DBSJ Journal, Vol. 7, No. 3
December 2008

効率のよい子供／子孫問合せ処理のためのグラフデータ格納手法の提案

Proposal of a Graph Data Storage for Efficient Processing of Child/Descendant Queries

只石 正輝♡　森嶋 厚行♢　田島 敬史♠

Masateru TADAISHI　Atsuyuki MORISHIMA　Keishi TAJIMA

今日，XML や RDF データ等の普及により，エッジラベル付き有向グラフに対する効率的な問合せ処理が重要な問題となっている．本論文では，大規模なグラフに対する正規パス式の演算を実現するための重要な構成要素である子供/子孫問合せに注目し，それらの問合せを効率的に処理するためのグラフ格納方式を提案する．

Today, we have a large amount of XML and RDF data, and efficient processing of queries against edge-labeled directed graphs is important. This paper proposes a graph storage scheme for efficient processing of child/descendant queries, which are important primitives in regular path queries.

1. はじめに

本論文では，エッジラベル付き大規模グラフデータに対する正規パス式を効率よく評価するためのディスク上でのデータ格納方式について議論する．XML や各種オントロジデータなど，近年はエッジラベル付きの大規模グラフデータの扱いが重要となっている．また，正規パス式は半構造データ言語等の文脈で議論されてきたが，近年 XML データに対しても Regular XPath 式という形で再び注目されており [1]，その効率よい処理が求められている．

一般に，正規パス式は，子要素を求める問合せと，閉包を求める問合せに展開できる．本論文では閉包に関しては特に需要の大きいと考えられる「特定ラベルの繰り返し」のみに焦点を当て，下記の 2 つの演算及びその連結を扱う．

(1) 子供演算：$a \xrightarrow{l} X$: ノード a から，ラベル l を持つエッジを 1 回辿ることで到達可能なノード集合．

(2) 子孫演算：$a \xrightarrow{l*} X$: ノード a から，ラベル l を持つエッジを 0 回以上辿ることで到達可能なノード集合．

演算の連結は $\cdot$ を用いて記述する．例えば，$a \xrightarrow{l1} \cdot \xrightarrow{l2} X$ とは，ノード a から $l1$ というラベルを持つエッジを 1 回辿り，到達したノードから $l2$ を持つエッジを 1 回辿ることで得られるノードを返す．

提案手法は，我々が [2] で提案した手法を次のように拡張したものである．(1)[2] では木を対象としていたが，本論文では非木エッジを含む一般のグラフに対応する．ただし非木エッジの数はグラフのサイズと比較して小さいと仮定する．(2) 複数の起点ノードを許可する．具体的には下記のように拡張した演算 $A \xrightarrow{l} X$ と $A \xrightarrow{l*} X$ を処理可能にする．

$A \xrightarrow{l} X = \bigcup_{a \in A} a \xrightarrow{l} X$

$A \xrightarrow{l*} X = \bigcup_{a \in A} a \xrightarrow{l*} X$

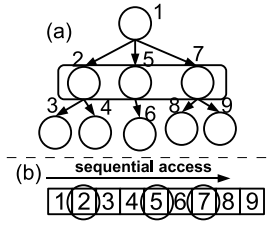


図 1 深さ優先順でのノード配置
Fig. 1 Depth-First-Order Arrangement

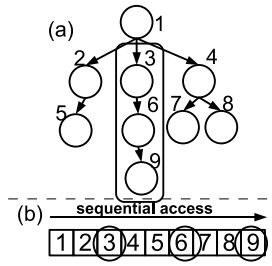


図 2 幅優先順でのノード配置
Fig. 2 Breadth-First-Order Arrangement

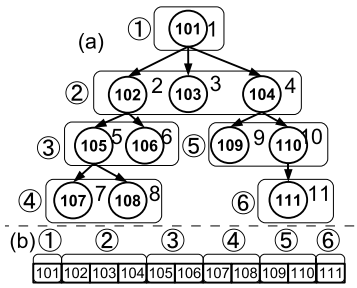


図 3 提案手法でのノード配置
Fig. 3 Proposed Order Arrangement

本研究の新規性. 本研究の新規性は 2 つある．第 1 に，通常の XML 問合せ処理などと異なり，**任意のノードを起点とする問合せの効率的な処理**を問題とすることである．近年，ブラウジングと問合せを柔軟に組み合わせることが重要になってきているが，例えば，SNS データベースに対する問合せとして下記のようなものが考えられる．

$prof[name ='' fernandez''] \xrightarrow{advises*} . \xrightarrow{name} X$

上記の問合せは，名前が"fernandez"の直接的／間接的な弟子を求める．

第 2 に，この問題に対して，**グラフを構成するノードのディスク上での配置順序を工夫**することによって効率的に解を取得する手法を提案している事である．具体的には，問合せの解となるノードができるだけ連続して配置されるようなノードの配置順序を提案する．この配置順序によって，解となるノードが記録された狭い範囲のみを読むことによって，解を取得することが可能となる．解となるノードが記録された領域のみを読むため，問合せはデータのサイズではなく解の数に比例する．

