効率的なXQuery処理のためのDTMに基づくXMLストレージ

油井 誠[†] 宮崎 純[†] 植村俊亮[††]

本稿では，XMLを表形式で表現するDocument Table Model（DTM）に基づくXMLデータの二次記憶への格納手法を提案する．大規模XMLデータに対するXML問合せ処理では，二次記憶上のXMLデータ格納方法とXMLデータへのアクセス手法が，問合せ処理性能に大きく影響する．そこで，我々はDTMの一形式で内部表現されたXML文書をブロック化して二次記憶に配置し，問合せ実行中に必要なブロックを主記憶に読み込む機能を特徴とするXQuery問合せ処理手法を開発した．提案手法では，参照するブロックの局所性が高い問合せを効率的に処理するためにエクステントを，参照するブロックの局所性が低い場合に対処するために逆経路索引をそれぞれ利用する．実験結果から，提案手法がデータサイズに対してほぼ線形の性能が得られることがあることを示し，さらに，限られたメモリ環境下における提案手法の有効性を示す．

XML Storage Based on DTM for Efficient XQuery Processing

Makoto Yui,[†] Jun Miyazaki[†] and Shunsuke Uemura[††]

In this paper, we propose an XML storage scheme based on DTM (Document Table Model) which expresses an XML by a table form. On query processing for large-scale XML data, XML storage schemes on secondary storage and their access methods greatly affect the entire performance. For this reason, we developed an XQuery processing scheme in which XML data is internally represented as a set of DTM blocks, and can directly be stored on secondary storage. Moreover, we adapted the use of extents for queries whose locality of referred blocks is high, and introduced Reverse-Path index for queries whose locality of referred blocks is low. Our experimental results show that the proposed scheme can often obtain almost linear scalability in performance as the data size increases, and is especially adaptable to limited resource environments.

1. はじめに

W3Cで標準化されたXMLは，組織間のデータ交換形式として標準の地位を確立し，計算機上のデータ永続化形式としても利用が広がっている[1)]．今後，蓄積されるXMLデータの増加に比例して，XMLデータベースシステムに求められるXMLデータ処理能力も高まっていくことが予想される．

これまでXMLデータベースの研究は，XMLの木構造（XML木）のノード符号化方式，索引方式，XPathのパス処理の効率化といった問合せ処理の最適化を中心に研究されてきた[2)]．大規模XMLデータベース処理を実現するにあたっては，問合せ処理手法にも影響を与えるXMLデータの二次記憶への格納方式も欠かせない因子である．しかし，これまでXMLデータベースの研究は関係データベースを活用した手法が主流であり，XMLストレージ方式の研究事例は少ない．一方で，近年では関係データベースにおけるXMLデータ処理においても，関係データ処理エンジンとは別に，XML専用のデータ処理エンジンを利用する研究が注目されている[3)]．

XML木の関係表への写像は，一般的に，XML木のノードを基本単位とする．そのため，データサイズの増加に比例して関係表のタプル数が増加することが問題として指摘されている[4)]．また，現在の関係データベースの行領域を確保する仕組みを利用した場合，クラスタ表の実装や断片化した表の自動再編成機能の有無によって，タプルに写像されたXMLノード群のディスク上における配置が疎らになり，XML木における近接関係が失われる可能性があることにも留意が必要である．

XML木の近接関係は，XML問合せ処理に重要な役割を果たす．XML問合せ処理においてはXML木

† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology
†† 奈良産業大学情報学部情報学科
Nara Sangyo University

の軸をたどる操作が中核操作であるが，軸をたどる操作は起点ノードを中心とする局所性の高いものであることが多いからである．また，XML 問合せ処理では問合せ結果の直列化（シリアライズ）という特有の処理が発生する．たとえば，*/auction//site* はルート要素 *auction* の子孫の *site* 要素を指定する XPath 問合せであるが，検索結果を XML として直列化して出力するうえでは，*site* 要素とその子孫すべてのレコードに，文書順にアクセスする必要がある．このようなことから，XML 問合せを処理するためには，二次記憶上での効率的なノード配置手法をとることが重要である．

本稿では，DTM（Document Table Model）に基づく二次記憶への XML データ格納方式と，XML データアクセス手法を提案する．提案手法では，参照するブロックの局所性が高い問合せを効率的に処理するためにエクステントを，参照するブロックの局所性が低い場合に対処するために逆経路索引をそれぞれ利用する．本稿の貢献の 1 つは，抽象レイヤで行われていた局所性の議論の一部を二次記憶へのデータアクセスにまで展開し，実験により既存提案を超える高いスケーラビリティを示すことにある．また，XML 問合せにおけるアクセスパターンの分析を通して，XML 問合せ処理においてこれまであまり注目されてこなかった直列化処理に焦点を当て，ノードの二次記憶における物理配置が問合せ性能に大きく影響することを示す．

本稿の構成は次のとおりである．2 章では関連研究について述べる．3 章では提案システムの概念モデルについて述べ，続く 4 章で内部モデルについて説明する．5 章では，逆経路索引を利用したアクセスパスの最適化について述べる．6 章で提案手法に評価を与え，7 章で本稿のまとめと今後の展望について述べる．

2. 関 連 研 究

本章では，関連研究として，まずアクセスの局所性について着目した研究について述べる．次に，これまでに提案されてきた XML の物理的格納手法について述べ，最後に，本提案システムにおいても利用する XML 木のラベル付け手法について述べる．

2.1 アクセスの局所性に着目した研究

大規模 XML データ処理を実現するための手がかりの 1 つが，参照局所性である．問合せを処理するにあたって参照する必要があるノード群の情報量は，必ずしも XML 文書のデータ量に比例しない．このことに着目した研究はすでにいくつかなされている．中島らは DOM を対象として，DOM 操作に必要な部分を部分的に主記憶に保持する SPlitDOM を提案している[5]．Amer-Yahia らは XQuery[6] を対象として，問合せを静的解析し，XML データから必要な部分だけを射影する手法（Document Projection）を提案している[7]．これらは主記憶上に展開される XML データのサイズを軽減する目的の研究であり，必要とされる部分 XML データが主記憶のサイズを超えるような場合は考慮されていない．天笠らは，Strong DataGuide に基づくリージョンディレクトリを利用し，問合せ式や参照頻度に応じて関係表に格納されるノード群をファイルとマッピング（アンマッピング）する手法を提案している[4]．関係表への写像手法におけるノード数の増加に対処する目的の研究であるという点，問合せ実行前の最適化であるという点で，問合せ実行中の参照局所性を問う本研究とは対象とする問題領域が異なる．

2.2 物理格納手法に関する研究

XML の物理的な格納方法に着目した研究は，これまでにもいくつか提案されてきた[8)~11)]．以下では，その代表的なものを説明する．

スキーマ情報に基づく格納手法

Meng らはスキーマを用いた効率的な物理格納手法を提唱している[10]．OrientStore ではあらかじめスキーマ情報を与え，スキーマグラフをブロック化する．そしてその断片に属するノードをディスク上で近接配置することで，問合せ処理において効率的なデータアクセスを実現している．スキーマグラフごとにブロック化を行う手法は，XPath の単純経路問合せ（ロケーションパス）によって指定される部分 XML データを選択する処理の効率化を主眼とするものである．スキーマグラフごとの格納戦略は，XML 問合せ処理で必要となる直列化処理に対して必ずしも効率的とはいえない．

単純経路アクセスにおいて該当レコードのアドレスを特定する処理については，経路索引を用いて補完できる．我々の提案手法における格納手法では XML 文書における文書順を保存し，経路索引を利用する．

部分木ごとの格納手法

Natix[8] は，部分木ベースの物理格納戦略をとる XML ネイティブデータベースの包括的な研究である．XML 木を物理ページサイズに適合するように分割し，分割した部分木を基本単位として格納する．Natix の手法では，ページをまたぐ問合せのためにポインタとして機能する仮想ノードや集約ノードといった冗長なノードの導入が必要である．一方で，我々の論理表に基づくアプローチでは，軸操作はすべてオフセット計

算に基づく．ブロックあたりのノード充填率が高いことは，読み込みページ数の削減につながる．

参照局所性のある問合せを効率的に処理するために効果的なノードの二次記憶への配置方法は，問合せ処理方式に依存するため，XML 木としての近接関係をそのまま配置することが必ずしも効率的であるとは限らない．たとえば，child::site/child::regions という child 軸を 2 度たどる処理をする場合，set-at-a-time 方式で処理する場合には子ノードがまとめて近接配置されている方が効率的である一方で，tuple-at-a-time 方式でパイプライン処理をする場合では，site[1]/region[1]，site[1]/region[2]，...，site[last()]/region[last()] という順にレコードアクセスされるため，文書順にノードが配置されている方が効率的となる．我々の XQuery プロセッサは，代表的な XQuery 処理系である BEA/XQRL XQuery プロセッサ[12] や Saxon[13] で採用されているものと同様のイテレータ木に基づく処理モデルを採用しており，多くの問合せがパイプライン処理される．そのため，ノードを文書順にディスク上に配置するという方針をとっている．

巡航アクセスのための格納手法

Zhang らは，Next-of-Kin（NoK）Pattern Match という木構造における近親ノードを効率的に選択するパターンマッチ手法を提案している[9]．NoK パターンマッチの主張は次のとおりである．XQuery UseCase[14] に現れるクエリの構造関係のうち 2/3 が child 軸であることに代表されるように，一般的に XML 問合せにおける構造関係は局所性が高いものが多く，選択率が低い．こうした局所性が高く選択率が低い問合せに対して，構造結合（Structual-Join）に代表される XML 問合せ処理手法[15] を用いることは非効率であり，近親ノードを選択する軸評価（たとえば，child や next-sibling）に巡航アクセスを用いている．そして NoK パターンマッチを効率的に評価するために，XML 木のノードをディスク上で（XML 木の）前置順に配置し，予想される IO コストを抑える物理格納手法を提案している．Zhang らの研究と我々の研究の主な違いとしては，論理的な文書表現の違いがあげられる．Zhang らの研究は Subject-tree と呼ばれる文字列表現により XML を表現するのに対し，我々のアプローチは，物理データアクセスについても論理的な表である DTM へのアクセスに基づく．Subject-tree ではページにノード名（要素名，属性名）と構造情報を格納するため，XML の格納効率が悪い．また，軸をたどる操作の効率性の面で DTM に優位性がある．3.2.2 項において文献 9) で説明されている軸処理について，我々の提案手法におけるアルゴリズムを取り上げる．

2.3 XML 木のラベル付け手法

XML 木におけるノードの先祖子孫関係や親子関係を効率良く処理するために，これまで，様々なラベル付け手法が提案されてきた[16)~19)]．近年では，なかでも挿入に対して強固なラベリング手法が注目されている[20)~22)]．

これらの中で，広く利用されているのが Dewey Order に基づくラベリング手法[17] とその派生[20),21)] である．Ordpaths[20] は，マイクロソフト SQL Server において採用されているラベリング手法である．挿入に対して強固な性質を持ち，また，ファノ符号化に基づく接頭辞スキーマに従うビットレベルの符号化を行うことで，ラベルサイズの増加を抑制するという特徴を持つ．Ordpaths の符号化方式を見直し，ハフマン符号を用いる研究も存在する[23]．Böhme らによって開発された Dynamic Level Numbering（DLN）[21] は，Dewey Order に基づくビットレベルのエンコーディングスキーマという点で Ordpaths と同一であり，ローカルオーダ（兄弟間の関係を表す数値）のエンコーディングには prefix-free の符号化スキーマを用いる．Ordpaths と比べて，挿入無制限特性を実現するために 1 ビットのセパレータを利用するという違いがある．この点で，DLN は Ordpaths 方式と Kobayashi らが提案する VLEI 方式[22] の利点を組み合わせたものといえる．局所への連続した挿入に対する強固さは Ordpaths に劣るが，バルクロード直後のラベルサイズを Ordpaths の場合と比べて相対的に小さくすることができる．Ordpaths では，奇数をバルクロード時のローカルオーダとして利用するため，単純な Dewey Order の倍速でローカルオーダに割り当てる数値領域を消費していく．ローカルオーダが大きくなることは，消費する記憶領域の増加につながる．

我々の提案システムにおけるラベリング手法としては，DLN の採用を見据えて，現在はバルクロード時の番号付けスキームが DLN と共通する Dewey Order の UTF-8 符号化方式[17] を利用している．なお，DLN はオープンソースの XML ネイティブデータベース eXist でもすでに採用されている[24]．

3. 提案システムの概念モデル

3.1 Document Table Model（DTM）

提案手法では，XML 文書を Document Table Model（DTM）の一形式で内部表現する．

Document Table Model とは，Apache Xalan-J[25] XSLT Processor に実装されたことで注目された，実

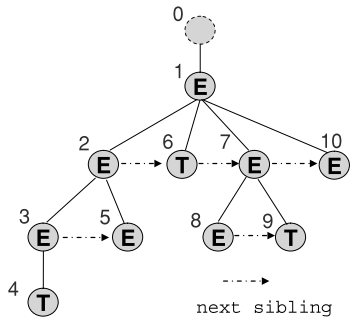

図 1 深さ優先順にラベル付けされた XML 木
Fig. 1 XML tree with depth-first order.

