データに対する問合せ処理について述べる．圧縮後の XML データは圧縮されている部分を含むため，あらかじめ問合せを解析することで，解凍が必要かどうかを判別する．問合せの解析は圧縮時に作成した圧縮部分の構造情報を利用して行う．問合せを解析し，圧縮データ内に必要なノードが含まれていなければ，解凍を行わずに問合せを評価できることが保証される．また，解析の結果圧縮データ内に必要なノードが含まれていた場合，構造情報から解凍するデータが記述された要素の識別子を抽出し，その識別子を持つ要素の圧縮データのみを解凍すれば問合せを評価することができる．たとえば，表 4.2 のような圧縮部分の構造情報がある場合に問合せ式 /books/magazene/title が発行された場合，識別子 3 を持つ圧縮データのみを解凍するばよいことがわかる．また，問合せ式 /books/book が発行された場合，経路式との先頭からの部分マッチングから，識別子 1 および 2 を持つ圧縮データのみを解凍するばよいことがわかる．このように，発行された問合せ式と構造情報内の経路式をマッチングすることでどの圧縮データを解凍すべきかを判断できる．

解凍したデータをどのように評価し結果を得るかは処理方法によって異なる．SAX による処理の場合，構造情報から得た識別子を持つ要素の開始イベントを受取ったときにその要素に記述されたテキストデータを解凍し，復元された部分木へ問合せを発行し，部分木内の結果を得る．これにより，全体をパースしながら，必要な部分のみを解凍しながら問合せ処理を行うことができる．なお，パースと同時に逐次解凍を行うことから，問合せの結果は元の XML データを問合せた場合と構造や文書順が等しい結果となる．例として，SAX での処理例を図 4.7 に示す．

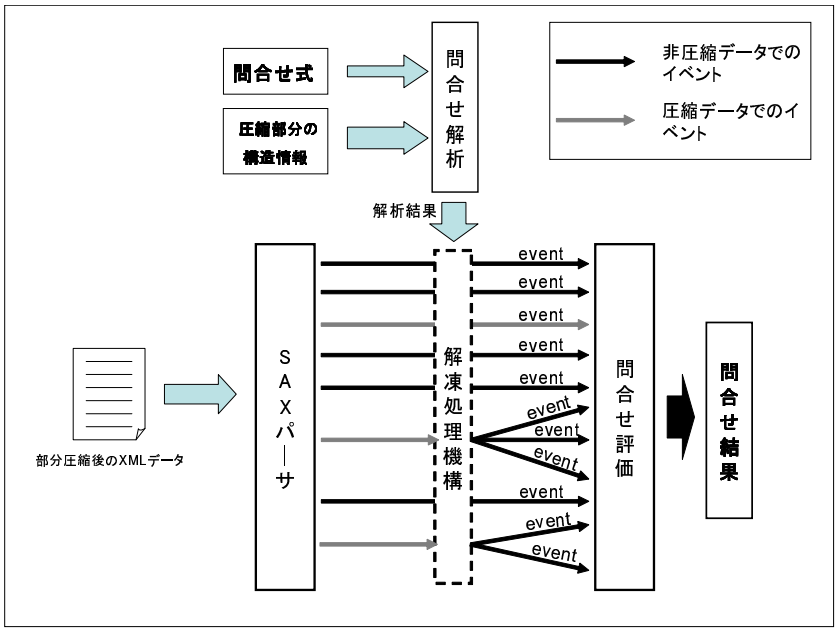

図 4.7 部分圧縮された XML データの SAX での処理例

図 4.7 の例では，問合せの解析結果から解凍が必要な要素を判断し，その要素がアプリケーションに通知される際には解凍処理を行い，そのデータのパース結果をアプリケーションを通知して処理している．DOM による処理の場合，問合せの評価に解凍処理が必要無い場合であれば部分圧縮後の XML データを DOM 木に展開して処理すれば結果を得られる．また，解凍処理が必要な場合は圧縮後の XML データをメモリに展開した上で，必要な圧縮部分のみを解凍してパースしてその部分木を復元する．これにより，問合せに必要な部分木が全て DOM 木として展開され，元の XML データへの問合せと同じ結果を得ることができる．例として，DOM での処理例を図 4.8 に示す.

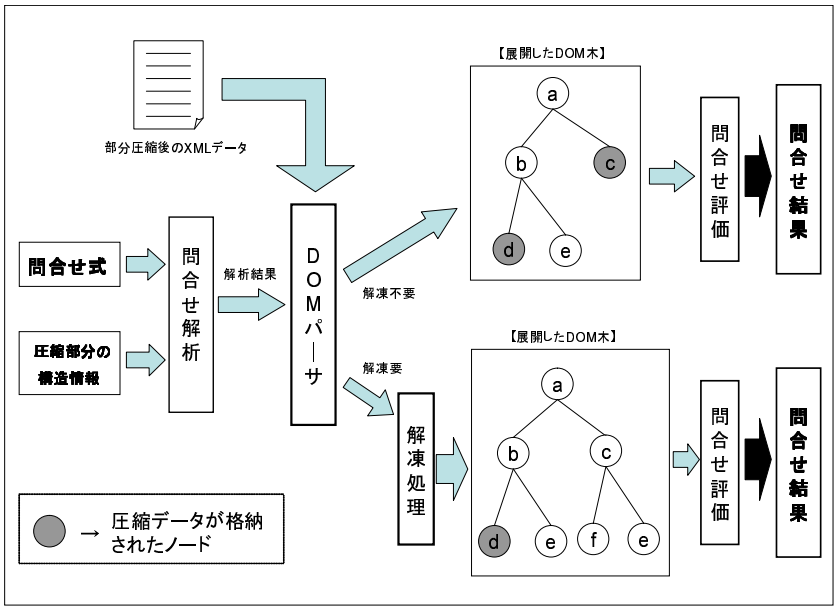

図 4.8 部分圧縮された XML データの DOM での処理例

図 4.8 の例では，問合せ解析の結果を受け，解凍処理が必要無い場合は部分圧縮されたデータをそのまま DOM 木として展開して処理を行う．また，解凍処理が必要な場合，処理に必要な部分のみを解凍した上で DOM 木を展開している．

なお，両方の処理方法について，解凍したデータを別途処理し，複数の結果を最終的にソートや結合によって統合して問合せ結果を得る処理等，他の方法も考えられるが，ここでは議論しない．

4.4 評価実験

4.4.1 実験環境

実験は表 4.3 で示す環境で行った．圧縮に用いたアルゴリズムは gzip であり，圧縮後の出力がバイナリ形式であるため，base64 アルゴリズムによってテキスト化を行った．

表 4.3 使用した計算機に関する情報

CPU	Pentium 4 1.80 GHz
メモリ	1 GB
HDD 容量	120 GB
圧縮アルゴリズム	gzip
テキスト化アルゴリズム	base64

XMark

実験データとして用いた XML データとその元となった研究について説明する．XMark (An XML Benchmark Project)[26] では， XML データベースの性能評価指標のために，規模変更可能 (scalable) な XML データの生成と，その XML データに対する二十種類の XQuery[24] 問合せを提供している．XML データの生成は，公開されたプログラムである xmlgen によってサイズ可変で自動生成することができる．また，xmlgen で生成される XML データのスキーマとして，DTD が公開されている[2] ．具体的にはインターネットの競売サイトに関するデータベースを模した構造となっており，根要素として site 要素があり，その子要素で region，categories，catgraph，people，open_auctions，closed_auctions の六つの要素へと枝分かれし，各要素下で更に複雑な構造を形成している．階層スキーマの概略図は図 4.9 のようになっている．

[2] 付録にて記載する

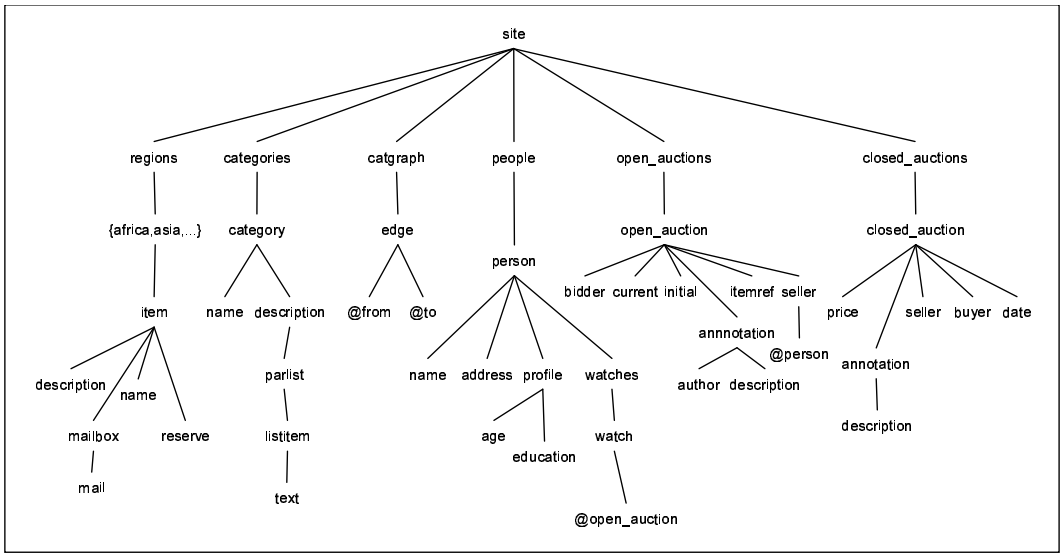

図 4.9 XMark のスキーマ

4.4.2 実験内容

実験は，図 4.7 で示すような SAX による XPath 問合せプログラムを実装し，xmlgen を用いて生成した 1 〜 100 MB の五つの XML データに対する問合せ処理時間を圧縮の前後で比較して行った．なお，プログラム言語には Java （J2SE 1.4.2）を用い，SAX パーサは公開されている Xerces2 Java[3] を用いた．

