ネットワーク上での XPath 問い合わせの計算・通信コストの最適化

福井　佳紀† 　　田島　敬史†

† 北陸先端科学技術大学院大学　〒 923–1292 石川県能美郡辰口町旭台 1-1
E-mail: †{y-fukui,tajima}@jaist.ac.jp

あらまし　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ることがある．そのため，これらの解をサーバが別々にクライアントに送信すると，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，与えられた問い合わせ集合全てに答えることができるサイズ最小のビューをサーバからクライアントに送ることで，通信コストを最適化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は増加する場合がある．そこで，本稿では，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．
キーワード　XML，XPath，問い合わせ処理，分散 DB，最適化

Optimization of computation/communication cost for answering XPath queries over a network

Yoshiki FUKUI† and Keishi TAJIMA†

† Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292 Japan
E-mail: †{y-fukui,tajima}@jaist.ac.jp

Abstract　When a client issues a set of XPath queries, or one XPath query, to a XML database over a network, a set of answer sets that are sent back to the client may include redundancy. Therefore, sending those answer sets separately to the client over a network is not optimal with respect to the communication cost. To solve this problem, in our previous paper, we proposed a method of minimizing the communication cost by sending from the server to the client a miminal-size view set that can answer to all the original queries. Although that approach reduces the communication cost, in some cases, it increases the computation cost at the server or the client instead. In this paper, we propose a method that takes both the communication cost and the computation cost into consideration.
Key words　XML, XPath, Query Processing, Distributed Database, Optimization

1. はじめに

今日，XML フォーマットは，インターネット上でのデータ交換やデータ発信における標準データ形式として広く使われるようになっている．そのため，ネットワーク上に大量に存在する XML データを効率的に問い合わせるための技術が必要になっている．

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] や XML ストリームサービス [5] 等が挙げられる．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システムである．また，XML ストリームサービスでは，サーバが XML データのストリームを配信し，クライアント側で問い合わせ処理を行う．

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るが，その中で，XPath1.0 [6] と呼ばれる問い合わせ言語が，既に W3C の勧告となり世界中で幅広く利用されるようになっている．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．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，部分木に子エレメントを追加したり，部分木から一部の子エレメントを取り除いたり，タグ名やテキストデータを変更したりといったデータの変更は一切行うことができないという特徴がある．

前述の連続問い合わせシステムのようにサーバ側で問い合わ

せを処理するシステムでは，問い合わせの解のみがクライアントに送られるので，クライアント側で問い合わせが処理されるシステムに比べ，通信コストの上では効率が良いと思われるが，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，クライアントが複数の XPath による問い合わせを行う場合，返送される解集合に冗長性が含まれることがあるためである．さらに，複数ではなく，単一の XPath 問い合わせを発行する場合にも，その解の中に自己冗長性が含まれる場合がある．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，我々は，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ことを目的として，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場合を考える．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．しかし，これまでに提案した手法は，通信コストの最適化のみを考慮しており，サーバで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対応する問い合わせは，オリジナルの問い合わせと比べて複雑になるのが原因である．そこで本論文では，これらの複雑な問い合わせを，より簡単な問い合わせに変形することで，計算コストもできる限り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．上述のサイズ最小のビューに変換する手法では，クライアント側で，受け取ったサイズ最小のビューからオリジナルの問い合わせの解集合を取り出す処理が必要となる．クライアントが携帯電話や PDA といったマシン性能が制限された機器を利用している場合を考えると，この計算コストは好ましくない．そこで，このコストを軽減するための手法も提案する．

以下，次の第 2. 節では，本稿で取り扱う XPath の部分言語について説明する．第 3. 節では，XPath 問い合わせの集合を，文献 [1] で示したアルゴリズムを使って，サイズ最小のビューに変換する例をいくつか挙げ，その評価実験の結果を第 4. 節で示す．第 5. 節と第 7. 節では，それぞれ，サーバに手を加えられない場合と手を加えられる場合について，文献 [1] で提案した手法で発生する計算コストの増大を改善する手法について提案し，それぞれの評価実験の結果を第 6. 節と第 8. 節で示す．最後の第 9. 節では，全体のまとめと今後の課題について述べる．

