ROM 上の XML データに対するメモリ使用量の少ない検索処理

西川　英毅†　　田島　敬史†

† 北陸先端科学技術大学院大学　〒 923-1292 石川県能美市旭台 1–1
E-mail: †{n-hideki,tajima}@jaist.ac.jp

あらまし　本論文では，DVD-ROM 等の ROM ディスク媒体によって巨大な XML データが与えられ，一方，利用できる書き込み可能記憶の容量はデータのサイズに比べて小さいという環境において，XPath による検索を実現する手法を提案する．従来の，ネットワーク上を流れる XML データに対する検索に関する研究では，データは記憶しない限り一度しか読めないため，オンラインアルゴリズムが必要であったが，上記のような環境ではデータ自体は何度でも読めるため，空間計算量の小さいオフラインアルゴリズムを考える必要がある．しかし，メモリ消費量と ROM へのアクセスの I/O コストはトレードオフの関係にあり，多くの ROM ディスク媒体ではアクセス速度が遅いことを考えると，I/O コストも無視できない．そこで，本論文で提案する手法では，使用可能メモリ量をパラメータとして受け取り，メモリ使用量がその値を超えない範囲内で，I/O を必要最小限に押さえて，与えられた問合わせを評価する処理手法を提案する．

キーワード　XML, 問い合わせ処理, 最適化

Query Processing with a small memory usage for XML data on ROM

Hideki NISHIKAWA† and Keishi TAJIMA†

† Japan Advanced Institute of Sciene and Technology
Asahidai 1–1, Nomi, Ishikawa, 923–1292 Japan
E-mail: †{n-hideki,tajima}@jaist.ac.jp

Abstract　In this paper, we show algorithms for XPath processing in environments where target XML data is huge data on ROM disc media, such as DVD-ROM, but the size of writable memory is relatively small compared with the size of the target data. In existing XML stream processing systems for XML data on network, we need on-line algorithms because we can read the data only once, but in the environments explained above, we can read data as many times as we want, and therefore, what we need is an off-line algorithm with low space complexity. However, there is a trade-off between the amount of memory needed and the amount of I/O to data on ROM. Because the access to ROM discs is usually slow, the cost of I/O is also an important factor for efficient query evaluation. To balance these two factors, we show an algorithm which, given the size of available memory, evaluate XPath queries with using given size of memory and by scanning the data as smaller number of times as possible.

Key words　XML, query processing, optimization

1　は じ め に

現在，地図データや辞書データなど，多くのデータコンテンツが CD-ROM や DVD-ROM の形で流通している．また，近年，汎用データのためのデータ形式やデータ交換用のデータ形式として XML が普及しつつあり，そのため，今後，CD-ROM や DVD-ROM などの ROM 媒体に XML 形式で記録されたデータコンテンツの流通が増大し，それに応じて，このようなデータに対する効率の良い問い合わせ処理の必要性も生じると思われる．一方，現在，カーナビや情報家電など，ROM から読み込むデータは非常に大きいが，それに比べて搭載している RAM は小さいというシステムが存在する．このようなシステムでは，できるだけメモリの消費量が少ない検索処理手法が必要となる．そこで，本研究では，検索言語として XPath [8] を想定し，ROM ディスク媒体上の XML データに対する，メモリ効率のよい XPath 検索処理手法を提案する．

メモリ消費量を考慮する問い合わせ処理の研究としては，XML ストリーム処理の研究がある [2], [3]．近年，ネットワーク上を流れる XML 形式のデータに対するストリーム処理のメモリ消費の効率化を測る研究が数多く行われている．しかし，ストリーム処理の場合，一度読み込んだデータを再び読み込むことはできないため，ある要素が問い合わせの解となるかどうか確定するまでは，候補となる要素の内容と，その要素が条件を満たすかどうかを判定するための計算状態を，一時的にメモリに格納しておかなければならない．そのため，問い合わせによっては，最悪，データを最後まで読み終わるまで，データの全体をメモリに保存しないといけなくなる可能性もある．

一方，扱うデータがストリームではなく ROM 上にある場合

は，同じデータを何度も読み込めるため，ストリーム処理とは状況が異なる．すなわち，ストリーム処理では，使用可能なメモリ量が制限されている場合，オンラインアルゴリズムが必要だが，データが ROM 上にある場合は，単純に空間計算量が少ないオフラインアルゴリズムを考えれば良い．よって，本研究では，空間計算量が少ない XPath の処理手法を開発する．

本研究では，以上のように，XML データの検索時のメモリ消費の効率化を目的としている．しかし，比較的アクセス速度が遅い場合の多い，ROM ディスク上のデータを対象としているため，I/O コストも無視できない．しかし，両者はトレードオフの関係にあり，メモリ消費を削減しようとすると，同じデータを何度も読み込むことになり I/O コストが増え，I/O コストを減らそうとすると，メモリ消費は増えることになる．そこで判断の基準を，「使用可能なメモリ量に応じた処理を行えること」とする．つまり，「**アルゴリズムの入力のパラメータとして使用可能なメモリ量を与え，そのメモリ量のみを使って問い合わせを処理しつつ，その範囲内で，データの入出力量は最低限に押さえる**」ことを目指す．よって，メモリ使用量はデータのサイズには依存しないことになる．