ノードの配置順序としては，深さ優先順や幅優先順が考えられるが，深さ優先順では子供問合せの解が，幅優先順では子孫問合せの解が連続して配置されない．例えば，図 1(a) の木を深さ優先順で配置すると，図 1(b) に示す順序となるが，この順序ではノード a の子供，たとえば 1 の子供 {2,5,7} は連続して配置されない．また，図 2(a) の木を幅優先順で配置すると，図 2(b) に示す順序となるが，この順序ではノード a の子孫，たとえば 3 の子孫 {3,6,9} は連続して配置されない．このように，単純な深さ優先順でも幅優先順でも，子供／子孫問合せのいずれかが連続して配置されず，不必要なディスクアクセスが発生する．

本論文で提案する配置順序の場合，子供／子孫問合せの両方が連続して配置される．よって，解となるノードが記録された領域のみを読むことで問合せを処理することが可能となる．

関連研究. 提案手法は，CAD/CAM アプリケーションを対象とした J. Banerjee らの研究 [3] の処理の一般化になっているが，我々

♡ 学生会員　筑波大学大学院 図書館情報メディア研究科
tada@slis.tsukuba.ac.jp
♢ 正会員　筑波大学大学院 図書館情報メディア研究科
mori@slis.tsukuba.ac.jp
♠ 正会員　京都大学大学院 情報学研究科
tajima@i.kyoto-u.ac.jp

の知る限り，グラフデータに対する正規パス式の処理に関してこのようなアプローチで取り組んだ研究は存在しない．

子供／子孫問合せの高速化に関しては，(1) 到達性判定 [4], (2)*Structural Joins* [5]，(3)XPath 問合せ処理のための索引 [6] などが提案されている．このうち，(1) 到達性判定や (2)Structural Join はいずれも実行時間がデータのサイズに比例する．一方，提案手法の実行時間は解のサイズに比例する．実際，データが巨大な場合，Structural Join では子供の計算にも多大な時間が必要となる (4 章の実験参照). また (3) エッジラベルの閉包を対象とした索引付けは我々の知る限り提案されていない．

本論文の構成は次の通りである．2 章では，グラフに対して効率的に子供／子孫問合せを処理するためにどのようにグラフをディスクに格納するかについて提案する．また単一のノードを起点とした問合せの処理方法についても説明する．3 章では，複数ノードを起点とした問合せの処理手法を提案する．4 章では，実験について述べる．5 章はまとめと今後の課題である．

2. 提案手法

本章では，子供／子孫問合せを効率的に処理するためのディスク格納方式，及び単一のノード a からの子供／子孫問合せの具体的な処理方法について説明する．ディスク格納方式において基本となるアイデアは，1 章でも述べたように解となるノードができるだけ連続して配置されるような順序で，ディスクに各ノードを格納することである．

まず，単純なケースとして，エッジラベルを持たない木の各ノードをどのようにディスクに格納するかについて 2.1 節で説明する．その後，エッジラベル付き木をどのようにディスクに格納するかについて 2.2 節で説明する．最後に非木エッジを含むグラフのノードをどのようにディスクに格納するかについて 2.3 節で説明する．

2. 1 エッジラベルを持たない木

2.1 節では，エッジを持たない木をどのようにディスクに格納するかについて説明する．まず，ノードの配置順序について説明し，次に，各ノードをディスクに格納する際にどのような情報を記録するかについて説明する．そして最後に，問合せの処理方法について述べる．

2.1.1 ノード配置順序

提案するノードの配置順序は以下の通りである．

ノードの配置順序:同じ親を持つ兄弟はグループとしてまとめる．各兄弟グループに対して深さ優先順に番号を振り，番号順に兄弟グループを格納する．兄弟グループ内のノードの順序は元々の木での順序と同じとする．

図 3(a) の木を提案する順序で配置した場合，図 3(b) のようになる．各ノードに書かれた数字 101–111 はそのノードの ID を表している．丸で囲まれた数字 1–6 は，兄弟グループに対して深さ優先順に振られた番号を表している．また，各ノードの脇に記述された数字は提案手法でのノードの配置順序を表している．提案手法により，ノード 102 の子供は，5,6 と連続し，その子孫もまた，自身 102 を除き 5,6,7,8 と連続する．

以降，ノードのディスク上での位置をアドレスと呼び，ノード n のアドレスを $addr(n)$ と記述する．たとえば，図 3 の木では $addr(109) = 9$ である．

2.1.2 ディスクに格納する情報

次にノードをディスクに格納する際どのような情報を記録するかについて述べる．提案手法では，以下の 3 つの情報を各ノードごとに記録する．(a) ノード ID．(b) ノードの親のアドレス．(c) ノードの firstChild のアドレス．図 4 に，図 3 の木をディスクに格納した際のイメージを示す．各ノードごとに，(a)，(b)，(c) の情報が順に記録されている．

