社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2008－DBS－146　(39)
2008／9／22

大規模グラフ構造に対する効率的な子供／子孫問合せの処理手法の提案

只石　正輝[†]　　森嶋　厚行[†]　　田島　敬史[††]

† 筑波大学大学院図書館情報メディア研究科　〒 305–8550 茨城県つくば市春日 1–2
†† 京都大学大学院情報学研究科　〒 606–8501 京都市左京区吉田本町
E-mail: †{tada,mori}@slis.tsukuba.ac.jp, ††tajima@i.kyoto-u.ac.jp

あらまし　今日，XML や RDF 等のエッジラベル付き有向グラフが生成，蓄積されている．また，生成，蓄積されるグラフは大規模となってきており，グラフに対する効率的な管理/検索が重要な問題となってきている．本論文では，大規模なグラフに対する問合せの一つである子供/子孫問合せに着目し，それらの問合せを効率的に処理するためのノード格納方式を提案する．

キーワード　グラフデータベース，問合せ処理

Proposal of an Efficient Method to Process Child/Descendant Queries for Large Graph Structures

Masateru TADAISHI[†], Atsuyuki MORISHIMA[†], and Keishi TAJIMA[††]

† Grad. Sch. of Library, Information and Media Studies, Univ. of Tsukuba.　1–2 Kasuga, Tsukuba, Ibaraki, 305–8550 Japan
†† Grad. Sch. of Informatics, Kyoto University.　Yoshida–Honmachi, Sakyo–ku, Kyoto 606–8501, Japan
E-mail: †{tada,mori}@slis.tsukuba.ac.jp, ††tajima@i.kyoto-u.ac.jp

Abstract　Today, we have many edge-labeled directed graphs such as ones written in RDF and XML. Because the size of such graphs is getting larger, efficient processing of queries against graphs is important. This paper proposes a novel node storing scheme for the efficient processing of child/descendant queries.

Key words　Graph Databases, Query Processing

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 式という形でまた注目されており [4]，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 度辿ることで得られるノード集合．

(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 度以上辿ることで到達可能なノード集合．

演算の連結は · 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 度辿り，到着したノードから $l2$ を持つノードを 1 度辿ることで得られるノードを返す．

本論文での提案手法は，我々が [6] で提案した手法を次のように拡張したものである．(1) [6] 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．

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

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

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．

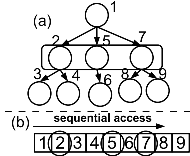


図 1　深さ優先順でのノード配置

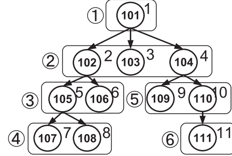


図 2　提案手法でのノード配置

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る解を提供する事である．これによって，少ないディスクアクセスでこれらの問合せの処理を実現する．

第 2 の特徴について説明する．まず単純なノードの配置順序として，深さ優先順でのノード配置を考える．ここでは，対象のグラフを図 1(a) のようなエッジラベルを持たない木とする．この木を深さ優先順で配置すると，図 1(b) の順でノードが配置される．この場合，ノード a の子孫は連続して配置されるが，子供，例えば 1 の子供 {2,5,7} は連続して配置されない．一方，幅優先順でノードを配置した場合でも同様の問題に直面する．すなわち，あるノードの子供は連続して配置されるが，子孫は連続して配置されない．よって，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で多くのランダムアクセスが必要となるため，問合せを効率的に処理する事が出来ない．本論文で提案する配置順序を利用することにより，子供／子孫問合せの両方が効率的に処理可能となる．

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 [2] の処理の一般化になっているが，我々の知る限り，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．

XML データやグラフデータに対しては，到達性判定 [3]，*structural joins* [1]，XPath 問合せ処理のための索引 [5] などが提案されてきた．到達性判定や structural join では，実行時間はデータのサイズに比例するが，提案手法では実行時間は解のサイズに比例する．前者は特に子供の計算にも多大な時間がかかってしまうという重大な問題がある．(4. 章の実験参照)．XPath 問合せ処理は，//a/b のような形の問合せの処理は効率的だが，ルート以外を起点とする問合せやエッジラベルの閉包には対応しない．

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のディスク格納方法について提案する．また単一ノードを起点とした問合せの処理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，グラフに対する子供／子孫問合せの効率的な処理方式を提案する．基本となるアイデアは，解となるノードができるだけ連続して配置されるようにノードをディスクに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．また，それぞれのグラフに対して単一のノードを起点とした子供／子孫問合せの処理方法についても述べる．