また，アルゴリズムの基本的な動きとしては，ストリーム処理のようにデータを先頭から走査しながら問い合わせを処理し，必要に応じて前に戻って，データの走査を繰り返すという手法を取る．これは，CD-ROM，DVD-ROM 等の ROM メディアにおいては，ランダムアクセスを行うと，先頭からシーケンシャルに読んでいく場合に比べて，アクセスヘッドのシーク時間のために，アクセス時間が長くなる特性があることから，できるだけランダムアクセスを減らすことを狙ったものである．

以下，次章では，関連研究について簡単に説明し，続く章では，この研究で対象とする XPath の部分言語について説明する．また，続く第 4 章で，提案する検索手法について詳しく説明する．また，続く章で，実験，評価について触れ，最後の章はまとめと今後の課題である．

2 関 連 研 究

XPath の計算量に関しては，XPath 式のあるステップまでの中間解の個数 (XPath の last() 関数) や，各ノードのその中間解集合の中の位置 (XPath の position() 関数) に関する演算，および文字列操作を含まない，純粋な木構造のパターンのみを問う部分言語 (Core XPath [4], [5]) に関して，問合わせ式とデータのサイズに対して多項式時間，多項式領域で解けることが示されている [4], [5]．また，上記の制限に加えて，さらに否定を含まない場合 (positive Core XPath [5] あるいは XPath- [6]) には，その計算量は LOGCFL に属することが [5], [6] に示されており，[6] には，問合わせ式の深さに比例した長さを持つ，データ中の要素へのポインタを記憶するスタックを用いたアルゴリズムが示されている．ただし，このアルゴリズムは，ある一つのノードが解となるかどうかを判定する，非決定性アルゴリズムであり，これを単純に決定的に実行すれば，指数時間の計算量になる．

本研究では，positive Core XPath よりさらに制限された XPath の部分言語について，実用的な時間計算量で全ての解を求める具体的なアルゴリズムを開発した．また，単にメモリ消費量を最小化するのではなく，メモリ消費量と I/O コストにはトレードオフの関係があることに注目し，使用可能メモリ量をパラメータとして与えられて，そのメモリ量の範囲内で，最小限の I/O コストで問合わせを処理するアルゴリズムを提案している．

3 XPath

XPath は，ロケーションパスと呼ばれる式によって XML 文書中の特定の要素を指定するための言語である．XPath 式では，文書木の根要素を「/」で表し，以下「/」でパスを区切って要素をたどっていく．「/」で区切った単位をロケーションステップと呼び，ロケーションステップは以下の三つの要素からなる．

- 軸: 選択するノードの，現在見ているノードからの木構造上の相対位置を指定する．
- ノードテスト: 選択するノードの型と名前を指定する．
- 述語: 条件式で，選択するノードをさらに絞り込む．

ロケーションステップはこれらを，「軸::ノードテスト [述語]」の形で表わす．よって，XPath 式全体は以下の形で記述される．

$XPath$ 式 ::= $/locStep/\ldots/locStep$

$locStep$ ::= 軸 :: ノードテスト [述語 (条件式)]

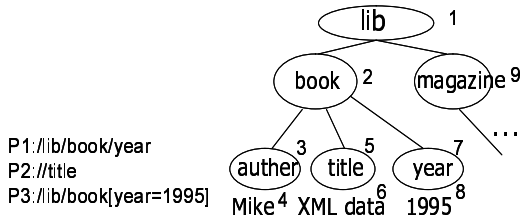

図 1　XML ツリーと XPath 式

また，XPath ではよく使用する軸とノードテストについて簡略化された構文を使うことができ，その一部を図 1 に示す．

ただし，本研究では，対象とする言語として，以下の制限を加えた XPath の部分言語を考える．

- 軸としては，/と // のみを含む．
- ノードテストはタグ名によるテストのみで，ワイルドカードである * は含まない．
- 述語中には，単純な相対パス式のみを許す．入れ子となる述語，論理演算子 (和，積，否定) は許さない．

軸に関しては，/と // のみでも，実用上必要となる問合わせの多くをカバーできると一般的に考えられている．述語に関しては，述語の入れ子を許さない場合，複雑な木パターンは記述できなくなり，表現能力が大きく下がるが，入れ子の述語を含む問い合わせ式への対応については，今後の課題である．論理否定演算子を言語に含むか含まないかは，関連研究の章でも述べたように，問合わせ評価の計算量に大きく影響する問題であり，この点に関する拡張は難しいと思われる．

ここで，XPath を用いた問い合わせについて，例を挙げて説明する．図 1 は XML データを木構造で表現したもので，図中の楕円で囲まれた lib，book 等は各 XML 要素のタグ名，囲まれていない Mike，XML data，1995 はテキストノード，それらの横の 1，2，...は本文中で各ノードを参照するために便宜上割り振った番号である．また，図中の P1，P2，P3 はそれぞれ省略構文を用いて記述した XPath 式である．P1 は，根要素の「子」である「lib」ノードの，その「子」である「book」ノードの，その「子」である「year」ノードが根となる部分木を解とする．つまり，ノード 7 を根とする部分木が解となる．次に，P2 は根要素の「任意の深さの子孫」である「title」ノードを根とする部分木を解とし，この図では，ノード 5 を根とする部分木が解

表 1　軸とノードテストの省略記法

軸とノードテスト	省略構文
self::node()	.
child::	何も書かない
/descendant-or-self::node()/	//

となる．最後に P3 は，根要素の「子」である「lib」の，その「子」である「book」を根とする部分木で，かつ，その「子」の「year」のテキストノードが 1995 であるもの，すなわち，この図では，ノード 2 を根とする部分木が解となる．