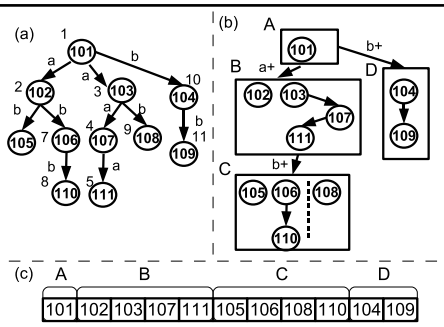


図 5 エッジラベル付き木
Fig. 5 Edge-Labeled Tree

2.1.3 問合せ処理

最後に，ノード a が与えられた時，そのノードの子供／子孫をどのように取得するかについて順に説明する．

[$a \rightarrow X$ の処理] ノード a の子供は連続して配置されている．よって，解となるノードが記録された領域のみを読めばよい．その領域の先頭は，firstChild のアドレスと対応し，その領域の終端は親のアドレスを参照することで判明する．以下に子供問合せを処理する手法を記述する．

1. $addr(a)$ へ移動し，a の firstChild のアドレス $fcAddr$ を取得する．
2. $fcAddr$ へ移動し，各ノードの親のアドレス $pAddr$ が $pAddr = addr(a)$ を満たす限り，順にノードを読み続ける．
3. **2.** で読んだノード集合を解として返す．

[$a \xrightarrow{*} X$ の処理] ノード a の子孫は a を除き連続して配置される．よって，子供問合せ同様，解となるノードが記録された領域のみを読む．領域の先頭，終端も子供問合せのときと同様，firstChild のアドレス, 親のアドレスを参照することによって判明する．そして，最後に読んだノード集合とノード a を解として返せばよい．以下に子孫問合せを処理する方法を記述する．

1. $addr(a)$ へ移動し，a の firstChild のアドレス $fcAddr$ を取得する．
2. $fcAddr$ へ移動し，各ノードの親のアドレス $pAddr$ が $pAddr = addr(a) \vee pAddr \geq fcAddr$ を満たす限り，順にノードを読み続ける．
3. **2.** で読んだノード集合とノード a を解として返す．

2.2 エッジラベル付き木

2.2 節では，2.1 節の手法を拡張し，エッジラベル付きの木をどのようにディスクに格納するかについて説明する．2.1 節と同様，まず，各ノードの配置順序を提案する．その後，ディスクに格納する情報を説明する．最後に，ラベルを含んだ問合せ，すなわち $a \xrightarrow{l} X$ と $a \xrightarrow{l*} X$ を処理する手法を提案する．

2.2.1 ノードの配置順序

ノードの配置順序の決定は，2 つのステップからなる．まず，**Step 1.** エッジラベルごとにクラスタリングを行う．クラスタリングによって，各クラスタ内の部分グラフは同一のエッジラベルのみで構成される．次に各クラスタに対して，**Step 2.** 各クラスタ内のノードの配置順序を決定する．

例えば，図 5(a) のエッジラベル付き木のノードの配置順序を決定する場合，まず，**Step 1.** 図 5(b) のようにクラスタ A,B,C,D の 4 つにクラスタリングする．その後，**Step 2.** 各クラスタ内のノードの配置順序を決定する．その結果，図 5(c) のような順序となる．

以下では，各ステップの具体的な処理について説明する．

Step 1. エッジラベルごとにクラスタリング. まず，木に含まれるノードを特定のエッジラベルで到達可能なノードごとにクラスタリングする．具体的には，以下のようにクラスタリングを行う．

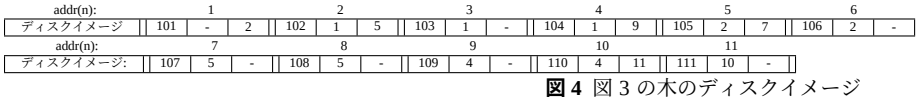


図 4 図 3 の木のディスクイメージ
Fig. 4 Disk Image of the Tree in Fig. 3

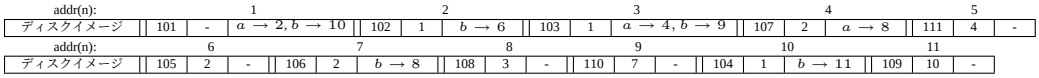


図 6 図 5 の木のディスクイメージ
Fig. 6 Disk Image of the Tree in Fig. 5

1. ルートノードを 1 つのクラスタとする．

2. ルートノードから特定のエッジラベルのみで到達可能なノード集合を 1 つのクラスタとしてまとめる．

3. 別のラベルが出現したとき，**2.** で得られたクラスタに含まれるノード集合をルートノードと見なし，再び **2.** を適用する．

例えば，図 5(a) の木を上記に従って図 5(b) のようにクラスタリングすることを考える．まず，**1.** に従いルートノード，101 を 1 つのクラスタとする．次に **2.** によってルートノードからラベル a のみで到達可能なノード集合，$\{102, 103, 107, 111\}$ を 1 つのクラスタとしてまとめる．また同様にラベル b でのみ到達可能なノード集合，$\{104, 109\}$ を 1 つのクラスタとしてまとめる．そして **3.** に従い，$\{102, 103, 107, 111\}$ でのラベル，a とは異なる別のラベル b が出現したとき，$\{102, 103, 107, 111\}$ をルートノードとみなし，これらのノードから到達可能なノード集合，$\{105, 106, 110, 108\}$ を 1 つのクラスタとしてまとめる．