2. XPath

前節で述べたように，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．本稿では，XPath 問い合わせの解集合は，Ans エレメントを根とし，解集合中の各要素をその子供とする 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．

$$<Ans><a>\cdots\cdots\cdots</Ans>$$

2.1 使用する XPath の部分言語の文法

本稿では，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．

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

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その下に持たないようなエレメントがマッチする．

3. 通信コスト最適化のための問い合わせ変換例

次に，この節では，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] で提案したアルゴリズムでは，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例は次のようなものである．

$$CASE_1 = \begin{cases} Q_1 : /a/*/c \\ Q_2 : /a/b/c \\ Q_3 : /a/b[e]/c/d \end{cases}$$

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 の解集合のみを見ても，そこに現れる 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判定できず，そこから Q_2 の解集合を取り出すことはできない．これは，Q_1 の解集合は c エレメントを根とする部分木の集合であり，もともとのデータベース中にあった文脈に関する情報，すなわち，この場合で言うと親エレメントのラベルの情報が失われているためである．

一方，Q_3 の解となるエレメントは，必ず Q_2 の解中のあるエレメントの部分木として含まれている．よって，Q_2 の解と Q_3 の解を別々に送信すると，Q_3 の解に含まれるデータは二回送られることになり，通信コストの上で冗長である．また，この場合も Q_1 と Q_2 の場合と同様，Q_2 の解集合のみを見て Q_3 の解を取り出すことはできない．これは，Q_2 の解は c エレメントを根とする部分木の集合であるが，どの c エレメントからは，その子供の d エレメントを Q_3 の解として取り出すべきなのかを判定するには，c エレメントの親の b エレメントの子供の情報が必要になるためである．

そこで，この場合，以下の問い合わせをサーバに送ればよい．

$$CASE_1 = \begin{cases} Q_{1-2} : & /a/\overline{\{b\}}/c \\ Q_{2-3} : & /a/b\overline{[e]}/c \\ Q_{2\cap 3} : & /a/b[e]/c \end{cases}$$

本稿では，問い合わせの解集合は，解集合中の各要素を子とするような 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_3 の解の生成は，$Q_{2\cap 3}$ の解集合に対して，$/Ans/c/d$ という問い合わせを実行すればよい．本稿では，今後，これを $Q_3 \leftarrow (Q_{2\cap 3}, /Ans/c/d)$ のように表記することにする．また，Q_1 の解は，クライアント側で $Q_{1-2}, Q_{2-3}, Q_{2\cap 3}$ のそれぞれの解の和集合を取ることで生成でき，Q_2 の解は，$Q_{2-3}, Q_{2\cap 3}$ のそれぞれの解の和集合を取ることで生成できる．

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

3.2 再帰を含む問い合わせの例

次に，再帰を含む問い合わせの解の冗長性について考える．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．

$$CASE_2 = \left\{ \ Q : \quad //a \right.$$

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できる．

$$CASE_2 = \left\{ \ Q^{\top} : \quad //a - //a//a \right.$$

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 Q の解はクライアント側で，次のようにして取り出すことができる．

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

// 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．

$$CASE_3 = \left\{ \ Q : //a/b/a/b \right.$$

この場合，解中の自己冗長性を取り除くには，$CASE_2$ と同様に，次の問い合わせを送信すればよい．

$$CASE_3 = \left\{ \ Q^{\top} : //a/b/a/b - //a/b/a/b//b \right.$$

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．

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

$Q^{\top}$ 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，その直下に a/b というパスが現れたら，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．

再帰的な問い合わせを考える場合，クライアントが一つの問い合わせのみを実行する場合でも，これを複数の問い合わせに分解する必要がある場合がある．例えば，以下の例を考える．

$$CASE_4 = \left\{ \ Q : //\mathsf{a}/*/* \right.$$

すなわち，任意の深さに現れる a エレメントの孫エレメントを取り出したいとする．この場合，以下の問い合わせの結果からだけでは，元の問い合わせの結果を生成できない．

$$\left\{ \ Q^{\top} : //\mathsf{a}/*/* - //\mathsf{a}/*/*//* \right.$$

末尾にある $-//\mathsf{a}/*/*//*$ は自己重複を取り除くための物である．この場合，以下の処理のみでは，Q の解を全て取り出

すには不十分である．

$$Q \leftarrow (Q^{\top}, \texttt{/Ans//a/*/*})$$

この処理のみでは，たまたま $Q^{\top}$ 中のある解の子供であったような解を取り出すことができない．例えば，`//a/a/b/c` というパターンにマッチするようなパスがデータ中にあったとする．この場合，b エレメントと c エレメントの双方が Q の解となるが，$Q^{\top}$ には b エレメントのみが含まれる．しかし，b エレメントの親のラベルがなんであったかの情報は $Q^{\top}$ の解中には含まれていないため，$Q^{\top}$ の解中のどのエレメントからは子供を取り出すべきなのかわからない．