4　提 案 手 法

この章では，XPath 式を幾つかのパターンに分けて，簡単な例の処理方式から初めて，最終的にこの論文で対象とする任意の XPath 式に対するアルゴリズムを示す．

4.1　問い合わせ式が「/」だけで構成されている場合

最初に，問合わせ式が「child」軸だけを含む場合，つまり

$$Q = /P_1/P_2/\ldots/P_l$$

という形式で表現できる場合を考える．この場合は

- $stat$: 現在問い合わせ式のどのステップを評価しているか
- $current\text{-}depth$: 現在，読み込んでいるノードの深さ

の 2 つだけを記憶しておくだけでよい．例として図 2 のデータと問い合わせ「Q = /a/b/c」を用いて説明する．図 2 の見方は前出の図 1 と同様である．また，本稿では各ノードの番号と名前の組み合わせを (1，a) や (5，b) のように記述する．

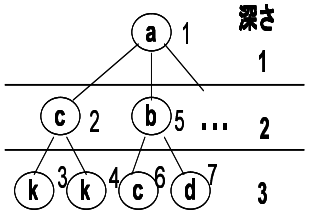

図 2　XML データの例 1

まず，初期状態として $stat$=0，$current\text{-}depth$=0 とする．$stat$=0 は，まだ問い合わせの最初のノードにマッチしていない，つまり P_1 の「a」にマッチするノードを探している段階であることを表す．$current\text{-}depth$ は開始タグを読み込むと 1 増やされ，終了タグを読み込むと 1 減らされる．

次に，(1,a) の開始タグを読み込むと，$current\text{-}depth$=1 となる．今，$stat$ は 0 のため，ノード「a」を P_1 である「a」と比較し，マッチすることがわかる．また，ここで対象とする「/」だけを含む問い合わせの場合，P_i がマッチすべきノードの深さは常に i である．よって，現在，評価しているステップ P_i の i と $current\text{-}depth$ の比較も行う．今の場合，$i = current\text{-}depth = 1$ であるため，このノードは P_1 にマッチすることがわかる．そこで，$stat$ を更新し，$stat$=1 とする．これは，P_1 にマッチするノードが見つかり，P_2 にマッチするノードを探している段階であることを表す．この処理を繰り返し行うと，(5，b) を読み込んだ時点で $stat$=2 となり，(6，c) を読み込んだ時点で $stat$=3 となり，問い合わせの解が発見されたことがわかる．

この手法で，変数 $stat$ と $current\text{-}depth$ に必要なメモリ量は，XPath 式の長さを l，XML データの深さを d として，各々

$\log l$ bit と $\log d$ bit

となる．巨大な XML データは，幅は広くても深さはそれほど深くない場合がほとんどなため，このメモリ量は現実上，問題にならない量と考えられる．

4.2　問い合わせ式が「//」だけで構成されている場合

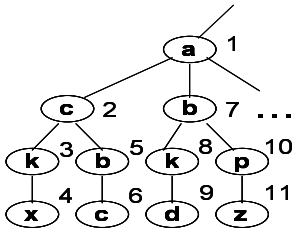

図 3　XML ツリーの例 2

次に，問い合わせ式が「//」だけで構成されている場合について考える．この場合は，

- $counter[i]$: 各ステップ (P_i) のマッチノード数のカウンタ

を用いる．前節の場合と異なり，各ステップにマッチするノードの出現をカウントしていき．カウントしたノードの終了タグが出現するとカウンタを減らす．

例として，図 3 のデータと問い合わせ「Q = //a//b//c」を考える．この場合，カウンタ counter=[0，0，0]，つまり，左から順に「a」,「b」,「c」の出現カウンタを用意する．次に，図 3 のノードを順に読み込むと，まず (1，a) が P_1 にマッチし counter[1]=1 とする．次に (2，c) を読み込むと P_3 にマッチするが，今，counter=[1，0，0] と，P_3 より左にある P_2 のカウンタが 0 であるため，(2，c) はカウントしない．同様に，既にカウントされているタグ名の終了タグが現れた場合は，対応するカウンタを減らし，まだカウントされていないタグ名の終了タグが現れた場合は無視する．対応するカウンタが 0 でない状態の時に現れる終了タグは，必ずそのカウンタでカウントされた要素の終了タグであるはずであることに注意されたい．

この処理を繰り返し行うと，(5，b) で P_2 が，(6，c) で P_3 がカウントされ，問い合わせの解が発見される．

この手法では，必要なメモリは各ステップのカウンタのみであり，XPath 式の長さ (ステップの数) を l，XML データの深さを d とすると，各カウンタに $\log d$ bit 必要で，総メモリ量は

$$l \log d \text{ bit}$$

となり，これも，現実上，問題にならない量と考えられる．

4.3　問い合わせ式が「/」と「//」を両方含んでいる場合

次に，問い合わせ式が「/」と「//」を両方含んでいる場合について考える．この場合は，以下の変数を用意すればよい．

- $counter[i]$: 各ステップ (P_i) のマッチノード数のカウンタ
- $depth[i]$: カウントした時の深さを保持
- $current\text{-}depth$: 現在，読み込んでいるノードの深さ

