XML 文書分散管理システムにおける position 関数の実装と評価

平山 雅之† 樋口 健† 都司 達夫†

†福井大学工学研究科 〒910-8507 福井県福井市文京 3 丁目 9 番 1 号

E-mail: †{hirayama, tsuji, higuchi}@pear.fuis.u-fukui.ac.jp

あらまし XML は, 構造化された様々なデータの標準的な記述ファーマットの一つとして広く普及しており, XML 文書を高速に検索するための様々な手法やシステムが開発されている. 本稿で扱う, XML 文書分散管理システムは, XML における要素の親子関係を索引化し，非共有メモリ型 並列計算機に分散配置して管理するシステムである.

XML 文書には重要な概念の一つとして文書の記述順があり，XPath 式では述語の条件式である position 関数が文書順に対しての指定方法となる．文書順に対する処理を行うためには，position 関数直前までのシーケンスを一時的に集計してソートする必要があり，並列処理環境では特にその処理方法が課題となる．そこで本稿では，position 関数を効率的に処理するための３つの手法を提案する．また，それぞれ提案手法を用いた検索時間を評価する．

キーワード XML，XPath，並列処理

Masayuki HIRAYAMA† Ken HIGUCHI† Tatsuo TUJI†

†Graduate School of Engineering, University of Fukui, 3-9-1, Bunkyo, Fukui, 910-8507 Japan

E-mail: †{hirayama, tsuji, higuchi}@pear.fuis.u-fukui.ac.jp

1. はじめに

近年, XML[1]は構造化された様々なデータを柔軟に表現可能なため，標準的な記述フォーマットとして多くの分野で広く普及している. XML はタグを用いてユーザが独自にデータ構造を定義するこが可能であり，またそのデータはテキスト形式であるため，あらゆる計算機の環境において汎用的に取り扱うことができる. それに伴い，大規模な XML 文書を高速に検索するために，さまざまなシステムが開発されている. XMLDB としては，eXist-db[2]および NeoCore[3]などがよく知られている.

索引方式の１つとして，XPath[4]など経路情報を検索条件に含むような検索において，マルチインデックス手法に相当する XML 文書中の要素間の親子関係に対する索引付けがある.

本稿で扱う XML 文書分散管理システムは，索引として子ノードから親ノードへの参照を保持する PBT(Parent B+Tree)と親ノードから子ノードへの参照を保持する CBT(Child B+Tree), 属性を持つノードから属性ノードへの参照を保持する ABT(Attribute B+Tree)を用いる. これらの索引を非共有メモリ型 並列計算機に分散配置して管理し，並列処理を行うことで，大規模で動的な XML 文書に対して高速な検索の実現を目指している.

XML 文書には重要な概念の一つとして文書の記述順があり，XML 文書を検索するシステムでは，その検索結果の文書順を保証する必要がある．文書順に対する問い合わせとして，XPath 式では述語の条件式である position 関数での指定方法がある. position 関数の処理で文書順に対する処理を行うためには，position 関数直前までのシーケンスを一時的に集計して文書順にソートしなければならない．しかし，並列処理環境では，異なるプロセス間の検索結果が必要になるため，その処理方法が課題となる．そこで本稿では，並列処理環境である XML 文書分散管理システムにおいて効率的に position 関数を処理するための，３つの手法を提案する.

本稿の構成を以下に示す. まず２.節において本稿で扱う XML 文書分散管理システムについて述べる．次に３.節で，提案する３つの position 関数処理方法を説明する．４.節では，提案手法の比較実験とその考察について述べる. 最後に５.節で本稿のまとめと今後の課題について述べる.

2. XML 文書分散管理システム

2.1. 概要

XML 文書分散管理システムは，XML データを３種に索引化し，その索引をハッシュ法によって分割して複数の端末で管理するシステムである．索引を分散管理することで，並列処理で検索を行うとともに，データの更新時には局所的な変更に抑える．そのため，更新が頻繁に起きる動的な XML 文書，更新コストが大きいパスインデックスが使用できない大規模な XML 文書などに対し，有効なシステムである．

並列環境としては，メッセージ通信を用いた非共有メモリ計算機で，各端末は１プロセスの逐次処理である．

端末は PE(Processor Element)と呼び，PE はそれぞれ HOST，RPE，DETECTOR として役割を割り振って運用する(図 1).

