メタデータDBのための到達ノード問合せと更新処理の効率化

石川　憲一†　　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
†† 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
〒 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 36–1
E-mail: †{ishiken,mori}@slis.tsukuba.ac.jp, ††tajima@i.kyoto-u.ac.jp

あらまし　近年，デジタルデバイスの機能強化やコンピュータネットワークを通じた情報共有技術の発達により大量のメタデータが生成・蓄積されるようになってきた．そのため，このようなメタデータの管理は重要な問題であると言える．メタデータは様々な形式で記述されるが，RDF 等に代表されるエッジラベルを持つグラフ構造は代表的なフォーマットの一つである．本論文では，エッジラベルを持つグラフ構造のデータを対象として，あるノードから到達可能であるノードの問合せに焦点を当てる．到着可能ノード問合せの例としては RDF スキーマにおけるサブクラスの関係をもちいて，あるクラスの全ての派生クラスを求めたりするなど応用範囲は広い．本論文では，到達ノード問合せを効率よく実行するためのデータ格納・問合せ処理手法，およびデータ更新手法について提案する．
キーワード　メタデータ，性能評価，問合せ処理，半構造データ

Efficient Evaluation of Reachable Node Queries and Updates for Metadata Databases

Kenichi ISHIKAWA†, 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 Library, Information and Media Studies/ Research Center for Knowledge Communities, Univ. of Tsukuba.　1–2 Kasuga, Tsukuba, Ibaraki, 305–8550 Japan
††† Grad. Sch. of Informatics, Kyoto University
36–1 Yoshida-Honmachi, Sakyo-ku, Kyoto 606–8501, Japan
E-mail: †{ishiken,mori}@slis.tsukuba.ac.jp, ††tajima@i.kyoto-u.ac.jp

Abstract　The recent development of information sharing technologies and the advances of various digital devices make a large amount of metadata generated and accumulated. Therefore, how to manage such metadata is an important problem. Edge-labeled graphs, such as RDF graphs, are one of the major formats for metadata. This paper focuses on the problem of the efficient processing of reachable node queries against edge-labeled graphs. The problem has a wide range of applications, including computing all of the subclasses of a given class with its sub-class relationships. This paper proposes techniques for storing, querying, and updating metadata for the efficient execution of reachable node queries.
Key words　Metadata, Performance Evaluation, Query Processing, Semistructured Data

1. はじめに

近年，データに関するデータであるメタデータが注目を集めている．デジタルデバイスの機能強化やコンピュータネットワークを通じた情報共有技術の発達により，大量のメタデータが生成・蓄積されるようになってきた．

メタデータは様々な形式で記述されるが，RDF [1] は代表的なフォーマットの一つである．RDF データは，エッジラベルを持つグラフ構造としてモデル化できる．これまで，RDF データベース等，数多くのメタデータデータベースにおける問合せ処理の研究が行われてきたが，これらにおいては，SPARQL [2] の basic pattern に代表されるような，閉包を含まないパターン

にマッチする部分グラフを効率よく発見する処理に主眼が置かれてきた．

本論文では，エッジラベルを持つグラフ構造のデータを対象として，あるノードから到達可能ノード問合せに焦点を当てる．具体的には，本論文で議論する到達可能ノードの問合せとは，次の 4 種類からなる問合せのクラスである．

(1) $a \xrightarrow{t} x$: 与えられたノード a を起点としたラベル t を持つエッジの終点となるノードの集合

(2) $a \xrightarrow{t*} x$: 与えられたノード a を起点としてラベル t を持つエッジを 1 回以上たどって到達可能なノードの集合

(3) $x \xrightarrow{t} a$: 与えられたノード a を終点としたラベル t を持つエッジの起点となるノードの集合

(4) $x \xrightarrow{t*} a$: 与えられたノード a までラベル t を持つエッジを 1 回以上たどって到達可能な起点となるノードの集合

このクラスの問合せの応用範囲は広い．例えば，RDF スキーマにおけるサブクラスの関係をもちいて，あるクラスの全ての派生クラスを求めたり ($a \overset{subclassof*}{\rightarrow} x$)，上位クラスを求める ($x \overset{subclassof}{\rightarrow} a$) といった演算は，このクラスの問合せに含まれる．また，論文間の参照関係を表すメタデータなどに対しても，このクラスの問合せが重要となる．

本論文では，上記問合せクラスを効率よく実行するためのデータ格納・問合せ処理手法，およびデータ更新手法について提案する．現実に存在するグラフ構造データの多くは疎であり，非木エッジ (non-tree edges) の個数 t は総ノード数 n と比較したときに $t << n$ と考えられる [4]．そこで，本手法では，対象となるグラフの構造は木とし (以下**データ木**と表記)，非木エッジの扱いは特殊ケースとして扱うことにする．