例として，図 4 のデータと問い合わせ「Q =//a/b//c」を考える．この場合，前節と同様，counter=[0，0，0] を用意する．また，XPath 式中の二番目のステップ P_2(すなわち「b」) にマッチするノードをカウントする際,「b」は「a」の子でなければならない．よって式中の一番目のステップ P_1(すなわち「a」) にマッチするノードをカウントする際は，その時の $current\text{-}depth$ を $depth[1]$ に保持する．この深さ保持が必要なのは，後ろに「/」が続くステップのみであり，この例では P_1 だけである．

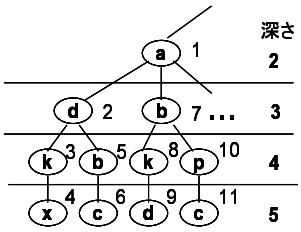

図 4　XML ツリーの例 3

図 4 のノードを順に読み込むと，まず (1，a) が P_1 にマッチし，counter[1]=1 とする．この際，$current\text{-}depth$=2 を一時的に保持する．さらに読み込んでいくと，次に (5，b) の「b」が P_2 とマッチするが，先ほどカウントした P_1 の保持した深さ 2 と $current\text{-}depth$=4 を比較し，この (5，b) は P_1 の「子」でないことが分かり，(5，b) はカウントしない．この処理を繰り返すと，(7，b) で P_2 が，(11，c) で P_3 がカウントされ，解が発見される．この手法では，消費メモリは各ステップのカウンタと，現在サーチしているエレメントの深さを保持する変数，深さ保持の必要なステップの深さを一時的に保持する変数だけであるため，XPath 式の長さを l，XML データの深さを d とすると，必要な総メモリ量は

$$l \log d + (l-1) \log d + \log d = 2l \log d \text{ bit}$$

となり，この場合も，現実上は問題にならない量と考えられる．

また，この場合の処理の，開始タグを読み込んだ際のアルゴリズムをまとめると表 2 のようになる．一方で，読み込んだノードが終了タグである場合は，開始タグを読み込んだ場合と逆の動作を行う．このアルゴリズムをまとめると表 3 のようになる．また，このアルゴリズムで使用する変数，関数は開始タグを読み込んだ場合と同じである．

4.4　問い合わせ式が述語を含む場合

次に，問い合わせ式が述語を含む場合について考える．この場合，ある要素を走査する時点では，その要素が解になるかどうか確定しない場合があるため，一旦，解候補として記憶する必要がある．しかし，解候補の数が増えると使用可能なメモリ量を超える可能性がある．この問題に対応する処理方針の概略について，まず説明する．

4.4.1　解候補オフセット，解オフセットの保存

問合わせが述語を含む場合，まず，変数としてさらに

- $candidateOffset$: 解候補のオフセットを保持する
- $answerOffset$: 解オフセットを保持する
- m: 使用可能メモリ量を表すパラメータ

を使用する．述語以外の部分の処理はこれまでと同様である．問い合わせ式の述語を除く部分が満たされた時は，ファイル中の現在位置のオフセットを解候補として保持する．そして，その後，述語の条件式が満たされて，解候補として保持しておいた要素が解となることがわかった場合，保持したオフセットを元に，データを取り出す．例として，図 4 のデータと問い合わせ「Q = //a/d[.//c]」を使ってこの動作の概略について説明する．この場合，まず問い合わせ式を「//a/d//c」に変換し，これまでに説明した手法 (5.1〜5.3 を用いて) で評価していく．この問い合わせはまず，(1，a) にマッチし，次に (2，d) にマッチする．取り出す解はこの (2，d) を根とする部分木であるため，ここで (2，d) のオフセットを解候補オフセット ($candidateOffset$) として保持しておく．そして，述語の中の条件式に対応する P_3 を評価するためデータを走査していき，P_3 は (6，c) でマッチする．

この時点で問い合わせの解が発見されたが，解が発見されるたびに，発見された解のオフセットの位置まで戻り解を出力すると，ROM のシーク時間のためにアクセス時間が長くなってしまうことが考えられる．そこで，すぐに解となる部分木を取り出さず，この解のオフセットを，先ほど保持した解候補オフセット ($candidateOffset$) から取り出し，それを解オフセット ($answerOffset$) として保持し直す．そして，データの走査を続け次の解を探していく．

保持する解候補オフセットや解オフセットの数は，現在のメモリ使用量がパラメータ m を超えない範囲で保持を続ける．逆にメモリ使用量が m を超える可能性が生じたら，データの走査を中断し，保持した解オフセットをもとに，これまでに発見した解を取り出し出力し，$answerOffset$ をクリアして，データの走査を中断した場所から再開する．

あるいは，$answerOffset$ をクリアしても，$candidateOffset$ のみでメモリ使用量を超えてしまう場合もありうる．このよ

表 2　XPath 処理のアルゴリズム (”/”と”//”混合)

```
Algorithm slaAndSlaSlaStart
function
check-counter() : counter を調べ，未カウントの最初の
          インデックスを返し，なければ'-1' を返す関数
check-hold(Pi) : Pi の一つ前のロケーションステップが
          深さ保持をしているかどうかを調べる．保持していれば
          最も深い値 (深さ) を返し，していなければ'-1' を返す
slaAndSlaSlaStart(node)
Input: node
1   current-depth++.
2   j ← check-counter().
3   if j = -1 do
4      for i ← 最後のインデックス to 0 do
5         if Pi = node & depth ← check-hold(Pi) &
6              depth + 1 = cuurent-depth do
7            counter[i]++.
8            if i = 最後のインデックス do
9               another-match ← ture.
10           end if
11           if Pi が深さ保持の必要なステップ do
12              Pi の深さ保持.
13           end if
14        goto end.
15        end if
16        else if Pi = node & check-hold(Pi) = -1 do
17           7〜14 と同様
18        end else if
19        else do
20           i−−.
21        end else
22     end for
23  end if
24  else do
25     for i ← j to 0 do
26        7〜21 と同様
27        #9 match ← ture.
28     end for
29  end else
Output: match，another-match
```

