修 士 論 文

ネットワーク上でのXML問い合わせ集合の最適化

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

福井 佳紀

2004年3月

修 士 論 文

ネットワーク上でのXML問い合わせ集合の最適化

指導教官　田島敬史 助教授

審査委員主査　田島敬史 助教授
審査委員　大堀淳 教授
審査委員　二木厚吉 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

210078 福井 佳紀

提出年月: 2004 年 2 月

概 要

ネットワーク上の XML データベースに対して，クライアントが複数，あるいは，単一の XPath による問い合わせを行う場合，返送される解集合には冗長性が含まれている可能性がある．これらの解をサーバが別々にクライアントに送信する場合，ネットワークの通信コストに無駄が生じる．そこで我々は，通信コストを最適化するために，与えられた問い合わせ集合を，それらの問い合わせ全てに答えることができるサイズ最小のビューに変換する手法をこれまでに提案した．しかし，この方法では通信コストは低減されるものの，サーバやクライアントでの計算コストが増加してしまう場合もある．そこで，本論文では，通信コストと計算コストの双方を考慮した最適化手法について提案する．

目 次

第 1 章　はじめに **1**

第 2 章　XPath **4**
2.1 XPath の解集合 4
2.2 使用する XPath の部分言語の文法 4

第 3 章　通信コスト 最適化のための問い合わせ変換の例 **6**
3.1 非再帰的な問い合わせによる例 6
3.2 再帰を含む問い合わせの例 8

第 4 章　通信コスト の最適化に関する評価実験 **9**
4.1 実験環境 9
4.2 非再帰的な問い合わせによる評価実験 9
4.3 再帰的な問い合わせによる評価実験 16

第 5 章　サーバ側での計算コスト の改善 **18**
5.1 非再帰的な問い合わせにおける計算コストの最適化 19
5.2 再帰的な問い合わせにおける計算コストの最適化 20
5.2.1 XML データ全体の統計情報を利用した再帰の展開 20
5.2.2 XML データ中の各エレメントの統計情報を利用した再帰の展開 . . 22

第 6 章　サーバ側での処理によるクライアント 側での計算コスト の改善 **24**
6.1 解集合のデータ加工 24
6.2 インデックス情報 26

第 7 章　計算コスト の最適化に関する評価実験 **28**
7.1 サーバ側での計算コストの改善 28
7.1.1 実験環境 28
7.1.2 非再帰的な問い合わせによる評価実験 28
7.1.3 再帰的な問い合わせによる評価実験 31
7.2 クライアント側での計算コストの改善 36
7.2.1 実験環境 36

7.2.2　解集合のデータ加工とインデックス情報 36

第 8 章　まとめと今後の課題 **41**

8.1　まとめ 41

8.2　今後の課題 41

謝辞 **43**

第1章　はじめに

今日，XML フォーマットは頻繁に利用されるようになり，インターネット上でのデータ交換やデータ発信の標準ともいわれるようになった．そして，XML データはネットワーク上に散在するようになり，ネットワーク上の XML データを効率的に問い合わせるための実現方法が求められている．その結果，XML データを用いた情報サービスシステムに関するさまざまな研究が行われるようになった．

XML データを用いた情報サービスシステムの例として，連続問い合わせシステム (Continuous Query System)[4][5] や XML ストリーミングサービス [6][7][8][9] などが挙げられる．

連続問い合わせシステムとは，各クライアントが問い合わせ内容を記述したプロファイルをサーバに登録しておき，サーバが定期的に問い合わせを評価して，その結果を各クライアントに送信するという方式を取っている．一方，XML ストリーミングサービスとは，サーバが XML データをストリーム形式で配信し，クライアントが XML データの断片を受け取りながら，必要に応じた処理を逐次行っていく方式を取っている．

上述のような XML データの情報サービスシステムでは，なんらかの XML への問い合わせ言語を利用している．XML への問い合わせ言語にもさまざまなものがあり，日進月歩で研究が進められている．その中で，XPath1.0[10][11] と呼ばれる問い合わせ言語のみが唯一 1999 年に W3C の勧告となり，既に世界中で幅広く利用されるようになった．XPath は，もともとは他の標準規格，例えばスタイルシート型 XML 変換言語 XSLT[12] や，汎用型 XML 問い合わせ言語 XQuery[13] などの一コンポーネントとして設計されたものだが，現在では，XML 情報システムのための独立した問い合わせ言語としても広く用いられるようになった．

XPath は，XML データ中の特定のノード集合をパス式によって選択することができる非常にシンプルな問い合わせ言語である．XML データは通常，ラベル付き木で表現され，データ中のエレメントのうち，問い合わせのパス式にマッチするエレメントを根とする部分木の集合が返される．XPath は，あるエレメントを根とする部分木の集合を取り出す機能しかなく，解に子エレメントを追加したり，解から一部の子エレメントを取り除いたりする，といったデータの変形を行うことができないという特徴がある．

XML 情報サービスシステムは，大きく，二つのタイプに分類できる．オンライン XML データベースや連続問い合わせシステムのように，問い合わせをサーバ側で処理するタイプのものと，XML ストリーミングサービスのように，問い合わせをクライアント側で処理するタイプのものである．

前者のサーバ側で処理するタイプでは，問い合わせの解のみがクライアントに送られる

ので，後者のクライアント側で XML データを受信しながら処理するタイプのものと比べれば，通信コストの上では効率が良い．しかし，サーバ側で処理するタイプのものでも，通信コストは必ずしも真に最適化されているわけではない．

これは，ネットワーク上の XML データベースに対して，クライアントが複数の XPath による問い合わせを行う場合を考えると，返送される解集合には冗長性が含まれている可能性があるためである．第一に，ある解集合に含まれているあるエレメントと，別の解集合に含まれているあるエレメントが全く同一のものであり，重複している場合がある．第二に，ある解集合中のあるエレメントが，別の解集合中のあるエレメントの部分木となっている場合がある．この 2 つのケースが XPath の問い合わせの解集合に発生し得る冗長性である．さらにいえば，複数の XPath ではなく，単一の XPath 問い合わせを発行する場合にも冗長性が発生する場合がある．これは，その問い合わせの解に含まれるあるエレメントが，同じ解に含まれる別のエレメントの部分木になっている場合があるためである．サーバがこれらの冗長性をもった解をそのままクライアントに送信すると，ネットワーク上に同じデータが何度も流されることになり，通信コストの上では最適とはいえない．そこで，本研究では，ネットワーク上で XPath 問い合わせを実行する場合に生じる通信コストを最適化するための手法を提案する．本研究では，サーバに手を加えられない場合と，サーバに手を加えられる場合の二種類の場合を想定し，それぞれに対して研究を行っている．

まず，サーバに手を加えられないシステムを考える．ネットワーク上の XML データベースがサーバとしてクライアントからの XPath の問い合わせを待ち受けており，受け取った問い合わせを評価し，解集合をクライアントに返すというシステムを前提に考えている．そのため，クライアントが自由にサーバを変更することができないので，計算コストを最適化するためには，問い合わせ内容を変更するしかない．我々は，上述のような解の冗長性による通信コストの増大を防ぐために，XPath 問い合わせの集合を与えられた場合，それらの問い合わせ全てに答えることができるサイズ最小のビューを求め，これをサーバからクライアントに送信し，クライアント側でこのビューから，オリジナルの問い合わせによって得られるはずであった解集合を生成する方法をこれまでに文献 [1] で提案した．これまでに提案した手法では，通信コストの最適化に特化されており，サーバの計算コストは増大してしまう場合がある．これは，サイズ最小のビューに変換された問い合わせは，オリジナルのものと比べて複雑な計算を必要とするのが原因である．例えば，サイズ最小のビューでは，問い合わせ集合間のすべての共通部分を取り除くために，可能な限り解を分割して取り出せるような問い合わせに変換し，解中に共通部分が発生しないようにする．これによって，否定を求める演算や集合の共通部分を取る演算の数が増え，計算が複雑になる．また，自己冗長性を取り除く演算には，根からみて一番浅い場所にあるエレメントを取り出す必要があるため，さらに複雑な処理が必要となる．そこで，本論文では，サイズ最小のビューに変換された問い合わせ集合の中で頻繁に現れるパターンの部分について，サーバが保有する XML データ中の統計情報を利用することによって，より簡単な問い合わせに変換し，計算コストを軽減する手法を提案する．

一方，サーバ側に手を加えられる場合では，上述の手法に加えて，さらにクライアント側での計算コストも減らすことが可能である．一般的なデータベース問い合わせシステムでは，サーバが返信する解データをそのまま所望する解としてクライアントが利用することが可能である．しかし，上述のサイズ最小のビューに変換する手法では，クライアントが，受け取ったサイズ最小の解からオリジナルの問い合わせで得られるはずであった解集合を取り出す追加処理が必要となる．クライアントが携帯電話や PDA といった，組込み機器を利用しており，マシン性能がある程度制限されている環境にあれば，この解を取り出す追加の計算は好ましくない．そこで，これを解決するために二つの手法を提案する．まず，はじめに，クライアント側での計算コストを軽減するために，解集合を簡単な問い合わせで取り出せるように，サーバがデータを加工する手法を提案する．次に，XML 問い合わせにおいて，大きな負荷がかかるパーズ処理に着目し，この負荷を取り除くため，取り出すべき解の位置のバイトオフセットをサーバが送信することによって，クライアントがパーズ処理を行わずに解を取り出せる手法を提案する．実験の結果，後者の方がより計算コストが改善された．

以下，次の第 2 章では，本論文で取り扱う XPath について説明する．第 3 章では，XPath 問い合わせの集合を，文献 [1] で提案した手法を使って，サイズ最小のビューに変換する例をいくつか挙げ，その実験結果を第 4 章で示す．第 5 章，第 6 章では，文献 [1] で提案した手法で発生する計算コストが増大してしまう問題を取り上げ，サーバを操作できない場合と，操作できる場合の双方を考え，計算コストを改良する手法について提案し，その実験結果を第 7 章で示す．最後の第 8 章で，全体のまとめと今後の展開について述べる．

第2章　XPath

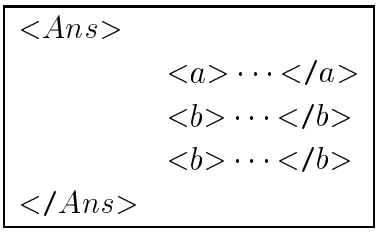
2.1　XPathの解集合

前章で述べたように，XPath は木パターン言語の一種である．XPath 問い合わせは，XML で記述されたデータベース木に対して評価され，パターンにマッチするエレメントを根とする部分木の集合を返す．本論文では，XPath 問い合わせの解集合は，Ans エレメントを根とし，解集合中の各エレメントをその子供とする XML 木の形で返されるものとする．これは，一部の処理系で実際に用いられている方法である．

例えば，問い合わせの解集合が次のようであったとする．

$$\{<a>\cdots, \cdots, \cdots\}$$

この場合，次のような XML 木が解として返される．



2.2　使用するXPathの部分言語の文法

本論文では，XPath の主要な機能のみを含む部分言語を用いる．この言語では，問い合わせ式 q は以下の文法で定義される．

$$\begin{aligned} q &::= /p \mid //p \mid q \cup q \mid q - q \\ p &::= a \mid \overline{\{a_1, \ldots, a_n\}} \mid * \mid p/p \mid p//p \mid p[p] \mid p\overline{[p]} \end{aligned}$$

q は，$/p$ または $//p$ という形か，二つの問い合わせ集合の和演算 $q \cup q$ か，二つの問い合わせ集合の差演算 $q - q$ のいずれかである．このうち，$/p$ または $//p$ の形をしたものを，一般に絶対ロケーションパスと呼ぶ．