アクセスの偏り方（局所性および頻度）と提案手法との関連を調べるため，圧縮を判断する二つの閾値は同じ条件のもとで，アクセスの偏りが異なる二種類の問合せセット（A，B）とそれらの発行頻度を用いて XML データを部分圧縮処理し，それぞれに実験を行った．具体的な問合せセットの内容を次の表 4.4，4.5 で示す．A は局所性が高く，B は局所性が低い問合せセットとなっていることがわかる．また，圧縮を判断する閾値 α を 0.1 ，ホットスポットを判断する閾値 β を 0.3 とした．なお，4.1.3 項で述べた圧縮によって軽量化されない小さなデータを圧縮してしまわないよう，データサイズが 200 バイト以下の部分木は圧縮しないこととした．

[3] http://xml.apache.org/xerces2-j/

表 4.4 問合せセット A

問合せ式	発行頻度
/site/people/person	0.3
/site/open_auctions/open_auction/bidder	0.25
/site/open_auctions/open_auction/interval	0.15
/site/closed_auctions/closed_auction/annotation	0.15
/site/closed_auctions/closed_auction/price	0.1

表 4.5 問合せセット B

問合せ式	発行頻度
//item	0.1
//name	0.1
//mail	0.1
//person	0.1
/site/people	0.1
/site/open_auctions	0.1
/site/closed_auctions	0.1
/site/regions/namerica	0.1
/site/regions/europe	0.1
/site/categories/category/description/parlist	0.05

4.4.3 結果と考察

圧縮率

図 4.10 は，問合せセット A，B に基づいてアクセス頻度を算出し，部分圧縮を行った際の圧縮率を示すグラフである．圧縮率は，圧縮後のデータサイズと圧縮前のデータサイズの除算した結果を百分率で表している．

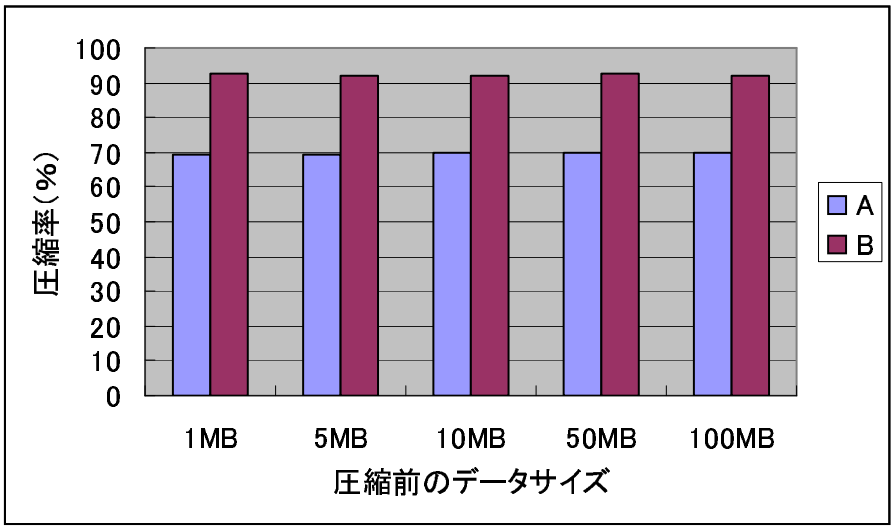

図 4.10 圧縮率

問合せセット A を用いた場合はおよそ 70 %，B を用いた場合はおよそ 92 %の圧縮率となった．局所性の高い A の方が圧縮される部分木が多くなることから圧縮率が高いことがわかる．具体的には，表 4.4 からもわかるように，A は site の六つの子要素のうち，三つの要素及びその子孫要素へのアクセスに集中している．一方，B では局所性が低いため圧縮される部分木の大きさが小さく，圧縮率が低い．このように，本手法における圧縮率は，用いる問合せセットのアクセスの局所性が深く関わる．

問合せ処理時間

部分圧縮後のデータへの問合せ処理時間結果を図 4.11，図 4.12 に示す．A，B に基づいて圧縮して得た二つの XML データに対し，特徴の異なる五種類の問合せを発行し，処理時間を計測した上で，圧縮前の XML データへの問合せ処理時間との比を算出した．この値が 1 未満の場合，圧縮前よりも問合せ処理時間が高速化している．なお，発行した問合せは問合せ処理時の解凍処理の特徴によって五つに分類できる．その特徴を次の表 4.6，4.7 に示す．

表 4.6 発行した問合せの特徴

問合せ	問合せ式	解凍処理
a	/site/people/person	必要無
b	/site/catgraph	必要有
c	/site/regions/asia	必要有
d	/site/closed_auctions/closed_auction/annotation	必要有
e	/site/open_auctions/open_auction/annotation/description/parlist/listitem/parlist	必要有

表 4.7 解凍処理が必要な問合せの特徴

問合せ	必要な解凍処理の回数	解凍したデータサイズの合計
b	少	小
c	少	大
d	多	小
e	多	大

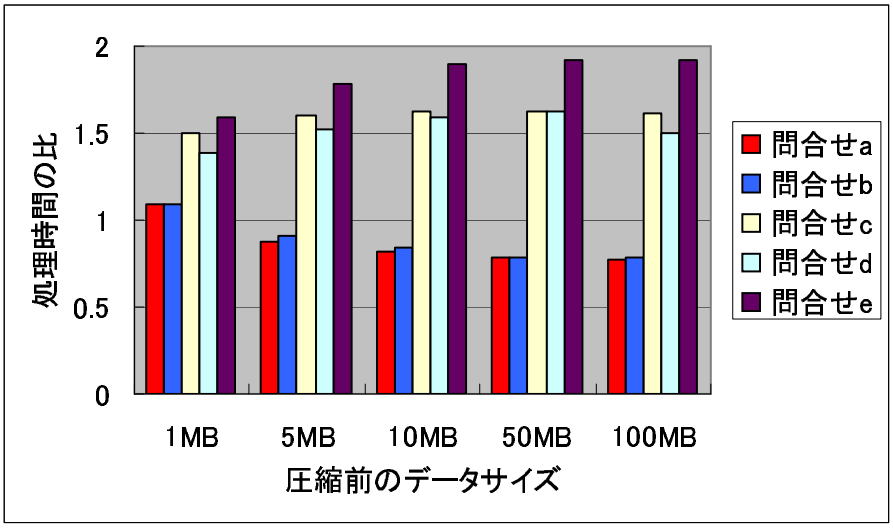

図 4.11 A に基づいて部分圧縮したデータへの問合せ結果

まず，A に基づいて部分圧縮したデータへの問合せ結果について考察する．問合せ処理時に解凍処理が必要無い問合せ a では，1 MB の XML データを使用した場合を除いて処理時間が高速化した．部分圧縮したデータへの問合せでは，問合せと構造情報から解凍に必要なデータが存在するかを解析する分の処理時間がかかる．1 MB のデータでは，この処理時間が問合せ処理時間全体に占める割合が高くなり，高速化していない．しかし，5 MB 以上のデータにおいては，処理時間が最大 22 % 高速化された．高速化の要因は，SAX 処理時のパース時間において，データサイズおよびノード数の減少によって発生イベントの数が減少したためである．

次に，問合せ処理時に解凍処理が必要な問合せ b から e では，b を除いて処理時間が遅くなった．問合せ処理に必要なデータが圧縮されていても，b のように解凍処理の回数が少なく，データサイズが小さい場合は圧縮されていても処理時間が高速化されることがわかった．しかし，解凍処理の回数や解凍するデータサイズが大きい場合では，解凍処理にかかる時間が大きくなるため，問合せ処理時間が遅くなった．しかし，圧縮が行われたデータはアクセス頻度が低いデータであるため，アクセス頻度に応じて処理時間が効率化されている．

なお，サイズの異なる 5 種類の XML データの間に存在する微量な誤差は，

xmlgen がランダムにデータを生成することによる誤差だと考えられる．

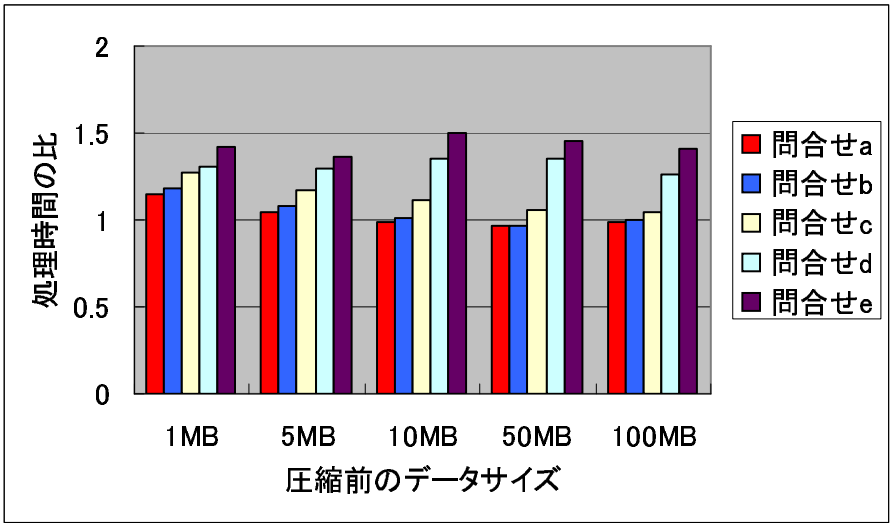

図 4.12 B に基づいて部分圧縮したデータへの問合せ結果

B に基づいて部分圧縮したデータへの問合せ結果について考察する．A とは異なり，ほとんど高速化していないことがわかる．これは，圧縮によるデータサイズの減少が A よりも少ない分，問合せ処理が高速化されないためである．a の場合，解凍処理が必要無いため，僅かではあるが問合せ処理時間は高速化されると考えられるが，構造情報から圧縮部分を解析するための処理時間が加わるため，ほとんどの場合で処理時間が遅くなっている．また，b から e の場合は A の場合と同様の理由で処理時間が遅くなっている．なお，処理時間の増加分が A の場合に比べて低くなっているのは，圧縮率から周知のように，圧縮されたデータ自体が A に比べて少なく，解凍処理の回数やそのデータサイズの絶対量が A に比べて少なくて済むためである．