クラスタリング終了後，各クラスタを深さ優先順にディスク上に格納する．

Step 2. クラスタ内のノード順序の決定: 最後に各クラスタに含まれるノードの配置順序を決定する．クラスタに含まれる部分グラフが木ならば，2.1 節と同様の手法で配置する．しかし一般には，クラスタ内の部分グラフは木ではなく森となる．例えば，図 5(b) のクラスタ D は木だが，クラスタ C は 3 つの木によって構成された森となっている．以下では，森となった部分グラフをどのような順でディスクに配置するかについて述べる．

まず森に含まれる部分木を，その部分木のルートの親のアドレス順にソートする．その後，以下のルールに従って，森のノードをグループ化し，各グループを深さ優先順に配置する．

ルール: 各部分木のルートノードを一つのグループとする．その後，残りのノードを兄弟ごとにグルーピングする．

例えば，クラスタ C に含まれるノード集合の順序を決定する場合，まず，ノード集合 {105,106,110,108} を上記のルールに従ってグループ化する．その結果，{105,106,108}，{110} という 2 つのグループができる．そして，それぞれのグループを深さ優先順に配置すると [105, 106, 108, 110] という順でノードが配置される．図 5(a) の各ノードに付与された数字 1–11 は提案手法でノードを配置した際の順序を表している．

2.2.2 ディスクに格納する情報

一般に木に含まれるエッジラベルは，任意の文字列で構成される．よって，エッジラベルごとに firstChild の情報を記録する．それ以外は，2.1 節と同じ情報を記録する．提案手法で図 5 の木をディスクを格納した際のイメージを図 6 に示す．

2.2.3 問合せ処理

最後に，$a \xrightarrow{l} X$，$a \xrightarrow{l*} X$ を処理する方法について順に説明する．

[$a \xrightarrow{l} X$ の処理] Step1 のクラスタリングにより，同一のエッジラベルで到達可能な子供は連続して配置される．$a \xrightarrow{l} X$ の解は連続して並ぶ．よって，2.1 節と同じ方法で処理可能である．

[$a \xrightarrow{l*} X$ の処理] 以下に，子孫問合せを処理する手法を以下に記す．

1. 子供問合せ，$a \xrightarrow{l} X$ を処理する．

2. **1.** で得られた解からラベル l で到達可能な firstChild のうち，最もアドレスが小さい値，$mAddr$ を取得する．

3. $mAddr$ へ移動し，親のアドレスが既にアクセスした範囲に存在する限り読み続ける．

4, ノード a，**1.**，**3.** で読んだノード集合を解として返す．

例えば，図 5 に対して，$102 \xrightarrow{b*} X$ を処理することを考える．上記の手法に従い，まず **1.** 子供問合せ，$102 \xrightarrow{b} X$ を処理する．子供問合せの解は，{105,106} となる．次に，**2. 1.** で得られた解から，ラベル b で到達可能な firstChild のうち，最もアドレスが小さい値，$mAddr$ を取得する．{105,106} のうち，b で到達可能な firstChild は，ノード 110 のみのため，$maddr$ は $addr(110) = 8$ となる．その後，**3.** $mAddr$ へ移動し，親のアドレスが既にアクセスした範囲に存在する限り読み続ける．この場合，ノード 110 のみが読まれる．最後に，**4.** ノード 102，**1.** で読んだ {105,106}，3. で読んだ {110} が解として返される．

2.3 非木エッジを持つグラフ

本節では，非木エッジを含む一般のグラフに対して子供／子孫問合せを効率的に行うためのディスク格納手法について説明する．まず非木エッジの情報を持つグラフをどのように格納するかについて述べ，その後，問合せを処理する方法について述べる．

2.3.1 格納手法

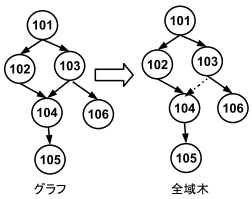
非木エッジを持つグラフは，グラフを構成する全域木と非木エッジのそれぞれの情報を保持しなければならない．提案手法では，全域木を 2.2 節で示した手法でディスクに格納し，非木エッジの情報をメモリ上に保持する．

このプロセスを図 7 を用いて説明する（単純化のため，グラフのエッジラベルは全て l であるとする）．図 7 のグラフは，103 から 104 への非木エッジと全域木によって構成されている．まず，図 7 の全域木を 2.2 節の手法でディスクに格納する．全域木をディスクに格納した際のイメージを図 7(右上) に示す．その後，非木エッジの情報を図 7(右下) のような，表で保持する．以降，図 7(右下) のように非木エッジの情報を持つ表を非木エッジ表と呼ぶ．

非木エッジ表は，以下のタプルによって構成されている．

nt-edge($label$,s,d, d_region)