アイデアを要約すると，(1) 可能な限りシーケンシャルアクセスで問合せを実行できるようにディスク上にデータ木のノードを配置すること，および (2) 関連ノードを出来るだけ近くに配置すること，である．具体的に説明するため，データ木に対して，問合せとして子を求める $a \xrightarrow{t} x$ と 子孫を求める $a \xrightarrow{t*} x$ を実行することを考える．この場合，ノードを深さ優先順にディスクに配置すれば，どちらもシーケンシャルアクセスで答えを求めることが出来る．しかし，子を求める問合せ $a \xrightarrow{t} x$ の処理効率に関しては工夫の余地がある．なぜなら，深さ優先順の配置では，一般に子ノードの位置は連続しないからである (図 1)．特に，子孫ノードの数が多い場合，子ノードの位置はお互いに遠く離れてしまう．本提案手法では，ノードの配置を工夫することにより，子ノードの検索も効率よくできるようにする．また，他の工夫も組み合わせて，4 つの問合せを効率よく実行可能な手法を提案する．

ここで提案している「関連ノードを近くに連続して配置することにより可能な限りシーケンシャルアクセスで解を求める」手法では，ノードの適切な配置を維持することがポイントであり，ノードの追加などではノードの配置をずらす必要がある．したがって，更新コストが大きくなってしまう．そこで，本論文では，更新に起因するノード再配置を「遅延」する手法についても提案する．これにより，実際の再配置は，計算リソースが空いているときに実行することが出来る．

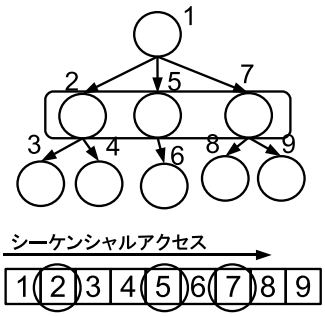

図 1　深さ優先順記録

関連研究．的野らの研究 [3] では，RDF グラフを述語の種類によって複数の部分グラフに分割し，それぞれ異なる方式で RDB に格納することを提案している．そこでは，subClassOf や subPropertyOf の関係は，インターバルナンバリングスキームを利用して格納することにより，到達可能なノードの問合せを実現している．我々の研究は，このような種類のグラフに対してより特化したデータ格納方式・問合せ処理方式を開発しようとするものである．また，H. Wang らはグラフデータベースにおいて 2 つのノードが与えられたとき，それらの到達可能性を判定するための効率のよい手法を提案している [4]．これに対し，本研究はあるノードが与えられたときにそのノードから到達可能な全てのノードを列挙するための高速な手法を実現するデータ格納方法を提案する．S. Al-khalifa らは Structural Join の処理方式の議論において，通常の深さ優先順では子問合せの解が効率よくまとまらない事を指摘している [5]．本論文で扱う問合せのクラスは一般の Structural Join とは異なるが，一つの起点ノードからの到達ノード問合せの問題に対してこの問題への解を与えるものである．油井ら [6] は，XML データを DTM に基づいてディスク上に配置し，XQuery 問合せを効率よく実行することを提案している．対象とする問合せが本研究と異なるため単純な比較は出来ないが，ナイーブな DTM ベースでは子ノードが分散して配置されるため，大規模データにおいて全ての子ノードを求める場合の実行コストが大きくなるのではないかと推測される．

本稿の構成は次の通りである．2 章では本研究のアプリケーションの一つである「情報空間統治のためのメタデータデータベースシステム」について簡単に説明する．本研究は特定のシステムに特化したものではないが，このシステムは，本研究の動機となったものである．3 章では，可能な限りシーケンシャルアクセスで問合せを処理可能なノード配置方式について提案する．4 章では，データ更新の際にノードの再配置を遅延するための方式について提案する．5 章では実験について述べる．6 章はまとめである．

2. アプリケーション例：情報空間ガバナンスのためのメタデータデータベース

現在，我々が進めているコミュニティ情報空間ガバナンス (Governance of Community Information Spaces) プロジェクトは，各クライアント PC やファイルサーバに格納されている大量の