以上，二つの結果から，提案手法の特徴と有効性を見ることができた．結果の考察から，アクセスするデータの局所性が高い XML データほど，提案手法により，アクセス頻度に応じて問合せ処理を効率化できる．また，XML データ内において，アクセス頻度が高く圧縮されない部分木に対する問合せはもちろん，圧縮された部分木への問合せであっても，解凍処理コストによって処理の高速化がみられた．

4.5 まとめ

本章では，アクセスの偏りに基づいて XML データを部分的に圧縮し，問合せ処理を効率化する手法について述べた．従来の処理系では，XML データへの問合せ処理時に XML データ全体を処理するものが多く，問合せの評価に必要な部分が小さくても XML データ全体のサイズが処理コストに大きな影響を与えてしまうという問題があった．特に，近年大規模な XML データが増加してきているため，このようなデータを効率的に処理したいという要求も高まってきている．

本章では，これらの要求と問題を解決するための手法として，アクセス頻度に基づいて XML データを部分木単位で圧縮してサイズを減らし，アクセス頻度の高いデータへの問合せ処理を効率化する手法を提案した．典型的な問合せセットとそれらの発行頻度を元にノード毎のアクセス頻度を算出し，圧縮の可否の判断に用いた．アクセス頻度の算出にデータサイズを考慮することで，圧縮効率に基づくアクセス頻度を算出した．また，アクセスが多くサイズが小さな部分をホットスポットとすることで，これらの部分が圧縮されないような工夫も行った．

評価実験を通して，アクセス頻度の低い部分木を圧縮することで，アクセス頻度の高い部分木への問合せを高速化できることが実験により確認できた．また，アクセス頻度が低く圧縮された部分木内のデータへの問合せであっても，解凍処理の規模が小さければ問合せ処理を高速化できることも分かった．

第5章　アクセス頻度を考慮した XML データの分割手法

5.1 提案手法の概要

前章で述べたように XML データの処理コストは XML データ全体のサイズに大きく左右されるため，大規模な XML データでは処理コストが高い．また，大規模な XML データを分散環境において処理したいという要求も近年増加してきている．ゆえに本手法では，XML データを複数の断片（XML データ）へと分割し，処理に必要なデータを含む断片のみに問い合わせることで処理を効率化する手法を提案する．分割は，元の XML データからノード集合（部分木）を抽出し，新たな断片（XML データ）として作成することで行う．たとえば，図 5.1 のような XML 木 で示す XML データを，図 5.2 と図 5.3 のような XML 木で示される二つの断片に分割した場合を考える．

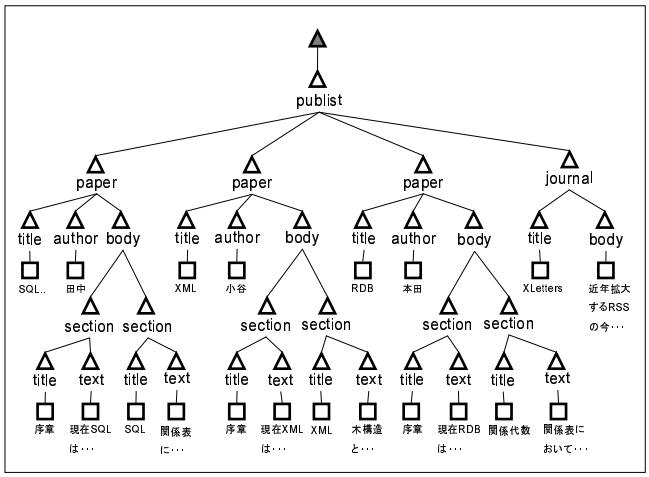

図 5.1　分割前の XML 木

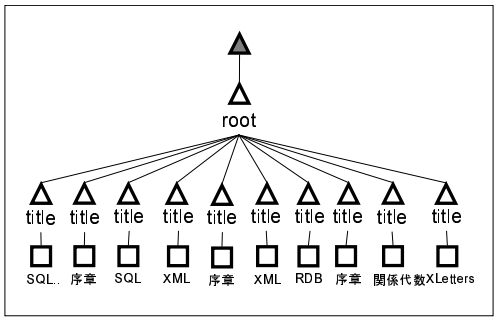

図 5.2 分割後の断片 (1)

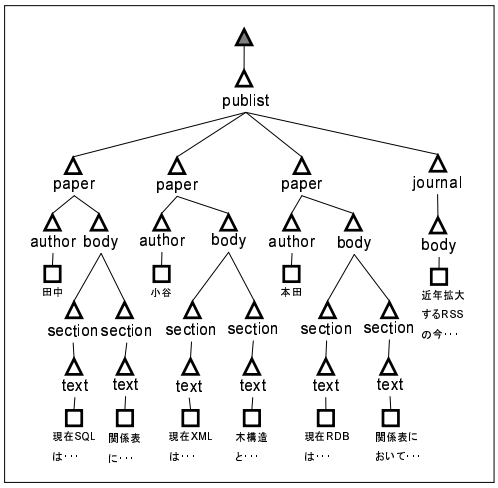

図 5.3 分割後の断片 (2)

例では，元の XML データを，title 要素のみを含む断片 (1) とそれ以外の部分を含む断片 (2) に分割している．これによって，title 要素のみを問い合わせる場合，元の XML データよりもサイズが小さい断片 (1) のみを処理すれば結果を得ることができ，処理の高速化が期待できる．また，author 要素を問い合わせる場合でも，断片 (2) のみを処理すればよいため高速化が期待できる．しかし，section 要素を問い合わせる場合，分割により子要素の title 要素が断片 (2) のみに存在しないため，両方の断片を処理する必要がある．このように，問合せに必要なデータが複数の断片に分かれて存在する場合，複数の断片から得られた問合せ結果を統合する必要がある．よって，データサイズの縮小による処理コストの

減少だけでなく，結果の統合にかかるコストも考慮する必要がある．本手法では，分割後の処理コストを評価するコスト関数をアクセス頻度を考慮して算出し，問合せ処理を効率化するように分割を行う．4 章と同様に，元の XML データへの典型的な問合せセットとその発行頻度が既知であることを仮定し，それらを元に各問合せでアクセスされる部分木の情報を得る．その情報を元に処理時のコストを考慮し，問合せ処理を効率化する断片を規定し，分割する．

また，分割前の XML データに対応する構造概要を DataGuide[6] に基づいて生成し，断片の規定や分割後の問合せ処理で利用する．分割時に，元のノードがどの断片に記述されているかを構造概要上で管理しておくことで，問合せ時に処理する断片の限定や各断片へ発行する問合せの変換に利用できる．

5.2 XML データの分割および構造概要の定義

まず，問合せ式に基づく XML データの垂直分割 (*vertical partitioning*) を次のように定義する [4].

【定義 (1)】XML データの垂直分割

垂直分割で規定される断片 f は $f = sf - \{ef_1, ..., ef_m\}$ の式で定義される．sf は選択される断片を指定する問合せ式で表現される．また，ef は除かれる断片を，sf が指す部分木の根ノードからの相対パスで指定した問合せ式によって表現される．ただし，$\{ef_1, ..., ef_n\}$ は無くてもよい．

[4] では XML データの水平分割 (*horizontal partitioning*) も定義しており，それは垂直分割で規定した断片 f において，sf, ef を表現する問合せ式が述語を含んでいる時，f を水平分割としている．本論文では問合せ式に述部を含まないため，水平分割は考えない．

次に，本手法で用いる XML データの構造概要を示す．XML を含む半構造のための構造概要はこれまで多く提案されているが，ここでは最も単純な Strong DataGuide[6] を用いる．

【定義 (2)】Strong DataGuide

情報源 s の DataGuide d とは，1) 全てのラベル経路が d において一つのデータ経路インスタンスを持ち，2) d の全てのラベル経路は s のラベル経路になっているものをいう．ここで，ノード o のラベル経路とは，o から辿ることのできるエッジ $(e_1, e_2, ..., e_n)$ のラベル系列 $(l_1, l_2, ..., l_n)$ である．また，ノード o のデータ経路とは，o から辿ることのできるノードとラベルの系列 $l_1, o_1, l_2, o_2, ..., l_n, o_n$ である．Strong DataGuide sd とは，このような d のうち，情報源における全ての経路式を任意の経路式のターゲット集合という視点から，s と d が区別できないものをいう．

図 5.1 の XML 木に対する構造概要を図 5.4 に示す．ただし，構造概要上の各ノードの左上に付加してある数字は，構造概要上での各ノードを識別する値であり，後に利用する．

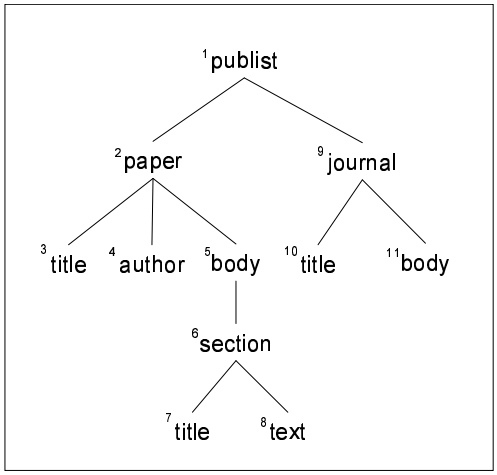

図 5.4 構造概要の例

5.3 分割処理

5.3.1 問合せ式による断片の規定

本手法では，4 章と同様に XML データに対する典型的な問合せセットとそれらの発行頻度が既知であることを仮定する．図 5.1 の XML 木で示した XML データに対する典型的な問合せセットとその発行頻度の例を表 5.1 に示す．