各タプルの要素について順に説明する．$label$ とは，非木エッジのラベルである．s，d はそれぞれ非木エッジの始点，終点と対応する．d_region は，全域木に対して $d \xrightarrow{label*} X$ を処理した際に，d を除く全ての解のアドレス範囲を表す．アドレス範囲は，



addr(n):

	1			2			3			4			5			6		
ディスクイメージ:	101	-	$l \rightarrow 2$	102	1	$l \rightarrow 4$	103	1	$l \rightarrow 6$	104	2	$l \rightarrow 5$	105	4	-	106	3	-

非木エッジ表

label	s	d	d_region
l	103	104	{5-5}

図 7 グラフとその全域木
Fig. 7 A Graph and its Spanning Tree

($start$–end) というリージョンの形式で保持される．上記のタプルを非木エッジの数だけメモリ上に保持する．そのため非木エッジ表を保持するために必要な領域は，非木エッジの数に比例する．しかし実際のアプリケーションでは，非木エッジの数は少ないため [4]，非木エッジ表を保持するために必要な領域はさほど大きくならない．

2.3.2 問合せ処理

本節では，非木エッジを持つグラフに対して子供／子孫問合せを処理する方法について説明する．

[$a \xrightarrow{l} X$ の処理] 子供問合せを行う場合，全域木での子供問合せに加えノード a から非木エッジでつながれた子供を取得しなければならない．例えば，図 7 のグラフにおいて，$103 \xrightarrow{l} X$ の解 X は，全域木での $103 \xrightarrow{l} X$ の解 {106} だけでなく，非木エッジを辿った先のノード，{104} も解となる．よって，全域木での解に加え，非木エッジを辿った先も解として加えなければならない．

以下に，子供問合せを処理する方法を記す．

1. 全域木に対して $a \xrightarrow{l} X$ を処理する．

2. 非木エッジ表を参照し，$s = a \wedge label = l$ となる全ての d を取得する．

3. **1.** で取得した解と **2.** で取得した d の集合の和を解として返す．

[$a \xrightarrow{l*} X$ の処理] グラフに対する子孫問合せは，その問合せを全域木に対する問合せに変換することで処理を行う．具体的には，以下の条件を満たすノード集合 A' を計算する．

条件:グラフに対して $a \xrightarrow{l*} X$ を処理した際の解を X とする．そのグラフから非木エッジを取り除いた全域木に対して，$A' \xrightarrow{l*} X$ を処理したとき，その問合せの解がグラフに対する問合せの解と同じになる．

たとえば，図 7 グラフに対して，$103 \xrightarrow{l*} X$ を処理することを考える．この問合せの解 X は $X = \{103, 104, 105, 106\}$ である．今，$A' = \{103, 104\}$ として，右の全域木に対して $A' \xrightarrow{l*} X$ を処理すれば，$103 \xrightarrow{l*} X$ の解が {103,106}, $104 \xrightarrow{l*} X$ の解が {104,105} のためグラフに対する問合せの解，$X = \{103, 104, 105, 106\}$ と同じ解を取得することが可能となる．

この変換によって，全域木に対してノード集合 A' を起点とした問合せを処理する必要がある．本節では，ノード集合 A' を求める方法のみを記述し，ノード集合 A' からの子孫問合せの処理方法は 3 章で述べる．

以下に，ノード集合 A' を求める方法を記す．

1. $A' = \{a\}$ とする．

2. 全域木での $a \xrightarrow{l*} X$ の解となるアドレス範囲，r を取得する．

3. 非木エッジ表を参照し，r に $addr(s)$ が含まれ，かつ $label = l$ となる全てのタプルを取得する．

4. **3.** で取得した各タプルの d が A' に含まれているか否かをチェックする．含まれていない場合，以下を行う．

(1). A' に d を加える．

(2). タプルの d_region を r として，**3.** を適用する．

例えば，図 7 のグラフに対して，先ほど例として用いた $103 \xrightarrow{l*} X$ を処理する事を考える．

まず，上記の **1.** に従い $A' = \{103\}$ とする．次に **2.** に従って，$a \xrightarrow{l*} X$ の解となるアドレス範囲，$r = \{3\text{-}3\}, \{6\text{-}6\}$ を取得する．その後，**3.** に従って非木エッジ表を参照し，r に $addr(s)$ が含まれ，かつ $label == l$ となるタプルを取得する．$addr(103) = 3$ は，r に含まれているため取得するタプルは，始点 103, 終点 104 の非木エッジが記録されたタプルとなる．その後，**4.** に従って，**3.** で取得した各タプルの d が A' に含まれているか否かをチェックする．d =104 が A' に含まれていないため，**(1).** に従い A' に 104 を加える．結果，$A' = \{103, 104\}$ となる．そして，**(2).** に従い $d\_region = \{5\text{-}5\}$ を r として再び **3.** を適用する．$r = \{5\text{-}5\}$ に $addr(s)$ が含まれるタプルは存在しないため，処理を終了する．以上により，$A' = \{103, 104\}$ を取得することが可能となった．