図 2　インフォメーションスペースの例

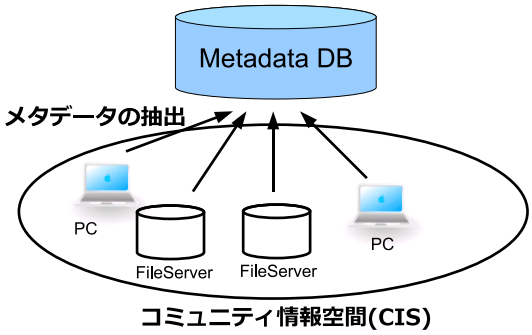

図 3　コミュニティ情報空間管理のためのメタデータデータベース

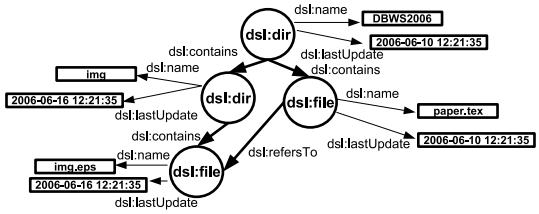

図 4　情報空間コミュニティメタデータの例

ファイル群の管理を行うシステム (InfoSpace Governor) を開発している．

本システムの第一の特徴は，個々のファイルシステムではなく，**コミュニティ情報空間** (CIS) と呼ぶ複数のファイルシステム群を含む空間を管理の範囲としていることである．図 2 はある大学の研究室における CIS の例である．一般に，CIS には複数の計算機が含まれており，それぞれが情報を分散して管理している．例えば，この研究室で行われているプロジェクトに関連する情報は，それら複数の計算機によって分散されて管理されている．それらに管理されている情報は，お互い関連しているものの，それらはシステムレベルでは明示的には関連づけられていない．例えば，そのプロジェクトに関連するあるファイルの最新バージョンはどれか，といった情報や，そのファイルが参照するファイルはどれか，といった情報をシステムはもっていない．したがって，これらの問にシステムが答えることは出来ない．ネットワークを通じてこれらの情報源が物理的には接続されているにもかかわらず，先に述べたような関連のレベルでは実は接続されていないのである．

本システム第二の特徴は，これらの関連を明示的に保持することにより，情報空間の統治を行う事である．具体的には，今説明したような情報資源間の関連を明示的に保持しするためのメタデータデータベースを構築し，コミュニティ情報空間の管理を行う (図 3).

2 章で詳しく説明するが，このメタデータデータベースには図 4 のような RDF [1] データが格納される．dsl:dir, dsl:file のノードはそれぞれディレクトリとファイルを表す RDF のクラスであり，dsl:name, dsl:lastUpdate のエッジは各ノードのファイル名や最終更新日時を表す RDF のプロパティである．dsl:contains はディレクトリとファイルの包含関係を表し，dsl:refersTo はノード間の参照関係 (例えば latex のソースと画像ファイルなど) を表すプロパティである．

本システムでは，図 4 のようなメタデータに対し，特定のラベルを持つエッジを通じて到達可能なノードを対象とした問合せが多く使われる．例えば，図 4 のファイル「img.eps」を削除したときに問題となるファイル群は，1 章の記法では $x \overset{refersTo*}{\rightarrow} a$ という演算で求められる．コミュニティ情報空間に含まれるファイル総数は数十万以上になることが多いため，大規模なメタデータに対してこれらの演算を効率よく行うことが要求される．

3.　効率よい問合せ処理のためのノード配置方式

3.1　概　　要

本提案手法では，まず，与えられたグラフ G をエッジの種類で射影した部分グラフ $G_{type_1}, G_{type2} \ldots$ を作成 (図 5) する．これらが木でない (非木エッジ持つ) 場合には後述する方法によって木に変換する．次に，各 G_{type_i} を次に説明する方法で格納する．

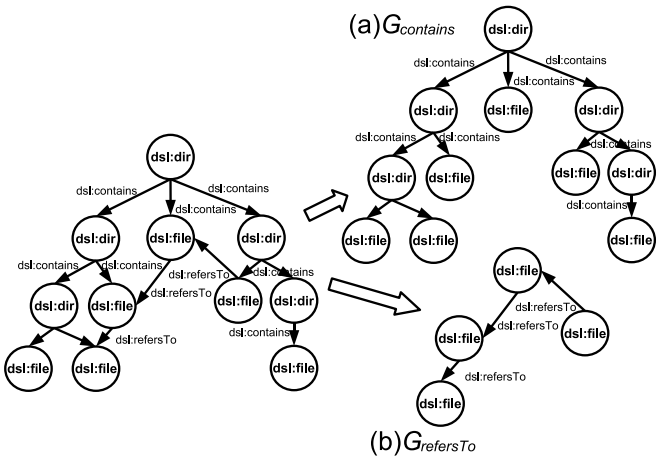

図 5　グラフの分割

3.2　各 G_{type_i} の格納方法

ここでは，図 5(a) の $G_{contains}$ を例にとり説明する．図 6 は，$G_{contains}$ を提案手法で格納する場合のデータをテーブルの形で表現したものである．後述するように，図中の各テーブルは，実際にはそれぞれ 1 ファイルとして格納され，ファイル内部では固定長のタプルが上から順にディスク上に並べられている (図 7)．次に，これらのテーブルについて順に説明する．

(1) 子孫データ．これは，親，子，および子孫間合せ ($a \overset{t}{\rightarrow} x$,

（a）子孫データ

node_id	parent	firstChild
101	-	2
102	1	9
103	1	-
104	1	5
105	4	-
106	4	7
107	6	-
108	6	-
109	2	11
110	2	-
111	9	-

（b）先祖データ

node_id	node_id list
101	-
102	101
103	101
104	101
105	104, 101
106	104, 101
107	106, 104, 101
108	106, 104, 101
109	102, 101
110	102, 101
111	109, 102, 101

図 6　子孫データ（a）および先祖データ（b）

①			②									③		
1			2			3			4			5		
101	-	2	102	1	9	103	1	-	104	1	5	105	4	-

…

図 7　ディスク上に記録された子孫データのイメージ

$x \xrightarrow{t} a, x \xrightarrow{t*} a,$) を処理するために利用する．

1 章で説明したように，基本的な考え方は，小さな範囲のシーケンシャルアクセスで必要な情報を得るようにノードを配置することである．単純に深さ優先順にノードを並べると子の計算に時間がかかるため，提案手法では次のような順で番号を振り，その順にノードを配置する．