絶対ロケーションパス $/p$ は，データとなる XML 木の根からスタートし，相対ロケーションパス p にマッチするパスを通って到達可能なエレメントにマッチする．一方，$//p$ は，p にマッチするパスが根からスタートしなくても良く，任意の深さからスタートできる．

また，$q_1 \cup q_2$ は集合の和演算であり，q_1 でマッチしたものと q_2 でマッチしたものの和を取ったものが返される．$q_1 - q_2$ は，q_1 にマッチするが q_2 にマッチしないエレメントの集合が返される．集合の和演算，集合の差演算は，絶対ロケーションパスの一番外側のレベルにのみ現れると仮定している．

相対ロケーションパス p は，木パターンを表現している．a は a をラベルとするエレメントにマッチするラベルテストである．同様に，$\overline{\{a_1, \ldots, a_n\}}$ は $a_1, \ldots, a_n$ を除くエレメントにマッチする否定のラベルテストである．また，$*$ は任意のラベルにマッチするワイルドカードである．

p_1/p_2 は，二つのロケーションパスの連結で，例えば，$/a/*$ はデータベース木の根に当たる a エレメントの任意の子供のエレメントにマッチする．$p_1//p_2$ も二つのパスの連結だが，この場合は，p_2 にマッチするパスが p_1 にマッチするパスのすぐ下に現れる必要はない．例えば，$/a//b$ は，a エレメントの子孫になっている任意の深さにある b エレメントにマッチする．// は，ある種の再帰を表現するもので，// を含む問い合わせを再帰的な問い合わせ，含まない問い合わせを非再帰的な問い合わせと呼ぶ．

$p_1[p_2]$ は，述語表現と呼ばれ，p_1 にマッチするパスを通って到達可能なエレメントの集合のうち，その下に少なくとも一つ，p_2 にマッチするパスを持つようなエレメントにマッチする．例えば，$//a[b/c]$ は，任意の深さにある a エレメントのうち，b エレメントを子供に持ち，さらに，その b エレメントが c エレメントを子供に持つようなものがマッチする．$p_1\overline{[p_2]}$ は否定の述語表現で，p_1 にマッチするパスを通って到達可能で，かつ，p_2 にマッチするようなパスを子供として持たないようなエレメントがマッチする．

なお，上の定義には，集合積演算 $q_1 \cap q_2$ は含まれていないが，これは，$q_1 - (q_2 - q_1)$ で求めることができる．また，q の補集合演算も $//* - q$ で求められる．

第3章　通信コスト最適化のための問い合わせ変換の例

次に，この章では，どのような場合に，XPath の問い合わせの解中に冗長性が生じるか，また，文献 [1] で提案した手法では，それらの冗長性を防ぐために，与えられた問い合わせ集合をどのようなビューに変換するかについて，ごく簡単に例を使って解説する．

3.1　非再帰的な問い合わせによる例

再帰を持たない XPath 問い合わせ集合で冗長性を生じる物の最も簡単な例は次のようなものである．

$$CASE_1 = \begin{cases} Q_1: & /a/* \\ Q_2: & /a/b \end{cases}$$

Q_2 の解集合は Q_1 の解集合の部分集合となっているのは明らかである．サーバがクライアントに，これら二つの解集合を別々にネットワークを介して送信するのは通信コストの上では最適ではない．この場合，簡単な解決方法として，Q_1 の解集合のみ送信するという方法が考えられる．Q_1 の解は，送られてきたものがそのまま Q_1 の解として利用できる．一方，Q_2 の解はクライアント側で Q_1 の解集合から b のエレメントのみを抜き出して生成することが可能である．本論文では，問い合わせの解集合は，解集合中の各エレメントを子とするような Ans エレメントを根とする XML 木の形で返されると仮定しているので，Q_2 の解の生成は，Q_1 の解集合に対して，$/Ans/b$ という問い合わせを実行すればよい．本論文では，今後，これを $Q_2 \leftarrow (Q_1, /Ans/b)$ のように表記することにする．

しかし，一方の問い合わせの解がもう一方の問い合わせの解の部分集合になっていても，取り出せない場合がある．例えば以下のような例を考える．

$$CASE_2 = \begin{cases} Q_1: & /a/*/c \\ Q_2: & /a/b/c \end{cases}$$

Q_1 の解は，c エレメントを根とする部分木の集合であり，かつ，Q_2 の解を部分集合として含んでいる．しかし，この場合，Q_1 の解集合のみから，Q_2 を取り出すことができない．Q_1 の解中に現れる c エレメントのうち，どれが Q_2 の解に含まれるべきものなのか，c エレメントを根とする部分木の集合からでは判定できなくなってしまう．これは，Q_1 の

解中から，もともとのデータベース中にあった文脈に関する情報（この場合，親エレメントのラベル情報）が失われているためである．このようなケースでは，通信コストのデータ量を最小にするために，次のような二つの問い合わせをサーバに送ればよい．

$$CASE_2 = \begin{cases} Q_{1-2}: & /a/\overline{\{b\}}/c \\ Q_2: & /a/b/c \end{cases}$$

Q_1 の解は，クライアント側で Q_{1-2} の解と，Q_2 の解の和集合を取ることで生成できる．

$$CASE_2 = \begin{cases} Q_1 \leftarrow (Q_{1-2}, /Ans/c) \\ Q_1 \leftarrow (Q_2, /Ans/c) \\ Q_2 \leftarrow (Q_2, /Ans/c) \end{cases}$$

Q_{1-2} と Q_2 という組み合わせは，解に重複が無く，かつ，最終的な解に含まれるエレメントしか含んでいないので，通信コスト上では最適と言える．

次に，述語を使った例を考える．述語が現れる場合も，上の例と同様に少し複雑になることがある．

$$CASE_3 = \begin{cases} Q_1 : /a/b/e \\ Q_2 : /a/b[c]/e \\ Q_3 : /a/b[d]/e \end{cases}$$

Q_2 と Q_3 は，共に Q_1 の部分集合である．しかし，単純に Q_1 の解集合のみから，Q_2 と Q_3 は取り出せなくなる．これは，上の例と同様に解から文脈に関する情報が失われてしまうためである．Q_1 の解は，根の子供として複数の e エレメントが現れる集合である．しかし，どの e エレメントが Q_2，もしくは，Q_3 に現れるべきものなのかを判定するには，e エレメントの親の b エレメントの子供の情報（すなわち，e エレメントの兄弟の情報）が必要になる．そこで，このような場合，次のような四つの問い合わせに変換する．

$$CASE_3 = \begin{cases} Q_{2\cap 3}: & /a/b[c][d]/e \\ Q_{2-3}: & /a/b[c]\overline{[d]}/e \\ Q_{3-2}: & /a/b\overline{[c]}[d]/e \\ Q_{1-2-3}: & /a/b\overline{[c][d]}/e \end{cases}$$

この場合，オリジナルの解を生成するにはクライアント側で次のような問い合わせを行う．

$$CASE_3 = \begin{cases} Q_1 \leftarrow (Q_{2\cap 3}, /Ans/e) \\ Q_1 \leftarrow (Q_{2-3}, /Ans/e) \\ Q_1 \leftarrow (Q_{3-2}, /Ans/e) \\ Q_1 \leftarrow (Q_{1-2-3}, /Ans/e) \\ Q_2 \leftarrow (Q_{2\cap 3}, /Ans/e) \\ Q_2 \leftarrow (Q_{2-3}, /Ans/e) \\ Q_3 \leftarrow (Q_{2\cap 3}, /Ans/e) \\ Q_3 \leftarrow (Q_{3-2}, /Ans/e) \end{cases}$$

3.2 再帰を含む問い合わせの例

前章の例では，非再帰的な問い合わせ，すなわち //が XPath に現れない問い合わせのみを扱った．ここでは，再帰を含む問い合わせの重複について考える．再帰を含む問い合わせの場合，ネットワークを介して送信されるデータの重複は，たった一つの問い合わせのみでも頻繁に発生する．例えば，クライアントが次のような問い合わせを送信する場合を考える．

$$CASE_4 = \left\{ \; Q: \;\; //a \right.$$

この問い合わせは，データベースの XML 木中の a エレメントを根とするすべての部分木を返す．もし，ある a エレメントが，別の a エレメントを子孫として持つ場合，後者の a エレメントは複数回送信されることになる．このように，再帰的な XPath 問い合わせに対する解集合は，祖先子孫関係からくる自己冗長性を含みうる．

このような場合，次のような問い合わせを送ることで，ネットワーク上のデータ量を最適化する．

$$CASE_4 = \left\{ \; Q^{\top}: \;\; //a - //a//a \right.$$

これは，根からスタートする各パス中で，最初に現れる a エレメントのみを取り出すという意味になる．オリジナルの解と同じ物を得るにはクライアント側で，次のように取り出すことができる．

$$CASE_4 = \left\{ \; Q \leftarrow (Q^{\top}, /Ans//a) \right.$$

//の後のステップ数が長くなるとさらに複雑になる．例えば，次のような例を考える．

$$CASE_5 = \left\{ \; Q : //a/b/a/b \right.$$

この場合，解中の自己冗長性を取り除くには，$CASE_4$ と同様に，次の問い合わせを送信すればよい．

$$CASE_5 = \left\{ \; Q^{\top} : //a/b/a/b - //a/b/a/b//b \right.$$

末尾の，$-//a/b/a/b//b$ は，自己冗長性を取り除くものである．$Q^{\top}$ の解からオリジナルの解を取り出すには，クライアント側で次の 3 つの問い合わせが必要となる．

$$CASE_5 = \left\{ \begin{array}{ll} Q \leftarrow (Q^{\top}, /Ans/b) & -(1) \\ Q \leftarrow (Q^{\top}, /Ans/b/a/b) & -(2) \\ Q \leftarrow (Q^{\top}, /Ans//a/b/a/b) & -(3) \end{array} \right.$$

$Q^{\top}$ 中の b エレメントは $//a/b/a/b$ にマッチしたエレメントであるので，解中に a/b というエレメントが現れたら，その孫エレメントの b はデータベース中で $//a/b/a/b$ というパスにマッチするノードである．よって，(2)が必要となる．この例で示したような，解となる子孫エレメントの取り出し方法は，古典的な部分文字列探索のアルゴリズムである Knuth-Morris-Pratt アルゴリズム [2] での，接頭辞関数の計算に似ている．

第4章　通信コストの最適化に関する評価実験

我々は，上述のような手法を用いた問い合わせ評価の実験を行った．ここでは，その一部を紹介する．

4.1　実験環境

実験データとして，XMark(XML Benchmark Project)[14] で生成した約 50MB と約 233MB の XML データを用いる．XMark は，人工的に規模変更可能なオークションデータを生成する．ハードウェアや OS 等の環境によらず，同一の XML データを生成することが可能であり，XML のスキーマは (DTD) も固定であるという特徴をもっている．本論文で用いる主要な DTD の一部を図 4.1 に示す．オークションデータは，商品の出品地域ごとに分類された商品の詳細情報，参加者の登録情報などが主なコンテンツ内容である．