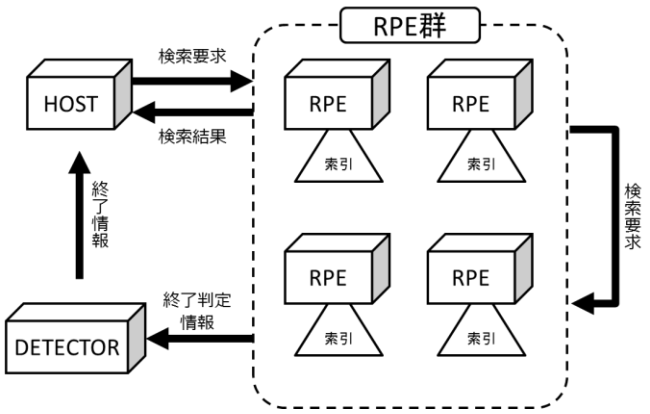

図 1　XML 文書分散管理システム

2.2. 索引

XML 文書分散管理システムでは親子関係に対する３種類の索引を用いて検索を行う．

索引のキー値には EID (Element ID)を用いる. EID とは XML 文書中のノードに対して割り当てられる識別子であり，TID (Tag ID：タグの種類に対して割り当てられる識別子) と，IID (Instance ID：タグの個数に対して割り当てられる識別子)から構成される．

索引のデータ値には，EID と NO (兄弟要素間に割り振る文書順の順序を表す値)を格納する．

３種類の索引について次に示す．

- PBT (Parent B+Tree)

  子ノードから親ノードへの参照を保持する索引である．指定したノードの EID に対しての親ノードの EID を得る．
- CBT (Child B+Tree)

  親ノードから子ノードへの参照を保持する索引である．指定したノードの EID に対しての子ノードの EID を得る．
- ABT (Attribute B+Tree)

  属性を持つノードから属性ノードへの参照を保持する索引．指定したノードの EID に対しての属性ノードの EID を得る．

2.3. PE (Processor Element)

2.3.1. HOST

HOST は，ユーザからのクエリを受け付けて RPE に検索要求を発行し，最終的な検索結果の集計・表示を行うといったユーザインターフェースに相当する PE である．以下に主な処理内容を示す．

- ユーザからのクエリを受け取り, 新規に RID (要求識別子)を割り当て RPE へ検索要求を発行する．同時に DETECTOR へ新規 RID の検索処理開始を通知する．
- RPE からの検索結果を受信し，集計を行う．
- DETECTOR から検索処理の終了通知を受信し，検索結果の出力を行う．

2.3.2. RPE (Retrieval Processor Element)

RPE は，分割した各索引（CBT，PBT，ABT）を管理し，実際に検索を行う PE である．以下に主な処理内容を示す．

- 検索要求を受信し，管理する索引を用いて検索処理を行って検索結果を得る．
- 得られた検索結果が最終検索結果になる場合は HOST へ，最終検索結果でない場合は新たな検索要求を対応する次の RPE へ発行する．
- 検索処理の進行情報を更新し，終了判定情報として DETECTOR へ送信する．

2.3.3. DETECTOR

DETECTOR は，RID ごとに検索処理の終了判定を行う PE である．XML 文書分散管理システムは，各 PE が並列に検索処理を行うため，その検索結果を集計するにあたってすべての検索要求の処理完了を判断する必要がある．DETECTOR は，すべて検索要求の発行数と処理完了数の情報を集計することで，この判定を行う．以下に主な処理内容を示す．

- HOST から新規 RID の検索処理開始の通知を受信し, RID にごとに終了判定マップを作成する．
- RPE から終了判定情報を受信し，RID に対応した終了判定マップの更新と終了判定を行う．
- 検索処理の終了が確定した RID を検出した場合, HOST に対し, 検索処理の終了を通知する．

2.4. 文書順の復元方法

2.4.1. ラベリング

ラベリングには様々な手法[5]が提案されているが，XML 文書分散管理システムでは，XML ツリー上でノード同士の関係にナンバリングを行う相対ラベリングを採用し，エッジに対しラベル付けを行っている(図 2)．このラベルの値は 2.2.節で説明する NO として索引のデータ値に格納する．