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 ノードの配置順序

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．各兄弟グループを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 2 は，あるエッジラベルを持たない木に対する提案手法でのノードの配置順序を表している．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．各ノードをこの順番でディスクに格納する．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 2 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 3 に，図 2 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．

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説明する．前述したように，$a \rightarrow X$ の解も，$a \xrightarrow{*} X$ の解 (ノード a 自身を除く) も，連続して配置される．これにより，少ない回数のランダムアクセスで解を取得することが可能である．以下では，それぞれの問合せをどのように処理するかについて説明する．

[$a \rightarrow X$ 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，子供ノードが記録された領域の先頭へ移動し，子供ノードのみを読み続ければよい．以下に子供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．

addr(n):		1			2			3			4			5			6			7			8			9			10			11	
ディスクイメージ	101	-	2	102	1	5	103	1	-	104	1	9	105	2	7	106	2	-	107	5	-	108	5	-	109	4	-	110	4	11	111	10	-

図 3　図 2 のディスクイメージ

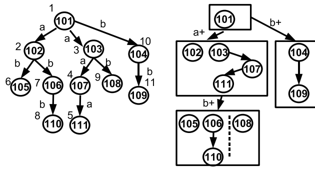


図 4　エッジラベル付き木

2.　$fcAddr$ へ移動し，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，子供問合せと同様に子孫ノードが記録された領域の先頭へ移動し，以降のノードを順に読み続ければよい．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，親のアドレス $pAddr$ が $pAddr = addr(a) \vee pAddr \geqq fcAddr$ 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し，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

ノードの配置順序は以下の 3 つのステップを順に適用することで決定される．

Step 1. エッジラベルごとに子供ノードをソート． まず，各ノードの子供を，その子供へ張られたエッジのラベルに従ってソートする．このようにエッジラベルごとにソートを行うことで，子供問合せ $a \xrightarrow{l} X$ の解は，連続して配置される．

Step 2. エッジラベルごとにクラスタリング． 次に，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．クラスタリングは以下によって実現可能である．

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の縮約ノードをルートノードと見なし，再び **2.** を適用する．

本手法に従って，図 4(左) の木の各ノードをクラスタリングしたものを図 4(右) に示す．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．そして **3.** によって別のラベル 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 3. 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．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，図 4(右) の {105,106,110,108} を上記のルールに従ってグループ化すると，{105,106,108}, {110} という 2 つのグループができる．それぞれのグループを深さ優先順に配置すると [105, 106, 110, 108] という順でノードが配置される．図 4(左) の各ノードに付与された数字 1–11 は選択肢 B. 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 と同じ情報を記録する．提案手法でノードをディスクを格納した際のイメージを図 5 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] エッジラベルごとに子供をソートしたため，$a \xrightarrow{l} X$ 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] 子孫は連続して配置されるとは限らない． 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．

2.3　非木エッジ

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

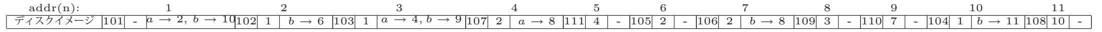
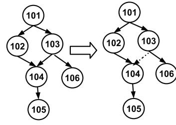
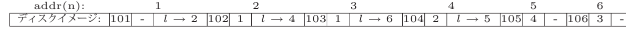


図 5　図 4 のディスクイメージ





非木エッジ表

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

図 6　非木エッジを持つグラフ

べる．

2.3.1　格 納 手 法

非木エッジを持つグラフを格納する方法は以下の通りである．まず，全域木を 2.2 節の手法でディスクに格納する．また，非木エッジの情報を持つ非木エッジ表をメモリ上に保持する．

このプロセスを図 6 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 6(左) 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 6 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 6(右上) に示す．その後，非木エッジの情報を図 6(右下) のような，非木エッジ表で保持する．

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，終点である．d_region は，全域木に対して $d \overset{label*}{\rightarrow} X$ を処理した際に，d を除く全ての解のアドレス範囲を保持する．アドレス範囲は，($start$–end) というリージョンの形式で保持される．

非木エッジ表の大きさは $O(|nt\text{-}tree|)$ となる．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 [3]，上記の表の大きさも小さくなる．

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．

[$a \overset{l}{\rightarrow} X$ 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．よって，以下のようにして子供問合せを処理する．