また，実験データの格納方法には，次の二種類のものを用意した．一つめに，文献 [3] で利用されているような，XML データを関係の形にエンコードする最も一般的な手法（Relational Encoding）を用いて，関係データベースである Oracle 9i 64bit に格納したものを用意した．XML データは，各エレメントの名前，深さ，前順序，後順序，親エレメントへの参照といった情報でエンコードされ，データベースに格納される．ここで用いたスキーマを簡単に，図 4.2 に示す．そして，この関係データベースに対しては，XPath を SQL に変換し，SQL で問い合わせを行う．次に，XML データを加工しないでそのままファイルに保存したものを用意した．この XML ファイルに対しては，Xalan[16] の DOM を用いて，XML データをメモリ上に読み込み，それに対して直接 XPath で問い合わせを行う．

そして，2CPU（900MHz UltraSPARC-CU），6GB のメモリを搭載した Sun Blade 2000 上で，この 2 つのデータベースシステムの実験を行った．

4.2　非再帰的な問い合わせによる評価実験

まず始めに，非再帰的な問い合わせによる実験を考える．クライアントは，次のような XPath の問い合わせ集合による問い合わせを行う．Q_1，Q_2 では，北アメリカとヨーロッ

```
<!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 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?)>
```

図 4.1: オークションデータの DTD の一部

文書 DOCUMENT

文書 ID docID	ファイル名 name
1	auction50M.xml
2	auction233M.xml

エレメントデータ DATA

文書 ID docID	名前 name	前順序（ID） pre	後順序 post	最大の子孫 ID rightMostChild	親 ID parentID
1	site	1	726516	726515	0
1	regions	2	245978	245977	1
1	africa	3	6419	6418	2

深さ depth	テキスト ID textID	開始位置 beginOffset	終了位置 endOffset	データサイズ dataSize
1	0	0	52053225	8
2	1	7	25436627	21
3	2	257	669225	145

テキストデータ TEXT

文書 ID docID	テキスト ID textID	データサイズ dataSize	内容 value
1	1	344	page rous lady idle authority capt ...
1	2	54	shepherd noble supposed dotage humble ...

図 4.2: Relational Encodingで用いたデータベースのスキーマ情報

パで出品されているオークション商品の「全情報（名前，詳細，連絡先など）」を問い合わせている．Q_3，Q_4 では，全地域で出品されているオークション商品の「名前」と「詳細」を問い合わせている．

$$EX_1 = \begin{cases} Q_1: & /site/regions/namerica/item \\ Q_2: & /site/regions/europe/item \\ Q_3: & /site/regions/*/item/name \\ Q_4: & /site/regions/*/item/description \end{cases}$$

上記の問い合わせ集合に，文献 [1] で提案したサイズ最小のビューに変換するアルゴリズムを適用すると次のようになる．Q_3 と Q_4 で問い合わせている「名前」と「詳細」は，Q_1 と Q_2 にも含まれており，重複しているのでそれを考慮している．

$$EX_1 = \begin{cases} Q_1: & /site/regions/namerica/item \\ Q_2: & /site/regions/europe/item \\ Q_{3-1-2}: & /site/regions/\overline{\{namerica, europe\}}/item/name \\ Q_{4-1-2}: & /site/regions/\overline{\{namerica, europe\}}/item/description \end{cases}$$

DOM を用いた実験では，ここで示した XPath 式をそのまま利用できるが，Relational Encoding を用いたものでは，SQL に変換する必要がある．EX_1 の問い合わせ集合を SQL に変換すると次のようになる．

Q_1：

```
SELECT
    d4.pre
FROM
    DATA d1, DATA d2, DATA d3, DATA d4
WHERE
    d1.name = 'site' AND d2.name = 'regions' AND d3.name = 'namerica' AND
    d4.name = 'item' AND d1.parentID = 0 AND d2.parentID = d1.pre AND
    d3.parentID = d2.pre AND d4.parentID = d3.pre AND
    d1.rightMostChild >= d2.pre AND d2.rightMostChild >= d3.pre AND
    d3.rightMostChild >= d4.pre
```

Q_2：

```
SELECT
    d4.pre
FROM
    DATA d1, DATA d2, DATA d3, DATA d4
WHERE
    d1.name = 'site' AND d2.name = 'regions' AND d3.name = 'europe' AND
    d4.name = 'item' AND d1.parentID = 0 AND d2.parentID = d1.pre AND
    d3.parentID = d2.pre AND d4.parentID = d3.pre AND
    d1.rightMostChild >= d2.pre AND d2.rightMostChild >= d3.pre AND
    d3.rightMostChild >= d4.pre
```

$\mathbf{Q_{3-1-2}}$：

```
SELECT
    d4.pre
FROM
    DATA d1, DATA d2, DATA d3, DATA d4
WHERE
    d1.name = 'site' AND d2.name = 'regions' AND
    ( d3.name != 'namerica' AND d3.name != 'europe' ) AND
    d4.name = 'item' AND d1.parentID = 0 AND d2.parentID = d1.pre AND
    d3.parentID = d2.pre AND d4.parentID = d3.pre AND
    d1.rightMostChild >= d2.pre AND d2.rightMostChild >= d3.pre AND
    d3.rightMostChild >= d4.pre
```

$\mathbf{Q_{4-1-2}}$：

```
SELECT
    d4.pre
FROM
    DATA d1, DATA d2, DATA d3, DATA d4
WHERE
    d1.name = 'site' AND d2.name = 'regions' AND
    ( d3.name != 'namerica' AND d3.name != 'europe' ) AND
    d4.name = 'description' AND d1.parentID = 0 AND d2.parentID = d1.pre AND
    d3.parentID = d2.pre AND d4.parentID = d3.pre AND
    d1.rightMostChild >= d2.pre AND d2.rightMostChild >= d3.pre AND
    d3.rightMostChild >= d4.pre
```

そして，上述の問い合わせを用いて，50MB の XML データに対して問い合わせたものが表 4.1 で，233MB の XML データに対して問い合わせたものが表 4.2 である．

50MB の表 4.1 を見ると，オリジナルの問い合わせでは 32.0MB ほどであった解集合のサイズが通信コストの最小化を行うと 21.9MB ほどになり，約 10.1MB も改善されており，31%近くものデータ量が削減されている．また，233MB の表 4.2 を見ると，通信コストが 216.2MB ほどであった解集合のサイズが 149.6MB ほどになり，約 66.6MB ものサイズの最適化が行われている．これは先ほどの例と同様に，31%近くデータ量が改善されていることになる．さらに，DOM と Relational Encoding のどちらをとっても，計算時間も改善されており，非常に効率的である．

Xalan の DOM を利用した問い合わせシステムによる問い合わせ処理では，得られる解集合のサイズに比例する計算時間がかかる処理が含まれている．Xalan の DOM はライブラリとして提供されており，問い合わせ処理の後，自動的に解となる部分木の集合をメモリ上に展開する．そのため，解集合の生成時間を除いた計測が不可能であり，解集合のサイズに計算時間が影響されてしまう．通信コストを最適化するために，問い合わせをある程度複雑なものに変換しても，解のサイズが小さければ計算時間は早くなる可能性がある．上記の変換では，Q_3, Q_4 の * というロケーションステップは $\overline{\{namerica, europe\}}$ に

変換された分，計算コストに悪い影響を及ぼしているが，解のサイズが小さくなったことで計算コストが低減したため，高速化されたと考えられる．

Relational Encodingを利用した問い合わせシステムでは，最終的な解集合のサイズに比例しないように計算コストを計測することが可能である．SQLでは，Q_3, Q_4の$*$は，表の直積演算を取るにあたって絞込みが行えないため計算コストが大きくなり，$\overline{\{namerica, europe\}}$に変換された分，タプルが制限され，表の積演算の計算コストが小さくなるため，高速化されていると考えられる．

一般に，変換後の問い合わせが複雑になるほど計算コストは増大する可能性があるのだが，日常的に使われる非再帰的な問い合わせの範囲では解集合のサイズのみではなく，計算コストも改善される可能性があるので，非常に効率的であるといえる．

50MB	DOM Time(sec)	RE Time(sec)	Size(KB)
Q_1	9.26	1.86	11458.89
Q_2	9.26	1.25	6914.51
Q_3	9.97	4.02	291.66
Q_4	9.96	4.56	13431.31
変換前 計	38.45	11.69	32096.37
Q_1	9.26	1.86	11458.89
Q_2	9.26	1.25	6914.51
Q_{3-1-2}	9.23	2.30	77.05
Q_{4-1-2}	9.27	2.47	3515.89
変換後 計	37.02	7.88	21966.34

RE : Relational Encoding

表 4.1: 50MB の XML データに対する非再帰的な問い合わせの実験結果

233MB	DOM Time(sec)	RE Time(sec)	Size(KB)
Q_1	166.09	8.94	78627.63
Q_2	147.74	6.69	47176.31
Q_3	180.15	20.97	1558.07
Q_4	150.69	32.17	88844.70
変換前 計	644.67	68.77	216206.71
Q_1	166.09	8.94	78627.63
Q_2	147.74	6.69	47176.31
Q_{3-1-2}	147.66	15.50	411.99
Q_{4-1-2}	148.32	19.31	23396.46
変換後 計	609.81	50.44	149612.39

RE : Relational Encoding

表 4.2: 233MB の XML データに対する非再帰的な問い合わせの実験結果

4.3 再帰的な問い合わせによる評価実験

次に再帰的な問い合わせによる実験を考える．クライアントは，次のような XPath の問い合わせを行う．Q は，任意の深さに現れる $parlist$ エレメントを根とする部分木を問い合わせている．$parlist$ とは，オークションの商品説明が記述されている文章リストの親のエレメントである（詳細は XMark[14] の DTD を参照のこと）．オークションデータの DTD 上では，$parlist$ は再帰を許されており，ある $parlist$ の子孫として，再び $parlist$ が現れることがあり，自己冗長性を含んでいる．

$$EX_2 = \left\{ \; Q : \;\; //parlist \right.$$

我々の通信コスト最適化のアルゴリズムに Q を適用すると，自己冗長性を取り除くために次のような問い合わせに変換される．

$$EX_2 = \left\{ \; Q^{\top} : \;\; //parlist - //parlist//parlist \right.$$

これを SQL に変換すると，次のようになる．SQL の**NOT EXISTS** 節は，自己冗長性を取り除いている．

$\mathbf{Q}^{\top}$:

```
SELECT
    d1.pre
FROM
    DATA d1
WHERE
    d1.name = 'parlist' AND
    NOT EXISTS (
        SELECT
            *
        FROM
            DATA p1, DATA p2
        WHERE
            p1.name = 'parlist' AND p2.name = 'parlist' AND
            p1.depth < p2.depth AND p1.pre < p2.pre AND
            p1.rightchild >= p2.pre AND p2.pre = d1.pre )
```

そして，上述の問い合わせを用いて実験を行った結果，表 4.3，表 4.4 のようになった．

50MB の XML データの実験結果の表 4.3 を見ると，オリジナルの問い合わせでは 24.8MB ほどであった解集合のサイズが通信コストの最小化を行うと 17.4MB ほどになり，約 7.3MB も改善されており，約 29%のデータ量が削減されたことになる．また，233MB の XML データの実験結果の表 4.4 を見ると，通信コストが 159.0MB ほどであった解集合のサイズ

50MB	DOM Time(sec)	RE Time(sec)	Size(KB)
Q	10.27	0.78	24801.50
$Q^{\top}$	10.52	28.25	17487.75

RE : Relational Encoding

表 4.3: 50MB の XML データに対する再帰的な問い合わせの実験結果

233MB	DOM Time(sec)	RE Time(sec)	Size(KB)
Q	193.87	1.67	158995.99
$Q^{\top}$	175.22	over 2(hour)	114840.66