この場合は，通信コストを最小にしながら，Q の解を正しく取り出せるためには，以下の問い合わせを送信すればよい．

$$\begin{cases} Q^{\top}_{\mathsf{a}}: & \texttt{//a/a/*} - \texttt{//a/*/*//*} \\ Q^{\top}_{\overline{\{a\}}}: & \texttt{//a/}\overline{\{a\}}\texttt{/*} - \texttt{//a/*/*//*} \end{cases}$$

そして，元の問い合わせの解は，以下のようにして取り出すことができる．

$$\begin{cases} Q \leftarrow & (Q^{\top}_{\mathsf{a}}, \texttt{/Ans/*}), (Q^{\top}_{\mathsf{a}}, \texttt{/Ans/*/*}), (Q^{\top}_{\mathsf{a}}, \texttt{/Ans//a/*/*}), \\ & (Q^{\top}_{\overline{\{a\}}}, \texttt{/Ans/*}), (Q^{\top}_{\overline{\{a\}}}, \texttt{/Ans//a/*/*}) \end{cases}$$

すなわち，$Q^{\top}_{\mathsf{a}}$ の解からのみ，その解の子エレメントも取り出すようにする必要がある．

4. 通信コストの最適化の評価実験

4.1 実験環境

実験データとしては，XMark [7] で生成した約 233MB の XML データを用いた．XMark は，オークションサイトを表現したある固定スキーマ (DTD) に基づく，任意のサイズのデータを生成できるツールである．データの内容は，出品地域ごとに分類された商品の詳細情報や参加者の登録情報などである．

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．まず，文献 [3] 等で用いられているような，XML データを，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属性を使って関係の形にエンコードする一般的な手法を用いて，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．この関係データベースに対しては，XPath を SQL に変換して問い合わせを行う．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，XML データを DOM 木の形でメモリ上に読み込み，この DOM 木の上で XPath を実行する処理系である Xalan [8] を用いて問い合わせを行う．計算機環境としては，2CPU（900MHz UltraSPARC-CU），6GB のメモリを搭載した Sun Blade 2000 を使用した．

4.2 非再帰的な問い合わせによる実験

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」のみを問い合わせている．

233MB	DOM Time(sec)	RDB Time(sec)	Size(KB)
変換前 計	644.67	68.77	216206.71
変換後 計	609.81	50.44	149612.39

表 1　非再帰的な問い合わせの実験結果

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

表 2　再帰的な問い合わせの実験結果

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

これを，サイズ最小のビューに変換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 にも含まれていて重複しているので，それを考慮している．

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

実験結果は表 1 のようになった．解集合の通信コスト，すなわち総データ量が約 216MB であったものが約 150MB になり，約 66.6MB(31%) の通信コストの軽減を実現している．さらに，DOM を用いる環境と関係データベース (RDB) を用いる環境のいずれにおいても，計算時間も改善されている．一般に，変換後の問い合わせは複雑になるため計算コストも増大する可能性があるのだが，解の小さい問い合わせに変換されることによる計算コストの減少もあり，ここでは計算コストも軽減されている．このように，日常的に使われる非再帰的な問い合わせの多くでは通信コストのみでなく，計算コストも改善される可能性があり，非常に効率的であるといえる．

4.3 再帰的な問い合わせによる実験

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，Q : $//parlist$ を問い合わせる．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 [7] 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．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 $Q^{\top}$: $//parlist - //parlist//parlist$ に変換される．表 2 に示した実験結果を見ると，約 159.0MB であった解集合のサイズが約 114.8MB になり，通信コストが約 44.2MB(28%) 削減されている．XPath では，一つの再帰的な問い合わせのみでも自己冗長性を含むことがあり，それを除去することで大きな通信コストの削減に繋がることが分かった．