表 5.1 典型的な問合せセットとその発行頻度の例

問合せ	XPath 式	発行頻度
q_1	//title	0.5
q_2	//section	0.3
q_3	//author	0.1

問合せセットが既知である場合，あらかじめそれぞれの問合せ式によって定義 (1) に基づいて断片を規定すれば，問合せ処理時に必要な部分木のみが記述された断片を処理できるため効率的である．しかし，問合せセットは一般に多数あり，その全ての問合せ式で断片を規定すると断片の数が余りにも多くなってしまう．また，後に述べる重複部分の問題も生じるので，一般には容易に断片を規定することができない．そこで本手法では XML データを n 個の断片へと分割することを考える．ここで n は，想定する環境によって自由に決めることができる．例えば，分散環境ではプロセッサの数，分散ディスクの場合はスピンドルの数などである．

以上より，問題は XML データを m 個の問合せ式に基づいて n 個の断片に分ける問題となる．ここで一般に $m > n$ である．提案手法では，まず与えられた m 個の問合せ式で規定される断片を，断片の候補としておき，その m 個の断片のいくつかを併合することで n 個にまで減らすことを考える．併合の決定は，問合せ処理のコストに問合せの発行頻度を考慮したコスト関数を用いる．これにより，問合せ処理を効率化する分割を行うことができる．

重複するノードの処理

一般に問合せ式による問合せ結果となるノード集合は独立ではなく互いに重複部分を持つ場合があるため考慮が必要である．例として，図 5.1 において問合せ $q_1 =$ //title と $q_2 =$ //section が指定するノード集合を考える．この場合，title 要素のいくつかが section 要素に含まれている．よって，このような q_1, q_2 に基づく断片 f_1, f_2 を求める場合，次のような方針が考えられる．

1) 重複部分を複製しない

2) 重複部分を複製して f_1 と f_2 両方にもたせる

2) は問合せの処理速度の面から有利であるが，余分な記憶容量を必要とすることや更新の際に余計なコストがかかることになる．以上の理由から本手法では，重複部分の複製は考えないこととする．

1) の方針はさらに下の二つのケースに分けることができる．

1a) 重複部分を新たな断片とし，元の問合せ式に対応する断片からはその断片を除く

1b) 重複部分をどの問合せ式に対応する断片に持たせるかを，頻度の高さや処理コストなどから一意に決定する

本手法では，1b) の方針を取り，重複部分は発行頻度が最も高い問合せ式に基づく断片が持つこととする．このことから，他の断片に完全に包含されてしまうような断片において，その断片を規定する問合せの発行頻度がより低ければその断片は除外されることとなる．重複部分を持つ二つの問合せによる断片の規定について以下に示す．問合せ式 q を評価するために必要なノード集合を $v(q)$ で示し q の発行頻度を $freq(q)$ で示すとき，q_1, q_2 $(freq(q_1) > freq(q_2))$ なる問合せから二つの断片 f_1, f_2 を規定する場合を考える．$v(q_1) \cap v(q_2) = \phi$ であれば重複部分が無いので，各断片は，$f_1 = q_1, f_2 = q_2$ で規定すればよい． $v(q_1) \cap v(q_2) \neq \phi$ のとき，重複部分 $v(q_1) \cap v(q_2)$ は発行頻度がより高い q_1 で規定された f_1 にのみに含める．重複部分を示す問合せ式は q_1, q_2 と元の XML データの構造概要から作成できる．その問合せ式を q_{2-1} とすると，各断片は，$f_1 = q_1, f_2 = q_2 - \{q_{2-1}\}$

で与えられる．たとえば，表 5.1 の問合せ式 //title と //section の場合，構造概要（図 5.4）から，重複部分が存在することがわかる．さらに，構造概要から重複部分を示す問合せ式を，section 要素からの相対パスで ./title のように表現できる．問合せの発行頻度から，重複部分は //title の規定する断片が持つため，重複部分が除かれた，//section を元に規定される断片の式は //section − {./title} となる．

問合せセットに基づく断片の候補

重複するノードを前述のように扱うことで，問合せセットに含まれる問合せ式の数だけ候補となる断片を決定できる．表 5.1 の問合せセットを用いて 図 5.1 の XML データを分割する場合，候補となる断片は表 5.2 のようになる．ただし，f_{remain} は，全ての問合せ式から指定されないノード集合からなる断片を示すものとする．表 5.3 のように，各断片を規定する式に加え，データサイズやノード総数，処理対象とされる問合せ等，問合せの処理コストに関わる情報を収集しておき，後で利用する．ただし，表 5.3 におけるデータサイズは図 5.1 を元に設定した概数である．

表 5.2 問合せセットの各問合せ式に基づく断片の候補

断片の候補	断片を規定する式
f_1	//title
f_2	//section
f_3	//author
f_{remain}	/publist − {./paper/title, ./journal/title, ./paper/author, ./paper/body/section}

表 5.3 断片の候補に関する情報

断片	データサイズ	ノード総数	処理対象とする問合せ
f_1	200	10	q_1, q_2
f_2	12000	12	q_2
f_3	60	3	q_3
f_{remain}	5000	9	-

断片の併合

断片の候補を併合することによって n 個の断片を決定する．断片の併合を考慮した XML データの垂直分割を，定義 (1) の拡張によって次のように定義する．

【定義 (1)'】併合を考慮した XML 文書の垂直分割

断片の併合によって規定される断片 f は $f = \{sf_1, ..., sf_n\} - \{ef_1, ..., ef_m\}$ の式で定義される．sf は選択される断片を指定する問合せ式，ef は除かれる断片を指定する問合せ式によって表現される． ただし，$\{ef_1, ..., ef_n\}$ は無くてもよい．なお，併合を考慮した断片の規定では，sf を表す問合せ式が複数存在するため，定義 (3) とは異なり ef は sf の問合せ式のいずれかを接頭辞とする絶対パスで指定した問合せ式によって表現する．

m 個の断片を併合によって n 個にまで減らす場合，併合する断片の組合せは多数考えられる．本手法では，併合後に問合せセットで問合せ処理を行った場合の処理コストとアクセス頻度からコスト関数を定めその値から併合の組合せを決定する．コスト関数が最小となる組合せで併合を行うことで，問合せ処理を最も効率化するように併合することができる．問合せセット全体を処理するコスト関数 C_{total} は次式によって算出する．

$$C_{total} = \sum_{k=1}^{m} freq(q_k) \cdot C(q_k)$$

ここで $freq(q_k)$ は q_k の発行頻度，$C(q_k)$ は q_k の評価にかかる処理時間コストである．$C(q_k)$ は問合せに必要な全ての断片に問合せを発行し，評価するコスト

C_{eval} と，必要に応じて結果を統合するためのコスト $C_{integration}$ の和で次のように与えられる．

$$C(q_k) = C_{eval}(q_k) + C_{integration}(q_k)$$

$C_{eval}(q_k)$ は 必要なノードが含まれる断片全てに対して q_k を評価するためのコストである．たとえば，断片 f_a 内に q_k の評価に必要な全てのノードがあれば，評価する断片は f_a のみでよい．しかし，そうでない場合，f_a だけでなく重複部分を持つ他の断片も評価する必要がある．f_a 以外で q_k にどの断片が必要になるかは，問合せセットから規定される断片の情報（表 5.3）から判断することができる．また，$C_{integration}(q_k)$ は，複数の断片から得た結果を結合やソートによって統合するためのコストである．

C_{eval} は問合せの処理方法によって異なる．処理方法が既知の場合はそれに関する情報を収集しておき，推測値として算出する．ただし，処理方法が未知の場合や，分割後のデータへの処理方法が複数ある場合は，ノード数やデータサイズ等から推測する．また，$C_{integration}$ についても処理方法の他，結合処理の方法によって異なる [1]．たとえば，Stack-Tree Join アリゴリズムの一手法である Stack-Tree-Desc に基づいて結合処理を行う場合，結合の候補となる先祖のノード集合のサイズ，子孫のノード集合のサイズ，結合結果のノード集合のサイズの合計に計算量が依存するため，表 5.3 で示したような断片の候補に関する情報と構造概要から該当するノード数を割り出して計算する．

併合の組合せ毎に C_{total} を計算する．併合の組合せの数だけ算出できる C_{total} の中で最小のものが問合せ処理を最も効率化する組合せとなる．しかし，m 個の断片の候補を n 個に併合するとき，その全ての組合せにおけるコスト関数を算出する計算量は $O\left(n^m\right)$ となってしまう．よって，本手法では，コスト関数の最小値の近似解を，併合の組合せ最適化問題の近似解法を適用して求める．

近似解法

本手法では，近似解法として欲張り法 (Greedy Method) に基づいてコスト関数の値が最小値に近くなるよう併合する断片の組合せを決定する．

欲張り法とは，目的関数値の良さを示す局所的評価（部分問題の解）に基づいて，可能解を直接構成していく方法である．部分問題毎に局所最適解を求め，それを繰り返すことで最適解の近似解を得る．

本手法では，欲張り法に基づいて，最適な併合の組合せの近似解である組合せを求める．具体的には，二つの断片のみを併合した場合の C_{total} を局所的評価（部分問題）とし，C_{total} が最も低くなる二つの断片の併合を行う．次に，$m-1$ 個となった断片に対し，同様に C_{total} を求め，併合を行う．このような一組ずつの併合を部分問題とすることで，併合を $m-n$ 回行うことで最終的な併合の組合せを決定する．これにより，組合せ数が m^3 に依存する数にまで限定できるため，コスト関数を算出する計算量は $O\left(m^3\right)$ となる．