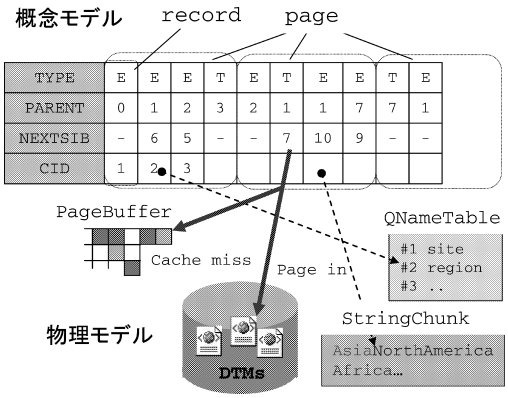

図 2 DTM の概念図
Fig. 2 Illustration of DTM.

行性能の効率化と記憶領域の最小化を目的とした計算機上での XML データの内部表現形式である．DOM（Document Object Model）に対して，オブジェクトを用いない数値型で構成される表で XML 文書を表現する特徴から Document Table Model と呼ばれる．DTM ではノード固有の整数（表の索引）をノードの識別に用いる．そのノード ID により，QName（名前空間とノードのローカル名の組合せ），整数のインデックス，そして文字列バッファにおけるそれぞれのノードのテキスト値へのオフセットなどを管理する．XML の木構造の保存はノード ID を用いたリンクによる．

原始データ型で XML の木構造を表現できるという特徴により，オブジェクト生成による実行効率の劣化，およびオブジェクトが消費するメモリフットプリントを抑制することができる．そのためオブジェクト利用負荷の高い Java や C#などにおける実装において XML を扱う場合の効率に優れる．主要な XQuery/XSLT 処理系[13),25)] が DTM と類似の内部表現形式をとっているため，DTM に基づく我々のアプローチは，これら実装手法における一次記憶上のデータ表現に対して二次記憶への拡張を行う際の透過性に優れる．

3.2 提案システムの概要

提案システムで用いる DTM の概要を**図 2** に示す．図 2 は，**図 1** の文書順に深さ優先順でラベル付けされた XML 木を DTM として表した例である．図 1 の E は要素ノード，T はテキストノードをそれぞれ指す．実際の DTM を説明のために図中では簡略して表記した．たとえば，図 2 の表は 4 行の配列からなるが，実際は 1 行の配列で表現される．実際に格納されている値と図表記の違いについては，説明で補い解説する．

3.2.1 表の構成

合計ノード数を n とすると図 2 の DTM は，$4 \times n$ の平坦な Int 配列として論理的に表現される．DTM の配列の添え字が 1 から始まると仮定すると，N 行 M 列の実際の添え字は $(M-1) \times 4 + N$ で計算される．たとえば，3 列 2 行目の添え字は $(3-1) \times 4 + 2$ で 10 である．配列の添え字がノードハンドルとなる．このように提案システムにおいては，1 ノードにつき 4 つの Int 値，16 バイトの表領域を利用する．

1 つのノードが構成する 4 つの要素は次のとおりである．1 列目は，ノード種別とノードの各種属性を表現する．下位 3 ビットで，7 種類のノード種別を表現する．下位 4 ビット目は FIRST_ENTRY フラグ，下位 5 ビット目は LAST_ENTRY フラグである．それぞれ，左右に兄弟ノードを持つかどうかを判定するのに利用される．下位 6 ビット目の HAS_CHILD フラグでは，ノードが子要素を持つかどうかを表現する．続く 0xFFC0 でマスクされる 10 ビットの領域は属性数を，0xFF0000 でマスクされる 8 ビットの領域は名前空間宣言の数をそれぞれ表す．0x7000000 でマスクされる 3 ビットの領域は，子要素数の概略を保持する予約スペースであり，残りの 5 ビットは将来の拡張のために予約した領域である．2 列目，3 列目には親ノードの索引，弟（next-sibling）ノードの索引の値をそれぞれ保持する．4 列目には ContentID（**CID**）を保持する．CID は，DTM 表とは別に管理する文字列の ID，あるいは QName を一意に表現する．XML 文書中の文字列についてはチャンク化したうえで，CID を振って一意に管理する（この文字列管理モジュールを以下，**StringChunk** と呼ぶ）．チャンク化閾値を超える文字列（デフォルトでは 512 バイト以上，設定可能）については，メモリ消費量を抑えるため，メモリ内表現としても LZ 圧縮して管理する．QName については文書単位ではなく，Collection と呼ぶ（ファイルシステムのディレクトリに相当する）集合単位に一意に管理し，スペース効率を高めている．この QName 管理

モジュールを **QNameTable** と呼ぶ．

3.2.2 表へのアクセス

表へのアクセスは，基本的に添え字を指定してノードに関連する数値を取得する操作 API を用いる．文字列値と QName については，それぞれノードハンドルから CID 値を取得し，その CID 値をキーとして，それぞれ StringChunk，QNameTable より取得する．XPath の軸の評価は，parent 値，next-sibling 値といったノードの属性値を参照するなど，オフセット計算をベースとする．問合せ処理器に対しては，論理的なデータ構造（DTM）と操作手段が提供される．

我々の DTM バリアントにおける軸処理の中核操作は，起点ノードの第 1 子を取得する firstChild 関数，末子を取得する lastChild 関数，弟を取得する nextSibling 関数，親を取得する parent 関数，兄を取得する previousSibling 関数の 5 種からなる．この 5 種の中核操作により XPath のすべての軸処理を実現する．ここでは，**図 3** に呼び出し頻度の高い firstChild 関数，および nextSibling 関数，parent 関数についてアルゴリズムを示す．

getCol 関数は，指定された配列添え字により指定される配列要素を取得する．hasChild 関数は，子を持つかビットフラグにより判断する関数である．getNamespaceCount 関数はノードの名前空間定義数を取得する，getAttributeCount 関数はノードの属性数を取得する関数である．BLOCKS_PER_NODE 値はノードが消費する配列の要素数であり，PARENT_OFFSET はノード ID に対して parent 値が格納されている配列添え字のオフセットを，NEXTSIB_OFFSET は next-sibling 値が格納されている配列添え字のオフセットをそれぞれ表現する．

Algorithm 代表的な軸アクセスのアルゴリズム

```
1. const BLOCKS_PER_NODE = 4;
2. const PARENT_OFFSET = 1;
3. const NEXTSIB_OFFSET = 2;

4. function firstChild(curnode) {
5.     code := getCol(curnode);
6.     if(!hasChild(code))
7.         return nil;
8.     namespaces := getNamespaceCount(code);
9.     attributes := getAttributeCount(code);
10.    addr := curnode + ((namespaces + attributes) + 1)
11.                * BLOCKS_PER_NODE;
12.    return addr;
13. }

14. function nextSibling(curnode) {
15.    return getCol(curnode + NEXTSIB_OFFSET);
16. }

17. function parent(curnode) {
18.    return getCol(curnode + PARENT_OFFSET);
19. }
```

図 3 代表的な軸アクセスのアルゴリズム
Fig. 3 Algorithm of primary axis-accesses.

4. DTM に基づく XML ストレージ

本章では，3 章の概念モデルを実現するための内部モデルについて述べる．

4.1 物理格納方法

3.2 節で説明したように，我々のシステムでは XML 文書を DTM として表現する．DTM の永続化は DTM の表（DTM 表）をブロック化し，二次記憶上に配置されるファイルに対してページングを行うことにより実現している（図 2）．提案手法では，XML 表の各ブロックを，文書順にページングファイル中に保存する．

ページング処理の形式には，大別して可変長方式と固定長方式がある．我々はこれまでに可変長ページング方式の DTM と固定長ページング方式の DTM について検討してきた[26] が，おおむね固定長 DTM が良い性能を導き出すことから，ここでは固定長のページングを採用する．

文書順での配置

提案手法では，XML 木のノードを文書順に配置する．文書順の配置は，ある親とその 2 番目以降の子供が遠く離れてしまう可能性があり，必ずしも XML 木での近接ノードを近くに配置する配置方式ではない．文書順による配置を採用したのは，次の 2 つの理由による．

- 結果の直列化処理や文字列値の計算に文書順での配置が適している．
- 軸アクセスをパイプライン処理するオペレータを採用しているため，文書順でのアクセスパターンが現れることが多い．

我々の XQuery プロセッサは，BEA/XQRL XQuery プロセッサ[12] や Saxon[13] と同様にイテレータ木に基づく問合せ処理モデルを採用しており，軸アクセスやループ処理をはじめ，多くの問合せはパイプライン処理される．個々のオペレータは標準的なイテレータのインタフェースを実装し，tuple-at-a-time の処理を行う．そのため，文書順でのアクセスが多く現れる．4.3 節で実際のアクセスパターンを取り上げる．

表の物理構成

3.2.1 項で説明した DTM 表は，主記憶上で物理的には，Int 型の二次元配列（配列の配列）として表現される（**図 4**）．二次元目の要素（Row Pages）が 1 つのページを構成し，一次元目の配列（Row Skeleton）がそのページへのポインタを保持する．一次記憶に読

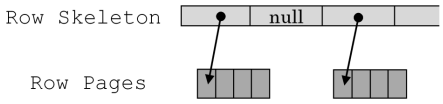

図 4 主記憶上での DTM 表の表現
Fig. 4 In-memory representation of DTM.

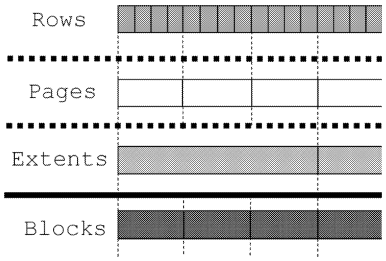

図 5 DTM の物理構成
Fig. 5 Physical layers of DTM.