一方，自己冗長性の除去を行う複雑な処理が必要となるために，計算時間は逆に増大してしまう可能性がある．表 2 を見ると，DOM を用いる場合は計算時間についても改善されている

が，RDB を用いる場合は，2 時間たっても結果を得ることができなかった．このように，通信コストの面では大きく改善されているが，RDB を用いる環境において計算コストが非常に増大してしまうため，このままでは実用的とはいえない．そのため，以後の節でこの問題の解決方法を提案する．

5. サーバ側での計算コストの改善

これまでに，冗長性がある XPath 問い合わせ集合の例と，その変換方法をいくつか挙げ，そして，実験によって実際にネットワークの通信コストを大幅に削減できることを示した．しかし，文献 [1] で示した変換アルゴリズムは，通信コストの最適化のみを考えており，前節で示したように，サーバの計算コストに関しては，逆に大きく増大してしまう場合がある．

そこで本節では，通信コストの最適化に加え，計算コストも可能な限り削減する手法を提案する．ここでは，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る場合，クライアント側で問い合わせを変換する際に，その情報を使って，より計算コストが低い問い合わせに変換するというアプローチを考える．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次節で示す.

しかし，問い合わせの計算コストは処理系によって大きく異なる．例えば，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．また，DOM は XML データをメモリ中で木構造に展開して処理を行う．そのため，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問い合わせは，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，述語表現が現れない，一度のスキャンで計算できる問い合わせに適している．また，RDB でデータを格納した場合，非再帰的な問い合わせが得意である．このように，処理系によって計算コストが大きく異なるため，一口に計算コストの良悪を言うことはできない．そこで本稿では，一般的に広く用いられている RDB を用いる方式を想定して，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

まず，非再帰的な問い合わせの場合について考える．$CASE_1$ の例で示したように，単純に全てを包含している問い合わせを実行するのでは，クライアント側でオリジナルの解を生成できない場合がある．我々は，このような場合，問い合わせの解の間の全ての重複分を取り除くように，解集合を細かく分割して取り出す問い合わせに変換している．これはちょうど，各解集合の間の関係をベン図で表し，そこに現れる全ての領域を別々に取り出す問い合わせに変換することに相当する．

この重複部分を別々に取り出すような問い合わせに変換するアルゴリズムでは，集合同士の差や共通部分に対応する問い合わせが頻繁に生成される．例えば，$CASE_1$ の場合 Q_{1-2}：$/a/\overline{\{b\}}/c$ は Q_1 の解集合と Q_2 の解集合の差に対応する．しかし，Q_{1-2} はそのような問い合わせを集合差演算ではなく否定を用いて表現している．処理系や問い合わせによっては，否定を含む問い合わせは，集合差演算よりもかえってコストがかかる可能性がある．そこで，状況に応じて，否定を含む問い合わせを生成するかわりに，実際に二つの問い合わせを別々に評価して差演算を行うことが考えられる．先ほどの例では，Q_1 と Q_2 を別々に評価し，Q_1 の解集合と Q_2 の解集合の差演算を行って $Q_{1-2} = Q_1 - Q_2$ を求めれば良い．後述の実験の結果，このような二つの解の差にあたる問い合わせを行う場合，二つの解集合のサイズが共に小さい場合には，否定を用いた問い合わせよりも集合の差演算を用いた問い合わせの方が効率が良いことがわかった．そこで，そのような場合は差演算を用いて解の差集合を求め，逆に二つの解集合のサイズが大きい場合は，これらを別々に評価すると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，否定を用いた問い合わせを使うようにする．

5.2 再帰的な問い合わせ

次に，再帰的な問い合わせを考える．この場合，自己冗長性を取り除く必要がある．例えば $CASE_2$ では，$//a$ という問い合わせを $//a - //a//a$ に変換して，根から最も近い a エレメントのみを取り出すことで，自己冗長性を取り除いている．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，実際には稀である．つまり，根から最も近い a エレメントを見つけるために，上記のような $//a - //a//a$ 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．

もしも，スキーマから XML データの最大の深さが分かれば，あるいは，サーバが，自身の保有する XML データの最大の深さを公開していれば，クライアントは，再帰的な問い合わせを非再帰的な問い合わせの和集合演算に変換することが可能である．例えば，XML データの深さが 3 である場合，$//a - //a//a$ は，次のように変換できる．

$$\{\ /a\ \cup\ /\overline{\{a\}}/a\ \cup\ /\overline{\{a\}}/\overline{\{a\}}/a$$