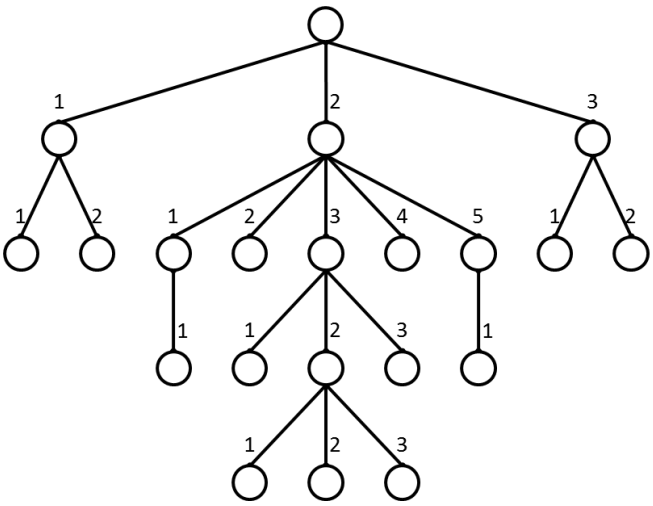

図 2　相対ラベリング

各ノードに文書順にナンバリングを行う絶対ラベリングに対し，相対ラベリングは，XML ツリーにおいてノードの追加や削除といった更新処理に強いラベリング手法である．

相対ラベリングでは，更新処理時にラベルの書き換えが必要となるノードが，更新ノードに対して後方の兄弟ノードのみである．一方で絶対ラベリングでは，更新処理時には更新ノードの後方ノード全てのラベルの書き換えが必要になる．

XML 文書分散管理システムでは，索引を分割して管理するため，低コストで更新処理を行うことができる相対ラベリングを採用している．

ただし文書順の並べ替えに関しては，絶対ラベリングがラベルの値に従って文書順を判定可能であるのに対し，相対ラベリングでは NO の情報単独で文書順を判定できない．そのため次に述べる NO 履歴を生成して文書順の復元を行う．

2.4.2. NO 履歴

NO 履歴は，あるノードから別のノードへの経路情報を，NO を用いて表した数字列である．XML 文書分散管理システムでは，RPE において検索の進行とともに得られるノードの NO を用いて NO 履歴を形成していく．NO 履歴は次のルールで形成する．

- NO 履歴の初期値は NULL を割り当てる
- 親方向へ進む場合は，NO を負の値で追加する
- 子方向へ進む場合は，NO を正の値で追加する
- 常に最短経路を表現するため，正の値と負の値が並ぶとき，その絶対値が等しければそれらを削除する

以上のルールで検索とともに XML ツリー上をたどると，すべてのノードにおいて NO 履歴を生成することが可能である(図 3).

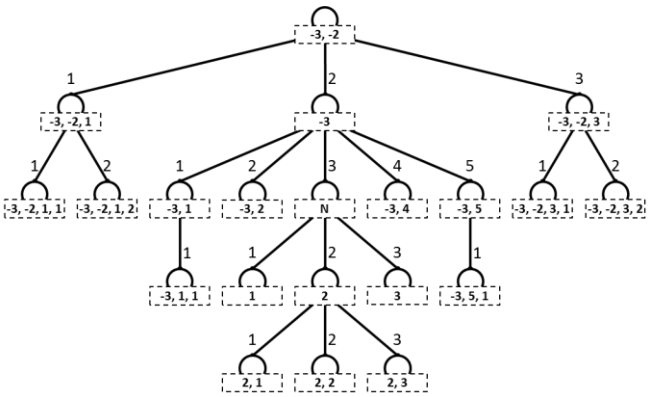

図 3　ノードと NO 履歴の対応

図 3 では，中央のノードを検索開始点の基点ノードとした場合を例に，各ノードの対応する NO 履歴を示している．NO 履歴は，0 個以上の負の値の後に 0 個以上の正の値が続く数字列となる．

2.4.3. 文書順の復元

NO 履歴を用いて文書順を判定するために，判定対象の NO 履歴を持つノードと基点ノードとの位置関係から次の 3 つパートを定義する．

- パート１：
  基点ノードより文書順で前方となるノード
- パート２：
  基点ノードと基点ノードの子孫ノード
- パート３：
  基点ノードより文書順で後方となるノード（基点ノードの子孫は含まない）

NO 履歴は，その数字列によって次のようにパートを判定できる．

- パート 1
  - 負の値のみで形成される場合
  - 負の値と正の値が隣接する点の前後の値

において，負の値の絶対値が正の値より大きい場合

- パート２
  - 先頭の NO が正の値である場合
- パート３
  - 負の値と正の値が隣接する点の前後の値において，負の値の絶対値が正の値より小さい場合

パートを判定した NO 履歴はパートごとに，パート１，パート２，パート３の順で文書順に並べ替えが可能になる．同じパート同士の NO 履歴の場合，次に数字列に含む負の NO の数で文書順判定を行う．この際，パートによって文書順の判定方法が異なる．