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

2.　非木エッジ表を参照し，$addr(s) = addr(a) \wedge label = l$ 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．

[$a \overset{l*}{\rightarrow} X$ の処理] 以下の条件を満たすノード集合 A' を取得する．

条件:グラフに対して $a \overset{l*}{\rightarrow} X$ 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，$A' \overset{l*}{\rightarrow} X$ を処理したとき，その問合せの解が X と同じになる

たとえば，図 6 左のグラフに対して，$103 \overset{l*}{\rightarrow} X$ 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 $A' \overset{l*}{\rightarrow} X$ を処理すれば，X と同じ解を取得することが可能となる．このように子孫問合せの処理では，複数ノードを起点とした問合せを処理しなければならない．その処理方法は，3. 章で述べる．

以下にノード集合 A' を求める方法を記述する．

1.　$A' = \{a\}$ とする．

2.　a のリージョン r を取得する．

3.　r に $addr(a)$ の情報を加える．

4.　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．

5.　**4.** 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，**4.** を適用する．

例えば，図 6 のグラフに対して，$103 \overset{l*}{\rightarrow} X$ を処理する事を考える．この場合，上記の手法を適用する事で以下のように A' を取得することができる．

まず，上記の **1.** に従い $A' = \{103\}$ とする．次に **2.** に従って，$a = 103$ のリージョン $r = \{6\text{-}6\}$ を取得する．そして **3.** に従い，r に $addr(103)$ を加える．その結果，$r = \{\{2\text{-}2\}, \{6\text{-}6\}\}$ となる．その後，**4.** 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．取得するタプルは図 6 右下の非木エッジ表のタプルとなる．その後，**5.** に従って，**4.** 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 **4.** 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を処理する際に問題となるのは，重複除去である．重複を除去するための最も単純な方法は，解となるノードの情報を全て保持する事であるが，この方法では $O(|X|)$ もの領域が必要となる．また，あるノードが解として含まれているかのチェックは，何の工夫もしなければ $O(|X|^2)$ の計算量が必要となる．そこで本章では，少ないメモリ量で，かつ高速に重複除去を行う手法を提案する．

本章で扱うグラフは全て木とする．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，本章では説明する問合せは子孫問合せのみとする．これは，木に対して複数ノードを起点とした子供問合せを処理しても，解が重複しないからである．

提案手法では以下の性質を使って重複の除去を行う．

性質:木に含まれる任意のノード $n1$ 及び $n2$ に対して，$n1 \xrightarrow{l*} X$ の解 X_1 と，$n2 \xrightarrow{l*} X$ の解 X_2 は，互いに素か，包含関係かのいずれかである．

重複が発生するのは、包含関係の場合のみである．よって，以下では A に含まれる各ノードの子孫が包含関係にあるか否かをチェックする方法について説明する．ここではまず，あるノード 2 つ $n1, n2 \in A$ の子孫集合同士の包含関係のチェックについて説明し，その後，任意のノード集合 A での重複除去について説明する．

包含関係をチェックするために，子孫問合せの解となるアドレスの範囲を利用する．2.3 節と同様，このアドレス範囲をリージョンで保持する．以下の条件を満たす時，ノード $n1$ と $n2$ の子孫問合せの解 X_1，X_2 は，$X_1 \supseteq X_2$ である．

条件:X_1 のリージョンに $addr(n2)$ が含まれている．

例えば，図 2 において，$n1 = 102$，$n2 = 105$ とした場合，$n1$ の子孫問合せの解 X_1 のリージョンに $addr(n2) = 7$ が含まれているため，子孫問合せの解は包含関係にある．

次に，任意のノード集合 A での重複除去について説明する．まず，A を子供を持つノード集合とそうでないノード集合に直和分割する．以降，子供を持つノード集合を intermediateNodes, そうでないノード集合を leafNodes とする．直和分割後，intermediateNodes は各ノードの firstChild のアドレスで，leafNodes は各ノードのアドレスでソートを行う．intermediateNodes を firstChild のアドレスでソートしたことにより，重複除去は，ひとつ前のリージョンにアドレスが含まれているか否かをチェックすることで実現することができる．また，leafNodes をアドレス順にソートしたことで，子孫問合せのリージョンに leafNodes のアドレスが含まれているか否かのチェックは，二分探索で行うことが可能となる．