欲張り法に基づいて併合を行うアルゴリズムを図 5.5 に示す．ただし，*addCombineFlag* (f_a, f_b) は，f_a と f_b を併合するようフラグ付けし，F に戻す関数とし，*omitCombineFlag* (f_a, f_b) はそのフラグを解除する関数とする．

```
アルゴリズム combineFragments (F : 断片の集合, Q : 問合せセット)
begin
  if m > n
    combineTwoFragments(F, Q)
  endif
  return F
end

アルゴリズム combineTwoFragments (F : 断片の集合, Q : 問合せセット)
begin
  foreach (f_a, f_b) ∈ F から選んだ 2 つの断片
    addCombineFlag (f_a, f_b)
    C_total = calculateCost (Q, F)
    if C_total < oldC_total or oldC_total == null
      oldC_total = C_total
      f_A = f_a, f_B = f_b
    endif
    omitCombineFlag (f_a, f_b)
  end
  addCombineFlag (f_A, f_B)
  m = m − 1
  return m, F
end

アルゴリズム calculateCost (Q : 問合せセット, F : 断片の集合)
begin
  C_total = 0
  foreach q in Q
    C_total = C_total + freq(q) · C(q)
  end
  return C_total
end
```

図 5.5 欲張り法による併合アルゴリズム

例として，表 5.3 の四つの断片の候補を二つの断片に分割するために併合する場合のコスト関数を考える．ただし，簡単のため，C_{eval} は問合せに必要な断片のノード総数のみに，$C_{integration}$ は問合せ結果のノード総数のみに依存するものとして計算する．たとえば，f_3 と f_{remain} を併合する場合の C_{total} を考える．この場合，問合せ毎の $C(q_k)$ はそれぞれ，$C(q_1) = C_{eval}(q_1) + C_{integration}(q_1) = 10 + 0 = 10, C(q_2) = (10 + 12) + 16 = 38, C(q_3) = 12 + 0 = 12$ となり，各々の問合せの発行頻度との積から $C_{total} = 17.6$ となる．また，同様に，f_1 と f_2 を選んだ場合は

$C(q_1) = 24+0 = 24, C(q_2) = 24+0 = 24, C(q_3) = 3+0 = 3$ となり，$C_{total} = 19.5$ となる．f_1 と f_3，f_2 と f_3 など，他の組合せについても同様に C_{total} を求める．このようにして最初の併合での C_{total} が最小となるもの見つけ，それを起点として併合を繰り返していく．この場合，f_3 と f_{remain} の併合が起点となり，併合後の三つの断片で再び併合を行っていく．最終結果として，f_1 と f_2，f_3 と f_{remain} を併合してできる二つの断片が C_{total} が低くなる結果として選ばれるため，分割後の二つの断片 f_A, f_B は表 5.4 のように規定できる．

表 5.4 併合後の断片を規定する式

断片	断片を規定する式
f_A	{/publist/paper/title, /publist/paper/body/section, /publist/journal/title}
f_B	{/publist} − {/publist/paper/title, /publist/journal/title, /publist/paper/body/section}

5.3.2 分割時の処理

併合後，断片を規定する式に基づいて XML データを n 個の断片へと分割する．ただし，断片は問合せ式から規定されるため，そのままでは最上位の要素が複数存在する場合がある．XML では最上位の要素を一つに定めているため，分割後のデータを XML 形式のまま保持したい場合は各断片の根要素となる要素を生成しておく必要がある．以下では，分割後のデータを XML 形式のファイルとして保持する場合について説明する．しかし，本手法は，XML データベース，リレーショナルデータベース，ファイルとしての保存等，XML データの格納（記述）方法に関わらず，様々な格納方法で利用できると考えられる．ただし，格納方法やその際の処理方法によって処理コストが何の影響を受けるかが変化するため，処理コストの推測に用いる要因を適宜選択する必要がある．

問合せ処理のための情報付加

分割処理時に，分割によって破断されるエッジの両端（親子）のノードと構造概要にそれぞれ情報を付加しておく．破断されるエッジの両端のノードには，二つの情報を付加する．一つは，結合時に利用する，エッジの識別子である．破断されるエッジの両端のノードは，必要に応じて再び結合によって構造を復元する必要があるため，互いが結合先のノードを特定できるような識別子を付加する．本手法では，元の XML 文書における文書順を識別子として用いる．これにより，結合だけでなく，文書順でのソートが必要な場合でも利用することができる．二つ目は，構造概要上でのノード識別子である．破断後の子となるノードでは，根から親までの経路に関する情報を失うため，根からの経路が異なる同名の要素との区別がつかない可能性がある．たとえば，図 5.1 の XML 文書を表 5.4 に示す式で分割した場合，f_A に含まれる title 要素は，そのままでは paper 要素の子要素なのか，journal 要素の子要素なのかを判別できなくなる．このため，構造概要上でのノード識別子により，同名のノードが識別不可能になることを防ぐ．

構造概要には，ノード毎に記述先となる断片の情報を付加する．図 5.6 は，表 5.4 のように分割した場合に，構造概要上の各ノードに記述先を付加したイメー

ジ図である．実際には各ノードに格納先を示す値を持たせるが，図 5.6 では格納先が同じノードを同じ色からなる範囲で示してある．これにより，問合せ時に問合せの解析を行うことができる．

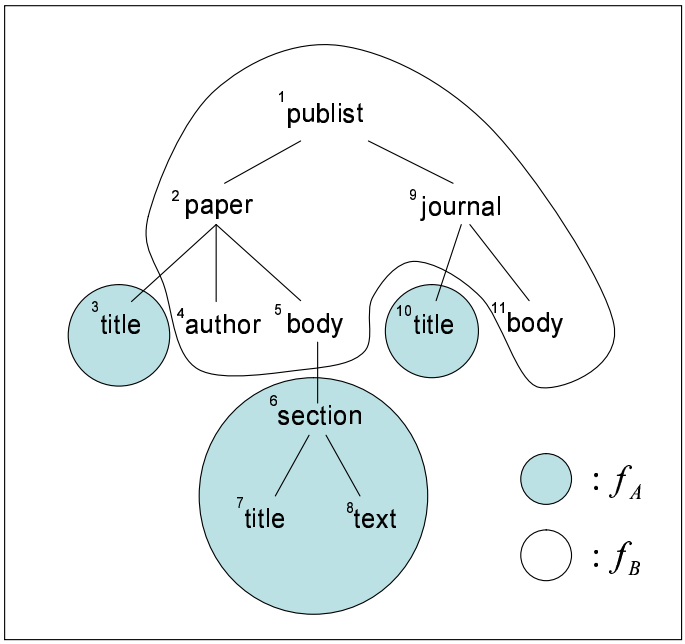

図 5.6 格納先を記述した構造概要

5.4 問合せ処理

分割後の各格納先への問合せ処理は図 5.7 のような流れとなり，問合せの解析，変換，結果の統合から実現される．

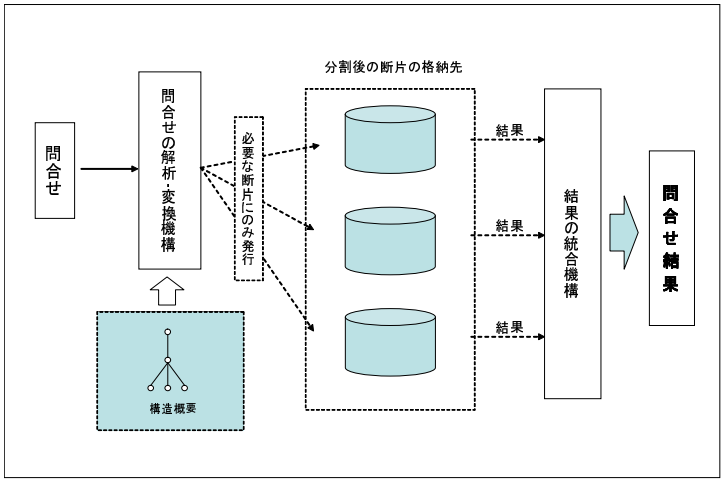

図 5.7 問合せ処理の流れ

5.4.1 問合せの解析と変換

分割後の問合せ処理では，構造概要を用いて問合せの解析と変換を行う．分割後，構造概要上の各ノードには 5.3.2 項で述べたように，各ノードの記述先である断片を示す値が付加されている．よって，構造情報を用いて問合せの解析，変換を行うことができる．具体的には，問合せ式の経路と一致するノードを元に構造概要をたどり，記述先の断片と，そこに発行すべき問合せを生成する．構造概要を用いて問合せを解析，変換するためのアルゴリズムを図 5.8 に示す．ただし，*MatchQuery(sd, q)* は構造概要 sd から問合せ式 q の経路式と一致するノードを取り出す関数とし，*addStep(v, q_v)* は，ノード v の名前からなるノードテストを先頭に加えて問合せ式 q_v を再構成する関数とする．また，*getExistFragment(v, sd)* は構造概要 sd からノード v の記述先を取り出す関数，*getParent(v)* は，ノー

ド v の親ノードを取り出す関数とする．また，返される Q_{trans} は変換した問合せ（発行先の情報を含む）の集合である．

```
アルゴリズム analyzeQuery (sd = (V_sd, E_sd) : 構造概要, q : 問合せ)
begin
  Q_trans = ∅
  foreach v in MatchQuery(sd, q)
    Q_trans = Q_trans ∪ makeQuery(v, sd)
    Q_trans = Q_trans ∪ makeQueryDesc(v, sd)
  end
  return Q_trans
end

アルゴリズム makeQuery (v : ノード, sd : 構造概要)
begin
  R = ∅
  q_v =addStep(v, q_v)
  ef_v = getExistFragment(v, sd)
  u = getParent(v)
  while ef_v == getExistFragment(u, sd)
    q_v =addStep(u, q_v)
    u = getParent(u)
  end
  R = (ef_v, q_v)
  return R
end

アルゴリズム makeQueryDesc (v : ノード, sd : 構造概要)
begin
  R = ∅
  q_v =addStep(v, q_v)
  ef_v = getExistFragment(v, sd)
  foreach u ∈ v の子ノード
    if ef_v == getExistFragment(u, sd)
      makeQueryDesc(u, sd)
    else
      q_v =addStep(u, q_v)
      ef_u = getExistFragment(u, sd)
      R = (ef_v, q_v)
    endif
  end
  returnR
end
```