- パート１：
  負の NO の数が多いものほど文書順で前方
- パート２：
  負の NO は存在しない
- パート３：
  負の NO の数が少ないものほど文書順で前方

さらに，負の NO の数が同じ場合は，正の NO 数字列全体を対象に，辞書順に比較を行うことで，文書順の判定が可能になる．

3. 提案手法

XML 文書分散管理システムで position 関数の処理を行うためのアプローチを説明する．

position 関数の処理は指定方法によって RPE の処理内でローカルに処理できる場合と，そうでない場合に分けることができる．以下の XPath 式(1)と(2)を用いて説明する．

```
/child::A/child::B/child::C[position()=10]          (1)
/child::A/child::B/descendant::C[position()=10]     (2)
```

式(1)の場合，position 関数を含むロケーションステップにおいて，B 要素の子要素となる C 要素集合それぞれがシーケンスとなり，それぞれの 10 番目の C 要素を選択する．ある B 要素からその子要素となる C 要素集合は，一度の索引検索ですべて取得できる．よって position 関数処理はその時点で可能で，一つの RPE でローカルに処理可能と言える．

一方で，式(2)では，ある B 要素からその子孫要素となる C 要素集合が，一つの RPE ですべて取得できない．そのため，一つの RPE のみでは position 関数処理できず，ローカルに処理不可能である．

以上をまとめると，position 関数処理を含むロケーションステップの軸ごとに次のように分けることができる．

- 一つの RPE がローカルに処理できる軸
  - child
  - parent
  - preceding-sibling
  - following-sibling
- 一つの RPE がローカルに処理できない軸
  - descendant
  - ancestor
  - preceding
  - following

また，position 関数直前までのシーケンスを対象に position 関数を処理する場合も，一つの RPE がローカルに処理できないケースとなる．例として，次のような XPath 式がある．

```
(/child::A/child::B/child::C)[position()=10]        (3)
```

式(3)では，“()”内の式にマッチするすべての C 要素を対象に 10 番目の C 要素を指定する XPath 式となる．そのため，一つの RPE がローカルに処理できないといえる．

本稿では並列処理環境で課題となる，ローカルに処理できない場合の position 関数処理を３つの方法で提案する．処理方式として，HOST のみで処理する方式と，DETECTOR 中心で処理する方式がある．

3.1. HOST のみで処理する方式

この方式は position 関数の処理を HOST のみが行い，RPE と DETECTOR はこの処理について関与しない．既存の検索処理方法に対し，HOST に position 関数に関わる処理を追加するのみで実装を行う手法である．具体的な処理方法を次に述べる．

まず HOST はユーザからのクエリを受けて，その検索パスに position 関数が含まれていた場合，検索パスを①position 関数前，②position 関数，③position 関数後，の３つに分割する．XPath 式(4)を例に説明する．

```
(books/book)[position()=2]/title                    (4)
```

式(4)にパス分割を適応すると次のように分けることができる．

①position 関数前　：　(/books/book)
②position 関数　：　[position()=2]
③position 関数後　：　/title

②と③は HOST が保持し，①は通常通りの処理で RPE へ検索要求を発行する．

RPE での処理が終わると，HOST では position 関数前までの検索結果がすべて集計され，検索結果が文書順にソートされる．この結果集合に対し，保持していた position 関数のパスを用いて処理を行い position 関数処理の最終結果を得る．position 関数後のパスが存在する場合は，そのパスについて再び position 関数の確認処理から繰り返す．パスが続かない場合は，HOST として検索結果を出力する．

この処理方法では，パスを分割することで position 関数処理対象となる検索結果をサブクエリとして取得する．そのため，従来の検索処理方法はそのまま利用して実装が可能である．一方で，HOST に負荷が集中するため，HOST がボトルネックになる可能性がある．

3.2. DETECTOR 中心で処理する方式

この方式は DETECTOR で position 関数の処理を行うが，RPE でもその補助となる処理を行う．この方式は，さらに２つの方法で提案する．

3.2.1. position 関数直前の処理完了を待つ方法

この方法では，RPE が position 関数処理を検出した時点で，結果候補を一時的に保持する．position 関数直前までの処理をすべて完了した後，DETECTOR で集計して position 関数処理結果を得る．