み込まれていないページ要素は null 値として状態表現されるため，一次元配列のみで DTM を表現する場合と違い，一次記憶の領域を過消費しない．なお，ノード数が $2^{32}-1$ を超える場合，我々の現在のシステムにおいては文書サイズが 2 G バイトを超える場合には，8 バイトの Long 配列を用いるように戦略を切り替える．二次元化した一次元配列へのアクセス方法は，一次元配列へのアクセス手法から導ける．以降，特に言及しない限り，説明では DTM を Int 型の一次元配列で構成したものとする．

エクステントを利用した物理構成

DTM の物理構成は，**図 5** のように 4 層から構成される．DTM 列は，DTM 表の物理モデルであり，ノード情報が格納される基本レコードである．ページは提案システムにおける基本 IO 単位である．ページはデフォルトの設定では 512 列で構成され，1 つのページで $512/4 = 128$ 個のノードを管理する．つまり，Int 配列で構成された DTM において最小ページサイズは 512×4（Int 長）で 2 K バイトとなる．**エクステント**（Extents）は連続するページをまとめた概念で，デフォルトの設定では 32 個のページをまとめた 64 K バイトの記憶領域である．一般に，エクステントとは論理的に連続した記憶領域を提供するものであり，XFS [27] などのファイルシステムや一部の商用関係データベースにおいて利用されている．我々の提案においては，連続するページをエクステントに連続して割り当てることでディスクブロックが連続するように促し，ディスクアクセスの効率化を図っている．

DTM におけるページング処理では，読み込みはエクステント単位に行い，書き込みはページ単位に行う．

Algorithm ページ読み込みアルゴリズム

```
IN: rowId   OUT: block

1. const PAGE_SHIFT := 9;
2. global BUFFER_CACHE;

3. pageAddress := rowId >> PAGE_SHIFT;(a)
4. block := BUFFER_CACHE.get(pageAddress);(b)
5. if(block == nil) {
6.     block := readInExtent(pageAddress);(c)
7. }
8. return block;
```

a) 列アドレスから，該当するページの先頭の添え字（ページアドレス）を計算する．

b) バッファにページがキャッシュされていれば，そのページを返す．

c) block 変数が値が nil ならば，pageAddress をキーとして readInExtent 関数を呼び出し，該当ページを取得する．readInExtent 関数は，要求ページアドレスの該当するエクステントに含まれるすべて（32 個）のページをページインし，要求ページアドレスの該当するページを返す．ページインされたページは，ページアドレスをキーとして BUFFER_CACHE に保存される．

図 6 ページ読み込みアルゴリズム
Fig. 6 Page-in algorithm.

ページとエクステントを別に設けたのは，(a) ページの読み込みを粗粒度で行うことにより IO 回数を抑えることと，(b) 細粒度の更新手段を提供するためである．さらに，エクステントを用いることは，(c) ページ読み出し時に要求ページに対しての先読み（**プリフェッチ**）として機能する．

4.2 物理アクセス方法

二次記憶に格納されたページングファイルへのアクセスは，参照する DTM の列データが一次記憶上に存在しない場合に行われる．具体的には，図 3 における getCol 関数にページング処理を行うフックを入れる．該当する列データが存在しない場合のページ読み込みのアルゴリズムは**図 6** のとおりである．

バッファのキャッシュ管理戦略は，提案システムのプロトタイプにおいては 2Q [28]（LRU も選択可能）を利用しており，デフォルトの設定では 128 M バイトの領域である．各ページサイズは 2 K バイトであるので，最大で 64,000 個のページがバッファ管理される．

4.3 アクセスパターンの分析

本節では，提案手法におけるページング戦略を決定するうえで判断材料とした予備実験について述べる．さらに，予備実験で問題となった問合せについてとっ

た対策を述べる．

DTM のページング処理は，基本的にファイルシステムにおいて利用される技術に基づくが，一度に読み込むページサイズは OS のページサイズに基づくよりも，実際の XML 文書アクセスにおける要求ページサイズに即したものである方がより効果的となる．Natix[8] において，Kanne らは 2 K〜32 K の固定長のページサイズを用いて実験を行っているが，ページサイズによってパフォーマンスに大きな影響が出ることを示している．

予備実験では，Natix の研究で示された結果にさらなる考察を与える目的で，XQuery 問合せ処理における DTM ページの要求パターンの抽出を行った．システムが用いるページ長を 512 列とし，XML ベンチマークツール XMark[29] のスケールファクタ 1 の XML 文書（113 M バイト）を用いた．検討したのは XMark で用意されている 20 個の XQuery 問合せである．ここでは，紙面の都合上，すべての問合せについて個別に見ることはできないため，全体的な傾向をつかむ目的で，参考までに 20 個すべてのページアクセスパターンを 1 枚に描画した**図 7** と特異的な Q9 と Q10 を除いた 18 個のページアクセスパターンを 1 枚に描画した**図 8** を提示し，平均的なページアクセスパターンであった XMark Q8（**図 9**）と，特徴的な XMark Q7（**図 11**）の問合せにおけるページアクセスパターン（**図 10**，**図 12**）について細かく見ていく．特異的な傾向を示した Q9，Q10 については，後述する実験（6.1 節）により別途詳しい考察を与える．

ページアクセスパターンを示す図中の縦軸は，要求されたページ番号を表し，横軸は 1 回のレコード要求を 1 単位時間とする時間軸である．ここで，縦軸のページ番号は，下限ほど文書ノードに割り当てられたページに近く，上限ほど XML 文書の末尾付近に現れるノードが格納されたページに近い．つまり，文書順が反映されている．

文書順にアクセスされる問合せ

図 10 は XMark Q8 の問合せ（図 9）を実行した際のページ要求パターンである．提案システムでは，Q8

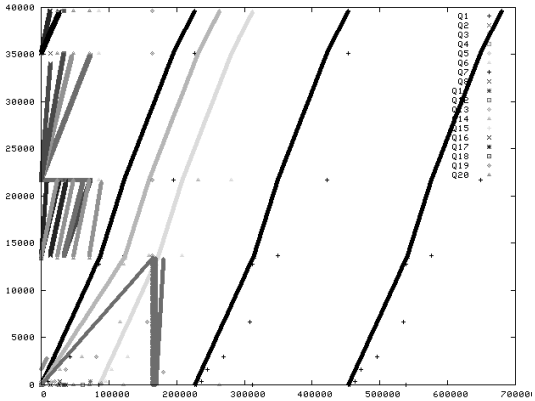

図 8 Q9 と Q10 を除く XMark 問合せのページアクセスパターン
Fig. 8 Page access pattern for XMark queries except Q9 and Q10.

XMarkQ8 構造結合を有する問合せ

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person(a)
 let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person(b) = $p/@id(c)
  return $t(d)
 return <item person="{ $p/name/text()(e) }">
       { count($a) } </item>
```

図 9 XMark 問合せ Q8
Fig. 9 Query No.8 of XMark.

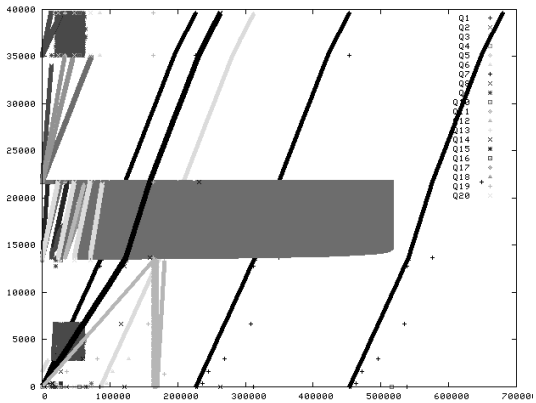

図 7 20 個すべての XMark 問合せのページアクセスパターン
Fig. 7 Page access pattern for all 20 XMark queries.

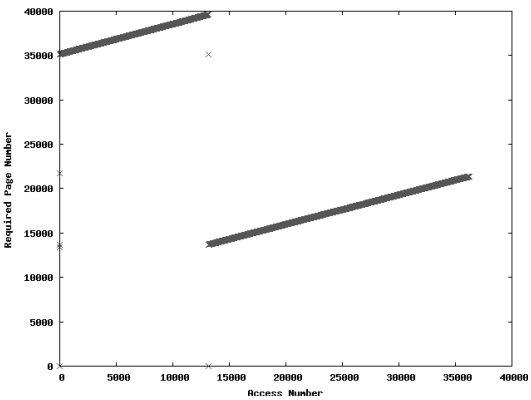

図 10 XMark Q8 のページ要求パターン
Fig. 10 Page access pattern of XMark Q8.

XMarkQ7 複数の // を有する問合せ

```
let $auction := fn:doc("auction.xml") return
 for $p in $auction/site return
   count($p//description(a)) + count($p//annotation(b))
 + count($p//emailaddress(c))
```

図 11 XMark 問合せ Q7
Fig. 11 Query No.7 of XMark.

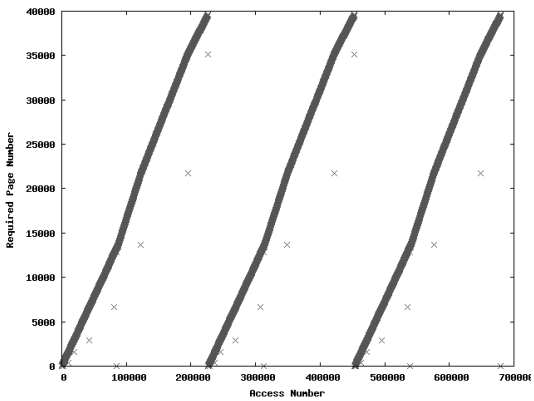

図 12 XMark Q7 のページ要求パターン
Fig. 12 Page access pattern of XMark Q7.

における内部ループの Join 属性 (b) と Join 属性に対応する値 (d) をあらかじめ計算し，次に外部ループ (a) を実行して (c) に対応する値 (d) を取得するという Join 処理をしている．図 10 の見方としては，y 軸の 35,000〜40,000 付近が (b) および (d) に関するアクセスで，y 軸の 14,000〜22,000 付近が (a)，(c) および (e) に関するアクセスである．

図 10 の 2 つの傾斜に代表されるように，XMark 問合せにおいて全体的な傾向（図 8）として右上がりのページ要求パターンが多く見られるのは，文献 9) で報告されているように，XQuery 問合せ中の構造関係の多くは child 軸へのアクセスであり，そのことが，図 10 や図 8 での右肩上がりへのアクセスに現れていると考えられる．このように規則的に文書順でのアクセスが行われる XQuery 問合せにおいては，エクステントの利用とプリフェッチが有効に働く可能性が高いことが分かる．

参照局所性の低い問合せ

XMark Q7 の問合せ（図 11）は，複数の //(descendant-or-self::node()/child::) を有する問合せであり，図 12 がそのページ要求パターンである．// を含む問合せは，参照局所性が低い問合せの代表的なものである．

y 座標の 0 から 40,000 付近までの XML 文書に割り当てた多数のページを要求する右上がりのアクセスパターンが見てとれる．これは，count 関数の引数における // のアクセスが図に表れている．$p 変数が，$doc(\text{“}auction.xml\text{”})/site$ ノードに割り当てられているが，この site ノードは文書ノードであるため，その子孫ノードは残りすべてのノードである．このような // を含む問合せにおいては，順次多くのページが読み込まれることとなる．原理的には，(a)，(b)，(c) すべての式がパイプライン処理可能な場合について 3 本のアクセスを 1 本にまとめることが可能と考えられるが，そうした最適化を行ったとしても文書に割り当てた 0〜40,000 付近までのページが読み込まれる．

XML ストレージ戦略に特化した場合においても，// のような問合せを効率的に管理するのは困難である．// を含むような参照局所性が低い問合せについては，逆経路索引などの索引手法との連携が不可欠といえる．提案手法においては，この問題に対処するために，アクセスパスの最適化として 5.1 節で述べる逆経路索引を導入する．

5. アクセスパス最適化

提案システムにおいては，DTM に対するアクセスが性能に支障をきたす一部のパス処理（顕著な例として // を含む問合せやステップ数が大きい経路式）について，逆経路索引を用いてアクセスパスを効率化する．

現在用意している索引は，5.1 節で説明する逆経路索引と 2.3 節で言及した XML 木の構造索引である．それぞれ，5.2 節で述べる B+木を利用して索引付けされる．

5.1 逆経路索引

経路索引を構築するうえで，Pal らが提案するコンパクトな逆経路索引[30] をベースとし，名前空間のサポートを加えた．**図 13** に逆経路表記 $ReversePath$ の定義を与える．

パス索引に関して，多くの既存の研究実装において名前空間への対応がなされていないという問題点がある．すべての要素指定アクセスにおいて，名前空間を意識する必要がある XQuery においては拡張が必要である．これについては，Pal らが提案している手法[20] におけるコンパクトな逆経路式中のロケーションステップに QNameTable の QName 識別子を割り当てることで対応した．一般に文書中に現れる QName の重複を除いた総数は限られるため，経路索引のサイズ低減にもつながる．また，Pal らの経路索引においては，// 軸をたどる操作について付加的な操作が必要であったが，Yoshikawa らが提案しているデリミタ[31]

表 1 XMark データに対するパス索引の重複キー割合
Table 1 Duplicate key ratio of path index on XMark dataset.

スケールファクタ（サイズ）	エントリ数	ユニーク数	重複キーの割合
0.1(12 MB)	205,594	536	0.997
1(114 MB)	2,048,193	548	0.999

定義 名前空間に対応する逆経路式

```
ReversePath ::= Step Separator
              | Step Separator ReversePath

Step ::= NameTest | '@' NameTest

NameTest ::= QNameID

QNameID ::= Integer

Separator ::= '/#'
```

図 13 名前空間に対応する逆経路式
Fig. 13 Reverse path expression supporting namespaces.