RE : Relational Encoding

表 4.4: 233MB の XML データに対する再帰的な問い合わせの実験結果

が 114.8MB ほどになり，約 44.2MB ものサイズの最適化が行われている．これは約 28%のデータ量の改善に相当する．XPath では，たった一つの再帰的な問い合わせのみでも，自己冗長性を含む可能性があるため，それを除去するだけでも大きなネットワーク流量の削減へと繋がることが分かった．

それとは逆に，計算時間が増大してしまう問題がある．これは，自己冗長性の除去を行うことにより，複雑な処理が必要となるためである．DOM を使った実験では，それほど大きな実行速度の差は出ていないように思える．ところが前述の通り，DOM の計算時間の一部は解集合のサイズにある程度比例している．50MB の場合，データ量が削減されたのにも関わらず，計算コストが増大しているため，計算に負荷がかかっていることが考えられる．

Relational Encoding に関して実行時間を比べると，容量が大きくなるほど莫大に計算コストが増大していることが分かる．233MB の場合，2 時間かかっても結果を得ることができなかった．確かに，通信コストの面では大きく改善されているが，Relational Encoding において計算コストが非常に増大してしまうため，そのまま実行するのは実用的とはいえない．そのため，以後の章でこの問題について取り上げ，その解決方法を具体的に提案する．

第5章　サーバ側での計算コストの改善

これまでに，実際に冗長性がある XPath 問い合わせ集合の代表的な例をいくつか挙げ，その変換方法を紹介した．そして，実験結果からネットワークの通信コストが大幅に最適化されていることが分かった．しかし，文献 [1] で示した変換アルゴリズムは，通信コストを最適化するという目的で開発されており，そのため，前章で示した通り，サーバの計算コストに関しいえば，逆に大きく増大してしまう場合がある．

もしも，サーバが，保有しているデータベースの統計情報をある程度公開していれば，クライアントは，そのデータの内容や状況によって，計算コストを最適化する問い合わせに変換できる．サーバ側が独自で最適化の処理を行うことが可能であるならば，自身が保有しているデータ内容を逐次，把握することができるため，それに従って，計算コストの最適な問い合わせに変換できる．

本論文では，通信コストを最適化し，さらに，計算コストもある程度最適化する手法を提案する．そして，提案した手法の有用性を検証するための評価実験を，第 7 章で示す．

ここで問題なのは，XML データへの問い合わせの計算コストは，問い合わせの処理系によって大きく異なる．たとえば，前述の DOM を使っている Xalan の場合，解集合のサイズに比例した処理の部分が総計算時間の支配的要因になっている場合があるため，その場合は，ほぼ解のサイズに比例した計算コストがかかる．一般的な DOM の処理系では，単純なパス式ほど計算コストが改善されると考えられる．DOM は，XML データをメモリ中で木構造に展開した後，処理を行うため，計算機のメモリ量が十分である限り，他の処理系に比べて致命的にコストが悪くなる問い合わせパターンは特にないといえる．しかし，データが実メモリより大きく，仮想メモリが利用される場合は，木構造中の離れた場所を行ったり来たりするような処理が発生するため，ディスクアクセスが増え，効率が悪くなる．また，SAX を利用した処理系では，ストリーム処理を行うため，XPath に述語が現れないような一度のスキャンで計算できる問い合わせのものに適している．また，Relational Encoding でデータを格納したものは，// を使ったような再帰的な問い合わせが得意である．このように，処理系によって大きく計算コストが異なるため，XML での問い合わせ計算コストの最適化というのは，一口に判定するのは難しい．そこで本論文は，Relational Encoding に重点をおきつつ，処理系全体を考慮に入れ，総合的に判断した上で，計算コストの良悪を判断することとする．

5.1 非再帰的な問い合わせにおける計算コストの最適化

まず，非再帰的な問い合わせのみを含む場合について考える．非再帰的なものでは，$CASE_2$ や $CASE_3$ のように，単純にすべてを包含しているもので問い合わせを行うと，解を受け取ったクライアントがオリジナルの解を生成できなくなってしまう問題がある．これは解を生成するのに必要なデータベース中の文脈に関する情報が失われてしまうのが原因である．前述のように XPath は，単にパス式にマッチするエレメントを根とする部分木の集合を返す機能しか提供されていないため，このような文脈に関する情報を解に残しておくことができないのが原因である．

我々は，このような場合，問い合わせ集合間のすべての共通部分を取り除くために，可能な限り解を分割して取り出せるような問い合わせに変換し，解中に共通部分が発生しないようにしている．これはちょうど，XPath の問い合わせ集合によって得られる解集合の分布を一つのベン図で表し，そこに生成されるすべての領域を別々に取り出せるような問い合わせに変換するのに似ている．

この重複部分を別々に取り出せるように問い合わせを変換するアルゴリズムでは，集合同士の差演算や積演算が頻繁に発生する．例えば，$CASE_2$ の場合，問い合わせ Q_1 を，Q_1 の解集合から Q_2 の解集合の差を取る Q_{1-2} : $/a/\overline{\{b\}}/c$ に変換している．しかし，サーバでは，集合同士の演算が実際には行われず，集合同士の演算を行ったときと同じものが得られるような解を探索している．この場合，サーバでの評価は，a エレメントの子供は，b というラベルを除いた任意のラベルのエレメントであり，さらに，その子供が c エレメントであるものにマッチするものを取り出している．決して，Q_1 と Q_2 を別々に評価し，Q_1 の解集合から Q_2 の解集合の差演算を行って Q_{1-2} を求めてはいない．

処理系によっては否定のエレメントの探索は，肯定のエレメントの探索よりもコストがかかると考えられる．この対策として，状況に応じてサーバが，否定のエレメントの探索を行わず，二つの問い合わせを別々に評価し，実際に集合同士の差演算を行うように工夫できる．先ほどの例では，Q_1 と Q_2 を別々に評価し，Q_1 の解集合から Q_2 の解集合の差演算を行うように $Q_{1-2} = Q_1 - Q_2$ と求めれば良い．

$Q_1 - Q_2$ といった集合の差演算を行う場合，Q_1 と Q_2 の解集合のサイズが共に小さい場合に，否定のエレメントの探索を行わず，両者を別々に評価した後，集合の差演算を行う方法に切り替えればよい．逆に，Q_1 と Q_2 の解集合のサイズが共に大きい場合，別々に評価していたらそれだけで計算コストがかかってしまう恐れがあるので，否定のエレメントの探索を行ったほうが効率が良いと考えられる．

サーバは，解集合のサイズの大小を判定するために表 5.1 のような統計情報を保持するとより効率が良いと考えられる．サーバはこのような階層的なサイズの統計情報を利用して，解の取り出し方を切り替えることができる．

パス	サイズ (KB)
/site	233,000
/site/regions	210,123
/site/regions/namerica	121,735
/site/regions/samerica	78,150
⋮	⋮

表 5.1: サーバが保持する階層的なサイズの統計情報

5.2 再帰的な問い合わせにおける計算コストの最適化

次に，再帰的な問い合わせを考えると，再帰的な問い合わせは，すべて自己冗長性を取り除く必要がある．例えば，$CASE_4$ では，$//a$ という問い合わせを $//a - //a//a$ に変換している．自己冗長性を取り除くには，根からみて一番浅い場所にある a エレメントを取り出す必要があるため，このような複雑な変形になっている．$CASE_5$ では，$//a/b/a/b - //a/b/a/b//b$ といった問い合わせが生成されており，オリジナルの $//a/b/a/b$ と比べて，さらに複雑なものとなり，計算コストが悪化する恐れがある．一般的に XPath での // の計算コストは大きく，何度も現れるのは好ましくない．$//a$ の場合，当初 1 個であった // の数が，$//a - //a//a$ になると，3 個となっている．さらに，$//a$ と $//a//a$ を計算した後に，その差を計算する処理も必要であり，処理時間は大きく増大する．

一般的な XML データを木構造で考えると，ちょうど B-tree のようにそれほど深くはならず，横に大きく広がる傾向がある．また，あるエレメントの子孫に再び同じエレメントが何度も現れるという再帰的な状況は，スキーマが許可していても，実際のデータ中にはそれほど頻繁に現れる傾向はなく，稀である．つまり，上記のような，根から見て一番浅い場所にある $//a/b/a/b$ エレメントを見つけるために，$//a/b/a/b - //a/b/a/b//b$ の問い合わせを行うにはあまりにも効率が悪いといえる．

そこで我々は，再帰的な問い合わせにおいて，自己冗長性を除去するコストが増大してしまうのを改善するために，データベースに格納されている XML データの統計情報を利用した最適化を提案する．利用する統計情報は，XML データ全体の統計情報を用いた方法と，Relational Encoding を利用し，XML データ中の各エレメントの統計情報を利用した 2 種類の方法を説明する．

5.2.1 XML データ全体の統計情報を利用した再帰の展開

XML データ全体の統計情報を利用する方法は，いたって単純であり，サーバは自身が保有する XML データの最大の深さを知っていれば良い．例えば，自身が保有する XML データの最大の深さが 4 であることをサーバが把握していれば，再帰を持つ問い合わせ $//a$ の通信コストを最適化した問い合わせ $//a - //a//a$ は，非再帰の問い合わせ集合の

和演算に変換することが可能である．

$$\begin{cases} /a & -(1) \\ /\overline{\{a\}}/a & -(2) \\ /\overline{\{a\}}/\overline{\{a\}}/a & -(3) \\ /\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}/a & -(4) \end{cases}$$

これは，任意の深さに現れるすべての a を取り出す問い合わせ $//a$ の再帰を展開して，4 つの再帰を持たない問い合わせ集合の和集合に変換している．(1) の $/a$ は，ルート直下に現れる，深さ 1 の a エレメントを根とする部分木を取り出している．(2) の $/\overline{\{a\}}/a$ は，ルート直下の a を除くエレメントを選択し，その子供である，深さ 2 の a エレメントを根とする部分木を取り出している．$/a/a$ の解は，(1) の $/a$ の部分木として含まれているため，(2) の $/\overline{\{a\}}$ は，自己冗長性を取り除くために必要となる．(3)，(4) も同様である．

この場合，サーバが XML データの最大の深さを知っているという前提であるが，たとえ，深さが分からなくても効率化が期待できる．一般的に XML データの深さは，それほど深くならないという傾向があるため，これを利用すれば次のような問い合わせでも効率化が期待できる．

$$\begin{cases} /a & -(1) \\ /\overline{\{a\}}/a & -(2) \\ /\overline{\{a\}}/\overline{\{a\}}/a & -(3) \\ /\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}/a & -(4) \\ /\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}//a - /\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}//a//a & -(5) \end{cases}$$

これは，深さ 4 までは一つ上のものと同様に，非再帰的な問い合わせの和集合に展開し，深さ 5 以降は従来通りの方法で取り出している．この手法では，もしも，データベース中に a というエレメントが一つも現れていなかった場合，逆にコストがかかってしまうという問題がある．しかし，通常，サーバ側でこのような変換を行うのであれば，自身が保有する XML データの最大の深さも把握することが可能であるため，一つ上で説明した適切な数の非再帰的な問い合わせを生成できると考えられる．