具体的な処理を次に述べる．

まず RPE では position 関数処理対象となる結果候補をすべて集計して保持する．一方で DETECTOR によって position 関数直前までの処理の完了を判定し，判定した時点で RPE に通知を行う．この通知を受けた RPE は保持している結果候補を一時的に文書順にソートし，position 関数処理に必要と判定できる結果候補のみを抽出して DETECTOR に送信する．RPE のこの結果候補の削減は，position 関数のパスを解析して処理する．

DETECTOR で RPE から受信するすべての結果候補を集計した後，position 関数の最終結果を抽出する．position 関数以降に再び検索が必要な場合は，DETECTOR から RPE へ検索要求を発行し，最終結果となる場合は HOST へ送信する．

この処理方法では，RPE と DETECTOR を利用して処理を分散して行うことが可能である．一方で，RPE が一時的に待ち状態になるため検索処理を遅延させる可能性がある．

3.2.2. position 関数直前の処理完了を待たない方法

この方法は，3.2.1 の処理方法を基本とするが，position 関数処理結果が得られる前に，検索処理を position 関数以降に進めるという点で異なる．

具体的には，RPE は結果候補を保持する一方で次への検索要求を発行する．しかし，この検索要求については position 関数処理結果が反映されないことから最終的に必要か判定できない不確定な検索要求となる．

この不確定な検索要求については，検索処理の優先度を下げることで，他の検索要求処理を阻害しない処置を行う．つまり，RPE は，他に処理する検索要求がないアイドル状態になる場合のみ，不確定な検索要求を処理する．

position 関数の最終結果については，3.2.1 と同様に DETECTOR で抽出するが，この結果は HOST と全 RPE に通知する．これを受信した HOST では，不確定検索結果を，RPE では不確定検索要求をそれぞれ正誤判定する．この正誤判定によって正しいもののみを残し，position 関数処理が完了となる．

この処理方法では，3.2.1 の処理方法に比べ，RPE が検索処理を遅延させることがない．一方で，正誤判定処理が必要になるため，不確定検索要求が大量に発行されると結果的に遅延を引き起こす可能性がある．

4. 評価実験

4.1. 実験環境

実験は，表 4.1 に示す計算機を 18 台用いて，それぞれ HOST1 台，RPE12 台，DETECTOR5 台を割り当てる．

表 4.1　実験に用いた計算機の仕様

CPU	Intel(R)Core(TM)i5
クロック周波数	3.20 GHz
メモリ	4.0 GB
OS	Fedora 14 Linux2.6.35

実験に使用した XML ファイルは XMark[6]プロジェクトで公開されている XML 文書作成ツールを用いる．検索対象の XML ファイルの情報を表 4.2 示す．

表 4.2　XML ファイル情報

ファイルサイズ	11,879,296 byte
総ノード数	167,864
タグ数	74
XML 木深さ	12

検索対象となる XML ファイルは，CBT，PBT，ABT の３種に索引化し，それぞれの索引を RPE 数と同じ 12 に分割したものを各 RPE に配置して実験を行う．

4.2. 検索時間

表 4.3 に示す 7 つのパスを用いたクエリセットを使用して検索時間を測定する．

表 4.3　クエリセット

RID	検索パス	検索結果数
1	/site/regions/africa/item/name	55
2	/site/closed_auctions/closed_auction	975
3	/site/people/person/descendant::interest	3832
4	position 関数を含むパス	-
5	/site/regions/australia/item/name	220
6	/site/categories/category/name	100
7	/site/people/descendant::interest	3832

RID 4 の位置に position 関数を含むパスとして表 4.4 に示す 3 種を用いる.

表 4.4　position 関数の指定方法

	RID 4 の検索パス	検索結果数
セット①	(/*/*/*)[position()<=100]/*/*/*	599
セット②	(/*/*/*)[position()<=1000]/*/*/*	6000
セット③	(/*/*/*)[position()<=10000]/*/*/*	13311

検索時間はクエリの受け付けからクエリセット全体の検索結果の集計までの時間を指し，40 回計測したものの平均検索時間を算出する.

なお position 関数の処理方法ごとに計測を行うが，便宜上ここでは各方法を次のように言い換える.

