情報処理学会論文誌 Vol.53 No.12 2815–2828 (Dec. 2012)

グラフ縮約に基づくSPARQLクエリ並列化方法の設計および予備評価

千代 英一郎[1,a)] 宮田 康志[1] 西山 博泰[1]

受付日 2012年4月9日, 採録日 2012年9月10日

概要：大規模化するRDFデータに対する検索処理を効率化するために，並列化による豊富な計算資源の活用が試みられている．しかしながら，RDFデータは非均質なグラフ構造を持つため，単純なデータ分割に基づくアプローチでは十分な並列化効果が得られない．本論文ではこの問題を解決する新しいSPARQLクエリ並列化方法を提案する．提案手法の特徴は縮約グラフとよぶRDFデータの要約情報を生成し，これを用いてSPARQLクエリ処理を負荷の均衡化された並列タスクに分割する点にある．提案手法のプロトタイプ実装を用いて予備評価を行い，期待する負荷分散効果および性能向上が得られることを確認した．

キーワード：RDF，SPARQL，並列化，データベース，クエリ最適化

Design and Preliminary Evaluation of SPARQL Query Parallelization Method Based On Graph Contraction

Eiichiro Chishiro[1,a)] Yasushi Miyata[1] Hiroyasu Nishiyama[1]

Received: April 9, 2012, Accepted: September 10, 2012

Abstract: While data parallelization is known as one of the most effective query optimization scheme in general, SPARQL query parallelization problem poses considerable challenges such as load-balancing due to the non-uniform graph structure of RDF data. In this paper we present a new approach to SPARQL query parallelization. Our idea is to create a structure-reflected summary information called *contracted graph* from RDF data and exploit this to divide SPARQL query into balanced and minimized parallel tasks. Preliminary experimental results shows that our method can achieve nearly optimal speedups.

Keywords: RDF, SPARQL, parallelization, database, query optimization

1. はじめに

1.1 研究背景

Web上の多様なリソース情報を統一的に表現することを目的として，RDF（Resoure Description Framework）[23]とよばれる枠組みがW3Cによって標準化されている．RDFはラベル付き有向グラフ構造に基づくデータ形式であり，すべての情報をトリプルとよばれる主語，述語，目的語の3つ組によって表現する．属性の追加が容易に行える等の高い柔軟性と表現力を備えており，近年急速に普及しはじめている[6]．

RDFデータの検索や分析を行うために，SPARQL[24]とよばれる問合せ言語が標準化されている．SPARQLはSQLに類似した言語であり，これを用いてクエリを記述することで，RDFデータを格納したデータベース（RDFストア）から，指定した条件を満たすデータをひきだすことができる．以下はSPARQLクエリの例である．

```
select ?n ?a where {
  ?x name ?n. ?x age ?a. filter (?a < 18)
}
```

[1] 日立製作所横浜研究所
Yokohama Research Laboratory, Hitachi, Ltd., Yokohama, Kanagawa 244–0817, Japan
[a)] eiichiro.chishiro.qp@hitachi.com

このクエリは，RDF ストアに格納されたデータから，年齢が 18 歳未満の人の名前と年齢を取得するクエリである．?ではじまる文字列（ここでは ?x，?n および ?a）は変数を表す．クエリを実行すると，`where` 以降に指定されたすべての条件を満たす変数の値が検索され，`select` の後ろに並ぶ各変数の値が解（結果）として返される．

RDF データは基本的にスキーマレスであり，格納されているデータの構造が事前には分からない．そのため，注目したい情報がどのような述語により，どのような形式で格納されているかを把握するために対話的に検索を繰り返しながらデータ分析を行うような利用形態が多くなる．すなわちデータ分析の一連の過程

(1) データの RDF ストアへの格納（ロードおよび検索効率化のためのインデックス生成）
(2) RDF ストアから目的とする情報を抽出するためのクエリの作成
(3) (2) で作成したクエリにより得られた結果の検討

のうち (2) において「クエリを記述し，RDF ストアへ入力し結果を取得し，結果を検討しクエリを修正」という試行錯誤を繰り返すことになる．このような過程を効率的に進めるためには個々のクエリ処理を可能な限り短時間で行うことが望ましい．しかしながら，SQL と比べ SPARQL クエリの処理技術は発展途上であり，少しクエリが複雑になると結果を得るのに長時間待たされるということがしばしば生じる．長時間待たされたあげく，単純な記述ミスのため結果が得られない，という事態は多くの利用者の経験するところであり，改善が望まれている．