表 3　XPath 処理のアルゴリズム (”/”と”//”混合 & 終了タグ)

```
Algorithm slaAndSlaSlaEnd
slaAndSlaSlaEnd(node)
Input: node
1   current-depth − −.
2   j ← check-counter().
3   if j = -1 do
4      for i ← 最後のインデックス to 0 do
5         if Pi = node & depth ← check-hold(Pi) &
6               depth - 1 = cuurent-depth do
7            counter[i]− −.
8            if i = 最後のインデックス &
9                  counter[i] = 0 do
10                 match ← false.
11           end if
12           else if i = 最後のインデックス do
13              another-match ← false.
14           end else if
15           else do
16              .
17           end else
18           if Pi が深さ保持の必要なステップ do
19              Pi の深さ解放.
20           end if
21           goto end.
22        end if
23        else if Pi = node & check-hold(Pi) = -1 do
24           7〜21 と同様.
25        end else if
26        else do
27           i− −.
28        end else
29     end for
30  end if
31  else do
32     for i ← j to 0 do
33        if Pi = node & depth ← check-hold(Pi) &
34              depth - 1 = current-depth do
35           counter[i]− −.
36           18〜21 と同様
37        else if Pi = node & check-hold(Pi) = -1 do
38           counter[i]− −.
39           18〜21 と同様
40        end else if
41        else do
42           i− −.
43        end else
44     end for
45  end else
Output: match，another-match
```

うな場合には，解候補を保持する度にメモリ使用量をチェックし，設定したパラメータ m を超えそうになれば，その時点での走査位置を記憶して一旦走査を中断し，これ以降は新たな $candidateOffset$ の探索は行わず，これまでに発見した $candidateOffset$ が解となるかどうかの検査のみを行いながら，データの走査を続け，データの終わりまで来たら，$candidateOffset$ をクリアして，先ほど記憶した位置から走査を再開する．例として図 5 と問い合わせ

Q = //a[b]//c//d

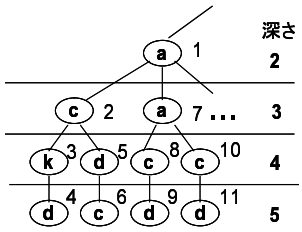

図 5　XML ツリーの例 5

を使用する．まず，前項で述べたように，この問い合わせを

$$Path = \begin{cases} //a//c//d \\ //a/b \end{cases}$$

と考える．そして，各ノードに対してカウンタを用い，これまでと同様に処理をしていく．すると，この図からこの問い合わせに対して解候補になるのは (4,d)(5,d)(9,d)(11,d) である．よって，$candidateOffset$=(4,d)(5,d)(9,d)(11,d) となり，さらに走査は続いていく．その後走査していく上で，(1,a) の子である「b」が出現すると，$candidateOffset$ は $answerOffset$ へと移動し，いつでも解として解放できる状態になるが，このまま (1,a) の子である「b」がいつまでも出現せず，(1,a) の終了タグも出現せず，$candidateOffset$ が増え続けると，メモリ使用量が初期に設定したパラメータ m を超える可能性がある．そこで，この場合は，この時点での走査位置を記録しておいて，(1,a) の子である「b」を (1,a) の終了タグが出現するまで，サーチする．仮に，(1,a) の子である「b」が出現すれば，これまで $candidateOffset$ として保持してきたオフセットを $answerOffset$ へと移動し，順次解放していく．逆に (1,a) の子である「b」が出現しなければ，$candidateOffset$ の値を順次クリアしていく．そして，それらの処理が終了すると，記録しておいた，中断時点の走査位置からデータの走査を再開する．

このような処理によって，あらかじめ上限を設けたメモリ使用量を超えない範囲で，I/O コストを最小限に押さえながら処理を行うことができる．以上が，処理の概略であるが，以下，その詳細について説明する．

4.4.2　述語のマッチング情報の保存

上の例では，述語でない部分の葉が先にマッチする場合を説明したが，実際には 述語の葉の部分が先にマッチする場合もあるため，その場合，どの 述語の葉が既にマッチしたかを記憶する必要がある．この処理は，問い合わせ式中のあるロケーションステップにマッチするノードがネストする (入れ子になる) 場合には少し複雑になる．例として図 6 と問い合わせ

Q = //a[.//b]/c」

使用する．図 6 を見ると，(1,a) と (5,a) および (7,a) の 2 つの「a」ノードがネストしていることがわかる．そして問い合わせ式の P_1=//a であることから，ネストしたノードをカウントすることになる．この事を前提とした上で，上記までの手法を用いてこの問い合わせを処理していくと，まず，この問い合わせを

$$Path = \begin{cases} //a/c \\ //a//b \end{cases}$$

と考える．そしてカウンタ [a,c]=[0,0] と [b]=[0] を用意する．ここで [a] のカウンタを 2 つ用意する必要はないので [b] では省略する．これらを用意し，順にデータを走査していくと，ま