ノード配置順序. 下記の順序に矛盾しないようノードに番号を振り，その順に並べる．

- (a) 各兄弟ノードごとにグルーピングしたとき，それらのグループは深さ優先順で並ぶ．
- (b1) グルーピングされた兄弟ノードは連続して並ぶ．ただし，(b2) 上記グループの深さ優先順におけるグループの子供の順序と逆順に並ぶ．

上の (a) と (b1) により，子および子孫問合せのいずれに対してもノードが連続して並ぶことになる．また，(b2) によって，後述するように問合せ処理の終了判定が容易になる．

例として，図 8 に $G_{contains}$ の場合の配置順序を表す番号付けを示す．①〜⑥までの番号は兄弟グループの記録順（深さ優先順）を示している．各ノードに付与された 1〜11 までの番号がノードの物理的な記録順を表す番号である．以下では，これら 1〜11 までのノード配置順序の番号を**アドレス**と呼び．ノード n のアドレスを $addr(n)$ と表記する．(101 から 111 までの番号はノードを一意に識別するための ID であり，**ノード ID** と呼ぶ)．このとき，子孫データは図 6(a) のようになる．このテーブルの一つのタプルは一つのノードに対応し，上記で決められた配置順に並んでいる．各タプルは属性として，各ノードのノード ID，その親ノードの位置を表すアドレス，および，その子供ノードの先頭位置を表すアドレスを持つ．

図 6(a) を見れば，各ノードの子ノードは，その親ノードの近くに連続して並んでいることが分かる．例えば，アドレス 1 のノードの次にはその子である 2, 3, 4 のノードが連続して記録されている．したがって，その次に記録されているノードからシーケンシャルにデータを読み込むだけで子ノードを取得する事ができる．ノードの $parent$ 値（親ノードのアドレス）が 1

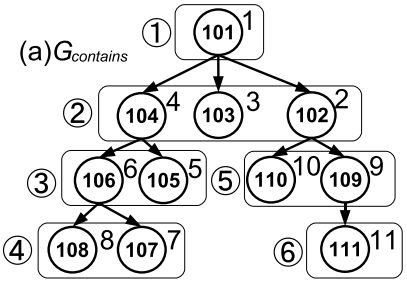

図 8　ノードの配置順序

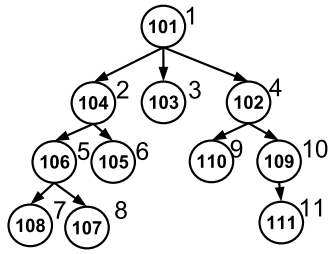

図 9　兄弟ノードの並び順が逆ではない場合

以外のノードに到達した時点で終了である．アドレス 2 の子である 9, 10 も親の記録場所の直後では無いものの一箇所に連続して記録される．2 の子を取得する場合も 9 番目のノードの場所に一度移動してしまえば，後はシーケンシャルアクセスで子ノードを取得する事ができる．

子孫ノードの計算 ($a \xrightarrow{t*} x$) に関しては，a の子供の後ろに子孫も連続して並んでいるため，そのままシーケンシャルアクセスでノードを取得する．ノードの $parent$ 値 (親ノードのアドレス）が a のアドレスよりも小さなものに到達した時点で終了である．兄弟ノードの配置順序を逆にしているのはこの子孫ノードの計算が理由である．図 9 は図 8 の木の兄弟ノードの配置順序を逆ではなくしたものある．この場合，例えばノード ID104 の子を同じように求めようとすると，5 番目のノードに移動しシーケンシャルアクセスを行う．しかし，5 番目のノード（ノード ID106) 以降には，ノード $parent$ 値 (親ノードのアドレス）が a のアドレス (この場合 2) よりも小さくなるノードは存在せず，どこまでが子孫ノードなのかが分からない．このように子孫ノードの計算をする場合には，兄弟ノードを逆順に並べる事が重要である．

親ノードの計算 ($x \xrightarrow{t} a$) はシーケンシャルには出来ないが，一回のランダムアクセスで可能なため短時間で実行可能と期待される．

図 7 に実際にディスク上に記録された子孫データのイメージを示す．上の段と真ん中の段はそれぞれ兄弟グループの記録順と各ノードの記録順を表す番号であり，実際に記録されるのは一番下の段だけである．なお，各ノードのアドレスはノードのデータとして明示的には記録されないことに注意して欲しい．理由は，アドレスはノードの配置順を表しているため，物理アドレスから計算できるからである．

(2) 先祖データ. これは，$x \xrightarrow{t*} a$（先祖問合せ) のためのデータである．$G_{contains}$ の先祖データを図 6(b) に示す．a に対応する

node_id	addrID(contains)	addrID(refers)
101	1	
102	2	
103	3	2
104	4	
105	5	3
106	6	
107	7	4
108	8	
109	9	
110	10	1
111	11	

図 10　ノード表

ノード ID 毎に，固定長のエントリを用意し，先祖を表すノード ID リストを格納する．先祖を表すノード ID リストはオーバラップすることが多いため，データ量の圧縮が可能であるが，ここでは最も単純な格納方法で説明している．