これは，クライアントが 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ある程度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よい．

$$\begin{cases} /a\ \cup\ /\overline{\{a\}}/a\ \cup\ /\overline{\{a\}}/\overline{\{a\}}/a\ \cup \\ /\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}//a - /\overline{\{a\}}/\overline{\{a\}}/\overline{\{a\}}//a//a \end{cases}$$

これは，深さ 3 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 4 以降は従来通りの方法で取り出している．

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

50MB		$Origin$(ms)	Opt(ms)	$Q_a - Q_b$(MB)
EX_1	Q_{2-3}	262062	179798	4.92 − 2.54
	Q_{3-2}	205812	146734	2.54 − 4.92
	Q_{1-2-3}	211672	184075	4.92 − 4.92 − 2.54
EX_2	Q_{1-2}	1516	1251	24.99 − 11.46
EX_3	Q_{1-2}	1829	2156	50.72 − 25.73

表 3　非再帰的な問い合わせの実験結果

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．

$$\begin{cases} /a/b/a \ \cup \ /*/a/b/a \ \cup \ /*/*/a/b/a - /a/b/a/b/a \ \cup \\ /*/*/*/a/b/a - /a/b/a/a/b/a - /*/a/b/a/b/a \ \cup \ \cdots \end{cases}$$

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 / しか含まない演算の方が計算コストは低いため，XML データがそれほど深くなければ，有効な変換である．

6. サーバ側での計算コストの最適化の評価実験

本節では，前節で提案したサーバ側の計算コストを改善する手法が，実際に有効なものか検証するための評価実験を示す．

本節での評価実験環境は，第 4. 節と比べて次の点で異なる．まず，XML の格納方法としては，前述の実験においてサーバ側の計算コストが問題になった，RDB に格納する方法のみを用いた．また，実験の XML データは 50MB のものを用いた．これは，このサイズで十分，提案手法による速度の違いが判定できたためである．

6.1 非再帰的な問い合わせによる実験

まず，非再帰的な問い合わせによる実験として EX_1，EX_2，EX_3 の三つの実験を行った．実験に用いた問い合わせは以下の通りである．

$$EX_1 \begin{cases} Q_1 : /site/people/person/name \\ Q_2 : /site/people/person[emailaddress]/name \\ Q_3 : /site/people/person[homepage]/name \end{cases}$$

$$EX_2 \begin{cases} Q_1 : /site/regions/*/item \\ Q_2 : /site/regions/namerica/item \end{cases}$$

$$EX_3 \begin{cases} Q_1 : /site/*/* \\ Q_2 : /site/regions/* \end{cases}$$

その実験結果を表 3 に示す．表 3 は，通信コストを最適化するために生成される，集合間の差にあたる各問い合わせについて，各解のサイズと，集合差演算を用いた場合の計算時間 Opt，および否定を用いた場合の計算時間 $Origin$ を示している．

この結果から，データサイズがそれほど大きくない場合は，なるべく否定を使わず集合差演算を用いる問い合わせに変換する方が効率が良く，データサイズが大きい場合は，逆に否定を用いる問い合わせに変換したほうが効率がよいことがわかった．そこで，クライアント側で問い合わせの解のサイズを予測し，否定を用いるか差演算を用いるかを適宜切り替えることが考えられる．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．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．

XPath	$Origin$(ms)	Opt(ms)
$//parlist$	273453	5125
$//parlist/listitem/parlist$	1028515	14219
$//emph$	8460922	39094
$//open\_auction//descript$	1797922	3078

表 4　再帰的な問い合わせの実験結果

6.2 再帰的な問い合わせによる実験

次に再帰的な問い合わせによる実験結果を表 4 に示す．RDB 上で // を実行する場合，// 直前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全てのエレメントを調べる必要があり，直積演算の負荷が高くなる．しかし，表 4 が示すように，負荷の高い // を展開し，非再帰な問い合わせのみに変換することで計算コストを大きく削減することができる．この実験で用いたデータでは，XML データの最大の深さは 12 であり，この値を使って // の展開を行っている．