ず (1,a) でマッチしカウンタ [a,c]=[1,0] となる．次に，(2,b) でマッチしカウンタ [b]=[1] となる．この時点で，(1,a) はステップ a[.//b] 中の 述語 [.//b] を満たすことが確定したので，これを (2,b) の終了タグを読み込んだ以降も，記憶する必要がある．この情報をどのように記憶するかは，一旦，置いておく．その後，(5,a)，(6,c) でマッチ ((6,c) は深さ保持した (5,a) の子であるため) し，カウンタは [a,c]=[2,1]，[b]=[1] となり，カウンタだけで判断すると，問い合わせが満たされているように考えられる．しかし，実際には (2,b) は (1,a) の子であり，同じくカウントした (5,a) の子でないため，この問い合わせは満たされていない．つまり，あるステップにマッチするノードがネストするような場合，カウンタが満たされていることが，必ずしも問い合わせが満たされたことと断定することができない．よって，このような場合には，[.//b] が満たされたのは，(1,a) の a に対してであることを明示的に記憶しておく必要があり，次の変数を用意する．

- $checkNodePath$: 述語中の葉にマッチするノードが，その述語を持つステップをどの深さのノードにマッチさせているか

この変数は述語中に葉として現れているステップの数分だけ用意し，述語の中の葉となる各ステップにマッチするノードをカウントする際に，問い合わせ式中でその述語を持っているステップにマッチしているノードの深さを保持する変数である．もう一度先ほどの例を用いて説明すると，まず，(1,a) でマッチしカウンタ [a,c]=[1,0] となる．次に，(2,b) でマッチし，カウンタ [b]=[1] となるが，この時にカウントしたこの「b」は深さが”2”(より正確には，2 以下) の「a」祖先ノードを//a[.//b] というステップにマッチさせる物としてカウントされているのだということを覚えておく必要がある．ここで先ほど用意した変数 $checkNodePath$ を用い，$checkNodePath$[1]=2 とし ($checkNodePath$[i] の配列のインデックス i は，述語の中の各ステップの順番に一致する．上記の Path では i =「b」のインデックス = 1 となる．)，カウントした「a」ノードの深さを保持しておく．また，この処理を可能にするため，述語を持つステップ (上の例では「a」) に対しては，(述語内または述語外でそれに続く軸として / がある場合は，これまで通りだが，そうでない場合にも必ず) 変数 depth を用意し，このステップに最後にマッチしたノードの深さを保持する．これによって，(6,c) をカウントした場合にカウンタ [a,c]=[2,1]，カウンタ [b]=[1] となり，(5,a) の子である (6,c) を取り出したいが，カウンタは満たされるが，$checkNodePath$[1]=2 を参照することで，カウントされた「b」ノードは深さ”4” の「a」ノード ((5,a)) の子ではなく，深さ”2” の「a」ノード ((1,a)) の子であることがわかり，カウントが満たされていても述語にマッチしないため，この問い合わせは満たされていないことがわかる．それとは別に，(1,a)，(7,a) にマッチし，カウンタ [a,c]=[2,0] となり，(8,b) を走査した後に，カウンタ [b]=[1] とすると同時に，$checkNodePath$[1]=3 と保持しておくことで，(10,c) をカウントしたときに，[a,c]=[2,1] となり，先ほど保持しておいた $checkNodePath$ の値”3” を参照することにより，カウントした述語 の「b」ノードは深さ”3” の「a」ノードの子であり，その「a」ノードの子である「c」ノードがいまカウントされたため，この問い合わせが満たされたことがわかる．

また，(8,b) の開始タグを走査した時点で，$checkNodePath$[1]=3 として「a」ノードの深さ保持し，その後 (8,b) の終了タグを走査した際，「b」のカウンタは [b]=[0] と解放されるが，

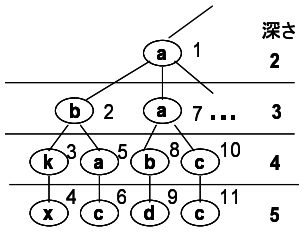

図 6　XML ツリーの例 4

$checkNodePath$[1]=3 は保持を続ける．これはまだこの問い合わせが満たされる可能性があるため，深さ”3” の「a」ノードの子孫に「b」ノードが出現したという情報を覚えておかなければならないからである．そして，(10,c) を走査すると保持した深さを元に，問い合わせ式が満たされる．その後も順に走査し，(7,a) の終了タグを読み込むと通常であれば $checkNodePath$[1]=3 を解放するのだが (問い合わせ式中でこの述語の葉を持っているステップの上位のステップは述語を含んでいないため)，「a」ノードがネストしており，また問い合わせ式の//a[.//b] という，「a」の子孫である「b」という関係から，(1,a) に対しても (8,b) は満たされているといえる．この場合は (7,a) の終了タグを走査した時点で，保持していた $checkNodePath$[1]=3 の値を $checkNodePath$[1]=2 と減らし (ここでは深さ 2 以下の「a」ノードに対してマッチしているという意味)，走査を続けていく．そうすることにより，今後 (1,a) の子に「c」ノードが出現しても，保持しておいた深さを元に問い合わせ式が満たされたと判断できる．