3.3　ノード表

本提案手法ではグラフ G がエッジタイプ毎のグラフに分割して格納されるため，分割されたグラフを統合するための情報が必要になる．本手法では，その情報をまとめてノード表として保持する (図 10)．各タプルは，ノード ID と各エッジのラベル毎にアドレスを保持している．ノード表を見ることにより，与えられた $node\_id$ とエッジのラベル $type_i$ から，グラフ G_{type_i} におけるアドレスを返すことが出来る．

3.4　問合せ処理の方法

本節では 4 種の問合せの具体的な問合せ処理方法について説明する．

[$a \xrightarrow{t} x$ (子問合せ)]

（1） ノード表を参照し，a のアドレス $addr_t(a)$ が示すアドレスにアクセスする．

（2） G_t の子孫データを参照し，a の最初の子を表すアドレス $firstChild$ の値を取得する．

（3） 取得した $firstChild$ が指すアドレスが表すノードのアドレスに移動する．

（4） シーケンシャルにノードを読み続ける．$parent$ の値が $addr_t(a)$ である限りデータを読み続ける

[$a \xrightarrow{t*} x$ (子孫問合せ)]

（1） ノード表を参照し，a のアドレス $addr_t(a)$ が示すアドレスにアクセスする．

（2） G_t の子孫データを参照し，a の最初の子を表すアドレス $firstChild$ の値を取得する．

（3） 取得した $firstChild$ が指すアドレスが表すノードのアドレスに移動する．

（4） シーケンシャルにノードを読み続ける．$parent$ の値が $addr_t(a)$ 以上である限りデータを読み続ける

[$x \xrightarrow{t} a$ (親問合せ)]

（1） ノード表を参照し，a のアドレス $addr_t(a)$ が示すアドレスにアクセスする．

（2） G_t の子孫データを参照し，a の親を表すアドレス $parent$ の値を取得する．

（3） 取得した $parent$ が指すアドレスが表すノードのアドレスに移動する．

（4） ノードを一つだけ読む．

[$x \xrightarrow{t*} a$ (先祖問合せ)]

（1） a の $node\_id$ から a の先祖の $node\_id$ リストが記録されている物理的な記録位置を計算しアクセスする．

（2） G_t の先祖データを参照し，シーケンシャルにノードを読み続ける．$node\_id$ リストが終了しない限りデータを読み続ける．

3.5　非木エッジの扱い

非木エッジとはグラフのスパニングツリーを作成する際に削るエッジの事である．G_{type_i} が非木エッジを持つ場合，複数の到着エッジを持つノードのコピーを作成する．コピーするノードの $node\_id$ はコピーしたノードの $node\_id$ と同一とする．この場合ノード表には，同一の $node\_id$ が複数出現することになる．そしてディスク上に記録されている $node\_id$ の値のうち上位 1 ビットにコピーノードが存在するというフラグを記録する．これにより問合せを実行する際にコピーノードが存在する事が分かる．

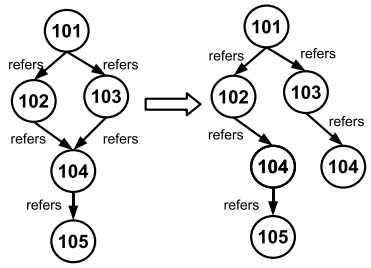

図 11　非木エッジの取り扱い例

図 11 に非木エッジが存在する場合の処理の例を示す．この場合，複数の到着エッジを持つノードは $node\_id$ が 104 のノードである．よってこのノードのコピーを作成する．そして，ノード表に $node\_id$ が 104 である両方のノード情報を記録する．最後に，ディスク上の $node\_id$ が 104 である両方のノードの $node\_id$ 値の上位 1 ビットにコピーが存在するというフラグを記録する．

例えば $node\_id$ が 103 であるノードの $a \xrightarrow{refers*} x$（子孫問合せ）の答えをを求める場合は次のような手順で問合せを実行する．最初に 103 の子である 104 を読み込む．この際，$node\_id$ の値上位 1 ビットにコピーノードが存在するというフラグが記録されているので，ノード表から $node\_id$ が 104 であるもう一つのノードを発見する．その上で，今度は新しく発見した 104 の子孫を取得する．最後に重複するノードを除去する事で 103 の子孫 $node\_id$ である 104 と 105 を取得する事ができる．

このように非木エッジが存在する場合はコピーされたノードが存在する分，処理回数が増えてしまう場合がある．しかし，1 章で述べたように現実のグラフ構造では非木エッジの個数は多くないと考えられるため，大きな問題にはならないと予想される．