```
1. for $p in
2.  RewriteInfo → 34/#33/#0/#(a)
3. return
4.    <item person="$p/child::name/child::text()" (e)>
5.    fn:count(
6.    RewriteInfo → 74/#73/#0/#(d)
7.          [$p/@id (c) = child::buyer/@person (b)]
8.    ) </item>
```

図 14 XMark Q8 の擬似アクセスプラン
Fig. 14 Pseudo acceess plan of XMark Q8.

を用いることでパターンマッチのみで // を処理可能とした．

ReversePath に現れる // は，B+木の検索を行う際に “%/” に置き換えて，関係データベースにおける LIKE 述語の処理（LIKE 処理）と同様に，B+木の検索と文字列照合を行う．しかし，単純に “%/” に書き換えただけではタグ名の接尾辞が同一である誤った経路が選択されてしまう可能性がある．ここでは，経路索引における一例をあげて説明する．検索する経路式が /site//description であったときの検索文字列として /site%/description を利用して LIKE 処理を行うと，/sitemap/description についても誤って選択してしまう可能性がある．この問題に対処するために，経路索引ではタグ名の終端記号としてデリミタが利用されている[31),32)]．/site/description を #/site#/description，/sitemap/description を #/sitemap#/description として格納し，/site//description を検索する際には，#/site#%/description として LIKE 処理を行うことで，// を含む問合せを経路索引だけで扱うことができる．

なお，逆経路索引の符号化による圧縮については，関連する文献 33) を参照されたい．

5.2 B+木索引

提案システムにおける B+木の改良点

逆経路索引と XML 木の構造索引には，Simple-Prefix B-trees[34)] に基づく B+木の拡張を用いる．Simple-Prefix B-trees は，内部ノードページのキーとして，B+木をたどるのに必要最小限の情報（Shortest Possible Separator）が保存される B+木の改良である．

提案システムにおいては，ノードページに含まれるキー群に共通する接頭辞を算出し，その接頭辞からの差分を保存することで，ノードページにおけるエントリ充填率を高めた．また，重複キーが存在する場合には，一次記憶上ではキーのインスタンスを共有し，ノードページを二次記憶にページアウトする際には，重複キーの代わりにキーが共有されているかの情報を書き込むようにすることで，ノードページにおけるエントリ充填率と B+木の格納領域の削減を図った．これらの拡張は，重複値の多い経路索引（**表 1**）や接頭辞の共有部分の多い木ラベリングにおいて特に有効である．これらのキーに対しては一般的に索引を付け，キーへのアクセスは索引アクセスが中心となるからである．ラベリング手法，エンコード手法におけるサイズ極小化にとどまらず，索引レベルのキー圧縮もアクセスの効率化に有効である．

なお，B+木においてはキーだけでなく各値のポインタも全体から見て大きな比重を占めることから，提案システムでは 8 バイトのポインタについて Variable-byte Coding[35)] で 1–9 バイトに符号化している．

5.3 索引を用いた問合せ処理

本節では，4.3 節で取り上げた XMark Q7 と Q8 を例に索引を用いた問合せ処理を解説する．

単純経路処理

XMark Q8（図 9）の問合せ処理で問合せ処理器は，最適化フェーズにおいて，次のアクセスパス書き換えを行う．以下，アクセスパスの書き換え後の問合せプランを**図 14** を用いて説明する．

```
1. for $p in RewriteInfo → 0/#(a)
2. return fn:count(
3.   RewriteInfo → 9/%#0/#, filter $p(b) )
4. + fn:count(
5.   RewriteInfo → 65/%#0/#, filter $p(c) )
6. + fn:count(
7.   RewriteInfo → 35/%#0/#, filter $p(d) ) )
```

図 15 XMark Q7 の擬似アクセスプラン
Fig. 15 Pseudo acceess plan of XMark Q7.

(a) /child::site/child::people/child::person を 34/#33/#0/#をキーとする逆経路索引へのアクセスに書き換える．数値は QName の識別子である．

(d) /child::site/child::closed_auctions/child::closed_auction を 74/#73/#0/#をキーとする逆経路索引へのアクセスに書き換える．

問合せ実行時には，(a)，(d) それぞれ逆経路索引をキーとして対応するノードのアドレス（DTM 配列の添え字）を求める．経路索引を用いた場合，通常は，索引に一致した結果アドレスを文書順に並べる必要がある．提案システムでは，バルクロード後に更新がなく，索引の完全一致探索の結果の順序が保証できる場合において，ソート処理を省略する．

ループ中で束縛される変数を持つ経路処理

XMark Q7（図 11）の問合せ処理では，問合せ処理器は最適化フェーズにおいて，次のアクセスパス書き換えを行う．以下，アクセスパスの書き換え後の問合せプランを**図 15** を用いて説明する．

(a) /child::site を 0/#をキーとする逆経路索引を利用したアクセスに書き換える．

(b) /child::site/descendant::description を 9/%#0/#をキーとする逆経路索引を利用したアクセスに書き換える．このとき，結果から$p 変数の子孫であるものだけを抜き出す．

(c) キーを 65/%#0/#とする (b) と同様の処理である．

(d) キーを 35/%#0/#とする (b) と同様の処理である．

Q7 は$auction/site の処理結果シーケンスからアイテムごとに return 句の処理を行う問合せである．return 句にある (b)，(c)，(d) はともに，(a) の結果に依存する．このため，for ループの処理中では$p 変数の値に依存する結果のみを抽出する必要がある．

このフィルタリングはノード ID から取得する木ラベルを用いた構造結合によるが，B+木の性質を利用することで効率的に処理している（以後，このフィルタを**構造フィルタ**と呼ぶ）．そのアルゴリズムを擬似コード（**図 16**）を用いて解説する．

```
IN: idxCond, ancestor   OUT: ptrs

1. global PathIndex;
2. global LabelIndex;