7. 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，この節では，サーバ側に手を加えられる場合を想定し，クライアント側の計算コストを軽減する二つの手法について考える．まず，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について説明し，次に，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サーバが解データと共に送信する手法について説明する．

7.1 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は，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しか含まず，計算コストはそれほど高くない．しかし，$CASE_1$ のように，オリジナルの問い合わせの数は三つであったのに，クライアント側で解を取り出す場合，問い合わせの数が六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，サーバ側はまったく通信コストや計算コストの最適化を考えないという前提であったが，本節では，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する．その場合，すべての解集合を単純に $<Ans>\cdots</Ans>$ というタグで囲んで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，解を取り出すのに必要な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，クライアント側でオリジナルの解を取り出すために実行する問い合わせの数を最小にすることができる．

ある XPath の問い合わせ Q を考える．

$Q : /a_1[p_1]/a_2[p_2]/\ldots/a_{n-1}[p_{n-1}]/a_n[p_n]$

ここで，a_i はラベルテスト，$[p_i]$ は述語，n はステップ数である．述語は，あるステップ a_i に現れても現れなくても良い．また，述語中に現れる p_i は，パス式である．この XPath によって得られる解は，a_n エレメントを頂点とする部分木の集合である．これまでは，Ans タグで解集合を囲んで <Ans>(解集合)</Ans> の形でクライアントに送信していた．この解中から，失われる文脈に関する情報は，$/a_1[p_1]/a_2[p_2]/\ldots/a_{n-1}[p_{n-1}]$ である．そこで，文脈情報を付与するために次のような形で解を返せばよい．

<a_1><p_1/>$\cdots$<a_{n-1}><p_{n-1}/>(解集合)</a_{n-1}>$\cdots$</a_1>

ここで <p_i /> は，述語の文脈情報である．p_i はパス式であるため，パスの文脈情報を付与すればよい．例えば，p_i が $a/b/c$ であれば，<p_i /> は，<a><b><c /></b></a> に置換すればよい．p_i が否定の述語であれば，あるステップが子供として p_i にマッチするものを含んでいなければ良い，という意味になるため，<p_i /> を取り除くことが，文脈情報を付与することになる．ラベルテストの文脈 a_i が否定の場合であれば，否定を表す文脈情報を付与すればよい．例えば，a_i が $\overline{\{a,b,c\}}$ であるなら，<a_i> は <$not\_a\_b\_c$> に置換すればよい．また，ラベルテストの文脈 a_i が * である場合，すべてのものにマッチするため，<ANY> で置換する．

次に，Q の各ステップが // である場合を考える．XPath の Query Containment の関係から，解中から失われた // に関する文脈を付与するには，/ で失われた文脈と同じものを付与すれば十分であるため，先ほどと同様に取り扱うことが可能である．

具体的な例として，$CASE_1$ 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 $Q_{1-2}, Q_{2-3}, Q_{2\cap3}$ の解データは，次のような形になる．

<Ans>(c タグに囲まれた Q_{1-2}の解集合)</Ans>
<Ans>(c タグに囲まれた Q_{2-3}の解集合)</Ans>
<Ans>(c タグに囲まれた $Q_{2\cap3}$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，クライアントは六つの問い合わせを行う必要がある．これは，受信した解データには，データベース中には含まれていた文脈に関する情報が失われてしまっているためである．そこで，サーバは，それぞれの解に対して文脈に関する情報を付与した後，この三つの解を適切にマージした，以下のような解集合 W を送信することにする．

<a>
　<not_b> (c タグに囲まれた Q_{1-2} の解集合) </not_b>
　<b> (c タグに囲まれた Q_{2-3} の解集合) </b>
　<b> <e /> (c タグに囲まれた Q_3 の解集合) </b>
</a>

クライアントは，文脈に関する情報が付与され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．

$$\begin{cases} Q_1 \leftarrow (W, /a/*/c) \\ Q_2 \leftarrow (W, /a/b/c) \\ Q_3 \leftarrow (W, /a/b[e]/c/d) \end{cases}$$