次に，もう少し複雑な再帰的な問い合わせ $//a/b/a$ の自己冗長性を取り除く変換を考える．通信コストを最適化する単純な方法では，$//a/b/a - //a/b/a//a$ となるが，新しい方法を利用すると次のような問い合わせに変換できる．

$$\begin{cases} /a/b/a & -(1) \\ /*/a/b/a & -(2) \\ /*/*/a/b/a - /a/b/a/b/a & -(3) \\ /*/*/*/a/b/a - /a/b/a/a/b/a - /*/a/b/a/b/a & -(4) \\ \qquad\qquad\qquad\qquad\vdots & \end{cases}$$

(1) と (2) は，それぞれ，深さ 1 と深さ 2 から探索を開始して，$/a/b/a$ の解を取り出している．(1) と (2) の解集合間には重複がない．しかし，(3) の $/*/*/a/b/a$ にマッチする

$/a/b/a/b/a$ は，(1) の問い合わせ $/a/b/a$ の部分木として現れるため，集合の差演算を行う必要がある．同様に，深さ 4 から探索を開始し $/a/b/a$ を取り出す，(4) の $/*/*/*/a/b/a$ は，(1) と (2) の問い合わせの部分木となり得るため，集合の差演算を行う必要がある．このように，深い階層に現れる $/a/b/a$ を取り出す問い合わせほど複雑なものになるが，オリジナルの // を含む問い合わせよりは，多少複雑でも / のみしか含まない演算の方が計算コストは低いため，有効な方法である．

5.2.2 XML データ中の各エレメントの統計情報を利用した再帰の展開

前項では，データベース中の XML データの最大の深さといった統計情報を利用して，再帰を含む問い合わせを再帰を持たない問い合わせに展開する効率化手法を紹介した．この方法は，XML データの最大の深さが大きくない場合ほど，取り出し方法を簡単に表現できるため，より有効となる．ここでは，この手法をさらに拡張し，XML データ中の各エレメントの統計情報を利用することによって，より効率化をはかる方法を説明する．

一般に XML データを Relational Encoding によってデータベースに格納する場合，第 4 章で説明したように，各エレメントの情報を詳細に保存することが可能である．例えば，各エレメントの名前，前順序，後順序，親エレメントへの参照，深さなどといった情報を計算し，統計情報として保存しておくのが一般的である．実際の XML データベースで用いられている様々な統計情報の例を挙げたが，ここで実際に利用するのは，前項と同様にエレメントの深さのみである．この深さの統計情報を上手く利用することによって，より効率的に，再帰を持つ問い合わせを非再帰的なものに変換できる．

まず，前項と同様に，再帰を持つ問い合わせ $//a$ を考える．前項では，XML データの深さが 4 として仮定されていたが，より大きな XML データを考え，最大の深さが 12 であるとする．これを前項の方法で展開すると，12 個の非再帰的な問い合わせ集合の和演算に展開できる．実験の結果，// を含むオリジナルの問い合わせ $//a - //a//a$ に比べれば，// を含まない 12 個の問い合わせに変換し，集合の和を取る前項の手法はかなり有効であった．しかし，前項の方法は計算コストの効率化が見込めるものの，最大の深さが大きくなればなるほど，問い合わせの数が増えコストは悪化していくことは避けられない．さらにいえば，もしも，a エレメントが深さ 4, 5, 6, 8 にのみ現れるとすれば，12 個非再帰的な問い合わせのうち，8 個の問い合わせの解は空集合となり，無駄な問い合わせを行っていることになる．そこで，サーバは a エレメントの現れる深さを統計情報から深さ 4, 5, 6, 8 であることを把握し，次のように新しい問い合わせに変換する効率化手法を考える．

$$\begin{cases} /*/*/*/a & -(1) \\ /*/*/*/\overline{\{a\}}/a & -(2) \\ /*/*/*/\overline{\{a\}}/\overline{\{a\}}/a & -(3) \\ /*/*/*/\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}/*/a & -(4) \end{cases}$$

この問い合わせの (1)，(2)，(3)，(4) は，それぞれ，深さ 4, 5, 6, 8 に対応している．(1) は，深さ 4 にある a エレメントを根とする部分木を選択している．すべてのエレメントに

マッチするワイルドカード $*$ を利用して，$/*/*/*/a$ とすれば，深さ 4 に現れる a を選択することが可能である．(2) 以降も同様である．

次に，複雑な問い合わせの例，$//a/b/a$ を考える．こちらも，上の例と同じ仮定で，a エレメントの現れる深さが 4, 5, 6, 8 と分かっているとすると，次のような問い合わせに変換することが可能である．

$$\begin{cases} /*/*/*/a/b/a & -(1) \\ /*/*/*/*/a/b/a & -(2) \\ /*/*/*/*/*/a/b/a - /*/*/*/a/b/a/b/a & -(3) \\ /*/*/*/*/*/*/*/a/b/a - /*/*/*/a/b/a/*/a/b/a & -(4) \\ \quad -/*/*/*/*/a/b/a/a/b/a - /*/*/*/*/*/a/b/a/b/a & \end{cases}$$

前項では，XML データの深さ 1 から最大の深さまでのすべての階層において，// 以下の XPath 式とマッチする部分木をすべて取得できるように展開している．これを，指定されたエレメントが現れる深さが分かっているならば，その深さから探索を開始するように変更し，そのエレメントが決して現れないような階層からは探索しないように変更する．この方法を用いれば，前項のように XML データの最大の深さの数だけ展開しなくても良く，指定されたエレメントが現れる深さの数だけ展開すればよいことになる．現れる深さを把握することによって，展開される問い合わせ集合の数が減れば，その分だけ計算コストの効率化が見込める．さらに，// が現れない非再帰的な問い合わせのみになるため，より効率的である．

第6章　サーバ側での処理によるクライアント側での計算コストの改善

前述のように，文献 [1] で提案した手法では，クライアント側の計算コストも増大する．そこで，この章では，クライアント側の計算コストを軽減する手法について考える．

ここでは，サーバが解集合を加工し，データベースの文脈に関する情報を付与する手法と，サイズ最小の解集合からオリジナルの解集合の取り出す方法を記述したインデックス情報をサーバが解データと共に送信する手法の二つを考え，クライアント側での計算コストを最適化する．

6.1　解集合のデータ加工

サイズ最小のビューに変換する方法では，サーバから受信したサイズ最小の解集合から，オリジナルの問い合わせで得られるはずであった解集合をクライアント側で取り出す必要がある．基本的にサーバから送信されるサイズ最小の解集合から，オリジナルの解集合を取り出す計算コストはそれほど高くない．受け取った解集合の和集合を計算したり，部分集合を抜き出す程度の計算ですむことが分かっている．しかし，$CASE_5$ のように，オリジナルの問い合わせの数は一つであったのに，クライアント側で解を取り出す場合，問い合わせの数が三つになる場合がある．これは，送信される解中からデータベースに含まれる文脈に関する情報が失われてしまったのが原因である．

文献 [1] では，サーバ側ではまったく通信コストや計算コストの最適化を行わないという前提であったが，本論文では，サーバ側で通信コスト最適化の変換を行う場合を想定することにした．そのため，すべての解集合を単純に $<Ans>\cdots</Ans>$ というタグで囲んで，サーバがクライアントに返送する必要は無く，ある程度送信する解集合を加工することも可能である．そこで，取り出すのに必要なデータベースの文脈に関する情報をサーバが送信するデータに付与することにより，クライアントがオリジナルの解を取り出すのに必要な問い合わせの数を最小にすることが可能となる．

例として，再び $CASE_5$ の場合を考える．通信コストを最適化した問い合わせの解をサーバがクライアントに送信する場合，次のような形式のデータを送信する．

$<Ans>$ （ b タグに囲まれた Q の解集合 ） $</Ans>$

この解集合から，オリジナルの問い合わせによって得られる解を取り出すには，クライアントが次の問い合わせを行う．

$$CASE_5 = \begin{cases} Q \leftarrow (Q^{\top}, /Ans/b) & -(1) \\ Q \leftarrow (Q^{\top}, /Ans/b/a/b) & -(2) \\ Q \leftarrow (Q^{\top}, /Ans//a/b/a/b) & -(3) \end{cases}$$

オリジナルの問い合わせが一つであったのに対して，クライアントが適切な解を取り出すために必要な問い合わせは三つに増えている．これは，解データ中には，階層構造 $/a/b/a$ の文脈に関する情報が残っていないためである．

そこで，文脈に関する情報を解に付与するために，階層構造 $/a/b/a$ を表す $<a><a>\cdots$ というタグで解集合を囲んだ，次のような解データ W を送信すると，文脈に関する情報を解データに残しておくことが可能である．

```
<a>
        <b>
                <a>
                        ( b タグに囲まれた Q の解集合 )
                </a>
        </b>
</a>
```

クライアントがこの解を受け取った場合，次の問い合わせ $Q \leftarrow (W, Q)$ によって，オリジナルの問い合わせで得られる解と同じ解集合を得ることが可能である．

$$CASE_5 = \left\{ \; Q \leftarrow (W, //a/b/a/b) \right.$$

この方法を利用すれば，オリジナルの問い合わせ一つに対して，それとまったく同じ問い合わせ一つで目的の解を取り出すことが可能になる．

次に，$CASE_2$ の場合を考える．サーバがクライアントに送信する通信コストを最適化した問い合わせの解データは次のようになる．

```
<Ans>
        ( c タグに囲まれた Q1−2の解集合 )
</Ans>
```

Q_{1-2}の解集合

```
<Ans>
        ( c タグに囲まれた Q2の解集合 )
</Ans>
```

Q_2の解集合

この解集合から，オリジナルの問い合わせによって得られる解を取り出すには，クライアントが次の問い合わせを行う．

$$CASE_2 = \begin{cases} Q_1 \leftarrow (Q_{1-2}, /Ans/c) \\ Q_1 \leftarrow (Q_2, /Ans/c) \\ Q_2 \leftarrow (Q_2, /Ans/c) \end{cases}$$

受信した解データには，Q_{1-2} の階層構造 $/a/\overline{\{b\}}$ と，Q_2 の階層構造 $/a/b$ に関する情報が無いため，このような問い合わせが必要となる．そこで，サーバは，それぞれに対して文脈に関する情報を付与した後，この二つの解を適切にマージした解集合 W を送信するとよい．

```
<a>
        <not_b>
                ( c タグに囲まれた Q1-2の解集合 )
        </not_b>
        <b>
                ( c タグに囲まれた Q2の解集合 )
        </b>
</a>
```

クライアントは，文脈に関する情報が付与した解を受け取った場合，オリジナルの問い合わせで解を取り出すことができる．

$$CASE_5 = \begin{cases} Q_1 \leftarrow (W, /a/*/c) \\ Q_2 \leftarrow (W, /a/b/c) \end{cases}$$

サーバが複数の解データを一つにマージすれば，一度にデータを送受信できる．また，クライアントは受信した一つの解データから，オリジナルの問い合わせと同じもので解を取り出すことが可能となる．文献 [1] で示したアルゴリズムでは，オリジナルの問い合わせ一つに対して複数の問い合わせで解を取り出す必要があるので，それとに比べると有効である場合も多いと考えられる．さらに，クライアントは，解の取り出し方を求めるための計算コストを抑えることも可能となる．

6.2 インデックス情報

クライアント側でのオリジナルの解集合の取り出しは，クライアントの環境によって大きな負担になる場合も考えられる．例えば，処理能力が制限された携帯電話や PDA といった組み込み機器では，取り出しに利用するパス式が少しでも複雑になれば，すぐに計算の負荷が大きくなり，処理できないという事態に陥る可能性がある．一般的な情報検索システムとは異なり，サーバから送信される，通信コストの上で最適化されたデータを単純にオリジナルの解として利用できないという点が問題になる．