3. Matched := PathIndex.find(idxCond);
4. ptrs := Matched.getMatchedSorted();
5. ptrs := filter(ptrs, ancestor);

6. function filter(ptrs, ancestor) {
7.   if(ptrs.length == 0)
8.     return nil;
9.   ancestorLabel := LabelIndex.getLabel(ancestor);
10.  for(i:=0;i<ptrs.length;i++) {
11.    targetLabel := LabelIndex.getLabel(ptrs[i]);
12.    if(!isDecendantOf(targetLabel, ancestorLabel))
13.      ptrs[i] := nil;
14.  }
15.  return ptrs;
16. }
```

図 16 **Algorithm** 構造フィルタ
Fig. 16 **Algorithm** structural filter.

まず，グローバル変数として定義した PathIndex，LabelIndex 変数はそれぞれ逆経路索引処理モジュール，XML 木の順序示すラベル（Dewey Order）の索引を処理するモジュールを指す．引数の idxCond は逆経路索引に対する問合せを示すオブジェクトであり，ancestor は，図 15 のケースを例にとるとフィルタの基準となるループ変数$p のノード識別子を示す．3 行目の Matched は，逆経路索引に対して問い合わせた結果のオブジェクトであり，4 行目で，結果オブジェクトから文書順にソートしたノード ID 群を取得し，ptrs に代入する．5 行目で呼び出す filter 関数が，構造フィルタの中核処理モジュールである．

filter 関数のアルゴリズムは次のとおりである．

9 行目 ancestor ノードの識別子からそのノードのラベルを取得する．

10〜14 行目 すべての ptrs ノード ID 群について走査．

11 行目 そのラベルを取得し，そのラベルを targetLabel とする．

12 行目 targetLabel と ancestorLabel を比較し，target が ancestor の子孫でないと判定されれば，13 行目へ．

13 行目 結果アドレスを無効化する．

15 行目 フィルタリングされた結果を返す．

ここで，結果ノード ID 群が文書順に整列されてい

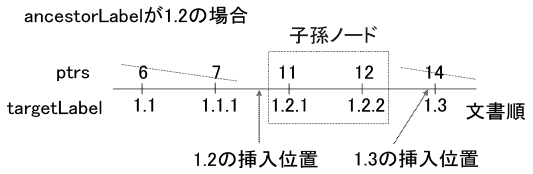

図 17 二分探索を用いた filter 関数処理
Fig. 17 Filter function processing with binary search.

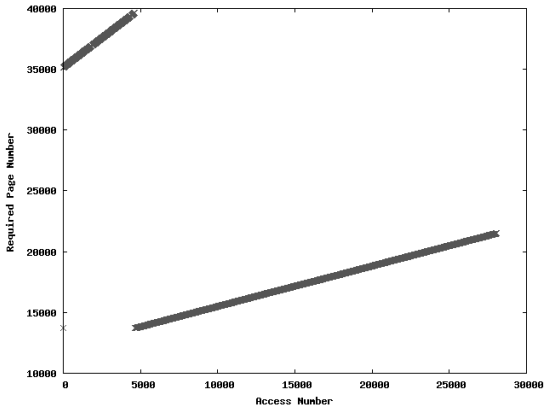

図 18 XMark Q8 におけるページ要求パターン（索引利用時）
Fig. 18 Page access pattern of XMark Q8 with index.

ることが保証される場合には，二分探索の原理で無効化するアドレスを発見し，ancestor の子孫を抽出するという filter 関数の最適化を行う（**図 17**）．図中の例を用いると，ancestorLabel（Dewey ラベル）が 1.2 であれば，1.2 と 1.3 の targetLabel における挿入位置を二分探索で求め，子孫ノード以外の ptrs の要素を無効化する（10〜14 行目の処理に相当）．

5.4 索引を用いた場合のアクセスパターン

本節では，逆経路索引を利用することで，4.3 節で検証した DTM におけるページアクセスパターンがどのように変化したか，どのような傾向を示すかについて説明する．

索引を利用したときの Q8 のアクセスパターンが**図 18** である．提案システムでは，図 14 における内部ループの Join 属性 (b) と Join 属性に対応する値 (d) をあらかじめ計算し，次に外部ループを実行して (c) に対応する値 (d) を取得するという Join 処理をしている．図 18 の見方としては，y 軸の 35,000〜40,000 付近が (b) および (d) に関するアクセスで，y 軸の 14,000〜22,000 付近が (a)，(c) および (e) に関するアクセスである．(b) および (d) に関するアクセス箇所については，索引を利用しない場合の図 10 と比べて，索引を利用した場合の DTM へのアクセス回数が減少している．外部ループのアクセス箇所については索引を利用した場合の方が，y 軸のアクセスされるページ数が若干減っている．これは (a) を索引アクセスにしているためである．DTM へのアクセス回数に大差が表れていないのは，索引を利用していない箇所 (c)，(e) に依存して変数 $\$p$ に束縛されたノード付近のページが読み込まれるためと考えられる．

Q7 については，fn:count 関数の呼び出しを索引に一致したエントリ数から計算するように最適化しているため，索引アクセスだけで問合せ処理が完結する．

CPU	Intel Pentium D 2.8 GHz
OS	SuSE Linux 10.2 (Kernel 2.6.18)
メモリ	2 GB
ディスク	SATA 7200 rpm
Java	Sun JDK 1.6
JVM option	-Xms1400m -Xmx1400m

$-Xms$ は，JVM に初期ヒープサイズを指定するオプションである．
$-Xmx$ は，JVM の上限ヒープサイズを指定するオプションである．

実験環境における提案システムの標準（初期）設定

- DTM のページングバッファのサイズ：128 M バイト
- StringChunk モジュールが利用するキャッシュサイズ：32 M バイト
- 索引(B+木)が利用するキャッシュサイズ：4 M バイト
- B+木のページサイズ：4 K バイト

図 19 実験環境と設定
Fig. 19 Experimental environment and settings.

6. 実　　験

提案する DTM に基づく XML ストレージ手法の有効性を測るために，XML データベースの代表的なベンチマーク指標である XMark [29] を用いて問合せ処理時間の計測を行った．

実 験 環 境

実験環境と実験環境における提案システムの標準設定は，**図 19** に示すとおりである☆．本章で行う実験では特に言及しない限り，この実験環境を用い，実験結果には 3 回の試行結果の平均値を用いる．

6.1 一次記憶に文書全体をロードして処理する場合との比較

提案手法の DTM（索引なし）と我々のシステムが動作するのと同じ Java VM（JVM）上で動作する XQuery プロセッサとして代表的な SAXON-SA [13] バージョン 8.9 を比較し，提案実装についての他の実装と比較した性能面での位置付けを与える．Saxon は TinyTree という XML 木の内部表現を利用している．

☆ 特に指定しない限り，我々の実験環境では Java VM は Client VM が利用される．

表 2 SAXON と提案システムの比較（計測単位：秒）
Table 2 Comparation between our system and Saxon.

	XMark 113 MB (SF: 1)				XMark 340 MB (SF: 3)			
	DTM	DTM^{mem}	pDTM	Saxon	DTM	DTM^{mem}	pDTM	Saxon
Q1	8.82	0.68	1.19	8.34	23.84	0.79	2.08	24.51
Q2	8.59	0.73	1.52	8.09	24.14	0.90	2.71	23.81
Q3	9.24	1.00	1.73	8.63	25.32	1.77	3.85	25.20
Q4	8.95	1.07	2.15	8.70	24.96	1.80	5.06	25.65
Q5	8.76	0.66	1.33	7.99	24.25	0.79	2.72	23.19
Q6	10.05	1.58	3.60	8.08	26.74	3.61	9.78	23.69
Q7	10.95	2.89	6.71	8.29	30.38	7.34	21.92	24.10
Q8	9.03	1.24	2.24	8.98	24.90	2.23	5.30	26.56
Q9	9.24	1.29	5.43	8.86	25.79	2.69	14.98	26.09
Q10	12.48	4.21	15.56	11.51	34.90	11.56	52.13	33.97
Q11	14.68	6.74	8.25	440.38	64.18	41.11	54.50	DNF
Q12	12.88	4.88	6.07	212.34	50.75	27.00	35.57	DNF
Q13	11.67	0.79	1.09	8.33	33.54	0.97	2.02	24.88
Q14	9.81	1.92	6.11	8.70	27.56	4.27	16.93	25.59
Q15	9.13	0.61	0.74	8.19	23.64	0.72	1.26	24.13
Q16	8.66	0.75	0.89	8.23	24.34	0.96	1.53	23.98
Q17	11.62	0.88	1.25	8.08	33.43	1.29	2.61	23.23
Q18	8.95	0.79	1.75	8.32	25.16	1.25	3.72	24.17
Q19	12.47	2.04	5.73	9.46	36.36	4.51	15.49	31.00
Q20	9.37	1.19	2.15	8.43	25.14	2.12	5.19	26.20

TinyTree はリンク情報などからなる複数の配列から構成されるが，論理的に表構造であり，本質的に DTM と同様の構造である．

実験環境において，$-Xmx$ オプションで JVM に割当て可能なヒープ量は 1.4 G バイトが限界であった．この制限のもと，Saxon を評価できたのは XMark におけるスケールファクタ 3 の約 340 M バイトのデータセットまでである．スケールファクタ 4 以上では，Saxon の問合せ実行中にメモリ不足によるエラーが発生する．そこで比較には，XMark スケールファクタ 1（113 M バイトの XML データ）とスケールファクタ 3 を用いた．

実験結果は**表 2** のとおりである．表 2 中の，DNF（Did Not Finish の略称）は 10 分経っても結果が返らないために計測を中止した項目である．DTM は，XML 文書から一次記憶に DTM を構築し，問合せを実行した項目である．DTM の計測結果には，XML データのパース時間と DTM の構築時間を含む．DTM^{mem} は，主記憶に DTM が構築済みの状態での問合せ処理性能を示す項目である．パース時間と DTM の構築時間は，計測項目 DTM から DTM^{mem} を引いたものとなる．pDTM は，あらかじめ二次記憶上に DTM を格納して問合せを実行した項目である．なお，SF はスケールファクタの略称である．

実験結果の考察

計測結果の DTM^{mem} と pDTM との違いに着目する．計測結果から，ディスクからの DTM 断片の読み込み負荷が問合せ性能に与える影響が大きかった問合せは，Q6，Q7，Q9，Q10，Q14，Q19 である．この中の，Q6，Q7，Q14，Q19 は // を含む問合せであり，そのことがページングによる性能の悪化に現れている．こうした // を含む問合せに対しては，逆経路索引を利用することが有効であると考えられる．6.3 節で，索引を利用した場合との比較を与える．残る Q9，Q10 は 20 個の問合せの中で，特異なアクセスパターン（図 7）を示す問合せである．

```
let $auction := fn:doc("auction.xml")
return
  let $ca := $auction/site/closed_auctions/closed_auction
  return
    let $ei := $auction/site/regions/europe/item
                for $p in $auction/site/people/person
                        let $a := for $t in $ca
               where $p/@id = $t/buyer/@person
               return
                              let $n := for $t2 in $ei
                          where $t/itemref/@item = $t2/@id
                          return $t2
              return <item>{ $n/name/text() }</item>
    return <person name="{ $p/name/text() }">{ $a }</person>
```

図 20 XMark 問合せ Q9
Fig. 20 Query No.9 of XMark.

Q9（**図 20**）は，3 重のループ（図 20 の下線部）を含む問合せであるため参照局所性を活かすことが困難であることが原因として考えられる．こうした問合せに対処するには，プログラミング言語のループ最適化で行われるブロック化技法を問合せ最適化に活かすことが有効となる可能性がある．