4.　データの更新手法

本システムでは子ノードを優先して並べるようにノードの物

理的な並び順を規定している．この方法ではノードの挿入が発生した場合，挿入位置よりも後に存在するノードを物理的に移動させ，再配置を行う必要がある．しかし，ノードを再配置するコストは非常に高いため更新コストが大きくなってしまう．そこで本システムではノードを挿入する際は物理的なノードの再配置を行わず，空いている記憶領域に挿入ノードの情報を記録する．そしてノードが本来挿入されるべき論理的なアドレスと実際にノードが存在するアドレスを対応付けるための論理アドレス変換表を用意する．これによりノードの挿入による再配置は生じず，論理アドレス変換表の値を書き換えるだけで済み，更新コストを下げる事ができる．そして物理的なノードの移動を伴う実際の再配置は計算リソースが空いているときに実行する．図 12 は論理アドレス変換表を利用した問合せのイメージ図である．問合せを行う方から見ると単一の木に見えるが実際には更新されたノードが空いている別の記憶領域に保存されている．

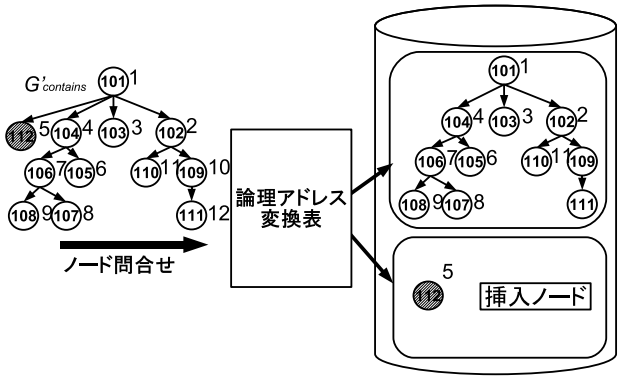

図 12　論理アドレス変換表の使用によるノード再配置の遅延

図 13 はノードの挿入が行われる以前の図 8 の $G_{contains}$ の論理アドレス変換表である．論理アドレス変換表は min, max, offset の 3 つの値の組からなっている．図 13 はアドレス 1 から 11 番目までのノードの実際のアドレスは offset の値 0 を足したものである事を意味している．つまり論理アドレスが 1 のノードは実際にアドレス 1 の位置に記録されており 2 のノードも実際にアドレス 2 に記録されている事を意味している．問合せ実行しデータを読み込む際には論理アドレスから実アドレスを求めながらデータの読み込みを行う．

min	max	offset
1	11	0

図 13　ノード挿入前の論理アドレス変換表

次に木にノードを挿入する場合の処理方法を説明する．図 14 に図 8 の $G_{contains}$ にノードが新しく一つ挿入された状態の木を示す (以下 $G'_{contains}$ とする)．アドレスが 5 であるノード（$node\_id$ は 112）が新しく挿入されたノードである．ノードが挿入されるので，それまで 5〜11 までのアドレスだったノードのアドレスが 6〜12 とそれぞれ 1 加算される必要がある．図 15 は $G'_{contains}$ の論理アドレス変換表である．この表は 1〜4 までのノードの offset が 0 であるので，論理アドレスの 1 から 4 までのノードは実際に 1 から 4 のアドレスに記録されている．

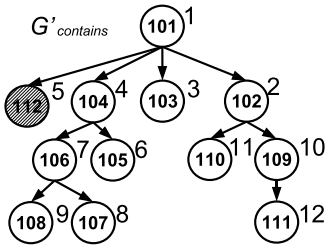

図 14　ノード挿入後の木

5 のノードは offset の値が 7 であるので，7 を加算した 12 のアドレスに記録されている事を意味している．同じように 6〜12 までのノードは offset の値-1 を加算した 5〜11 番目にそれぞれ記録されている事を表している．例えばこの表を利用して 1 の子を求める場合，1 の最初の子のアドレスは 2 であるので論理アドレス変換表の 2 の値を参照する．2 の value は 0 であるので 2 番目に記録されているノードのデータを読み込む．3 は 3 番目，4 は 4 番目というように同じようにデータを読み込む．5 のノードにアクセスする際は論理アドレス変換表の 5 の value は 7 であるので 5 に 7 を加算した値である 12 のアドレスに記録されている事が分かる．よって 12 番のアドレスの位置に飛び，データの読み込みを行う．

データを読み込む際には論理アドレスから実アドレスを求めながらデータの読み込みを行う．しかし，論理アドレス変換表はデータに比べ小さいためオンメモリでデータを持ち，論理アドレス変換表を参照するコストは大きくない．また，論理アドレス変換表はアドレスが小さい順で並んでいるため 2 分探索が利用でき，計算量は $O(\log N)$ で抑える事が可能である．

min	max	offset
1	4	0
5	5	7
6	12	−1

図 15　ノード挿入後の論理アドレス変換表

5. 性能評価実験と考察

本章では我々が開発したシステムの性能評価を行う．

5.1 実験環境

本実験では筑波大学森嶋研究室のファイルサーバ中のデータ 49,350 ノードを表現する情報コミュニティメタデータのうち，ディレクトリとファイルの包含関係を表す $G_{contains}$ で 4 種の問合せの実行し，速度を測定した．$G_{contains}$ には非木エッジは一つも存在していない．$G_{contains}$ 木の深さの最大値は 22，根ノードから葉ノードへのパスの平均長は 10.4 である．又，一つ以上の子を持つノード（葉ノード以外）の子の数の平均は 8.6，子ノード数の最大値は 819 である．$a \xrightarrow{t} x$，$a \xrightarrow{t*} x$ で実行した問合せの a のノードの深さは 3，子の数は 45，子孫の数は 2,5529 である．$x \xrightarrow{t} a, x \xrightarrow{t*} a$ で実行した問合せの a のノードの深さは 15，子と子孫の数は 0 である．本実験では，子孫データと先祖データはファイルとしてディスク上に記録されている．又，ノード表は RDB に格納されている．実験に使用した計算機環境を図 16 に示す．