この値保持もオフセット保持と同様で，現在のメモリ使用量がパラメータ m を超えない範囲で保持を続ける．

4.4.3　解候補のマッチング情報の保存

述語の葉に関するマッチング情報と同様，述語外の葉にマッチする要素，すなわち解候補となる要素のマッチング情報についても保存する必要がある．例として図 7 と問い合わせ

Q = //a[b]//c//d

を使用する．まず，前項で述べたように，この問い合わせを

$$Path = \begin{cases} //a//c//d \\ //a/b \end{cases}$$

と考える．そして，各ノードに対してカウンタ等を用い，これまでと同様に処理をしていく．すると，この図からこの問い合わせに対して解候補になるのは (4,d) と (9,d) である．よって，$candidateOffset$=(4,d), (9,d) となり，次に (10,b) が出現する．この (10,b) は $candidateOffset$ の (9,d) と対応しているが (4,d) とは対応していない．よって，(9,d) のみを $answerOffset$ に移動したいのだが，この対応付けはカウンタや深さ保持，$checkNodePath$ といった変数だけでは判断できない．なぜなら，一度カウントしたノードや深さ保持といったものは，これまで述べた手法を用いると，終了タグが出現すると解放するからである．よって，$candidateOffset$ と条件式の対応付けを記録する変数として，

- $solutionCorrespondence$: 保持したオフセットがどの述語にマッチするか

を用いる．

問い合わせの XPath 式に述語が含まれている場合，問い合わせ式を木構造で表現すると，述語の数だけ分岐ができ

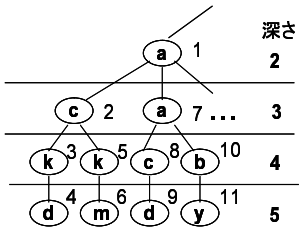

図 7　XML ツリーの例

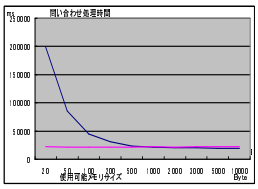

図 8　使用可能なメモリサイズと問い合わせ処理時間

る．この分岐点をもとに，$candidateOffset$ と条件式の対応付けを行う．具体的には，保持した解候補オフセットは，問い合わせを満たしたノード (上記では d) から問い合わせ式上を上に遡って，最も近い分岐点となるステップにマッチしたノードの深さを保持する．つまり，この問い合わせでは $candidateOffset$=(4,d), (9,d) はそれぞれ，(1,a)，(7,a) の深さ (2 と 3) を $solutionCorrespondence$=(4,d,2), (9,d,3) として保持しておく．この保持した値をもとに，$candidateOffset$ と述語を対応付けることによって，どの述語によりどの解候補が満たされるかということが，判断できると考えられる．

また，この値保持もオフセット保持と同様で，現在のメモリ使用量がパラメータ m を超えない範囲で保持を続ける．

また，これらのアルゴリズムをまとめると表 4 のようになる．

5　実験，評価，考察

この章では，4 章で提案した各アルゴリズムを用いた実験とその評価を考察する．実験環境として，OS：WindowsXP，CPU：Cerlon1.4GHz，Memory：256MB，XML データは DBLP の文献データで，記憶媒体は CD-ROM である．

図 8 にその様子を示す．グラフの縦軸が問い合わせの処理時間で，横軸が使用可能メモリサイズである．そして，述語を含む 2 種類の検索式を基に，使用可能メモリサイズを変化させた際の問い合わせ処理時間を計測した．まず，図 8 の左の問い合わせは，常に解の前に述語が出現するパターンである．つまり，解オフセットを保持しなくてよいため，使用可能メモリサイズを変化させても，常に一定の時間で問い合わせ処理が行えていることがわかる．一方で，図 8 の右の問い合わせは常に解の後に述語が出現するパターンである．つまり，使用可能メモリサイズを超えない範囲で解候補オフセット，解オフセットの保持を続けていく．この場合においては，使用可能メモリサイズを増やせば，ある一定を超えた所で，常に一定の時間で問い合わせ処理が行えているが，逆に使用可能メモリサイズを減らしていくと，問い合わせ処理時間が大幅に増加していることがわかる．

カーナビや情報家電といった機器のメモリサイズが今後も増加する傾向にあると思われることと，この実験結果だけから考えると，メモリ消費量よりも，I/O コストの制約の方が優先すべきファクターになるのではないかとも考えられる．しかし，今回用いた問合わせ式は，確かに使用可能メモリサイズを超えない範囲で解オフセットを保持していく必要が生じる問合わせ式ではあるが，述語がマッチする位置と解がマッチする位置が比較的近いデータであるため，メモリ使用量の制約がそれほど問題にならない問い合わせである．よって，メモリ消費量と I/O コストのどちらがどの程度，問合わせ処理全体の効率の支配的要因になるかについては，もっと様々な実験データと，様々な問い合わせパターンを試し，その変化を計測し，再び考察する必要があると考えられる．

表 4　XPath 処理のアルゴリズム (述語を含む & Path の処理)

Algorithm includePredicate
$type : 述語を除くシンプルパスのタイプ ”0”:パスに”/”と”//”の両方を含んでいる ”1”:パスは”//”だけで構成されている ”2”:パスは”/”だけで構成されている
$cuurent-offset : 現在のオフセット
$predicate[i] : 述語をステップごとに区切る
$dep-correspondence[i] : 最も近い分岐点の深さを保持
function
memory-check(m) : 使用メモリがパラメータ m を超えていないか 超えていなければ true を返す