```
let $auction := fn:doc("auction.xml")
return
  for $i in distinct-values(
    $auction/site/people/person/profile/interest/@category)
  let $p :=
    for $t in $auction/site/people/person
# $i に対して複数の personne が $p に束縛される
    where $t/profile/interest/@category = $i
    return
      <personne>
        <statistiques>
          <sexe>{ $t/profile/gender/text() }</sexe>
          <age>{ $t/profile/age/text() }</age>
          <education>{ $t/profile/education/text() }</education>
          <revenu>{ fn:data($t/profile/@income) }</revenu>
        </statistiques>
        <coordonnees>
          <nom>{ $t/name/text() }</nom>
          <rue>{ $t/address/street/text() }</rue>
          <ville>{ $t/address/city/text() }</ville>
          <pays>{ $t/address/country/text() }</pays>
          <reseau>
           <courrier>{ $t/emailaddress/text() }</courrier>
           <pagePerso>{ $t/homepage/text() }</pagePerso>
          </reseau>
        </coordonnees>
        <cartePaiement>{ $t/creditcard/text() }</cartePaiement>
      </personne>
  return <categorie>{ <id>{ $i }</id>, $p }</categorie>
# $p に束縛された personne[1],..., personne[n] のそれぞれが
# 直列化される．ここで，sexe，age などの $t に関連する箇所の
# 直列化で DTM へのアクセスが発生する．
```

図 21 XMark 問合せ Q10
Fig. 21 Query No.10 of XMark.

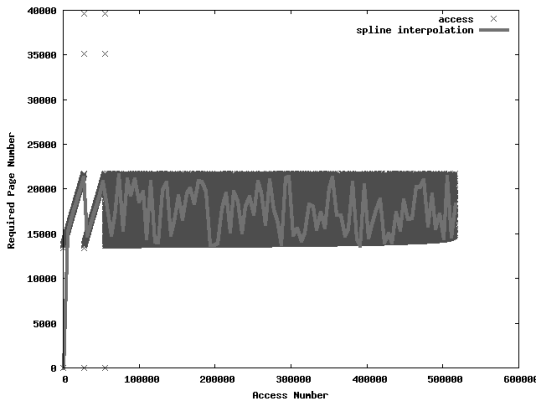

図 22 XMark Q10 のページ要求パターン（索引なし時）
Fig. 22 Page access pattern of XMark Q10.

Q10（**図 21**）では，pDTM に対して Saxon が若干勝る結果が出ている．Q10 の DTM のページ要求パターンを**図 22** に示す．図中の曲線にはアクセスがあった点のスプライン補間をとっている．アクセス番号 50,000 付近からのページ要求が直列化処理に関連するページアクセスである．図 21 の変数$p には複数の personne が束縛される．personne に含まれる$t 群へのアクセスが，外側の$i の繰返しごとに行われることで，右端のアクセスが発生している．

Q10 は，XMark の 20 個の問合せ中で最も出力結果が大きい問合せである．たとえば，スケールファクタ 1 では，XML 文書サイズの 20%弱の 21 M バイトの出力となる．そのため，直列化処理の段階で多数の（ノード）レコードにアクセスが必要となることが性能に影響している．我々の実装においては，3.2.1 項で説明した大きな文字列を主記憶上で圧縮する機能とは別に，StringChunk モジュールにおいてチャンク化した文字列をページアウトする際に，XML 文書をデータベースにロードした後のディスクスペースの消費を抑える目的で LZ 圧縮を施している．このため，問合せ結果のサイズが大きくなる場合に文字列チャンクでの解凍コストが発生していた．スケールファクタ 3 において Q10 をプロファイラを用いて計測したところ，約 3 割が解凍に関するコストであった．文字列の圧縮処理を行うことは，一次記憶と二次記憶の消費効率とのトレードオフである☆が，解凍スピードが速い LZO [36] などのデータ圧縮法を採用することで，文字列チャンクの解凍コストを下げることができる可能性がある．

Q11 と Q12 で Saxon の性能が劣っているのは，これらが Join を含む問合せであるが，Saxon の最適化機構が十分な Join 最適化を行っていないためである．これら Join を含む問合せについては，計算量が大きいため性能に大きな影響が出ている．

バッファ管理上の問題点

実際のディスク読み込みは，バッファ管理に要求するページが存在しない場合のみに発生するが，直列化処理においてはシーケンシャルスキャンが頻繁に発生するため，バッファ管理戦略においてシーケンシャルスキャンに対応する必要がある．シーケンシャルスキャンに対してナイーブな LRU などのバッファ管理手法を用いた場合，バッファ管理中の重要なページが追い出されてしまうからである．6.4 節で，バッファ管理手法とバッファサイズによる影響について考察を与える．

6.2 Natix との性能比較

Natix [8] との性能比較を行う☆☆．Natix は C++で実装された XML ネイティブデータベースで，XPath 1.0 による XML 問合せ処理機能を提供している☆☆☆．利用した Natix のバージョンは 2.1.1 である．今回，

☆ XML 文書を主記憶上に構築した場合，Saxon は提案システムと比較して 2 倍程度の領域を消費する．
☆☆ NoK [9] のプロトタイプ実装は parent-child 軸アクセスのみの問合せしか評価できないため，比較対象として不適である．
☆☆☆ XPath 1.0 は問合せ結果がセットであるのに対して，我々の実装が提供する XQuery 1.0/XPath 2.0 では問合せ結果は（順序が保障される）シーケンスとなる．そのため，各ストレージ手法の特徴を議論する目的の参考までの比較である．

表 3 XPath に変換した XMark 問合せ
Table 3 XMark queries converted into XPath.

Q1	/site/people/person[@id = “person0”]/name/text()
Q2	/site/open_auctions/open_auction/bidder[1]/increase/text()
Q4	/site/open_auctions/open_auction[bidder[personref/@person = “person20”]/following-sibling::bidder [personref/@person = “person51”]]/reserve/text()
Q5	/site/closed_auctions/closed_auction[price/text() >= 40]/price
Q6	count(//site/regions//item)
Q7	count(/site//description | /site//annotation | /site//emailaddress)
Q14	/site//item[contains(description, “gold”)]/name/text()
Q15	/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()
Q16	/site/closed_auctions/closed_auction[annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()] /seller/@person
Q17	/site/people/person[homepage/text()]/name/text()

表 4 Natix との XPath 問合せ処理性能比較
Table 4 Performance comparison of XPath query processing with Natix.

	XMark 568 MB (SF5)					XMark 1.1 GB (SF10)				
	出力 (KB)	pDTM^c	pDTM^s	pDTM^i	Natix	出力 (KB)	pDTM^c	pDTM^s	pDTM^i	Natix
Q1	1	2.56	2.96	2.54	1.84	1	4.60	5.02	4.44	3.22
Q2	241	3.14	3.43	3.14	4.11	482	7.68	7.25	6.14	10.64
Q4	0	7.73	7.53	7.80	4.70	0	15.38	13.60	15.49	9.03
Q5	666	4.12	4.61	4.14	7.32	1322	7.72	8.01	8.03	9.46
Q6	1	15.57	12.17	0.46	11.99	1	34.51	27.55	1.82	118.30
Q7	1	33.88	25.68	3.23	50.80	1	73.92	54.45	20.74	621.83
Q14	149	48.39	34.49	16.10	28.26	297	106.90	72.68	36.48	155.24
Q15	42	1.67	2.08	0.89	1.20	80	4.00	3.73	2.38	2.39
Q16	10	1.94	2.10	1.94	1.56	20	3.48	3.49	3.69	2.58
Q17	897	3.07	3.25	3.18	4.47	1789	6.71	6.25	7.09	13.13

我々が実験した範囲では Natix は索引をいっさい利用せずに，すべて巡航アクセスを行う．XQuery はサポートされていないため，比較には XMark のデータセットを用いた XPath ベンチマークである XMark の XQuery 問合せから XPath 1.0 に変換可能な 10 個の問合せを抽出し，**表 3** の XPath 問合せを作成し，利用した．問合せの番号は XMark の XQuery 問合せ番号に準ずる．比較に用いるデータセットとしては，XMark スケールファクタ 5 の 568 M バイトの XML 文書とスケールファクタ 10 の 1.1 G バイトの XML 文書を利用した．

実験結果は**表 4** のとおりである．表 4 の pDTM^c，pDTM^s，pDTM^i は，実行時の設定として，それぞれ索引を利用せずに JVM に Client VM を用いた場合，索引を利用せずに JVM に Server VM を用いた場合，索引を利用して JVM に Client VM を用いた場合である．pDTM^s は比較対象がネイティブアプリケーションであるため，提案手法の実行環境である JVM が性能に与える影響についての検討材料として導入した項目である．Natix は索引を利用しないため，pDTM^i は参考までの値である．

スケールファクタ 5 と 10 の実験結果をそれぞれグラフ化したものが，**図 23** と**図 24** である．グラフの

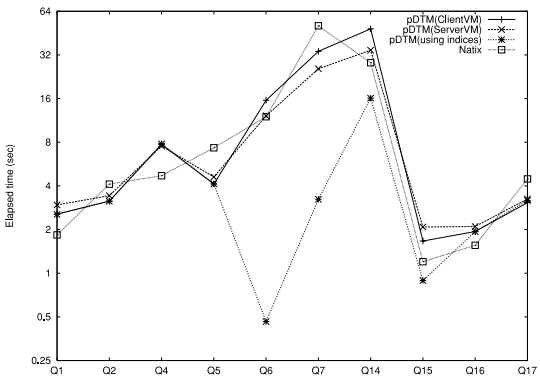

図 23 XMark SF5 のデータセットに対する XPath 問合せ性能
Fig. 23 Performance of XPath Queries on XMark SF5 dataset.

縦軸の実行時間（秒）は，底を 2 とした log スケールをとっている．

実験結果の考察

プログラミング言語と実行形態に違いがあるため，実行時間の短い問合せの場合，提案手法の良し悪しよりも実装の完成度やプログラミング言語の実行形態の差が現れる可能性がある．そこで，実験結果の差が大きい箇所に注目して考察を与える．ここでは，Q4，Q6，Q7，Q17 に注目し，考察を与える．

表 5 pDTM のスケール特性（計測単位：秒）
Table 5 Scalability of pDTM.

	113 MB (SF: 1)		340 MB (SF: 3)		568 MB (SF: 5)		1.1 GB (SF: 10)	
	索引なし	索引あり	索引なし	索引あり	索引なし	索引あり	索引なし	索引あり
Q1	1.19	1.12	2.08	1.98	2.59	2.55	4.86	5.09
Q2	1.52	1.26	2.71	2.58	3.44	3.43	9.82	13.50
Q3	1.73	1.73	3.85	3.86	4.67	4.61	10.76	10.59
Q4	2.15	2.17	5.06	5.06	7.99	7.91	15.55	15.34
Q5	1.33	1.38	2.72	2.74	3.90	3.96	7.56	8.37
Q6	3.60	0.49	9.78	0.62	15.54	0.49	33.08	2.35
Q7	6.71	0.81	21.92	1.63	36.87	1.02	75.84	10.24
Q8	2.24	2.28	5.30	5.19	8.61	8.08	16.78	15.55
Q9	5.43	5.52	14.98	14.81	24.52	24.41	50.61	57.46
Q10	15.56	15.47	52.13	51.51	88.17	87.14	185.56	186.77
Q11	8.25	8.23	54.50	50.95	132.92	140.97	513.48	509.91
Q12	6.07	6.11	35.57	35.71	90.30	89.16	329.21	323.21
Q13	1.09	1.12	2.02	2.02	2.86	2.99	6.04	6.57
Q14	6.11	3.85	16.93	10.03	28.11	16.87	58.50	36.69
Q15	0.74	0.59	1.26	0.76	1.77	0.96	4.01	4.18
Q16	0.89	0.91	1.53	1.56	2.18	2.18	4.06	5.25
Q17	1.25	1.33	2.61	2.52	3.70	3.67	7.30	6.78
Q18	1.75	1.77	3.72	3.76	5.68	5.72	11.91	13.18
Q19	5.73	5.05	15.49	13.43	25.38	21.92	64.42	57.26
Q20	2.15	1.88	5.19	4.19	8.01	6.45	17.18	16.35

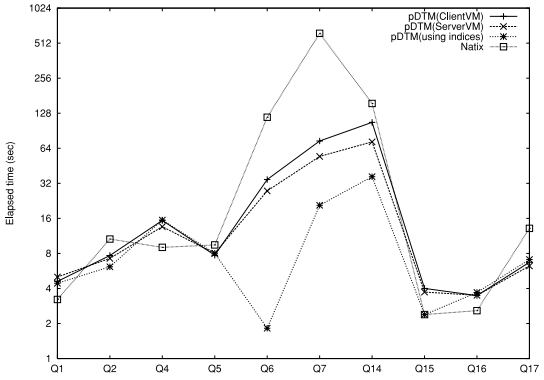

図 24 XMark SF10 のデータセットに対する XPath 問合せ性能
Fig. 24 Performance of XPath Queries on XMark SF10 dataset.

Q4 は following-sibling 軸を含む問合せであるが，この問合せではスケールファクタ 5，10 の場合ともに Natix が勝っている．この原因として，我々の提案手法が文書順でのブロック配置をするのに対して，Natix が部分木ベースのブロック配置をとっていることが影響していると考えられる．Natix では子ノードがまとめられているため，following-sibling の実行効率が良い．我々のシステムでは，XQuery UseCase [14] に現れる軸処理は child 軸と descendant(-or-self) 軸へのアクセスが中心であり，child 軸と descendant 軸へのアクセスが特に性能に与える影響が大きいという観点から，following-sibling 軸の処理には必ずしも適さない文書順の配置をとっている．

スケールファクタ 10 のときに，Q6 および Q7 で大きな性能差が現れており，我々の提案手法が Natix に勝る性能を出している．Q6，Q7 は // を含む問合せであり，再帰的に子孫ノードのブロック読み込みを行うため，ブロック読み込みの性能差が前面に現れていると考える．出力結果のサイズが相対的に大きいスケールファクタ 10 の Q17（および Q5）について，我々の提案手法が勝る性能を出しているのは，直列化処理の効率が Natix に比較して高いことが考えられる．