図 5.8 問合せの解析，変換アルゴリズム

たとえば，前節で示した例のように図 5.1 の XML データを表 5.4 に基づいて

分割したときに問合せ式 //section を処理する場合を考える．まず，構造概要上の section 要素から，分割後の section 要素が f_A のみに存在することがわかる．次に，構造概要の section 要素を先祖，子孫の方向へたどる．section 要素の親である body 要素は異なる断片に記述されているため，断片 f_A へ発行する問合せ式は断片の最上位に生成した要素を root として/root/section となる．また，子孫方向では異なる断片に記述された要素が無いことがわかる．よって，問合せ式 //section は，断片 f_A への問合せ式 /root/section に変換できる．また，//journal を処理する場合，同様に構造概要上の journal 要素から，まず記述先 f_B を得ることができる．この場合，先祖方向は publist 要素まで断片 f_B に記述されていることがわかるため，断片 f_B への問合せは /root/publist/journal となる．次に，子孫方向への探索から，title 要素が断片 f_A に記述されていることがわかるため，断片 f_A への問合せ /root/title も併せて生成できる．ただし，断片 f_A 内に存在する title 要素は，元の XML 文書において paper 要素の子要素であるものと journal 要素の子要素であるものの両方を含む．このため，破断したノードが持つ構造概要上の識別子が 10 という条件でフィルタリングを行う．最後に，得られた結果を，破断されたエッジ部分が持つ情報の比較によって統合することで，最終的な結果を得ることができる．結果の統合については，次で述べる．

5.4.2 結果の統合

問合せの変換後，発行先が複数の断片となった場合，それぞれの断片から得た結果をソートや結合によって統合する必要がある．結果の統合は破断したエッジ部分に付加した情報を利用することで実現できる．統合してできた結果が最終的な問合せ結果であり，分割前の XML データへの問合せ結果に等しいことが保証される．たとえば，図 5.1 の XML データを 表 5.4 に基づいて分割した後，前節のように問合せ //journal を評価する場合，f_A と f_B から得た結果のうち，破断されたエッジの両端にあたるノードを結合する必要がある．この場合，各断片から得られた結果のうち，title 要素と journal 要素は，5.3.2 項で示した，破断されたエッジの両端のノードであるため，結合に必要な情報を持つ．よって，title 要素と journal 要素からエッジの識別子が一致するものを結合する．また，// を

含むような問合せでは，複数の断片から得た結果を，必ずしも結合を行わなくてもソートする必要がある．本手法では，破断したエッジの識別子に文書順を利用しているため，その値によってソートを行うことができる．ただし，エッジの識別子には文書順以外にも，Dewey Order[11] に基づく値等，他の識別子を利用してもよい．

5.5 評価実験

5.5.1 実験環境

実験に用いた計算機は 4 章の表 4.3 と同様のものである．実験のため，分割および問合せのプログラムを SAX を用い，Java 言語により実装した．なお，用いた SAX パーサも 4 章 と同様に Xerces2 Java を使用した．また，実験に使用したデータは XMark プロジェクト [26] で公開配布されている XML 文書生成プログラム xmlgen によって 1 MB，5 MB，10 MB の XML データを生成して用いた．

5.5.2 実験内容

実験は，問合せセットとそれらの発行頻度，分割する数を与えた状況において，(1) 断片の候補の併合による組合せの決定，および (2) 分割前後での問合せ処理時間の比較を行った．

問合せセット及びそれらの発行頻度として，表 5.5 に示す 7 種類の問合せ式を用いた．これらの問合せ式に基づく表 5.6 のような 8 個の断片の候補を，三つの断片へと併合し，分割を行った．なお，今回は SAX によって問合せ処理（結果の統合を含む）を行うことから，コスト関数の処理コストは，イベント数に基づいて計算し，C_{eval} は処理する断片をパースする上で発生するイベント数，$C_{integration}$ は問合せ結果となるデータのパースで発生するイベント数に基づいて計算した．

表 5.5 実験で用いた問合せセットとそれらの発行頻度

問合せ	問合せ式	発行頻度
q_1	//date	0.3
q_2	//interval/start	0.2
q_3	//interval	0.15
q_4	//closed_auction	0.125
q_5	//person/name	0.1
q_6	/site/region/asia	0.05
q_7	/site/catgraph	0.025

表 5.6 問合せセットに基づく断片の候補

断片の候補	断片を規定する式	イベント数	処理対象とする問合せ
f_1	//date	3033	q_1, q_4, q_6
f_2	//interval/start	360	q_2, q_3
f_3	//interval − {./start}	962	q_3
f_4	//closed_auction − {./date}	7583	q_4
f_5	//person/name	766	q_5
f_6	/site/region/asia − {./item/mailbox/mail/date}	1597	q_6
f_7	/site/catgraph	30	q_7
f_8	/site から f_1〜f_7 を除いた部分（紙面の都合上省略）	53410	-

5.5.3 結果と考察

併合の結果

併合の結果決定した三つの断片は表 5.7 のようになった.

表 5.7 併合結果

断片	併合した断片の候補	イベント数	処理対象とする問合せ
f_A	f_1, f_4	10616	q_1, q_4, q_6
f_B	f_2, f_3, f_5, f_6, f_7	3715	q_2, q_3, q_5, q_6, q_7
f_C	f_8	52410	-

本手法では，コスト関数として処理コストに関わる要因（実験ではイベント数）と発行頻度が考慮されている．処理コストの面から併合が起こりやすい組合せは二つ考えられる．一つは併合によって問合せで処理する断片の数が一つになり，統合が不必要になるような組合せである．この場合，併合によって統合の必要が無くなるため，統合にかかる処理コスト分だけコスト関数の値が減少するため併合が起こりやすくなる．用いた問合せセットの場合，表 5.6 から統合の必要な問合せは q_3, q_4, q_6 であるとわかる．表 5.7 の結果において，断片の候補 f_1 と f_4，および f_2 と f_3，はこの理由によって併合が行われたと考えられる．

二つ目は，処理コストの少ない組合せである．この場合，併合によるコスト関数の値の増加が少ないため，併合が起こりやすくなる．表 5.7 では，比較的処理コストの少ない f_2, f_3, f_5, f_6, f_7 がこの理由によって併合されたと考えられる．

しかし，併合によって統合の必要が無くなるにもかかわらず，断片 f_1 と f_6 の併合は行われなかった．これは，発行頻度による影響が強いと考えられる．問合せ q_1 と q_6 では発行頻度の差が大きいため，併合によって q_1 にかかる処理時間のコスト増加の比重が大きくなるためコスト関数の値が高くなる．このことから，f_1 と f_6 の併合よりも他の併合が優先されたと考えられる．

問合せ処理時間の結果

次に，問合せセットの 7 つの問合せにかかる問合せ処理時間を表 5.7 に基づく分割の前後で比較した．実験結果はデータサイズごとに 図 5.9, 5.10, 5.11 のようになった．ただし，処理時間を分割前後の比で表現しているため，グラフの高さが 1 以下のとき，分割によって処理が高速化したことを示している．なお，問合せ処理時間を問合せの解析，断片の評価，結果の統合のそれぞれにかかった処理時間によって色分けしてある．