以下に，上記のアイデアに基づいて $A \xrightarrow{l*} X$ を処理するためのコードを記す．intermediateNodes，leafNodes は共にソート済みであるとする．

```
1.  List answer;
2.  currentRegion = null;
3.  for each n in intermediateNodes{
4.    if (currentRegion doesn't contains addr(n)){
5.      insert (descendants of n) to answer;
6.      currentRegion = region of n;
7.      remove node whose address is contained
8.          by currentRegion from leafNodes;
9.    }
10. }
11. return answer + leafNodes;
```

2–8 行目で intermediateNodes の各ノードから子孫問合せを行う．もし，現在のリージョン currentRegion に，$addr(n)$ が含まれていれば重複しているため，子孫問合せを行わない．そうでなければ，互いに独立であるため，子孫問合せを行う．(4–5 行目) その後，取得したノードのリージョンを currentRegion に代入し，currentRegion に含まれている全ての葉ノードを leafNodes から取り除く．(6–8 行目) 最後に，取得した子孫と，葉ノードを解として返す (11 行目)

提案手法で必要な計算量は，currentRegion のみのため，チェックを行うために必要な領域は $O(1)$ となる．また，包含関係のチェックは 1 ノードあたり，intermediateNodes の場合 $O(1)$，葉ノードの場合 $O(\log |leafNodes|)$ となる．よって，必要なチェックに必要な計算量は，$O(|intermediateNodes| + |leafNodes| \log |leafNodes|)$ となる．

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 11　実験 2. 問合せとランダムアクセス数

4. 評 価 実 験

提案手法を評価するための実験を行った．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 stack-tree join での子供問合せの実行時間を比較した，実験で用いたグラフはエッジを持たない完全二分木であり，以下では深さ 17,18,19 の 3 つの完全二分木の結果を記す．

実験結果． 実験では，$A_i \rightarrow X$ と，structural join $A_i/V (1 \leqq i \leqq 8)$ の実行時間を測定した．ここで，A_i は深さ i のノード (2^{i-1} ノード) であり，V は，完全二分木に含まれる全てのノードである．問合せの解，$|X|$ は 2^i ノードとなる．図 7 に $2^{19} - 1$ ノードでの実験結果を示す．また，図 8 に，3 つの完全二分木での $A_8 \rightarrow X$ と A_8/V の実行時間を示す．実験結果から，structural joins の実行時間が木のサイズに依存しているのに対して $A_i \rightarrow X$ の実行時間は，解の数に依存していることが分かる．また，既存の手法よりも高速に子供問合せを処理できることが分かる．

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 11 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，

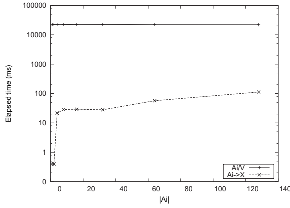


図 7　実験 1: 結果 1

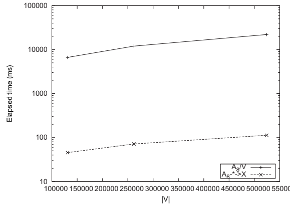


図 8　実験 1: 結果 2

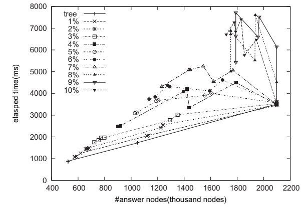


図 9　実験 3: 結果

非木エッジの割合 (%)	0	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 10　実験 3. 解 1 ノードあたりの実行時間

	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 12　実験 2. 実行時間

問合せを処理するために必要な最悪のランダムアクセス数を計算した．

DF: ノードを深さ優先順に配置する．各子供の順序は任意とする．

Proposal: 2.2 節で説明したノード順序で，ノードを配置する．

ランダムアクセス数. 図 11 の 3–4 列目に，その結果を示す．結果を見ればわかるとおり，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ランダムアクセスの回数が圧倒的に少ない．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．ランダムアクセス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 12 に示す．

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各実行時間がどのように変化するかを調べるために実験を行った．実験では，深さ 21 の完全二分木を全域木として，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 9 に非木エッジ 1–10%での実行時間を示す．また図 10 に非木エッジの割合が 0%(木構造)–10%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，木での実行時間の約 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.　まとめと今後の課題

本論文では，大規模なグラフ構造に対して正規パス式を効率的に処理するために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が考えられる．

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] 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

[2] 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)

[3] 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.

[4] 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)

[5] 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.

[6] 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.