6.3 異なるデータサイズに対する性能比較

表 5 は，二次記憶に格納した DTM に対して，XMark のスケールファクタ 1（113 M バイト），スケールファクタ 3（340 M バイト），スケールファクタ 5（568 M バイト），スケールファクタ 10（1.1 G バイト）のデータセットについての計測結果をまとめたものである．図中の計測項目 x3，x5，x10 はそれぞれ，スケールファクタ 1 の計測値を 3 倍，5 倍，10 倍したものを示す．

まず，索引なしの DTM に着目してみると，Q11，Q12 を除いてデータサイズの増加に対して，ほぼ線形の性能が出ていることが分かる（**図 25**）．Q11，Q12 の性能が線形に向上しないのは，Q11，Q12 は θ 結合を含む問合せであるが，提案システムの実装において θ 結合の性能が十分ではないことが考えられる．Q11，Q12 に対しては Saxon の処理性能も高くなく（表 2），これらの問合せを処理する我々のシステムの実装が相対的に著しく劣っているわけではないが，θ 結合の性能改善は今後の課題である．

表 6 バッファサイズが性能に与える影響
Table 6 Influence of buffer-size to the performance.

	出力	索引なし（BUF：64 M）			索引なし（BUF：128 M）			索引なし（BUF：256 M）		
	(KB)	処理時間	ブロック数	置換回数	処理時間	ブロック数	置換回数	処理時間	ブロック数	置換回数
Q1	1	4.62	80,737	38,138	4.86	80,673	0	4.67	80,673	0
Q2	3,425	10.89	133,889	91,290	9.82	133,889	48,692	6.74	133,825	0
Q3	1,688	10.80	134,849	92,249	10.76	134,849	49,651	10.46	134,817	0
Q4	0	15.63	134,881	92,282	15.55	134,881	49,684	15.76	134,817	0
Q5	1	7.50	45,082	2,483	7.56	45,018	0	7.59	45,018	0
Q6	1	33.25	531,912	489,311	33.08	531,784	446,586	31.94	397,329	223,512
Q7	1	75.31	1,192,181	1,149,579	75.84	1,191,893	1,106,693	76.57	1,191,667	1,021,269
Q8	9,543	17.60	125,723	83,124	16.78	125,626	40,428	17.27	125,530	0
Q9	12,067	64.54	1,651,866	880,565	50.61	369,574	123,598	48.80	163,002	0
Q10	250,181	208.38	1,731,362	835,627	185.56	80,673	0	183.72	80,673	0
Q11	9,974	521.52	215,522	172,923	513.48	215,425	130,216	514.12	215,329	44,900
Q12	1,773	329.65	215,522	172,923	329.21	215,457	130,215	326.32	215,361	44,906
Q13	62,002	6.11	13,889	0	6.04	13,889	0	5.71	13,889	0
Q14	594	57.62	397,601	354,815	58.50	397,505	312,121	59.14	397,409	226,827
Q15	104	4.06	45,090	2,491	4.01	45,026	0	3.98	45,026	0
Q16	57	4.36	45,090	2,491	4.06	45,026	0	3.98	45,026	0
Q17	4,876	6.99	80,737	38,138	7.30	80,673	0	8.13	80,673	0
Q18	694	12.11	134,881	92,282	11.91	134,881	49,684	12.56	134,817	0
Q19	10,814	80.91	2,500,737	2,040,281	64.42	1,400,289	718,922	50.78	134,913	0
Q20	1	20.15	322,756	280,157	17.18	80,673	0	16.61	80,673	0

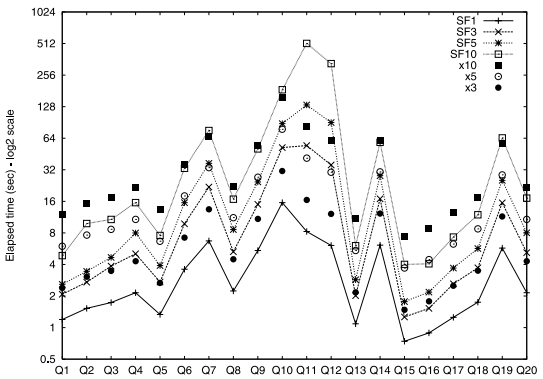

図 25 pDTM のスケール特性
Fig. 25 Scalability of pDTM.

スケールファクタ 5 までの計測項目については，おおむね索引ありの DTM が良好な性能を示すが，スケールファクタ 10 では索引を利用した方がむしろ性能が悪化しているケースが見られ，アクセスプランの選択と索引アクセスの実行効率に課題を残した．一方で，逆経路索引の導入理由となった // を含む問合せ Q6，Q7，Q14，Q19 については，すべて索引を利用した場合の性能が，索引を利用しない場合の性能に勝ることから，参照局所性の低い問合せに対して逆経路索引が有効であることが確認できた．

6.4 バッファ管理が性能に与える影響

我々の提案システムで利用できるバッファ管理手法は，**LRU** と **2Q** [28] である．特に指定しない場合，バッファ管理手法として 2Q が利用される．

バッファサイズが性能に与える影響の考察

バッファサイズが性能に与える影響を計測した結果が**表 6** である．表 6 は，XMark SF10 に対する XQuery 問合せを索引を用いずに処理する場合の (1) 処理に要した時間，(2) 読み込まれたブロック数，(3) ページの置換回数をまとめた表である．DTM に関連するバッファサイズはシステムのデフォルト設定では，128 M バイトである．ここでは，DTM に関連するバッファサイズを 64 M バイト，128 M バイト，256 M バイトと設定した際の各計測結果を示す．

実験結果から，バッファサイズの増減が性能に顕著な影響を及ぼしていた問合せ Q2，Q9，Q10，Q19（表 6 の下線部）に注目し考察を与える．下線部は，バッファサイズの増減によって，処理時間が 2 割以上変動した項目と処理時間が 10 秒以上変動した項目に着目したものである．

Q19 はバッファサイズの増加が，そのまま性能の向上に現れていた項目である．Q19 ではバッファサイズの増加によって，ブロックの読み込み数が減ることで性能の向上が見られた．Q9 と Q10 については，バッファサイズを 64 M バイトから 128 M バイトと変更したときにはブロックの読み込み数が大きく減り，性能の向上が得られた一方で，バッファサイズを 128 M バイトから 256 M バイトと変更したことで得られる性能の向上は軽微であった．Q10 については，バッファサイズを 128 M バイトとしたときに，すでにページの置換がなくなっていることから，128 M バイトが

表 7 バッファ管理アルゴリズムに LRU と 2Q を利用した場合の比較
Table 7 Comparison between LRU and 2Q for the use as the buffer management algorithm.

	索引なし（BUF：LRU 128 M）			索引なし（BUF：2Q 128 M）			性能向上
	処理時間	ブロック数	置換回数	処理時間	ブロック数	置換回数	(%)
Q1	4.73	80,737	15,201	4.86	80,673	0	−2.7
Q2	11.24	133,889	68,353	9.82	133,889	48,692	14.4
Q3	11.93	134,849	69,313	10.76	134,849	49,651	10.9
Q4	16.60	134,881	69,345	15.55	134,881	49,684	6.7
Q5	7.86	45,018	0	7.56	45,018	0	3.9
Q6	40.75	531,816	466,280	33.08	531,784	446,586	23.2
Q7	91.29	1,191,989	1,126,453	75.84	1,191,893	1,106,693	20.4
Q8	17.54	125,627	60,091	16.78	125,626	40,428	4.5
Q9	60.16	1,302,329	527,197	50.61	369,574	123,598	18.9
Q10	211.83	1,919,586	567,702	185.56	80,673	0	14.2
Q11	515.29	215,522	149,986	513.48	215,425	130,216	0.4
Q12	330.72	215,522	149,986	329.21	215,457	130,215	0.5
Q13	5.95	13,889	0	6.04	13,889	0	−1.5
Q14	65.81	397,537	331,815	58.50	397,505	312,121	12.5
Q15	4.63	45,026	0	4.01	45,026	0	15.7
Q16	4.47	45,026	0	4.06	45,026	0	10.1
Q17	7.54	80,737	15,201	7.30	80,673	0	3.3
Q18	12.61	134,881	69,345	11.91	134,881	49,684	5.9
Q19	65.19	2,020,193	1,429,137	64.42	1,400,289	718,922	1.2
Q20	21.22	322,756	257,220	17.18	80,673	0	23.5

割当てバッファサイズとして十分であったと考えられる．Q9 では，バッファサイズが 128 M バイトのときに 123,598 回のページ置換が発生しており，バッファサイズを 256 M バイトとすることでページの置換回数は減ったが，大きな性能の向上は得られていない．この理由として，バッファサイズが 128 M バイトのときに，バッファ管理により不要と判断されパージされたページが，再び呼び出されることがなかったためと考えられる．Q2 は，右上がりにほぼ一直線のアクセスパターンを示す問合せであり，一度読み込まれたページが再び読み込まれることが稀な問合せである．そのため，バッファサイズを 64 M バイトから 128 M バイトとしたときには読み込みページ数に変化が現れない．バッファサイズを 128 M バイトから 256 M バイトとしたときに性能の向上が見られたのは，読み込みページ数が若干減ったことと，割込みのパージ処理がまったく起らなかったことが一因として考えられる．

バッファ管理手法が性能に与える影響の考察

我々の提案システムで採用している 2Q[28] は，シーケンシャルスキャンに対して強固なバッファ管理アルゴリズムとして，近年では PostgreSQL でも採用されている．LRU は，シーケンシャルスキャンによって重要なページが追い出されてしまうという問題が知られている．2Q は，シーケンシャルに参照されたブロックはすぐに追い出し，ループ参照されるものは長く残すというページ置換ポリシがとられている．そのため，シーケンシャルかつループ参照される Q10 のようなアクセスパターン（図 22）に対しても強固である．XML 問合せ処理では，直列化処理や文字列値の計算などでシーケンシャルスキャンが頻繁に発生するため，シーケンシャルスキャンに対して強固なバッファ管理を行うことが重要である．

そこで，XMark スケールファクタ 10 を利用して，提案システムのバッファ管理手法に LRU を用いた場合と 2Q を用いた場合の比較を行う．比較には，DTM に関連するバッファサイズとして 128 M バイトを利用した．

実験結果が**表 7** である．表 7 では，(1) バッファ管理に LRU を採用した際の問合せ処理時間，(2) バッファ管理に 2Q を採用した際の問合せ処理時間，(3) 2Q を採用することによる LRU に対する性能向上比（%）をそれぞれ示す．

実験結果のうち，バッファ管理手法の違いが性能に顕著に影響を及ぼしていたのは，Q6，Q7，Q9，Q10，Q20 などであった．ここでは，図 22 でアクセスパターンの分析を行った Q10（図 21）に注目して考察を行う．

Q10 は図 22 のアクセス点に現れているように頻繁なシーケンシャルスキャンが発生する問合せである．一方で，スプライン曲線に現れているように，本来バッファが一杯になった際に追い出すページは，必ずしも最近利用されなかったページではない．表 7 で，2Q におけるページ置換回数は 0 である．2Q では，一度アクセスしたページは必ずバッファにヒットしている

一方で，LRU を利用した場合は置換回数が 567,702 あり，シーケンシャルスキャンにより重要なページのいくつかが追い出されていることが分かる．

Q1，Q13 で 2Q を利用した場合の性能が LRU に若干劣るのは，我々の提案システムで採用している (full-) 2Q [28] に，LRU と比較して若干のオーバヘッドが存在するためと考えられる．

以上から，XML 問合せ処理において，シーケンシャルスキャンに強固なバッファ管理を行うことの重要性が確認された．ページ利用効率はメモリ資源が限られている場合，特に重要となる．

6.5 割当てヒープ量による影響

ページングを独自に管理する提案アプローチでは，少ないメモリ容量においても動作が可能なことが利点である．ここでは，XMark のスケールファクタ 10（1.1 G バイト）において，JVM のヒープ限界値として 1,400 M バイト，256 M バイト，128 M バイト，64 M バイトをそれぞれ与えたときの性能を比較した．ヒープの限界値を考慮したバッファに利用する領域として，それぞれ，160 M バイト（提案システムの標準設定である DTM のページング用：128 M バイト，StringChunk のキャッシュ：32 M バイト），96 M バイト（DTM のページング用：64 M バイト，StringChunk のキャッシュ：32 M バイト），48 M バイト（DTM のページング用：32 M バイト，StringChunk のキャッシュ：16 M バイト），20 M バイト（DTM のページング用：12 M バイト，StringChunk のキャッシュ：8 M バイト）を設定した．

実験では索引なしの DTM を用い，二次記憶に配置した DTM に対する XML 問合せ性能を計測した．実験結果を示したのが，**表 8** である．図中の ERR はメモリ不足のエラーが発生した項目である．

JVM へのヒープ割当て量を 64 M バイトとした状況であっても，1.1 G バイトの XML 問合せ処理を，Q10 と Q19 を除く 18/20 の問合せで実行することができた．また，この中で Q9 を除いた 17 個の問合せについては，ヒープ割当て量を 1,400 M バイトとした場合と比べても大きな性能差が開くことなく問合せを実行することができている．この理由としては，4.3 節で考察を与えたように，提案システムで XMark の問合せを処理するうえでは，文書順のアクセスパターンが多く現れるが，我々のシステムが実際のアクセスパターンの分析から予想される文書順のアクセスパターンに対して強固な XML ノード配置手法をとっていることと，シーケンシャルスキャンに強固なバッファ管理が有効に働いていることによる．

表 8 JVM へのヒープ割当て量を限定した場合（計測単位：秒）
Table 8 Testcase on limited JVM heap space.

	$1400\mathrm{m}_{160}^{buf}$	$256\mathrm{m}_{96}^{buf}$	$128\mathrm{m}_{48}^{buf}$	$64\mathrm{m}_{20}^{buf}$
Q1	4.86	4.75	4.84	4.70
Q2	9.82	6.68	6.66	7.84
Q3	10.76	10.75	10.64	11.98
Q4	15.55	15.96	15.91	15.86
Q5	7.56	7.46	7.59	7.43
Q6	33.08	33.10	33.31	33.51
Q7	75.84	72.79	73.31	74.21
Q8	16.78	16.31	16.83	17.44
Q9	50.61	69.45	114.74	442.98