- Type1：
  HOST のみで処理する方法（3.2）
- Type2：
  DETECTOR 中心の処理で position 関数直前の処理完了を待つ方法（3.2.1）
- Type3：
  DETECTOR 中心の処理で position 関数直前の処理完了を待たない方法（3.2.2）

図 4 に実験結果を示す.

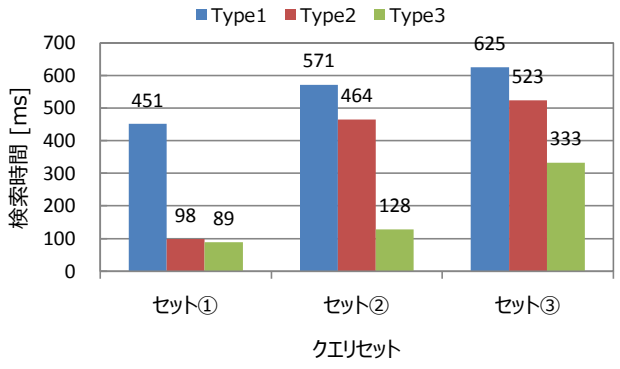

図 4　position 関数処理方法別の平均検索時間

いずれのクエリセットについても Type1 は最も時間を要し，処理の負荷が大きい HOST がボトルネックになってしまっていると考えられる．セット①では，Type2 が 98[ms]，Type3 が 89[ms]と DETECTOR 中心で処理する方式が非常に高速となった．これは position 関数で指定する値が小さいために RPE で結果候補を削減する効果が大きく，DETECTOR の処理負荷も小さいためと考えられる．しかし，セット②，セット③と position 関数で指定する値が大きくなるにつれ，Type2 は検索時間が 464[ms]，523[ms]となり，その増加率は高いと言える．Type2 は position 関数で指定する値に依存が強く，結果候補削減の効果が得られないほど検索時間の増加につながりやすいと考えられる．一方で Type3 は，セット②で 128 [ms]，セット③で 333[ms]と，Type2 に比べて検索時間の増加率も低く非常に高速であると言える.

またここで，全 RPE 合計の平均索引検索数を表 4.5 に示す.

表 4.5 全 RPE 合計の平均索引検索数

	セット①	セット②	セット③
Type1	6,326	18,758	53,414
Type2	6,326	18,758	53,414
Type3	8,330	46,346	101,694

表 4.5 から，Type3 は Type1 と Type2 に比べて不確定な検索要求を検索するために索引の検索数が増加している．不確定な検索要求は結果候補となる数が大きいほど増加するが，この検索要求は処理の優先度が下げられるため，アイドル状態になる RPE のみ処理される．Type2 が DETECTOR で処理する際に，処理の遅延が検索時間の増加に直接つながるのに対し，Type3 はその間にアイドル状態になる RPE を利用して不確定ながら検索を進行する効果は大きいと考えられる.

また Type3 は，Type2 に比べて正誤判定処理を必要とするが，この処理も検索時間を大きく遅延させるほどのコストはないと考えられる.

5. おわりに

本稿では XML 文書分散管理システムにおける position 関数の処理方法を 3 つの方法で提案した.

3 つの方法の中では，DETECTOR 中心に処理する方式で position 関数直前までの処理を待たずに検索を進行し，後に判定を行うという処理方法が最も高速であることが判った．この方法では，アイドル状態になる RPE を利用して検索を進行するため，並列処理下で資源を有効に活用して効率的に処理できると言える．ただし，ビジー状態となる RPE が多い場合は，効果が得られない可能性がある．また，position 関数以降に検索処理がないパスでは，position 関数の最終結果を得てから正誤判定を行う分の処理が非効率になる．この点は，今後の課題として，十分に検証を行い，改良可能であると考えられる.

また，本研究で用いた XML 文書は XMark で生成したもののみであるため，他の異なる構造を持つ XML

文書に対しての検証も行いたい.

参 考 文 献

[1] Extensible Markup Language, http://www.w3.org/XML/

[2] eXist-db Open Source Native XML Database, http://exist.sourceforge.net

[3] XML データベース NeoCore, http://www.neocore.jp

[4] XML Path Language, http://www.w3.org/TR/xpath/

[5] Subramaniam, Samini, Su-Cheng Haw, and Poo Kuan Hoong. "XML Labeling Schemes for Dynamic Updates: Strengths and Limitations." International Journal on Advanced Science, Engineering and Information Technology 1.3 (2011): 236-241.

[6] XMark-An XML Benchmark Project, http://www.xml-benchmark.org/