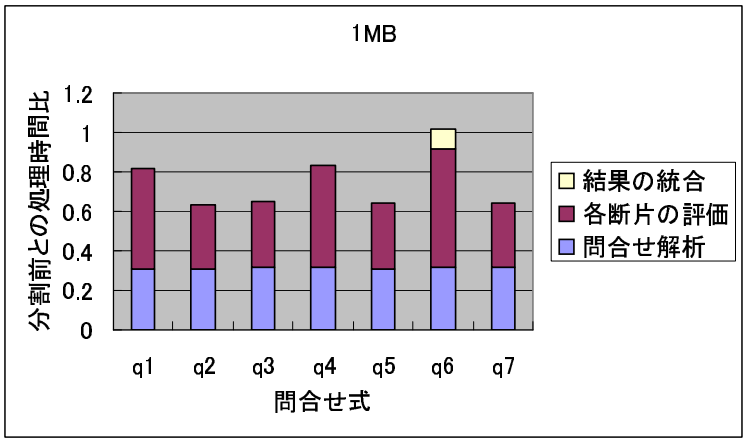

図 5.9 分割前後の処理時間比（1 MB）

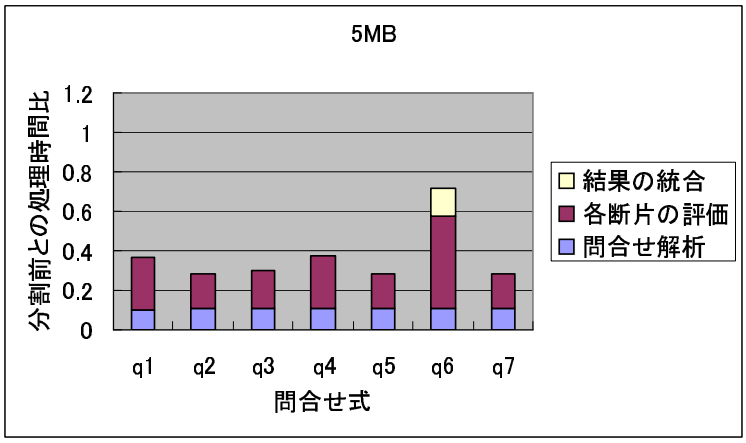

図 5.10 分割前後の処理時間比（5 MB）

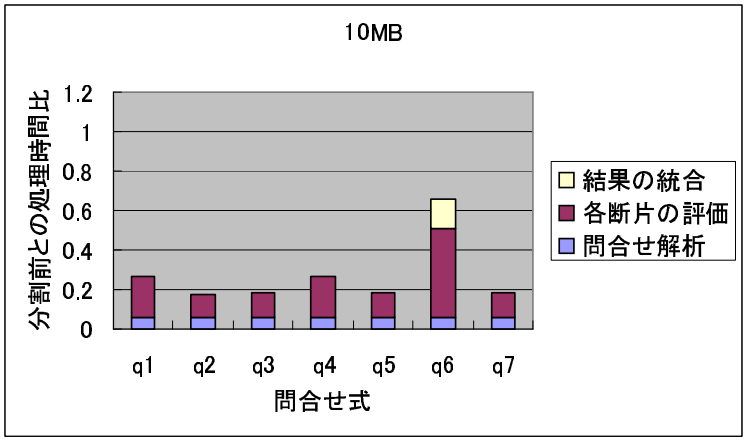

図 5.11 分割前後の処理時間比（10 MB）

実験の結果，ほとんどの場合で問合せ処理の高速化が確認できた．用いた XML データのサイズが大きくなるにつれて処理が高速化しているのは，パースやファイルの読み込みにかかるオーバヘッド分の処理時間の影響が少なくなることに起因すると考えられる．問合せ処理がどの程度高速化されるかは分割後の断片のデータサイズに依存する．これは，問合せに必要な部分の局所性に関連して変化する．

今回の実験では，元の XML データに対して f_A が約 17 %，f_B が約 6 %のサイズとなり，元の XML データのサイズが異なってもほぼ一定であった．このことから，元の XML データのサイズを大きくした場合，およそこれらの割合の程度まで高速化されると考えられる．

問合せの解析にかかる処理時間は元の XML データのサイズに関係無くほぼ一定であった．これは，使用したデータが同じスキーマに基づいて生成されていることから，XML データのサイズが増加しても構造概要のサイズがほぼ一定だったためである．このことから，構造概要を使用した本手法は，スキーマが同じデータであれば，XML データのサイズが大きいほど問合せの解析にかかる時間の割合が少なくなり有効であると考えられる．

結果の統合にかかる処理時間は 5.3.1 項で述べたように統合の処理方法によって異なる．今回の実験では，問合せ q_6 において統合が必要となった．q_6 の場合，問合せ結果の規模が小さいことや，統合に用いた SAX での処理の影響などの要因からそれほど統合の時間はかかっていないが，問合せ結果の大きさやそれに伴うファイルへの一時的な書き込みの必要性の有無など，実際にはより処理時間がかかるケースも考えられる．それらを判断し，併合時の処理時間コストを算出することで，処理コストが高い統合は併合時に回避される．

5.6 まとめ

本章では，アクセスの偏りに基づいて XML データを分割し，問合せ処理を効率化する手法について述べた．前章でも述べたように，従来の処理系では XML データへの問合せ処理時に XML データ全体を処理するものが多く，アクセスの偏りに関係なく，XML データ全体のサイズが処理コストに大きな影響を与えてしまうという問題があった．特に，近年大規模な XML データが増加してきているため，このようなデータを効率的に処理したいという要求も高まってきている．また，XML データを分散環境で管理，処理したいという要求も高まってきている．

本章では，これらの要求と問題を解決するための手法として，アクセス頻度に基づいて XML データを分割し，問合せ時に必要なデータのみが記述された断片（部分 XML データ）を処理することで問合せ処理を効率化する手法を提案した．典型的な問合せセットとそれらの発行頻度を元にアクセスの偏りを求め，分割部分の決定に利用した．XPath に基づく問合せ式による XML データの分割を定義し，与えられた問合せ式から断片の候補を決定した．それらの候補を元に，任意の数で分割可能にするため，断片の併合を考慮した分割を定義し，頻度と処理コストを元に算出するコスト関数によって併合の組合せを決定した．これにより，任意の数に分割する際に，問合せ処理を効率化されるような断片（分割後の部分 XML データ）を決定することができた．また，分割後の問合せを Strong DataGuide に基づく構造概要を利用することで実現した．構造概要を用いた，問合せ時の解析・変換方法や，そのために必要な構造概要および分割したデータ上で管理する情報についても提案を行った．

評価実験を通して，アクセスの偏りを元に，処理コストと頻度の両方を考慮して任意の数へと分割できることが確認できた．また，断片の候補の特徴と併合のされやすさについても知見を得ることができた．さらに，分割前後の問合せ処理時間の変化から，アクセスの偏りに基づいて問合せ処理を高速化できることを確認した．処理時間について，構造概要による問合せの解析処理が，スキーマが同じ XML データであればサイズの増加に対してほぼ一定であるという知見を得ることもできた．

第6章　結論

6.1 まとめ

本論文では，アクセスの偏りを考慮して XML データへの問合せ処理を効率化する二つの手法について述べた．

一つ目は，アクセス頻度を考慮した XML データの部分圧縮手法であり，XML データへの問合せセットから部分木のアクセス頻度を算出し，それをに基づいて部分圧縮を行うことで，問合せ処理を効率化できた．また，実験結果から，アクセスの局所性が高いほど提案手法が有効であることや，圧縮されたデータであっても，解凍処理のコストによって問合せ処理が高速化されることが確認できた．

二つ目は，アクセス頻度を考慮した XML データの分割手法であり，XPath に基づく問合せ式によって XML データの分割を定義した上で，問合せセットの各問合せ式から複数の断片の候補を決定した．それらの断片の候補を，処理コストと使用頻度を考慮したコスト関数に基づいて併合することで，任意の数の断片へと分割する手法を提案した．また，実験結果から，アクセスの偏りを考慮した分割と，それによる問合せ処理の高速化が実現できることを確認できた．

6.2 今後の課題

提案した二つの手法に共通して挙げられる課題として，述語をはじめとした，より複雑な問合せ表現への対応が挙げられる．述語の導入により，部分圧縮手法で利用した構造情報や，分割手法で利用した構造概要の拡張が必要になると考えられる．これらの拡張は，構造概要の表現に柔軟性を持たせることで，構造概要自体のサイズを多少犠牲にすれば実現できると考えられる．また，現在の構造情

報，構造概要についても，更なる改善点の調査が必要である．XML データには，要素の内容や，XML 木の構造に対する更新処理が行われるものも多いため，二つの手法の更新への適応も今後の課題といえる．

部分圧縮手法の課題としては，圧縮時の符号化やテキスト化で用いるアルゴリズムについての調査が挙げられる．部分木のテキストデータの特性や，処理速度等の特徴に応じて，符号化・テキスト化アルゴリズムを適宜選択できれば，問合せ処理を効率化できると考えられる．

分割手法の課題としては，問合せの処理コストと発行頻度による分割だけでなく，分割後の断片のデータサイズの指定や均一化など，より細かな要求に柔軟に対応できるような拡張が考えられる．また，併合の組合せを最適化するために用いたアルゴリズムについては，本手法で用いた欲張り法以外にも多くの手法が存在する．これらを調査し，本手法の性質や，最適化にかかるコストを考慮してより最適なアルゴリズムを選択できれば，問合せ処理をさらに効率化できると考えられる．問合せの指定するノードが重複した場合に，ノードを複製して複数の断片に保持させるケースの検討についても今後の課題である．

謝辞

本研究は，奈良先端科学技術大学院大学 情報科学研究科 植村 俊亮教授のもと，同研究科データベース学講座にて行われたものである．本研究の遂行にあたり，多くの皆様方に御指導と御協力を頂きました．ここに深く感謝の意を表すとともに，厚く御礼申し上げます．

植村 俊亮 教授には，御多忙中にもかかわらず主指導教官になって頂き，深く御礼申し上げます．植村先生には，私に本研究の機会を与えて頂き，さらに日頃から論文の執筆，発表の指導，進路の相談に至るまで数多くの御指導，御鞭撻を賜りましたことを心より感謝すると共に厚く御礼申し上げます．

関 浩之 教授には，御多忙中のところ，指導教官をお引き受け頂き，本研究に関する貴重な議論，御意見を頂くことができ，その後の研究に生かすことができました．心より感謝いたします．

松本 健一 教授には，御多忙の中にもかかわらず指導教官になって頂き，本研究に関して熱心なる御指導，御助言を頂きましたこと，心より感謝いたします．

宮崎 純 助教授には，本論文執筆の際だけでなく，研究室での研究会やその他の発表においても数々の貴重な御教示を頂きました．ここに深く御礼申し上げます．

天笠 俊之 助手には，本研究全般に関して，当初より懇切丁寧な御指導，御助言を賜りました．研究の方針決定から実装に至るまで，惜しみない御協力と御教示を頂きました．ここに深く御礼申し上げます．

波多野 賢治 助手には，研究会や学会に向けての発表練習の場を通じて本研究に関する多くの御助言と御指導を頂きました．ここに心より感謝の意を表します．

的野 晃整 氏には，本研究の方針の決定および提案手法の実装にあたり，多くの御協力，御助言を頂きました．また，発表での心構えや，その構成についても多くの御教示を頂きました．ここに深く御礼申し上げます．

最後に，データベース学講座の皆様には，本研究を行うにあたり，多くの御指導，御協力を頂きました．ここに重ねて深く御礼申し上げます．

参考文献

[1] Shurug Al-Khalifa, V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pp. 141–152, 2002.

[2] Encode and Decode Base64 Files. `http://www.fourmilab.ch/webtools/base64/`.（2005 年 3 月 URL 確認）.