Q10	185.56	221.63	DNF$^{(a)}$	ERR$^{(b)}$
Q11	513.48	559.69	555.32	553.55
Q12	329.21	351.68	354.15	353.93
Q13	6.04	5.94	5.35	5.18
Q14	58.50	59.78	56.63	56.70
Q15	4.01	3.43	3.14	3.03
Q16	4.06	4.09	4.06	3.82
Q17	7.30	7.20	7.26	6.84
Q18	11.91	11.16	11.14	10.99
Q19	64.42	ERR$^{(c)}$	ERR	ERR
Q20	17.18	19.06	18.92	18.78

Q9，Q10 は 20 個の問合せの中では，特異なアクセスパターンを示す問合せである．Q10 は出力が大きい問合せであるが，文字列バッファを限定した場合 (a) に計算時間が急増しているのは，文字列チャンクのキャッシュが限定されるために，文字列チャンクの解凍コストが高くなっているものと予想される．(b) でメモリ不足となったのは，主記憶上に保存している Join 表が割当て可能なヒープ領域を枯渇させたことが原因である．Join 表が大きくなると予想される際には，二次記憶へデータの退避を行うか，あるいは B+ 木を利用した Join 処理など，アクセスプランを工夫する必要がある．Q9 は，6.1 節で考察したように，多重のループを含む問合せであるため参照局所性を十分に活かせなかったことが原因として考えられる．

Q19 については，256 M バイトのヒープを利用した場合 (c) から，メモリ不足エラーが発生した．これは，Q19 がソート処理を含む問合せであったが，提案システムの実装において外部ソートが未実装であったためである．Q19 については外部ソートを実装することで，メモリ不足エラーを回避する必要がある．XML 問合せの中間結果はスカラー値ではなくノードであることもあるため，外部ソートを適用する際には，ディスクへの出力方法に工夫が必要となる．

7. ま と め

本稿では，DTM という論理表に基づく XML データの二次記憶への格納手法を提案した．鍵となるのは，

エクステントを利用することで必要とされたページに対して，文書順のアクセスパターンに適した先読みを行うこと，そして逆経路索引を利用することで局所性の低い問合せにも対応したことである．実験により，提案手法がデータサイズに対してほぼ線形の性能が得られることがあることを示し，限られたメモリ環境下における提案手法の有効性を明らかにした．XQuery 問合せ処理を実行するうえで，ボトルネックとなるのは従来指摘されてきた構造結合処理やパス処理ばかりでなく，直列化処理もその 1 つであることを示した．

今後の課題として，アクセスパターンの動的分析によって得られた情報により，プリフェッチやバッファ管理戦略を切り替えるといったデータベースの自動最適化を行うことや，XQuery Update [37] の更新処理への対応があげられる．XQuery Update は現在のところ仕様策定段階であるが，トランザクション隔離レベルとしてスナップショット隔離レベルが想定されている．そのため，スナップショット隔離レベルをサポートするための DTM の構造について検討を進めていきたい．

謝辞 Natix の実験を遂行するうえで多くの助言を与えてくれた Mannheim 大学 Brantner 氏に感謝する．本研究の一部は，科学技術振興機構戦略的創造研究推進事業（CREST）「情報社会を支える新しい高性能情報処理技術」ならびに科学研究費基盤研究費（A）(2)（課題番号：15200010），若手研究（B）（課題番号：17700109），文科省の大都市大震災軽減化特別プロジェクト（代表：河田惠昭），情報処理振興機構（IPA）平成 17 年度下期未踏ソフトウェア創造事業の支援による．ここに記して謝意を表す．

参 考 文 献

1) OASIS: Open Document Format for Office Applications (OpenDocument). http://www.oasis-open.org/committees/office/
2) 吉川正俊：データベースの観点から見た XML の研究，日本学術会議 2002 年情報学シンポジウム講演論文集 (2002).
3) Beyer, K., Cochrane, R.J., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G., Lyle, B., Özcan, F., Pirahesh, H., Seemann, N., Truong, T., der Linden, B.V., Vickery, B. and Zhang, C.: System RX: one part relational, one part XML, *Proc. SIGMOD conference*, pp.347–358 (2005).
4) 天笠俊之，植村俊亮：リージョンディレクトリを用いた関係データベースによる大規模 XML データ処理，日本データベース学会 Letters，Vol.3, No.2, pp.33–36 (2004).
5) 中島 哲，小田切淳一，井谷宣子，吉田 茂：XML 高速処理技術 SPlitDOM の機能拡張と評価，日本データベース学会 Letters，Vol.3, No.1, pp.125–128 (2004).
6) World Wide Web Consortium: XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/
7) Amer-Yahia, S., Cho, S., Lakshmanan, L.V.S. and Srivastava, D.: Minimization of tree pattern queries, *SIGMOD Record*, Vol.30, No.2, pp.497–508 (2001).
8) Kanne, C.-C. and Moerkotte, G.: Efficient Storage of XML Data, *Proc. ICDE*, p.198 (2000).
9) Zhang, N., Kacholia, V. and Özsu, M.T.: A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML, *Proc. ICDE*, pp.54–65 (2004).
10) Meng, X., Luo, D., Lee, M.-L. and An, J.: OrientStore: A Schema Based Native XML Storage System, *Proc. VLDB*, pp.1057–1060 (2003).
11) Tian, F., DeWitt, D.J., Chen, J. and Zhang, C.: The Design and Performance Evaluation of Alternative XML Storage Strategies, *SIGMOD Record*, Vol.31, No.1, pp.5–10 (2002).
12) Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M.J. and Sundararajan, A.: The BEA streaming XQuery processor, *VLDB Journal*, Vol.13, No.3, pp.294–315 (2004).
13) Saxonica Ltd: Saxon. http://www.saxonica.com/
14) World Wide Web Consortium: XML Query Use Cases. http://www.w3.org/TR/xquery-use-cases/
15) Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N. and Srivastava, D.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching, *Proc. ICDE*, pp.141–152 (2002).
16) Li, Q. and Moon, B.: Indexing and Querying XML Data for Regular Path Expressions, *Proc. VLDB*, pp.361–370 (2001).
17) Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C.: Storing and querying ordered XML using a relational database system, *Proc. SIGMOD conference*, pp.204–215 (2002).
18) Amagasa, T., Yoshikawa, M. and Uemura, S.: QRS: A Robust Numbering Scheme for XML Documents, *Proc. ICDE*, pp.705–707 (2003).
19) Chen, Y., Davidson, S.B. and Zheng, Y.: BLAS: An efficient XPath processing system.

20) O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G. and Westbury, N.: ORDPATHs: insert-friendly XML node labels, *Proc. SIGMOD conference*, pp.903–908 (2004).
21) Böhme, T. and Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS, *Proc. DIWeb*, pp.70–81 (2004).
22) Kobayashi, K., Liang, W., Kobayashi, D., Watanabe, A. and Yokota, H.: VLEI code: An Efficient Labeling Method for Handling XML Documents in an RDB, *Proc. ICDE*, pp.386–387 (2005).
23) Haustein, M.P., Härder, T., Mathis, C. and Wagner, M.: DeweyIDs — The Key to Fine-Grained Management of XML Documents, *Proc. SBBD*, pp.85–99 (2005).
24) Meier, W.: Index-driven XQuery processing in the eXist XML database, XML Prague (2006).
25) Apache Software Foundation: The Apache Xalan Project. http://xml.apache.org/xalan-j/
26) 油井 誠，宮崎 純，植村俊亮：効率的な XQuery 処理のための DTM に基づく XML ストレージ，電子情報通信学会技術研究報告，Vol.196, No.148, pp.73–78 (2006).
27) Sweeney, A.: Scalability in the XFS File System, *Proc. USENIX Annual Technical Conference*, pp.1–14 (1996).
28) Johnson, T. and Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, *Proc. VLDB*, pp.439–450 (1994).
29) Schmidt, A.R., Waas, F., Kersten, M.L., Florescu, D., Manolescu, I., Carey, M.J. and Busse, R.: The XML Benchmark Project, Technical Report INS-R0103, CWI (2001).
30) Pal, S., Cseri, I., Schaller, G., Seeliger, O., Giakoumakis, L. and Zolotov, V.V.: Indexing XML Data Stored in a Relational Database, *Proc. VLDB*, pp.1134–1145 (2004).
31) Yoshikawa, M., Amagasa, T., Shimura, T. and Uemura, S.: XRel: A path-based approach to storage and retrieval of XML documents using relational databases, *ACM Trans. Internet Technology* (*TOIT*), Vol.1, No.1, pp.110–141 (2001).
32) Jiang, H., Lu, H., Wang, W. and Yu, J.X.: Path Materialization Revisited: An Efficient Storage Model for XML Data, *Proc. Australasian Database Conference* (*ADC*) (2002).
33) 油井 誠，宮崎 純，植村俊亮：DTM に基づく XML データベースのための逆経路索引，第 18 回データ工学ワークショップ（DEWS2007）論文集，電子情報通信学会 (2007).
34) Bayer, R. and Unterauer, K.: Prefix B-Trees, *ACM Trans. Database Systems* (*TODS*), Vol.2, No.1, pp.11–26 (1977).
35) Witten, I.H., Moffat, A. and Bell, T.C.: *Managing Gigabytes*, 2nd ed., Mrogan Kaufmann (1999).
36) oberhumer.com GmbH: LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/
37) World Wide Web Consortium: XQuery Update Facility. http://www.w3.org/TR/xqupdate/

(平成 18 年 12 月 21 日受付)
(平成 19 年 4 月 12 日採録)

(担当編集委員 佐藤 哲司)

油井 誠（学生会員）

2003 年芝浦工業大学工学部工業経営学科卒業．同年（株）NEC 情報システムズ入社．グリッド・コンピューティングの研究開発に従事．2006 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．同年 10 月より同大学情報科学研究科博士後期課程入学．現在に至る．XML と大規模データベース処理，DBPL に興味を持つ．平成 15 年情報処理学会論文賞．日本データベース学会会員．

宮崎 純（正会員）

奈良先端科学技術大学院大学情報科学研究科准教授．1992 年東京工業大学工学部情報工学科卒業．1997 年北陸先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（情報科学）．同大学助手を経て，2003 年より現職．2003〜2007 年科学技術振興機構さきがけ研究員（兼務）．2000〜2001 年テキサス大学アーリントン校客員研究員．高性能・高機能データ工学システムの研究に従事．電子情報通信学会，日本データベース学会，IEEE CS，ACM SIGMOD 各会員．

植村　俊亮（フェロー）

奈良産業大学情報学部情報学科教授．1964 年京都大学大学院工学研究科修士課程修了．同年電気試験所（現産業技術総合研究所）．マサチューセッツ工科大学電子システム研究所客員研究員，東京農工大学教授，奈良先端科学技術大学院大学教授を経て，2007 年から現職．データ工学，データベースシステムの研究に従事．工学博士．IEEE Fellow，電子情報通信学会フェロー．