これを解決するための方法として，クライアントが簡単にオリジナルの解を取り出せるように，サーバ側であらかじめ，取り出すための計算を負担する方法が考えられる．サーバ側で取り出しの計算を行えば，その計算結果をインデックスのようなものとして，解データと共にクライアントに送信することができる．例えば，ある解 Q を Q' から取り出すための情報は次のような，データの先頭からのバイトオフセットの組の集合で表現できる．

$$\begin{cases} (256, 128) \\ (1024, 64) \\ (4000, 320) \end{cases}$$

これは，送信されてきた Q' の解から，Q の解を生成するためにどの部分を取り出すのかを指示したインデックス情報である．この場合，Q' の先頭からみて，256 バイト目から 128 バイトを取り出し，次に 1024 バイト目から 64 バイトを取り出し，4000 バイト目から 320 バイトを取り出したものが，Q の問い合わせに相当する解集合である，という意味である．バイトオフセットがソートされていれば，クライアントはインデックスを上から順番に見ながら，受信した解データからオリジナルの問い合わせの解データを一度のスキャンで取り出すことができる．つまり，クライアントは計算コストの高い XML のパーズ処理や，XPath による取り出しをまったく行う必要がなくなるのである．

一方，インデックス情報をサーバが送信するということは，インデックスデータの通信コストが新たに増加するため，サーバとクライアントがやり取りする通信コストの点からは負荷を増大することになる．そのため，XML のような木構造データに対して，抜き出すべき部分集合の個所を記述するための効率的なデータ構造と，そのデータ量を最適化する手法が必要であり，今後の研究課題である．

第7章　計算コストの最適化に関する評価実験

文献 [1] の通信コストのみを完全に最適化するアルゴリズムにおいて，計算コストが増大する問題を解決するために，第 5 章ではサーバ側の，第 6 章ではクライアント側の計算コストを改善する手法の提案を行った．本章では，これらの手法が実際に有効なものか検証するための，評価実験を示す．

7.1　サーバ側での計算コストの改善

7.1.1　実験環境

サーバ側での計算コストの評価実験環境は，第 4 章で紹介したものとほぼ同じであるが，次の点で異なる．まず，XML を格納しているデータベースは，Relational Encoding のみを用いる．これは，Xalan の DOM を利用すると，計算時間の一部が解集合のデータサイズに比例する場合があるため，正確な計算コストを計算できないためである．第 4 章では，通信コストの最適化を重視した実験であり，こちらは計算コストの最適化を重視した実験であるため，計算コストをより正確に評価できる Relational Encoding のみで実験を行う．また，実験の XML データは 50MB のみを扱うこととした．これは，50MB のみでも十分，計算コストの最適化による速度の違いが判定できたためである．

7.1.2　非再帰的な問い合わせによる評価実験

まず始めに，非再帰的な問い合わせによる実験を考える．クライアントは，第 3 章の例 $CASE_3$ を，オークションデータに適用した，次のような XPath の問い合わせを行う．

$$EX_3 = \begin{cases} Q_1 : /site/people/person/name \\ Q_2 : /site/people/person[emailaddress]/name \\ Q_3 : /site/people/person[homepage]/name \end{cases}$$

文献 [1] の通信コスト最適化のアルゴリズムを適用すると，次のような問い合わせに変換される．

$$EX_3 = \begin{cases} Q_{2\cap3}: & /site/people/person[emailaddress][homepage]/name \\ Q_{2-3}: & /site/people/person[emailaddress]\overline{[homepage]}/name \\ Q_{3-2}: & /site/people/person\overline{[emailaddress]}[homepage]/name \\ Q_{1-2-3}: & /site/people/person\overline{[emailaddress][homepage]}/name \end{cases}$$

一方，同じ問い合わせを 5.1 節で提案した手法で変換すると，以下のようになる．この場合は，否定演算は使用せずに，積演算および，集合間の差演算に変換されている．

$$EX_3 = \begin{cases} Q_{2\cap3}: & Q_{2\cap3} \\ Q_{2-3_{opt}}: & Q_2 - Q_3 \\ Q_{3-2_{opt}}: & Q_3 - Q_2 \\ Q_{1-2-3_{opt}}: & Q_1 - Q_2 - Q_3 \end{cases}$$

実験を行うため，これらの XPath を最終的に SQL へ変換する必要がある．ここでは，紙面の都合上，$Q_{2-3}, Q_{2-3_{opt}}$ の二種類のみを紹介する．前順序**pre** の比較演算で一部で冗長な部分が含まれているが，これは，関係データベースの最適化器にヒントを与え，高速化をはかるためのものである．

Q_{2-3}：

```
SELECT
    d4.pre
FROM
    DATA d1, DATA d2, DATA d3, DATA d4
WHERE
    d1.name = 'site' AND d2.name = 'people' AND d3.name = 'person' AND
    d4.name = 'name' AND d1.parentID = 0 AND d2.parentID = d1.pre AND
    d3.parentID = d2.pre AND d4.parentID = d3.pre AND d1.rightchild >= d2.pre AND
    d2.rightchild >= d3.pre AND d3.rightchild >= d4.pre AND d1.pre < d2.pre AND
    d2.pre < d3.pre AND d3.pre < d4.pre AND
    EXISTS (
        SELECT
            *
        FROM
            DATA p1
        WHERE
            p1.name = 'emailaddress' AND d3.pre = p1.parentID AND d3.pre < p1.pre ) AND
    NOT EXISTS (
        SELECT
            *
        FROM
            DATA p1
        WHERE
            p1.name = 'homepage' AND d3.pre = p1.parentID AND d3.pre < p1.pre )
```

$\mathbf{Q_{2-3_{opt}}}$:

```
SELECT
    rootset.pre
FROM (
        SELECT
            d4.pre
        FROM
            DATA d1, DATA d2, DATA d3, DATA d4
        WHERE
            d1.name = 'site' AND d2.name = 'people' AND d3.name = 'person' AND
            d4.name = 'name' AND d1.parentID = 0 AND d2.parentID = d1.pre AND
            d3.parentID = d2.pre AND d4.parentID = d3.pre AND d1.rightchild >= d2.pre AND
            d2.rightchild >= d3.pre AND d3.rightchild >= d4.pre AND d1.pre < d2.pre

    MINUS

        SELECT
            d4.pre
        FROM
            DATA d1, DATA d2, DATA d3, DATA d4
        WHERE
            d1.name = 'site' AND d2.name = 'people' AND d3.name = 'person' AND
            d4.name = 'name' AND d1.parentID = 0 AND d2.parentID = d1.pre AND
            d3.parentID = d2.pre AND d4.parentID = d3.pre AND d1.rightchild >= d2.pre AND
            d2.rightchild >= d3.pre AND d3.rightchild >= d4.pre AND d1.pre < d2.pre AND
            d2.pre < d3.pre AND d3.pre < d4.pre AND
            EXISTS (
                SELECT
                    *
                FROM
                    DATA p1
                WHERE
                    p1.name = 'homepage' AND d3.pre = p1.parentID AND d3.pre < p1.pre )
    ) rootset
```

そして，EX_3 の三種類の問い合わせ集合をそれぞれ実行し，比較評価実験を行った．加えて上述で解説した EX_3 以外の問い合わせ集合にも，同様の計算コスト最適化のアルゴリズムを適用して，いくつか実験した．この結果を表 7.1，表 7.2 に掲載している．

まず，EX_3 を取り上げる．Q_{2-3} と $Q_{2-3_{opt}}$，Q_{3-2} と $Q_{3-2_{opt}}$ をそれぞれ比較すると，最適化した後者のものが計算コストの面で改善されている．高速化された一つめの要因として，処理している述語の数が 2 つから 1 つに減少したことが考えられる．EX_3 の Q_1, Q_2, Q_{3-2} の述語の数は，それぞれ，0, 1, 2 個と増えていく．計算時間をそれぞれ比較すると，述語の数が増えるとそれだけ計算が複雑になり，全体的に計算速度が落ちている事がわかる．また，高速化された二つめの要因は否定演算が差演算に変換されたためであると考えられる．これは，実験 EX_4, EX_5 から判定できる．共に，計算コストを最適化した $Q_{1-2_{opt}}$ は

30%ほど高速化されている．これは，否定の演算を差演算に変換したことにより，処理が軽減したためであるといえる．

一方，EX_6 を見ると，最適化したものの計算時間が大きくなっている．Q_1 は，データベース中のほぼ全データを取り出すことができる問い合わせである．Q_1 のデータサイズは，約 50M であり，非常に大きい．また，Q_2 のデータサイズも約 25M と，全データのほぼ半分である．集合の差演算を行うには，前処理として，演算対象となる集合をそれぞれ取り出さなければならない．サイズが大きいデータを取り出す計算コストは一般に高いため，大きな集合同士の差演算は，否定とは逆に速度が落ちてしまったものと考えられる．

以上のことから，データサイズがそれほど大きくないものは，なるべく否定演算を含まないように肯定演算の差集合に変換したものは有効であることが分かった．データサイズが大きいものは，逆に速度が遅くなるため，否定演算を用いる問い合わせに変換したほうがよい．サーバ側では問い合わせで得られる解のサイズによって，そのまま否定演算を行うか，差演算に切り替えるか，適宜判断して最適化すれば効率的である．また，述語を複数個，問い合わせ中に含むものを，集合の差演算に変換することにより，述語の数を減らすことが可能となる．実験の結果からも，述語の演算が多いほど，処理速度が落ちるため，述語の数が減ることは，計算コストの面において有効である．

7.1.3 再帰的な問い合わせによる評価実験

次に再帰的な問い合わせによる実験を考える．クライアントは，第 4 章の EX_2 と同様に，次のような XPath の問い合わせを行う．Q_1 は，任意の深さに現れる $parlist$ エレメントを根とする部分木を問い合わせている．

$$EX_2 = \left\{ \; Q_1 : \;\; //parlist \right.$$

文献 [1] の通信コスト最適化のアルゴリズムを Q_1 に適用すると，自己冗長性を取り除くために次のような問い合わせに変換される．

$$EX_2 = \left\{ \; Q_1^{\top} : \;\; //parlist - //parlist//parlist \right.$$

一方，これに 5.2.1 項で提案した手法を適用すると，次のように変換される．

$$EX_2 = \left\{ \begin{array}{ll} Q_1^{\top}{}_{opt_1} : & /parlist \\ & /\overline{\{parlist\}}/parlist \\ & /\overline{\{parlist\}}/\overline{\{parlist\}}/parlist \\ & /\overline{\{parlist\}}/\overline{\{parlist\}}/\overline{\{parlist\}}/parlist \\ & /\overline{\{parlist\}}/\overline{\{parlist\}}/\overline{\{parlist\}}/\overline{\{parlist\}}/parlist \\ & /\overline{\{parlist\}}/\overline{\{parlist\}}/\overline{\{parlist\}}/\overline{\{parlist\}}/\overline{\{parlist\}}/parlist \\ & \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ & (/\overline{\{parlist\}}/\overline{\{parlist\}} \ldots /\overline{\{parlist\}})_{11_{step}}/parlist \end{array} \right.$$