3. 複数ノードからの問合せ

複数ノードを起点とした問合せでは，解の重複が発生するため，重複除去を行わなければならない．例えば，図 3(a) の木において，{102,104,105,109} を起点とした子孫問合せを行う場合，102 の子孫と 105 の子孫，及び 104 の子孫と 109 の子孫がそれぞれ重複している．

重複除去を行うための最も簡単な手法は，問合せの解となる全てのノードをメモリ上に保持する必要があるが，この方法は以下の 2 つの理由から効率的ではない．(1) 解となるノード集合，X を全てメモリ上に保持しなければならないため，$O(|X|)$ もの領域が必要となる点．(2) 何の工夫もしなければ問合せのチェックには，$O(|X|^2)$ もの計算量が必要となる点．

本章では，必要な領域を減らし，かつ高速に重複のチェックを行う手法を提案する．重複のチェックのために必要な領域は $O(1)$ であり，また計算量は起点となるノード集合を A とすると $O(|A|)$ となる．

本章で対象とするグラフは全て木とし，問合せは子孫問合せのみに限定する．本論文では非木エッジを持つ任意の有向グラフを対象としているが，2.3 節のアルゴリズムを適用すれば，全域木に対して本手法をそのまま適用することが可能となる．また木に対する子供問合せでは解が重複しないため，問合せを子孫問合せのみに限定する．

本提案手法における重複除去は，以下の 2 つのアイデアに基づいている．

1. 解となるノードが記録された領域をメモリに保持する

2. 事前にノード A をソート後，ソートした順に子孫問合せを行う

まず，問合せの解となるノード集合を全てメモリ上に保持するのではなく，**1.** 解となるノードが記録された領域をメモリに保持する．そして，あるノード a の子孫が重複しているか否かのチェックは，その領域に $addr(a)$ が含まれているか否かのチェックによって実現する．提案するノードの配置順序により，$a \xrightarrow{*} X$ の解となるノード集合は，a を除き，連続して配置される．よって解のう

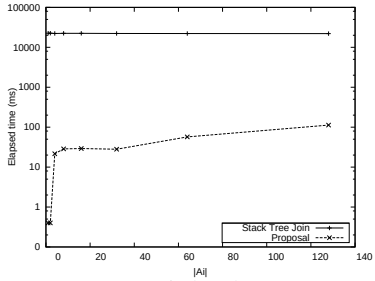


図 8 実験 1: 結果 1
Fig. 8 Exp. 1: Result 1

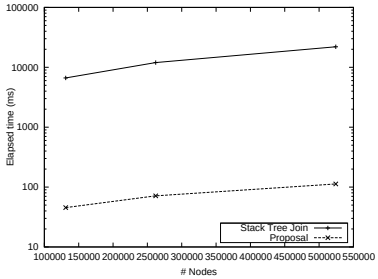


図 9 実験 1: 結果 2
Fig. 9 Exp. 1: Result 2

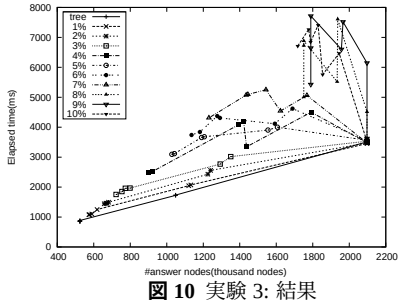


図 10 実験 3: 結果
Fig. 10 Exp. 3: Result

非木エッジの割合 (%)	0(木)	1	2	3	4	5	6	7	8	9	10
実行時間/解の数 (μ 秒)	1.65	1.85	2.06	2.28	2.55	2.69	2.9	3.04	3.04	3.27	3.43

図 11 実験 3: 解 1 ノードあたりの実行時間
Fig. 11 Exp. 3: Elapsed Time Per Node in the Answer

ち，a を除く解の範囲は全て $start$-end というリージョンの形式で保持することが可能となる．その結果，解となるノードを保持するというアプローチと比較して必要な領域が小さくなる．

例えば，図 3(a) の木において，$\{102, 105\} \xrightarrow{*} X$ を処理することを考える．この際，まず $102 \xrightarrow{*} X$ の解のうち，自身 102 を除く解が記録された領域，(5-8) をメモリ上に保持する．そして，$105 \xrightarrow{*} X$ を処理する際，解となる領域 (5-8) に $addr(105)$ が含まれているため，重複していることが分かる．

次に，問合せを処理する際，**2,** 以下のルールに従って A に含まれるノード集合をソート後，順に問合せを行う．

ソートルール:各ノードを $firstChild$ のアドレスでソートする．子供がない場合，自身のアドレスを $firstChild$ として同様にソートする

このソートにより，A に含まれるノードのうち，子供を持つ全てのノードは，$a \xrightarrow{*} X$ の解となる領域の先頭が常に昇順になるようにソートされる．例えば，$A = \{102, 104, 105\}$ を上記のルールでソートすると，$A = \{102, 105, 104\}$ となる．そして，$102 \xrightarrow{*} X$，$105 \xrightarrow{*} X$，$104 \xrightarrow{*} X$ が記録された領域はそれぞれ (5-8)，(7-8)，(9-11) となり，領域の先頭が昇順にソートされていることが分かる．