本研究の主目的は上記の (2) を支援するためのクエリ処理の高速化である．なおデータ分析においては (1) の処理時間短縮も重要であるが，(1) はデータ収集時に 1 度行えばよく，かつ夜間バッチ等による自動化も可能であるため，本研究では (2) を優先している．

クエリの処理効率を改善する様々な試みのうち，特に重点的に行われているのは並列化である．そのアプローチはおおきく 2 つ存在する．1 つは RDF ストア内部における主に演算レベルの並列化であり，関係データベースの場合と同様に，ボトルネックとなる結合演算の並列化や演算間のパイプライン並列化が行われている．ただし，これらは実行時にデータの分配やソートといった前処理が必要であり，オーバヘッドに見合う効果が得られる状況は限定的であることが指摘されている [9], [19]．

これに対して近年では MapReduce に代表される分散フレームワークを用いた並列化が注目されている．これは RDF データを物理的に分割して複数の計算機に配置しておき，クエリを計算機ごとに並列に処理するというアプローチである．この方法は計算機台数を増やすことでデータ量の増加に対応できるスケーラビリティを備えており，大規模データを効率的に処理するためのアプローチとして有望視されている．

このアプローチはクエリ処理が各計算機内で完結する場合，高い並列化効果が期待できる．しかしながら，対象である RDF データを分割した場合，必ず分割にまたがる処理が必要となるクエリが存在する[*1]．このようなクエリの処理においては，計算機間でのデータ転送が発生し，そのオーバヘッドが問題となる [12]．また MapReduce フレームワークの起動に数十秒を要するため，対話的なクエリ処理には不向きである．

1.2 提案手法

本論文では SPARQL クエリ処理の並列化に対する新しいアプローチを提案する．提案手法の特徴は，縮約グラフとよぶ RDF データの要約情報を事前に生成し，これを用いてクエリを排反な解集合を持つクエリ群に変換することで，クエリ処理を負荷の均衡化された並列タスクに分割する点にある．データの分割は不要であり，MapReduce アプローチのようなデータ分割にともなう効率低下の問題は生じない．

課題となるのは並列化により生成される各クエリの処理の負荷の大きさおよびばらつきである．SPARQL クエリ処理は部分グラフマッチングの一種であり，クエリが定めるグラフ構造にマッチする部分グラフのなかで，フィルタ節等で指定した条件を満たすものを探す処理である．そのため，RDF データの持つグラフとしての構造を考慮せず，単純に解空間を全ノード集合と見なして分割し並列処理（たとえば RDF グラフの全ノード集合 N を N_1，N_2 に均等分割して，クエリ変数の値が N_1 に含まれる場合と N_2 に含まれる場合に分けて並列処理）したのでは，処理の負荷が均等化されることは期待できず，十分な並列化効果が得られない．

ここで，もし入力されたクエリが定めるグラフ構造から，それとマッチする可能性の高い部分グラフ集合を含む領域を限定することができれば，その領域を均等に分割することで，負荷の大きさ・ばらつきともに大きく改善されることが期待できる．提案手法は，縮約グラフとよぶ元のグラフにおけるノードの接続構造（ノードに接続しているエッジのラベル）を保ちながら複数のノードを 1 つに集約したグラフを事前に生成しておくことでこれを実現する．

1.3 動機例

本節では，具体例を用いて提案手法の直感的なイメージを

[*1] 極端な例として述語が 1 つしかない RDF データを考える．このようなデータに対する以下のクエリ

```
select ?x1 ?x2 where {
  ?x1 p ?y1. ?x2 p ?y2. filter (?y1 <?y2).
}
```

を処理するためには，どのような分割を用いても必ず分割にまたがる処理が必要となる．

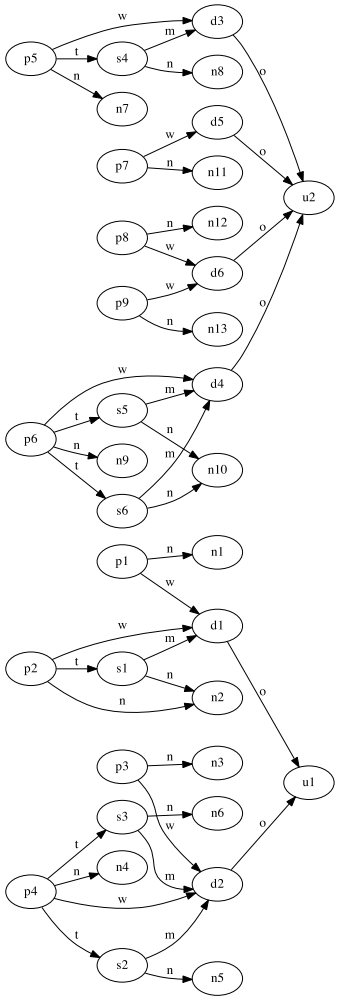

図 1 RDF グラフの例
Fig. 1 Example of RDF graph.