XPath		
EX_3	Q_1 :	$/site/people/person/name$
	Q_2 :	$/site/people/person[emailaddress]/name$
	Q_3 :	$/site/people/person[homepage]/name$
	$Q_{2\cap 3}$:	$/site/people/person[emailaddress][homepage]/name$
	Q_{2-3} :	$/site/people/person[emailaddress]\overline{[homepage]}/name$
	Q_{3-2} :	$/site/people/person\overline{[emailaddress]}[homepage]/name$
	Q_{1-2-3} :	$/site/people/person\overline{[emailaddress][homepage]}/name$
	$Q_{2-3_{opt}}$:	$Q_2 - Q_3$
	$Q_{3-2_{opt}}$:	$Q_3 - Q_2$
	$Q_{1-2-3_{opt}}$:	$Q_1 - Q_2 - Q_3$
EX_4	Q_1 :	$/site/regions/*/item$
	Q_2 :	$/site/regions/namerica/item$
	Q_{1-2} :	$/site/regions/\overline{\{namerica\}}/item$
	$Q_{1-2_{opt}}$:	$Q_1 - Q_2$
EX_5	Q_1 :	$/site/regions/*/item$
	Q_2 :	$/site/regions/africa/item$
	Q_{1-2} :	$/site/regions/\overline{\{africa\}}/item$
	$Q_{1-2_{opt}}$:	$Q_1 - Q_2$
EX_6	Q_1 :	$/site/*/*$
	Q_2 :	$/site/regions/*$
	Q_{1-2} :	$/site/\overline{\{regions\}}/*$
	$Q_{1-2_{opt}}$:	$Q_1 - Q_2$

表 7.1: 50MB の XML データに対する非再帰的な問い合わせの実験対象

50MB		Time(ms)	Size(KB)
EX_3	Q_1	1375	4921.78
	Q_2	128125	4921.78
	Q_3	166860	2538.57
	$Q_{2\cap 3}$	186313	2538.57
	Q_{2-3}	262062	2383.21
	Q_{3-2}	205812	0
	Q_{1-2-3}	211672	0
	$Q_{2-3_{opt}}$	179798	2383.21
	$Q_{3-2_{opt}}$	146734	0
	$Q_{1-2-3_{opt}}$	184075	0
EX_4	Q_1	1563	24988.36
	Q_2	1032	11458.89
	Q_{1-2}	1516	13529.47
	$Q_{1-2_{opt}}$	1251	13529.47
EX_5	Q_1	1344	24988.36
	Q_2	943	658.63
	Q_{1-2}	1692	24329.73
	$Q_{1-2_{opt}}$	1266	24329.73
EX_6	Q_1	1954	50716.99
	Q_2	125	24988.36
	Q_{1-2}	1829	25728.63
	$Q_{1-2_{opt}}$	2156	25728.63

表 7.2: 50MB の XML データに対する非再帰的な問い合わせの実験結果

用意した 50MB の XML データベースの最大の深さは 12 であったため，12 個の非再帰的な問い合わせの集合の和演算に変換されている．

次に，これを 5.2.2 項で提案した手法を適用すると，次のように変換される．

$$EX_2 = \begin{cases} Q_1^{\top}{}_{opt_2} : & /*/*/*/*/parlist \\ & /*/*/*/*/\overline{\{parlist\}}/parlist \\ & /*/*/*/*/\overline{\{parlist\}}/\overline{\{parlist\}}/parlist \\ & /*/*/*/*/\overline{\{parlist\}}/\overline{\{parlist\}}/\overline{\{parlist\}}/parlist \end{cases}$$

XML データベース中に現れる *parlist* エレメントが現れる深さは 5, 6, 7, 8 の 4 つであったため，4 個の非再帰的な問い合わせの集合の和演算に変換されている．

そして，それぞれの実験結果は表 7.3 のようになった．$Q_1^{\top}$ の通信コストの最適化のみでは，273.5 秒ほどの計算時間が，計算コストの最適化を行うと $Q_1^{\top}{}_{opt_1}$ では 11.3 秒となり，$Q_1^{\top}{}_{opt_2}$ では 5.1 秒となり，飛躍的に計算時間が改善された．前者では，約 24 倍高速化され，後者では，約 53 倍も高速化されていることが分かる．Relational Encoding では，問い合わせ中の再帰 // が少数なら高速なのだが，複数現れると非常に遅くなってしまう問題がある．これは，Relational Encoding 上で // が処理される場合，// 直下に現れるラベル名を持つ，エレメントの開始位置から終了位置までに現れるすべのエレメントを比較する必要があり，直積演算の負荷が高くなるのが原因である．文献 [1] で紹介したネットワーク流量の最適化は，再帰 // をたくさん生み出してしまうため，非再帰に展開することにより高速化される．

50MB	Time(ms)	Size(KB)
Q_1	784	24801.50
$Q_1^{\top}$	273453	17487.75
$Q_1^{\top}{}_{opt_1}$	11328	17487.75
$Q_1^{\top}{}_{opt_2}$	5125	17487.75

表 7.3: 50MB の XML データに対する再帰的な問い合わせの実験結果

同様に，いくつかの問い合わせで実験を行ったものが表 7.4 である．オリジナルの問い合わせを Q，通信コストを最適化した問い合わせを $Q^{\top}$，5.2.2 項の手法で通信・計算コストを最適化した問い合わせを $Q^{\top}{}_{opt_2}$ とし，それぞれを比較している．

表 7.4 を見ると，Q に対してすべての $Q^{\top}$ の処理時間は非常に遅くなっていることが分かるが，$Q^{\top}{}_{opt_2}$ は $Q^{\top}$ より高速である．Q_1, Q_2, Q_4, Q_5 のようにオリジナルの問い合わせ自体が高速に処理できるものは，計算コストを最適化しても，オリジナルのものよりは遅くなってしまうが，ネットワークの流量の上で最適化した $Q^{\top}$ と比べると劇的に高速化されていることが分かる．また，Q_3, Q_6 のように，オリジナルの問い合わせ自体が複雑なものは，計算コストの最適化を行った結果，オリジナルのものより高速になっていることが分かる．よって，この計算コストの最適化は非常に有効であるといえる．

XPath
Q_1 : //*parlist*
Q_2 : //*listitem*
Q_3 : //*item*//*listitem*
Q_4 : //*parlist*/*listitem*/*parlist*
Q_5 : //*emph*
Q_6 : //*open_auction*//*description*

50MB	Time(ms)	Size(KB)	Note
Q_1	784	24801.50	・再帰を持つ
$Q_1^{\top}$	273453	17487.75	・*parlist* は，深さ 5–8 に現れる
$Q_{1\ opt_2}^{\top}$	5125	17487.75	
Q_2	312	24591.84	・再帰を持つ
$Q_2^{\top}$	1569844	17356.40	・*listitem* は，深さ 6–9 に現れる
$Q_{2\ opt_2}^{\top}$	14734	17356.40	
Q_3	875484	11745.54	・再帰を持つ
$Q_3^{\top}$	1698125	8270.09	・*item* は，深さ 4 に現れる
$Q_{3\ opt_2}^{\top}$	7172	8270.09	・*listitem* は，深さ 6, 8 に現れる
Q_4	1719	7313930	・DTD 上では再帰を持つが，実際のデータに再帰は現れない
$Q_4^{\top}$	1028515	7313930	
$Q_{4\ opt_2}^{\top}$	14219	7313930	・*parlist* は，深さ 5–8 に現れる
Q_5	281	1950.49	・DTD 上では再帰を持たないが，深さが特定できないため // での問い合わせが必要
$Q_5^{\top}$	8460922	1950.49	
$Q_{5\ opt_4}^{\top}$	39094	1950.49	・*emph* は，深さ 8–12 に現れる
Q_6	396407	7520.50	・DTD 上では再帰を持たない
$Q_6^{\top}$	1797922	7520.50	・*open_auction* は，深さ 3 に現れる
$Q_{6\ opt_2}^{\top}$	3078	7520.50	・*description* は，深さ 4–5 に現れる

表 7.4: 50MB の XML データに対する再帰的な問い合わせの実験結果

7.2 クライアント側での計算コストの改善

クライアントは，サイズ最小の解集合をサーバから受信し，これに対して，オリジナルの問い合わせで得られる解を取り出す．第 6 章で提案した，解の取り出しを最適化するアルゴリズムの有効性を検証するため，これを適用したものと，最適化を行わなかったものとの比較実験を行った．

7.2.1 実験環境

クライアント側での計算コストの評価実験環境は，サーバでの実験と比べるといくつか異なる点がある．解の取り出しに関する計算コストを比較するために，Xerces[15] の SAX パーサを利用した XPath 問い合わせシステムを独自に構築した．この XPath 問い合わせシステムは，ファイルの先頭から末尾まで一度のスキャンで XPath の評価を行い，その解を返すものである．現在処理しているファイル位置から前方に戻ることは決してなく，常に後方に向かって処理をしている．複数の XPath 問い合わせ集合が与えられても，並列にマッチング処理されるため，常に一度のスキャンで XPath の問い合わせを行うことが可能である．

実験のマシン環境は，サーバで使われている高度なものではなく，一般にクライアントが用いるような環境を利用した．1CPU（Pentium III 600MHz），256MB のメモリを搭載した Microsoft Window 2000 上で，システムの実験を行った．

7.2.2 解集合のデータ加工とインデックス情報

第 6 章では，クライアントの計算コストを最適化する手法を提案した．それは，1. サーバが解集合を加工し，データベースの文脈に関する情報を付与する手法と，2. サーバが解データと共に，サイズ最小の解集合からオリジナルの解集合の取り出す方法を記述したインデックス情報を送信する手法の二種類である．ここでは，通信コストの最適化を行って，従来通りに解を取り出す方法と，上述の二種類の手法を適用したものから解を取り出す方法を比較するため計算コストの評価実験を行う．

まず，例 $CASE_2$ をオークションデータに適用した，次のような問い合わせを考える．

$$EX_a = \left\{ \begin{array}{ll} Q_1: & /site/regions/*/item \\ Q_2: & /site/regions/namerica/item \end{array} \right.$$

この問い合わせをネットワーク通信コストの上で最適化したものは，次のようになる．

$$EX'_a = \left\{ \begin{array}{ll} Q_{1-2}: & /site/regions/\overline{\{namerica\}}/item \\ Q_2: & /site/regions/namerica/item \end{array} \right.$$

この解集合をクライアントが受信した場合，オリジナルである EX_a と同様の解を取り

出すためには，次の問い合わせが必要となる．

$$EX_a = \begin{cases} Q_1 \leftarrow (Q_{1-2}, /Ans/item) \\ Q_1 \leftarrow (Q_2, /Ans/item) \\ Q_2 \leftarrow (Q_2, /Ans/item) \end{cases}$$

クライアントは，Q_1 を取り出すために，EX'_a の Q_{1-2} と Q_2 に対して二つの問い合わせが，Q_2 を取り出すために，EX'_a の Q_2 に対して一つの問い合わせが，それぞれ必要となり，合計三つの問い合わせを行う．これは，解集合が単純に Ans タグに囲まれて次のような形式で送信されるため，XML の文脈に関する情報が解から失われてしまうことが原因である．

<Ans>
(item タグに囲まれた Q_{1-2}の解集合)
</Ans>

Q_{1-2}の解集合

<Ans>
(item タグに囲まれた Q_2の解集合)
</Ans>

Q_2の解集合

第 6 章で提案した，サーバが解データを加工し，失われてしまった XML の文脈に関する情報を付与する手法を用いると，次の問い合わせでよい．

$$EX_a = \begin{cases} Q_1 \leftarrow (Q_{1-2} \cup Q_2, /site/regions/*/item) \\ Q_2 \leftarrow (Q_{1-2} \cup Q_2, /site/regions/namerica/item) \end{cases}$$