CPU	Intel PentiumM(1.73Ghz)
Memory	512MB
OS	Windows XP Professional SP2
RDB	MySQL 5.0.27 (InnoDB)
Java	JDK 1.5.0_10

図 16　実験に使用した計算機環境

5.2 比較手法

本節では我々のシステムと比較を行う各手法・システムの説明を行う．

5.2.1 XRel

XRel [7] は RDB を用いた XML データベースである．XRel では XML 文書を要素ノードを格納する Element，属性ノードを格納する Attribute，テキストノードを格納する Txt，XML 文書に出現する全ての経路を格納する Pth の 4 つのリレーションに分解して格納する．XPath 等で問合せが実行される際には SQL に変換され問合せが実行される．本実験では先ほど述べた $G_{contains}$ のグラフを XML 文書として表現しデータベースに格納した．格納する XML 文書の DTD を図 17 に，XML 文書の例を図 18 に示す．この XML 文書ではディレクトリとファイルの包含関係をタグの入れ子構造で表現し，ノード ID を id という属性ノードで表現している．本実験で格納した XML 文書の大きさは 12.4MB，要素ノード数は 289,679，属性ノード数は 49,366，テキストノード数が 232,977，経路数は 252 である．実験に際しては全てのタプルにインデックスを作成した上で問合せを行った．なお，XRel の実験では XPath を SQL に変換するためのコストは計算には入れていない．

```
<!ELEMENT dir (name,lastUpdate,path,dir?,file?)>
<!ATTLIST dir id NMTOKEN #REQUIRED>
<!ELEMENT file (name,ext,lastUpdate,path,size)>
<!ATTLIST file id NMTOKEN #REQUIRED>
<!ELEMENT lastUpdate (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT path (#PCDATA)>
<!ELEMENT size (#PCDATA)>
<!ELEMENT ext (#PCDATA)>
```

図 17　格納する XML 文書の DTD

```
<dir id="1">
  <name>root</name>
  <lastUpdate>20061212</lastUpdate>
  <path>/root</path>
    <dir id="2">
      <name>a</name>
      <ext></ext>
      <lastUpdate>20061212</lastUpdate>
      <path>/root/a</path>
    </dir>
    <file id="3">
      <name>hoge.txt</name>
      <ext>txt</ext>
      <lastUpdate>20061212</lastUpdate>
      <path>/root/hoge.txt</path>
      <size>2</size>
    </file>
</dir>
```

図 18　格納する XML 文書の例

5.2.2 eXist

eXist [8] は Java で記述されたオープンソースのネイティブ XML データベースである．eXist に格納する XML 文書は XRel と全く同一の物を格納した．問合せには XML:DB API を利用し，XPath で問合せを行った．実験に用いた eXist のバージョンは 1.1.1 である．図 20 にノード ID が$node_id であるノードの子・子孫・親・先祖ノードのノード ID を求める場合の XPath 式を示す．

```
子： //*[@id = $node_id]/*/@id
子孫： //*[@id = $node_id]//*/@id
親： //*[@id = $node_id]/parent::node()/@id
先祖： //*[@id = $node_id]/ancestor::node()/@id
```

図 20　eXist で実行した XPath 問合せ

5.3 問合せ性能

図 19 に問合せの実行結果を示す．実行時間はそれぞれ 10 回行った平均値である．$a \xrightarrow{t} x$(子問合せ）で取得されるノード数は 45，$a \xrightarrow{t*} x$（子孫問合せ）で取得されるノード数は 25529，$x \xrightarrow{t} a$（親問合せ）で取得されるノード数は 1，$x \xrightarrow{t*} a$（先祖問合せ）で取得されるノード数は 14 である．実行した問合せはノード ID を指定し 4 種の問合せでそれぞれ取得されるノードのノード ID を取得する問合せである．XRel と eXist は Java プログラムから接続し問合せ結果のセットを取得するまでの時間で，本システムでは $node\_id$ のリストを取得するまでの時間で計測した．

実験結果からは本システムは eXist と比較して 4 つ全ての問合せで極めて高速に問合せ処理を実行できている事が分かる．eXist の処理に膨大な時間がかかっており，親ノードを 1 つだけ取得する場合でも 3500 ミリ秒以上の時間がかかっているのはノード ID を指定する問合せの起点となるノードの取得に大きな時間がかかっているためと予測される．XRel では子と親の問合せはほぼ互角であるが，子孫問合せと先祖問合せで大きな差が出ている．