一般に，木に含まれる任意の 2 ノードの子孫問合せの解はそれぞれの解が互いに疎か，包含関係であるかのいずれかである．そして，上記に述べたソートにより，解が記録された領域を高々1 つメモリ上に保持するだけで重複のチェックを行うことができる．すなわち，必要な領域は $O(1)$ となる．また，重複のチェックはメモリに保持している高々1 つの領域に含まれているか否かをチェックするだけなので $O(|A|)$ となる．

本アイデアに基づいて $A \xrightarrow{*} X$ を処理するための抽象コードを記す．メモリ上に保持する領域は変数 currentRegion で保持する．また，A は既にソート済みであるとする．

```
1. currentRegion = null;
2. for each a in A {
3.     if (therecurrentRegion doesn't exist addr(a)){
4.       getDescendantsOf(a); //a →* X を処理
5.       currentRegion = a の解となるノードが記録された領域
6.     }
7. }
```

例えば，$A = \{102, 105, 104, 109\} \xrightarrow{*} X$ を処理することを考える．この場合，まず上記のルールに基づいて A をソートし，$A = \{102, 105, 104, 109\}$ となる．その後，上記のコードに従って処理を行うと，まず $102 \xrightarrow{*} X$ の解を取得し，currentRegion は (5-8) となる．次に処理すべき子孫問合せは $105 \xrightarrow{*} X$ であるが，$addr(105) = 5$ は currentRegion に含まれるため，処理を行わない．一方，次の子孫問合せ $104 \xrightarrow{*} X$ は，$addr(104)$ が currentRegion に含まれていないため，処理を行い，currentRegion を (9-11) に更新する．最後に，$109 \xrightarrow{*} X$ を処理するが，$addr(109)$ は，currentRegion に含まれているため処理を行わない．以上により，複数ノードからの問合せを処理することが可能となった．処理をする必要がある子孫問合せは $102 \xrightarrow{*} X$，$104 \xrightarrow{*} X$ のみであり，その解は重複せず，また漏れも存在しない．

4. 評価実験

提案手法を評価するために実験を行った．実験環境は，OS RedHat Linux, CPU Xeon 2.8GHz, メモリ 1GB である．実装は，Java 1.6.0 で行った．なお，本論文で示す実行時間は，いずれも 5 回の計測結果の平均である．

4. 1 実験 1: 既存手法との比較

提案手法と既存の手法の特徴の違いを明らかにするために，既存の手法との比較を行った．実験では，本手法と Strucutural Join の実装の一つである Stack-Tree Join でノードをディスクに配置し，子供問合せの処理時間を比較した，実験で用いたグラフはエッジラベルを持たない深さ 17,18,19 の完全二分木である．

実験結果. 実験では，完全二分木の深さ i のノード集合を問合せの起点 A_i として，A_i の子供を問合せ，その実行時間を測定した．問合せの解 X は 2^i ノードとなる．

図 8 に $2^{19} - 1$ ノードでの実験結果を示す．図 8 の X 軸は，問合せの起点となる A_i の数，Y 軸は $A_i \rightarrow X$ の処理時間である．

また，図 9 に，3 つの完全二分木において，深さ 8 のノード集合，すなわち A_8 からの子供問合せの処理時間を示す．図 9 の X 軸は，解となるノードの数，Y 軸は $A_8 \rightarrow X$ の処理時間である．

実験結果から，Stack Tree Join の実行時間が木のサイズに依存しているのに対して提案手法の実行時間は，解の数に依存していることが分かる．また，提案手法が既存の手法よりも効率的に子供問合せを処理することが出来る事が分かる．

4. 2 実験 2: ノード配置順序

次に，ノードの配置順序が問合せの処理にどのような影響を与えるかについて実験を行った．実験では，深さ 10 の完全 10 分木を用いた．木の各ノードは $l1$ というエッジでつながれた子供，$l2$ というエッジでつながれた子供をそれぞれ 5 ノードずつ持つ．図 12 の 1–2 列目に実験で用いた問合せと，問合せの解となるノード数を示す．

実験では，以下の 2 つの配置順序でノードを配置した際に，問合せを処理するために必要な最悪のディスクアクセス数を計算した．

問合せ	解の数	DF	Proposal
Q1: Root $\xrightarrow{l1}\cdot\xrightarrow{l2}$ X	25	31	3
Q2: Root $\xrightarrow{l1}\cdot\xrightarrow{l2*}$ X	2,441,405	2,441,417	4
Q3: Root $\xrightarrow{l1*}\cdot\xrightarrow{l2}$ X	2,441,405	4,882,811	4
Q4: Root $\xrightarrow{l1*}\cdot\xrightarrow{l2*}$ X	23,803,711	26,245,111	4

図 12 実験 2: 問合せとディスクアクセス数
Fig. 12 Exp. 2: Queries and the Number of Disk Accesses

	深さ	6	7	8
Q1	実行時間 (ms)	10.6	10.2	10.2
	解の数	25	25	25
Q2	実行時間 (ms)	46.4	95.6	268.4
	解の数	3905	19530	97655
Q3	実行時間 (ms)	62.2	159.2	478.6
	解の数	3905	19530	97655
Q4	実行時間 (ms)	115	340.4	1469.6
	解の数	22461	131836	756836