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も抑えることができる．

7.2 インデックス情報

本稿で示した枠組みでは，一般的な情報検索システムとは異なり，サーバからは通信コストの上で最適化されたデータが送信されるため，それをそのまま問い合わせの解としては利用できず，クライアント側で本来の解を取り出す処理が必要になる．しかし，この処理は，クライアントの環境によっては大きな負担とになりうる．例えば，処理能力が制限された携帯電話や PDA といった機器では，取り出しに利用するパス式が複雑になると，計算の負荷が高すぎて処理できなくなる可能性がある．

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解の取り出しの計算を行い，その結果をインデックスのような形で，解データと共にクライアントに送信する方法が考えられる．例えば，ある解 Q を Q' から取り出すための情報は，Q' の解からどの部分を取り出すのかを示した，$(256, 128), (1024, 64), (4000, 320), \ldots$ のような，データの先頭からのバイトオフセットの対の集合で表現できる．例えば，この例は，送信されてきた Q' の先頭からみて，256 バイト目から 128 バイトを取り出す，次に 1024 バイト目から 64 バイトを取り出す，...と取り出していったものが，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，クライアントはこの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データを一回の走査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い．

しかし，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．

8. 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最適化の評価実験

前節で提案した，クライアント側での解の取り出し処理を軽減する手法の有効性を検証するため，これを適用した場合と，しない場合との比較実験を行った．

クライアント側での計算コストの評価実験環境は，サーバ側に関する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 [9] の SAX

XPath 問い合わせ集合	実行時間 (ms)			
/site/regions/namerica/item				
/site/regions/europe/item				
/site/regions/*/item/name	従来	文脈	索引	Size(KB)
/site/regions/*/item/descript	4907	3926	2054	21,454
//parlist/listitem/parlist	1562	1602	520	7,143

表 5　実 験 結 果

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走査する間に 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行う．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を行うため，常に一度の走査で問い合わせの解を求めることができる．

実験のマシン環境としては，サーバで使われるような高性能なものではなく，一般にクライアントが用いるような環境として，1CPU（Pentium III 600MHz），256MB のメモリを搭載した Microsoft Windows 2000 上の環境を用いた．

8.1　解集合のデータ加工とインデックス情報による実験

前節で，クライアントの計算コストを軽減する手法として，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，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を提案した．そこで，通信コストの最適化を行って従来通りに解を取り出す場合と，上述の二種類の手法を適用したものから解を取り出す場合の計算コストの比較実験を行った．その結果を表 5 に示す．

文脈情報をサーバが付与する方法は，従来の場合と比べると，計算コストがやや低減している．これは，複数のデータに対して別々に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方が，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ものと考えられる．

従来の場合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．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り，さらに，インデックス情報を送信する手法は最も有効であると言える．

また，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．今回，実験したものでは，文脈に関する情報を解に追加するものでは，解データのサイズに対して 0.0001%未満，インデックス情報を送信するものでは，0.01%未満の増大しか見られなかったため，通信コストの上でも効率は良いといえる．

9. ま と め

本稿では，インターネット上でのデータ交換の標準データ形式となっている XML データを用いたインターネット上の情報サービスにおける，通信コストと計算コストの問題に焦点を当て，サーバに手を加えられない場合と手を加えられる場合を想定し，研究を行った．

サーバに手を加えられない場合については，クライアント側でサイズ最小のビューに対応する問い合わせに変換することによって，通信コストを最適化するアルゴリズムをこれまでに提案している．しかし，この手法は通信コストの最適化のみを考えており，サーバ側での計算コストは増大する場合がある．そこで，この手法で頻繁に発生する計算コストの高い演算，すなわち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を除去する演算を，クライアントがデータの統計的情報を利用して，より計算コストの低い演算に変換する方法を提案した．

次に，サーバに手を加えられる場合を想定して，クライアント側での解の取り出し処理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方法と，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．

文　　献

[1] K. Tajima, Y. Fukui, “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] T. Grust, “Accelerating XPath Location Steps”, In Proc. of ACM SIGMOD, pages 109–120, 2002.

[4] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda, “Monitoring XML Data on the Web”, In Proc. of ACM SIGMOD, pages 437–448, 2001.

[5] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou, “A Transducer-Based XML Query Processor”, In Proc. of VLDB, pages 227–238, 2002.

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

[7] 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, 2002.

[8] “Xalan”, `http://xml.apache.org/xalan-j/index.html`.

[9] “Xerces”, `http://xml.apache.org/xerces2-j/index.html`.