5.4 データ更新後の問合せ性能

本システムではデータ更新の際にノードの再配置を遅延させているため，再配置が実行されるまでは完全なシーケンシャルアクセスにはならず，問合せ性能が低下する事が予想される．そこで本節ではデータ更新後，再配置が行われる前と後での問合せ性能の差を測定した．本実験では (1)1000 ノードをまとめて一箇所に挿入する場合と (2)1000 ノードをランダムに挿入するという 2 種類のノード追加方法で実験を行った．論理変換表のサイズは (1) の場合が 3，(2) の場合は 2001 であった．なお，(1) の実験では子と子孫問合せの場合には問合せ対象に新しく挿入された 1000 個のノードが含まれるようにする．親と先祖を求める問合せでは，新しく挿入されたノードの親と先祖を求めた．(2) の実験では子・親・先祖問合せに関しては (1) の結果と大きく異なるとは思われないため，子孫問合せのみを実行した．実験の結果を図 21 と図 22 にそれぞれ示す．

(1) の実験結果から 1000 ノードをまとめて一箇所に追加した場合の問合せ処理時間は，再配置後と比べてあまり大きな変化は無い事が分かる．又 (2) の場合ではノード再配置後と比べて問合せ処理にかかる時間が 2 倍となっているが，それでも，XRel, eXist の子孫問合せにかかる時間よりも短い．

問合せの種類	取得ノード数 (個)	XRel(ミリ秒)	eXist(ミリ秒)	本システム (ミリ秒)
$a \xrightarrow{t} x$ (子問合せ)	45	4.8	4357.9	1.6
$a \xrightarrow{t*} x$ (子孫問合せ)	25529	2488.5	7048.4	80.9
$x \xrightarrow{t} a$ (親問合せ)	1	3.2	3598.5	3.2
$x \xrightarrow{t*} a$ (先祖問合せ)	14	2573.6	3609.2	1.6

図 19　実 行 性 能

問合せの種類	取得ノード数 (個)	1000 ノードを一箇所に追加 (ミリ秒)
$a \xrightarrow{t} x$ (子問合せ)	1045	3.2
$a \xrightarrow{t*} x$ (子孫問合せ)	26529	101.5
$x \xrightarrow{t} a$ (親問合せ)	1	3.1
$x \xrightarrow{t*} a$ (先祖問合せ)	14	1.6

図 21　1000 ノードを一箇所に追加後の実行性能

問合せの種類	取得ノード数 (個)	1000 ノードランダム追加 (ミリ秒)
$a \xrightarrow{t*} x$ (子孫問合せ)	26529	156.3

図 22　1000 ノードランダム追加後の実行性能

6. ま と め

本論文ではエッジラベルを持つグラフ構造のデータを対象として，到達ノード問合せを効率よく実行するためのデータ格納・問合せ処理手法，およびデータ更新手法について提案と実装，およびその評価を行った．データ格納方法の特徴としては (1) 可能な限りシーケンシャルアクセスで問合せを実行できるようにディスク上にデータ木のノードを配置すること，および (2) 関連ノードを出来るだけ近くに配置すること，である．本提案手法では，ノードの物理的配置が意味を持つため，データ更新時にノード再配置のコストが大きいものとなる．そこで，データ更新に伴う再配置を遅延するための手法を提案した．実験の結果，到達ノード問合せを効率よく実行可能なことが分かった．

今後の課題としては，数 100 万ノード以上の大規模データに対する実験評価，および，より複雑な問合せに対応するための機構の検討，更新遅延の実用性の検討，などがあげられる．

謝　　辞

ゼミなどでご議論いただいた，筑波大学図書館情報メディア研究科杉本重雄教授，阪口哲男助教授，永森光晴講師に感謝いたします．また，XRel システムのコードを提供して頂いた京都大学情報学研究科吉川正俊先生ならびに筑波大学システム情報工学研究科天笠俊之先生に感謝いたします．

文　　献

[1] W3C. Resource Description Framework (RDF). http://www.w3.org/RDF/.
[2] W3C. SPARQL http://www.w3.org/TR/rdf-sparql-query/.
[3] 的野晃整，天笠俊之，吉川正俊，植村俊亮: “ 経路式に基づく RDF データの関係データベースへの格納 ”, 電子情報通信学会論文誌 vol.J88-D-I NO.3 pp.590-603, (2005).
[4] H. Wang, H. He, J. Yang, P. S. Yu, J. Xu Yu(2006): “ Dual Labeling: Answering Graph Reachability Queries in Constant Time ”, Proc ICDE, pp.75-86, (2006).
[5] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava and Y. Wu: “ Structural Joins: A Primitive for Efficient XML Query Pattern Matching ”, Proc ICDE, pp.141-152, (2002).
[6] 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 ”, 情報処理学会研究報告, Vol.2006, No.77, 2006-DBS-140(I), pp.87-94, (2006).
[7] M. Yoshikawa, T. Amagasa, T. Shimura and S. Uemura: “ XRel: Apath-based approach to storage and retrieval of XML documents using relational databases ”, ACM TOIT, 1(1), pp.110-141, (2001)
[8] eXist http://exist.sourceforge.net/