説明する．例として用いる RDF グラフを**図 1** に示す．これは標準的な RDF データセットの 1 つである LUBM [20] の一部を取り出したものである．ただし，述語名等は表記を簡略化するために変更している．

図 1 のグラフに対し，2 章で示す方法を用いて生成される縮約グラフを**図 2** に示す．なお，元のグラフと縮約グラフの対応が分かりやすくなるように，縮約グラフのノードには対応する元のグラフのノードを：の後ろに記している．たとえば，「1:d1-2」は縮約グラフのノード 1 が，d1 および d2 をまとめたものであることを表している．また，縮約グラフのエッジは，aw のように対応するエッジ名に a を付加したものとしている．図 1 および図 2 を比較すると，元のグラフのノードが持つ入力エッジおよび出力エッジのラベルとその縮約ノードの入力エッジおよび出力エッジのラベルは 1 対 1 に対応しており，接続構造が保存されていることが分かる．たとえば，ノード d1 および d2 はいずれも入力エッジラベルとして w および m，出力エッジラベルとして o を持つ．一方，これらに対応する縮約グラフのノード 1 は，入力エッジラベルとして aw および am，出力エッジラベルとして ao を持っている．

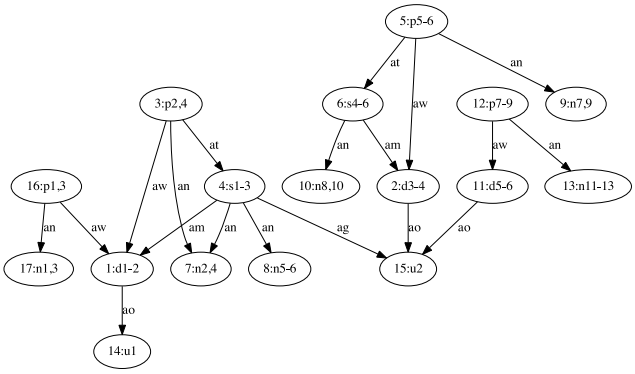

図 2 縮約グラフの例
Fig. 2 Example of contracted graph.

提案手法では，この縮約グラフの特性を用いてクエリの定める構造にマッチする部分グラフ集合を求め，それを分割して並列処理する．例として以下のクエリ q を考える．

`select` $?x\ ?y\ ?z$ `where` { $?x\ w\ ?y.\ ?z\ m\ ?y.$ }

最初に q を対応する縮約グラフ上のクエリ q_A に変換する．

`select` $?x\ ?y\ ?z$ `where` { $?x\ aw\ ?y.\ ?z\ am\ ?y.$ }

次に，縮約グラフから q_A にマッチする部分グラフ集合を求める．具体的には，q_A に含まれるクエリ変数の値を求める．図 2 では

$[?x \to 3, ?y \to 1, ?z \to 4]$
$[?x \to 16, ?y \to 1, ?z \to 4]$
$[?x \to 5, ?y \to 2, ?z \to 6]$

という 3 つの解が得られる．この 3 つの解はそれぞれ，元のグラフにおいてクエリ構造にマッチしうる排反な部分グラフの集合を表している．そのため，これらを個別に処理し，その結果をあわせることで，クエリにマッチするすべての部分グラフ集合を得ることができる．

変数の値域を限定するには，通常は RDF ストアの実装の修正が必要となるが，これは容易な作業ではない．しかしながら，縮約グラフのノードを用いることで，以下に述べるように RDF ストアの実装に手を入れることなく値域限定が可能となる．アイデアは単純である．あらかじめ，縮約グラフのノード a と元のグラフのノード r の対応関係を $a\ abs\ r.$ という形のトリプルとして RDF ストアに登録しておく．これにより，たとえば $a\ abs\ ?x.$ というトリプルパターン（限定節とよぶ）をクエリに追加することで変数 $?x$ の値域を縮約グラフのノード a に縮約される範囲に限定することができる．一般的な RDF ストアは，定数ノードに対する各種のインデックスや結合順序変更に基づく最適化機能を有しており，これによりほぼ意図した限定効果を得ることが可能である．

縮約グラフは特定のクエリ構造に依存するものではないため，任意のクエリに対して適用可能である．いくつかの

表 1 縮約グラフに対するクエリの結果例
Table 1 Example of queries for contracted graph.