XML データ中の $Q_1 : /site/regions/*$，$Q_2 : /site/regions/namerica$ という文脈に関する情報を解に付与して送ることにより，クライアントはオリジナルの EX_a の問い合わせ内容と全く同じものによって解を取り出すことが可能となる．また，サーバが複数の解データを一つにマージすれば，一度にデータを送受信できる．

<site>
<regions>
<not_namerica>
(item タグに囲まれた Q_1の解集合)
</not_namerica>
<namerica>
(item タグに囲まれた Q_2の解集合)
</namerica>
</regions>
</site>

さらに，XML データを Relational Encoding でデータベースに格納する際，各エレメントの開始バイトオフセットと終了バイトオフセットを統計情報として保存しておけば，第 6 章の後半で説明したインデックスの手法を適用することが可能である．

これまで説明した例をはじめとする，様々な問い合わせの比較評価実験を行った結果が表 7.5–7.7 である．EX_a, EX_b が，非再帰的な問い合わせ集合に対する実験で，EX_c, EX_d が，再帰的な問い合わせ集合に対する実験である．

まず，従来の取り出し方と文脈情報を利用した取り出し方を比較してみると，EX_b を除いて，後者のほうが遅くなっている．文脈情報を解に付与すると，追加されたエレメントの分だけ，SAX でパーズする計算時間が増大する．そのため，取り出す問い合わせの数の差がそれほど無かったこれらは，パーズのコストが増加したためであると考えられる．また，EX_b に関していえば，取り出しの問い合わせの数が 8 個から 4 個に減ったことにより，速度が上がったものだと考えられる．

次に，文脈情報をサーバが付与し，送る解集合を一つにマージしたものを比較してみると，EX_a, EX_b 共に，計算コストが低減している．これは，複数のデータに対して別々問い合わせを行う場合よりも，一つのデータに対して一度に問い合わせを行うほうが，オーバヘッドが少ないため，計算コストが軽減されていると考えられる．

従来のものとインデックスを利用して解を取り出すものと比較すると，約 2 倍から 3 倍，高速になっている．一般に，XML データに問い合わせを行う場合，計算コストの大部分がパーズ時間であることが文献 [7] で示されている．取り出し方を記述したインデックス情報を利用すれば，XML データをパーズすることなく，解を取り出すことができるため，非常に高速である．

以上の実験から，サーバが解を送信する場合，クライアント側である程度取り出しやすいように，データを加工するのは有効であるといえる．ただし，文脈に関する情報を解に追加したり，インデックス情報をデータと共に送ったりすると，ネットワークの通信コストは増大する．文脈に関する情報をデータに付与する手法では，それほど大きな通信量の変化はない．しかし，インデックス情報を用いる場合，取り出し方が複雑であれば，それだけ通信量が増大する．今回，利用したインデックスでは，EX_b では，最大 214KB もの通信量が増大した．そのため，今後の課題として，通信コストを最適化するデータの取り出し方の記述構造を考案する必要がある．

XPath		従来の取り出し方
EX_a	$Q_1 : /site/regions/*/item$	$Q_1 \leftarrow (Q_{1-2}, /Ans/item)$
		$Q_1 \leftarrow (Q_2, /Ans/item)$
	$Q_2 : /site/regions/namerica/item$	$Q_2 \leftarrow (Q_2, /Ans/item)$
EX_b	$Q_1 : /site/regions/namerica/item$	$Q_1 \leftarrow (Q_1, /Ans/item)$
	$Q_2 : /site/regions/europe/item$	$Q_2 \leftarrow (Q_2, /Ans/item)$
	$Q_3 : /site/regions/*/item/name$	$Q_3 \leftarrow (Q_1, /Ans/item/name)$
		$Q_3 \leftarrow (Q_2, /Ans/item/name)$
		$Q_3 \leftarrow (Q_{3-1-2}, /Ans/name)$
	$Q_4 : /site/regions/*/item/description$	$Q_4 \leftarrow (Q_1, /Ans/item/description)$
		$Q_4 \leftarrow (Q_2, /Ans/item/description)$
		$Q_4 \leftarrow (Q_{4-1-2}, /Ans/description)$
EX_c	$Q_1 : //parlist/listitem/parlist$	$Q_1 \leftarrow (Q_1^{\top}, /Ans/parlist)$
		$Q_1 \leftarrow (Q_1^{\top}, /Ans/listitem/parlist)$
		$Q_1 \leftarrow (Q_1^{\top}, /Ans//parlist/listitem/parlist)$
EX_d	$Q_1 : //open\_auction//desceiption$	$Q_1 \leftarrow (Q_1^{\top}, /Ans/desceiption)$
		$Q_1 \leftarrow (Q_1^{\top}, //open\_auction//desceiption)$

第 6 章の文脈情報を用いる場合は，オリジナルの問い合わせで同じもので取り出す

表 7.5: オリジナル問い合わせと通信コスト最適化後のオリジナルの解の取り出し方

XPath		
EX'_a	Q_{1-2} :	$/site/regions/*/item$
	Q_2 :	$/site/regions/\overline{\{namerica\}}/item$
EX'_b	Q_1 :	$/site/regions/namerica/item$
	Q_2 :	$/site/regions/europe/item$
	Q_{3-1-2} :	$/site/regions/\overline{\{namerica, europe\}}/item/name$
	Q_{4-1-2} :	$/site/regions/\overline{\{namerica, europe\}}/item/description$
EX'_c	$Q_1^{\top}$:	$//parlist/listitem/parlist - //parlist/listitem/parlist//parlist$
EX'_d	$Q_1^{\top}$:	$//open\_auction//description - //open\_auction//description//desceiption$

表 7.6: 通信コストを最適化した問い合わせ

実行時間 (ms)		従来	文脈	文脈 $\bowtie$	索引	サイズ (KB)
EX'_a	Q_{1-2}	2704	2734	-	1122	13,213
	Q_2	2073	2373	-	1081	11,191
	計	4777	5107	4036	2203	24,404
EX'_b	Q_1	2274	2263	-	1122	11,191
	Q_2	1492	1492	-	661	6,753
	Q_{3-1-2}	330	310	-	10	76
	Q_{4-1-2}	811	782	-	261	3,434
	計	4907	4847	3926	2054	21,454
EX'_c	$Q_1^\top$	1562	1602	-	520	7,143
EX'_d	$Q_1^\top$	1652	2043	-	551	7,345

従来： 従来の取り出し
文脈： 文脈情報を利用した取り出し
索引： インデックス情報を利用した取り出し
$\bowtie$: 複数の解集合を一つにマージした

表 7.7: 実験結果

第8章　まとめと今後の課題

8.1　まとめ

本論文では，インターネット上でのデータ交換の業界標準ともなった，XML データに関する通信コストと計算コストの問題に焦点を当てている．ネットワーク上の XML データベースに対してクライアントが XPath による問い合わせを行う場合，返送される解に冗長性が生じ，通信コストが増大することがある．これを解決するために，我々はクライアント側で通信コストを最適化するサイズ最小のビューに変換するといった，通信コスト最適化のアルゴリズムをこれまでに提案している．

ところが，この手法では，通信コストの最適化に特化されており，サーバ側での計算コストは増大する場合がある．また，クライアント側では，オリジナルの問い合わせで得られるはずの解集合を，受信した解から独自で取り出すための，追加処理が必要となる問題がある．そこで，第一に，通信コストの最適化を行うことによって頻繁に発生し，計算コストの高い，非再帰的な問い合わせ集合に現れる否定演算と，再帰的な問い合わせ集合に現れる自己冗長性の除去する演算を，サーバ側がデータの状況や内容を把握することによって，計算コストを最適化する方法を提案した．次に，クライアント側での解の取り出し演算の計算コストを軽減するために，サーバ側が，XPath によって失われるデータベースの文脈情報を解に付与するといった方法や，クライアントが解を効率的に取り出せるようなインデックスを付与するといった方法を提案した．そして，これらの評価実験を行い，この有効性を証明した．

8.2　今後の課題

文献 [1] で提案した通信コスト最適化のアルゴリズムは，1. 任意個の非再帰的な問い合わせ集合が与えられた場合，2. 任意個の再帰を含む問い合わせが一つ与えられた場合，3. 非再帰的な問い合わせ一つとある制限された形の再帰を含む問い合わせが一つ与えられた場合，以上の三つの場合についてのみ，適用可能である．よって，任意の場合について適用できるアルゴリズムを開発することが通信コストの最適化の面では，最重要課題である．

クライアント側での計算コストの改善のため，データの取り出し方をインデックス情報として，解と共に送信する方法を提案したが，インデックス情報を送信するため，その分，通信コストが増大してしまう．本論文では，インデックス情報として，開始位置と終

了位置のバイトオフセットで表現する単純なものを考案したが，通信コストを最適化するための，XML データに対して効率よく取り出し方を記述できるインデックス構造の開発も今後の研究課題である．

謝 辞

本研究を行うに当たり，ご指導・ご鞭撻を頂いた田島敬史助教授に心から感謝し，深くお礼申し上げます．また，日常，有益な議論をして頂いた研究室の皆様に感謝します．

参考文献

[1] Keishi Tajima, Yoshiki Fukui, "Efficiently Answering XPath Queries over a Network by Sending Minimal Views", 投稿中.

[2] D. E. Knuth, J.H. Morris, and V. B. Pratt., "Fast pattern matching in strings.", SIAM Journal of Computing, pages 6:323–350, 1977.

[3] Torsten Grust, "Accelerating XPath Location Steps", In Proc. of ACM SIGMOD, pages 109–120, 2002. .

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A scalable Continuous Query System for internet databases", In Proc. of ACM SIGMOD, pages 379–390, 2000.

[5] Benjamin Nguyen, Serge Abiteboul, Grégory Cobena, and Mihaí Preda, "Monitoring XML Data on the Web", 2001.

[6] Bertran Ludascher, Pratik Mukhopadhyay, and Yannis Papakonstantinou, "A Transducer-Based XML Query Processor", In Proc. of VLDB, pages 227–238, 2002.

[7] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu, "Processing XML Streams with Deterministic Automata", In Proc. of ICDT, 2003.

[8] D. Olteanu, H. Meuss, T. Furche, and F. Bry, "Xpath: Looking forward", In Proc. of EDBT Workshop on XML Data Management(XMLDM) LNCS, pages 109–127, 2002.

[9] M. Altinel, and M.J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information", In Proc. of VLDB, pages 53–64, 2000.

[10] J. Clark, and S.D., editors, "XML Path Language (XPath) Version 1.0 - W3C Recommendation", `http://www.w3.org/TR/xpath.html`, Nov, 1999.

[11] J. Clark, and S. DeRose, editors, "XML Path Language (XPath) Version 2.0 - W3C Working Draft", `http://www.w3.org/TR/xpath20/`, Nov, 2003.

[12] J. Clark, editor, "XSL Transformations (XSLT) Version 1.0 - W3C Recommendation", http://www.w3.org/TR/xslt, Nov, 1999.

[13] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon, editors, "XQuery 1.0: An XML Query Language - W3C Working Draft", http://www.w3.org/TR/xquery/, Nov. 2003. .

[14] A. Schmidt, F. Waas, M.L. Kersten, M.J. Carey, I. Manolescu, and R. Busse, "XMark: A benchmark for XML data management", In Proc. of VLDB, pages 974–985, Aug. 2002.

[15] "Xerces", http://xml.apache.org/xerces2-j/index.html.

[16] "Xalan", http://xml.apache.org/xalan-j/index.html.