6　まとめと今後の課題

本稿では，ROM 上の XML データに対するメモリ使用量の少ない検索処理アルゴリズムについて提案した．また，提案した処理方式をもとに実験を行い，問合わせのサイズとデータの深さが定数オーダ内で制限されている場合においては，実際に定数メモリで処理されていることを確認し，メモリ使用量と I/O コストの支配的要因についての考察を行った．今後はさらなる評価実験を行い，メモリ使用量と I/O コストのトレードオフのバランスの取り方について，さらに検討を行いたい．また，完全な XPath をサポートするアルゴリズムへの拡張も，今後の重要な課題である．

文　献

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, editors. *Extensible Markup Language (XML) 1.0 (Second Edition) – W3C Recommendation*. `http://www.w3.org/TR/2000/REC-xml-20001006`, Oct. 2000.

[2] Ashish Kumar Gupta and Dan Suciu. Stream Processing of XPath Queries with Predicates. In *Proc. of ACM SIGMOD*, pp. 419–430, May 2003.

[3] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD*, pp. 431–442, 2003.

[4] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, pp. 95–106, 2002.

[5] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The Complexity of XPath Query Evaluation. In *PODS*, pp. 179–190, 2003.

[6] Luc Segoufin. Typing and Querying XML Documents: Some Complexity Bounds. In *PODS*, pp. 167–178, 2003.

[7] Simple API for XML. `http://www.saxproject.org/`

[8] James Clark and Steve DeRose, eds. *XML Path Language (XPath) Version 1.0 – W3C Recommendation*. `http://www.w3.org/TR/xpath` Nov. 1999.

[9] James Clark, eds. *XSL Transformations (XSLT) Version 1.0 – W3C Recommendation*. `http://www.w3.org/TR/xslt` Nov. 1999.

[10] Ron Daniel Jr., Steve DeRose, and Eve Maler, eds. *XML Pointer Language (XPointer) Version 1.0 – W3C Candidate Recommendation*. `http://www.w3.org/TR/WD-xptr` Jun. 2000.

```
includePredicate(node)
Input: node
1   current-depth++.
2   if type = 0 do
3       goto 12.
4   end if
5   else if type = 1 do
6       goto 68.
7   end else
8   else (type = 2) do
9       goto 88.
10   end else
11
12   j ← check-counter().
13   if j = -1 do
14       for i ← 最後のインデックス to 0 do
15           if Pi = node & depth ← check-hold(Pi) &
16                depth + 1 = cuurent-depth do
17               counter[i]++.
18               if i = 最後のインデックス do
19                   if memory-check(m) do
20                       candidateOffset ← cuurent-offset.
21                   end if
22                   else do
23                       if answerOffset に解がある do
24                           解を取り出していく.
25                           goto 20.
26                       end if
27                       else do
28                           while candidateOffset の値がある do
29                               nomSolOffset と solutionCorrespondence
30                               分岐ノードの深さを対応付けて解をサーチする.
31                               if candidateOffset が解であった do
32                                   解を取り出していく.
33                               end if
34                               else candidateOffset が解でなかった do
35                                   candidateOffset の値を削除.
36                               end else
37                           end while
38                           goto 20.
39                       end else
40                   end else
41               end if
42               if Pi が深さ保持の必要なステップ l
43                    Pi のステップに述語が含まれる do
44                   Pi の深さ保持.
45               end if
46               if Pi の上位に分岐ノードがある do
47                   dep-correspondence[i] ← current-depth.
48               end if
49               goto 102.
50           end if
51           else if Pi = node & check-hold(Pi) = -1 do
52               17〜49 と同様.
53           end else if
54           else do
55               i–.
56           end else
57       end for
58   end if
59   else do
60       for i ← j to 0 do
61           15〜56 と同様.
62       end for
63   end else
64   goto 102.
65
66   j ← check-counter().
67   if j = -1 do
68       for i ← 最後のインデックス to 0 do
69           if Pi = node do
70               counter[i]++.
71               18〜49 と同様.
72           end if
73           i−−.
74       end for
75   end if
76   else do
77       for i ← j to 0 do
78           if Pi = node do
79               counter[i]++.
80                   if check-counter() = -1 do
81                       19〜49 と同様.
82                   end if
83           end if
84           i–.
85       end for
86   end else
87
88   if stat が満たされている do
89       goto 102.
90   end if
91       else if stat が示す Pi = node &
92            stat が示す Pi の深さ = current-depth do
93       stat++.
94       if stat が満たされる do
95           19〜49 と同様.
96       end if
97   end else if
98   else do
99       goto 102.
100   end else
101
102   for i = 最後のインデックス to 0 do
103       if predicate[i] = node &
104            このステップがカウントされている (path) do
105           predicate[i]++.
106           19〜42 と同様.
107           dep-correspondence[i] ← このステップの
108                ノードの深さ (最も近い分岐点)
109           while nomSolOffset is not empty do
110               それぞれ同じステップの dep-corresondence[i] の値をチェック.
111                   if 全てのステップにおいて一致 do
112                       solOffset ← 一致した nomSolOffset.
113                   end if
114           end while
115       end if
116   end for
Output: solution(解)
```