[3] Jan-Marco Bremer and Michael Gertz. An efficient XML node identification and indexing scheme. `http://citeseer.lcs.mit.edu/bremer03efficient.html`. Techn. Report CSE2003–4, Dep. of Computer Science, Univ. of California, Davis, 2003（2005 年 3 月 URL 確認）.

[4] Jan-Marco Bremer and Michael Gertz. On Distributing XML Repositories. In *In 6th International Workshop on the Web and Databases (WebDB)*, pp. 73–78, 2003.

[5] bzip2. `http://sources.redhat.com/bzip2/`.（2005 年 3 月 URL 確認）.

[6] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pp. 436–445. Morgan Kaufmann, 1997.

[7] GNU zip (gzip). `http://www.gzip.org/`.（2005 年 3 月 URL 確認）.

[8] Hartmut Liefke and Dan Suciu. XMill: An Efficient Compressor for XML Data. In *Proceedings of the 2000 ACM SIGMOD International Conference*

on Management of Data, May 16-18, 2000, Dallas, Texas, USA, pp. 153–164. ACM, 2000.

[9] Amélie Marian and Jérôme Siméon. Projecting XML Documents. In *VLDB 2003: Proceedings of 29th International Conference on Very Large Data Bases, September 9–12, 2003, Berlin, Germany*, pp. 213–224, Los Altos, CA 94022, USA, 2003. Morgan Kaufmann Publishers.

[10] MSXML 4.0. `http://www.xml.com/pub/a/2002/06/05/msxml4.html`. （2005 年 3 月 URL 確認）.

[11] Online Computer Libraly Center. Introduction to the Dewey Decimal Classification. `http://www.oclc.org/dewey/versions/abridgededition14/intro.pdf`. （2005 年 3 月 URL 確認）.

[12] Regular Language description for XML (RELAX). `http://www.xml.gr.jp/relax/`. （2005 年 3 月 URL 確認）.

[13] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). `http://www.w3.org/TandS/QL/QL98/pp/xql.html`, September 1998. （2005 年 3 月 URL 確認）.

[14] Simple API for XML. `http://www.saxproject.org/`. （2005 年 3 月 URL 確認）.

[15] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A Query-friendly XML Compressor. In *Proceedings of ICDE 2002 San Jose, California, USA*, pp. 225–234, 2002.

[16] uuencode. `http://www.webopedia.com/TERM/U/Uuencode.html`. （2005 年 3 月 URL 確認）.

[17] W3Schools Online Web Tutorials. Document Type Definition. `http://www.w3schools.com/dtd/dtd_intro.asp`. （2005 年 3 月 URL 確認）.

[18] World Wide Web Consortium. Document Object Model (DOM). http://www.w3.org/DOM/.（2005 年 3 月 URL 確認）.

[19] World Wide Web Consortium. Hyper Text Markup Language (HTML) 4.01. http://www.w3.org/TR/REC-html40/, December 1999. W3C Recommendation 24 December 1999（2005 年 3 月 URL 確認）.

[20] World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, November 1999. W3C Recommendation 16 November 1999（2005 年 3 月 URL 確認）.

[21] World Wide Web Consortium. XML Pointer Language (XPointer) Version 1.0. http://www.w3.org/TR/WD-xptr, January 2001. W3C Last Call Working Draft 8 January 2001（2005 年 3 月 URL 確認）.

[22] World Wide Web Consortium. XML Schema. http://www.w3.org/XML/Schema, May 2001. W3C Recommendation 05 May 2001（2005 年 3 月 URL 確認）.

[23] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/TR/2004/REC-xml-20040204/, February 2004. W3C Recommendation 04 February 2004（2005 年 3 月 URL 確認）.

[24] World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, October 2004. W3C Working Draft 29 October 2004（2005 年 3 月 URL 確認）.

[25] Xalan-Java. http://xml.apache.org/xalan-j/.（2005 年 3 月 URL 確認）.

[26] An XML Benchmark Project (XMark). http://monetdb.cwi.nl/xml/index.html, 2003.（2005 年 3 月 URL 確認）.

[27] XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999. W3C Recommendation 16 November 1999（2005 年 3 月 URL 確認）.

[28] 天笠俊之, 植村俊亮. リージョンディレクトリを用いた関係データベースによる大規模 XML データ処理. 日本データベース学会 Letters, pp. 33–36, September 2004. Vol.3, No.2.

[29] 横山昌平, 太田学, 片山薫, 石川博. SAX-GTR: 高速 XML ストリーム読み込み手法. 研究報告 - データベースシステム, 第 2004-71 巻, pp. 213–220, OPTannote = , July 2004.

[30] 中尾伸章, 天笠俊之, 的野晃整, 植村俊亮. アクセス頻度を考慮した XML 文書分割方式の提案. 電子情報通信学会第 16 回データ工学ワークショップ (DEWS2005) 論文集, 5A-i5, February 2005.

[31] 中尾伸章, 天笠俊之, 植村俊亮. 部分圧縮を用いた大規模 XML データ処理方式の提案. 電子情報通信学会技術研究報告, 第 104-102 巻, pp. 1–6, june 2004. DE2004-1.

[32] 東原正智, 田島敬史. 問い合わせ処理に適した XML データ圧縮形式に関する研究. 電子情報通信学会第 15 回データ工学ワークショップ (DEWS2004) 論文集, 4-C-4, March 2004.

[33] 中島哲, 小田切淳一, 井谷宣子, 吉田茂. XML 高速処理技術 SPlitDOM の機能拡張と Web アプリケーションへの適用評価. 電子情報通信学会第 15 回データ工学ワークショップ (DEWS2004), I-5-5, March 2004.

[34] 日本規格協会（JIS）. 文書記述言語 : Standard Generalized Markup Language (SGML), 1992. JIS X 4151:1992.

[35] 富士通株式会社. XML 高速処理技術 SPlitDOM の機能拡張と Web アプリケーションへの適用評価. http://xml.fujitsu.com/jp/tech/split/index.html. （2005 年 3 月 URL 確認）.

[36] 富士通株式会社. XML CSV 圧縮. http://xml.fujitsu.com/jp/tech/csv/, November 2002. （2005 年 3 月 URL 確認）.

付録

A XMark データの DTD

```
<!ELEMENT site (regions, categories, catgraph, people, open_auctions, closed_auctions)>

<!ELEMENT categories (category+)>
<!ELEMENT category (name, description)>
<!ATTLIST category id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (text | parlist)>
<!ELEMENT text (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph (#PCDATA | bold | keyword | emph)*>
<!ELEMENT parlist (listitem)*>
<!ELEMENT listitem (text | parlist)*>

<!ELEMENT catgraph (edge*)>
<!ELEMENT edge EMPTY>
<!ATTLIST edge from IDREF #REQUIRED to IDREF #REQUIRED>

<!ELEMENT regions (africa, asia, australia, europe, namerica, samerica)>
<!ELEMENT africa (item*)>
<!ELEMENT asia (item*)>
<!ELEMENT australia (item*)>
<!ELEMENT namerica (item*)>
<!ELEMENT samerica (item*)>
<!ELEMENT europe (item*)>
<!ELEMENT item (location, quantity, name, payment, description, shipping, incategory+, mailbox)>
<!ATTLIST item id ID #REQUIRED
featured CDATA #IMPLIED>
<!ELEMENT location (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT payment (#PCDATA)>
<!ELEMENT shipping (#PCDATA)>
<!ELEMENT reserve (#PCDATA)>
<!ELEMENT incategory EMPTY>
<!ATTLIST incategory category IDREF #REQUIRED>
<!ELEMENT mailbox (mail*)>
<!ELEMENT mail (from, to, date, text)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT itemref EMPTY>
<!ATTLIST itemref item IDREF #REQUIRED>
<!ELEMENT personref EMPTY>
<!ATTLIST personref person IDREF #REQUIRED>

<!ELEMENT people (person*)>
<!ELEMENT person (name, emailaddress, phone?, address?, homepage?, creditcard?, profile?, watches?)>
<!ATTLIST person id ID #REQUIRED>
<!ELEMENT emailaddress (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT address (street, city, country, province?, zipcode)>
```

```
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT province (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT homepage (#PCDATA)>
<!ELEMENT creditcard (#PCDATA)>
<!ELEMENT profile (interest*, education?, gender?, business, age?)>
<!ATTLIST profile income CDATA #IMPLIED>
<!ELEMENT interest EMPTY>
<!ATTLIST interest category IDREF #REQUIRED>
<!ELEMENT education (#PCDATA)>
<!ELEMENT income (#PCDATA)>
<!ELEMENT gender (#PCDATA)>
<!ELEMENT business (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT watches (watch*)>
<!ELEMENT watch EMPTY>
<!ATTLIST watch open_auction IDREF #REQUIRED>

<!ELEMENT open_auctions (open_auction*)>
<!ELEMENT open_auction (initial, reserve?, bidder*, current, privacy?, itemref, seller, annotation,
quantity, type, interval)>
<!ATTLIST open_auction id ID #REQUIRED>
<!ELEMENT privacy (#PCDATA)>
<!ELEMENT initial (#PCDATA)>
<!ELEMENT bidder (date, time, personref, increase)>
<!ELEMENT seller EMPTY>
<!ATTLIST seller person IDREF #REQUIRED>
<!ELEMENT current (#PCDATA)>
<!ELEMENT increase (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT interval (start, end)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT end (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT amount (#PCDATA)>

<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction (seller, buyer, itemref, price, date, quantity, type, annotation?)>
<!ELEMENT buyer EMPTY>
<!ATTLIST buyer person IDREF #REQUIRED>
<!ELEMENT price (#PCDATA)>
<!ELEMENT annotation (author, description?, happiness)>

<!ELEMENT author EMPTY>
<!ATTLIST author person IDREF #REQUIRED>
<!ELEMENT happiness (#PCDATA)>
```