図 13 実験 2: 実行時間
Fig. 13 Exp. 2: Elapsed Time

DF: 深さ優先順でのノード配置

Proposal: 提案手法でのノード配置

ディスクアクセス数. 図 12 の 3–4 列目に，その結果を示す．実験結果から，深さ優先順と比較して我々の提案手法では問合せを処理する際に必要なディスクアクセスの回数が圧倒的に少ない事がわかる．よって，子供／子孫問合せを効率的に処理する事ができる．

実行時間. 以上の見積もりから，提案手法でノードをディスクに格納し，その実行時間を測定した．実験で用いたのは，深さ 6,7,8 の完全 10 分木である．ディスクアクセス数を見積もった時同様，完全 10 分木は，$l1$ というエッジでつながれた子供，$l2$ というエッジでつながれた子供をそれぞれ 5 ノードずつ持つ．実験結果を図 13 に示す．実験から，問合せの解が 70 万ノード程度でも約 1.5 秒程度で問合せを処理できることが判明した．

4. 3 実験 3. 非木エッジ

最後に，非木エッジが増えた場合に実行時間がどのように変化するかを調査した．実験では，深さ 21 の完全二分木を全域木として，その木に 1-10%の非木エッジを加えたグラフを利用した．非木エッジの始点，終点はランダムに選択した．全域木の全てのエッジ，非木エッジともにエッジラベルは l のみで構成されている．実験では，全域木の以下の 7 ノードから，子孫問合せを行った．(1) ルートノード, (2–3) 深さ 2 のノード 2 つ，(4–7) 深さ 3 のノード 4 つ．

図 10 に木での実行時間と非木エッジ 1%–10%での実行時間を示す．図 10 の X 軸は解となるノードの数 (千ノード単位)，Y 軸は子孫問合せの実行時間である．

また図 11 に各グラフでの解 1 ノードあたりの実行時間を記す．実行時間の値の単位は，μ 秒であり，それぞれの値は 7 ノードから子孫問合せを実行した際の平均である．

実験結果では，たとえ非木エッジが 10% 程度となったとしても，木に対する問合せ処理時間の約 2 倍程度で処理可能であることが判明した．また実験では，2.3 節で示した A' を計算する手法を拡張し，重複として取り除かれることが自明なノードは A' に加えなかったため，ルートノードからの問合せは非木エッジの割合にかかわらず，一定の速度となった．

5. まとめ

本論文では，大規模なグラフに対して正規パス式を効率的に処理するためのディスク上でのノードの配置順序について提案した．提案手法により，ディスクアクセス数が減少し，効率的に問合せを処理することが可能となった．実験では，本手法が効率的であることを示した．今後の課題としては，グラフの更新への対応や他の問合せへの処理方法の検討等が挙げられる．

謝辞

ゼミなどでコメントいただきました筑波大学大学院図書館情報メディア研究科の杉本重雄教授，阪口哲男准教授，永森光晴講師に感謝いたします．本研究の一部は科学研究費補助金若手研究 (B)(#20800076) による．

[文献]

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener: The Lorel Query Language for Semistructured Data. Int. J. on Digital Libraries 1(1): 68-88 (1997)

[2] 只石正輝，森嶋厚行，田島敬史. 大規模木構造データに対する正規パス式の効率的な処理方式の検討. 第 70 回情報処理学会全国大会講演論文集 (第 1 分冊), pp. 603-604, 茨城，2008 年 3 月.

[3] Jay Banerjee, Won Kim, Sung-Jo Kim, Jorge F. Garza: Clustering a DAG for CAD Databases. IEEE Trans. Software Eng. 14(11): 1684-1699 (1988)

[4] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, Jeffrey Xu Yu: Dual Labeling: Answering Graph Reachability Queries in Constant Time. ICDE 2006: 75-86.

[5] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, Divesh Srivastava: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002: 141-152

[6] Wei Wang and Haifeng Jiang and Hongzhi Wang and Xuemin Lin and Hongjun Lu and Jianzhong Li, “Efficient processing of XML path queries using the disk-based F&B Index,” VLDB 2005, 145-156, 2005.

只石 正輝　Masateru TADAISHI
筑波大学大学院 図書館情報メディア研究科博士前期課程在学中．情報処理学会学生会員，日本データベース学会学生会員．

森嶋 厚行　Atsuyuki MORISHIMA
筑波大学大学院 図書館情報メディア研究科/知的コミュニティ基盤研究センター准教授．筑波大学大学院 工学研究科修了．博士 (工学)．ACM，IEEE-CS，情報処理学会，電子情報通信学会，日本データベース学会各正会員．

田島 敬史　Keishi TAJIMA
京都大学大学院 情報学研究科社会情報学専攻准教授．東京大学大学院 理学系研究科情報科学専攻博士後期課程修了．博士（理学）．ACM，IEEE-CS，情報処理学会，日本データベース学会各正会員．