クエリの条件節	解
$?x\ t\ ?y.$	$[?x \to 3, ?y \to 4], [?x \to 5, ?y \to 6]$
$?x\ g\ ?y.\ \ ?x\ n\ ?z.$	$[?x \to 4, ?y \to 15, ?z \to 7],$
	$[?x \to 4, ?y \to 15, ?z \to 8]$
$?x\ w\ ?y.\ \ ?z\ m\ ?y.\ \ ?x\ n\ ?v.\ \ ?z\ n\ ?v.$	$[?x \to 3, ?y \to 1, ?z \to 4, ?v \to 7]$
$?x\ w\ ?y.\ \ ?z\ t\ ?x.\ \ ?z\ m\ ?v.$	なし

例に対して縮約グラフから得られる部分グラフを**表 1** に示す．クエリの構造に応じて異なる分割が得られていることが分かる．なお，最後の例では，縮約グラフのみから解がないことが分かるため，元のグラフの検索は不要となる．解がないクエリの処理はしばしば長時間を要することがあり，そのような場合，提案手法によって大幅に処理速度が向上する．

1.4 貢献点

本論文の貢献点は以下のとおりである．

- 縮約グラフとよぶ元の RDF データの構造の要約情報を生成し，クエリ変数の値域をクエリの定めるグラフ構造に基づいて限定することで，クエリ処理を効率的に並列化する方法を示した．
- 提案手法のプロトタイプを用いた予備評価を行い，生成される並列タスクの負荷のばらつき，および負荷の大きさの点で，一般的なノード集合の均等分割による方法と比べて 2 倍以上優れていることを確認した．

なお，本論文では単一計算機上での並列化を想定しており，複数計算機による分散並列化は扱わない．ただし，分析対象データの複製を計算機ごとに用意しておくことで，複数計算機を用いた並列化へ拡張することは容易である．また，データの更新については，一連のデータ分析中にデータの更新は生じないことを前提としている．すなわち，リアルタイムデータ分析のように，データ収集とデータ分析を並行して行う利用形態は想定していない．

1.5 準備

本節では，以降で用いる用語を定義する．RDF データ G に対し，G の定めるラベル付き有向グラフとは，G に含まれるトリプル (s, p, o) をノード s からノード o へのラベル p を持つエッジと見なしてできるグラフのことである．これを RDF グラフとよび，RDF データと同一視する．G のノード集合を $\mathcal{N}(G)$，エッジラベル集合を $\mathcal{L}(G)$ で表す．

SPARQL クエリ q に対し，q が定めるグラフ（クエリグラフ）とは，q の条件節に含まれるトリプルパターンの集合を RDF データの場合と同様にグラフと見なしたものである．本論文では，説明が煩雑になるのを避けるため，クエリの条件節はトリプルパターンおよびフィルタパターンの集合で構成されているものとする．また `order by` 等の解修飾子を含まないものとする．ただし，提案手法は一般のクエリへも容易に拡張可能である．クエリに含まれる変数をクエリ変数，クエリの結果を保持する変数を結果変数とよぶ．また，クエリ変数とその値の対応関係を変数束縛とよぶ．クエリのトリプルパターン集合に対し，その変数束縛は，1 つの部分グラフを定める．そのため，本論文では変数束縛とそれによって定まる部分グラフを同一視する．クエリの結果は変数束縛の集合となる．

2. 縮約グラフの生成方法

1 章で述べたように，提案手法ではクエリ変数の値域を分割し，互いに排反な解空間を持つクエリ群に展開するために縮約グラフとよぶ元の RDF データの要約情報を用いる．本章では，最初に 2.1 節で縮約グラフの定義を与えた後，2.2 節で縮約グラフを生成するためのアルゴリズムを示す．

2.1 縮約グラフの定義

最初に縮約グラフを，その基礎となる縮約関数とよぶ概念とあわせて定義する．

定義 1（縮約関数および縮約グラフ）．RDF グラフ G_1，G_2 に対し，以下を満たす関数 α を G_1 の縮約関数とよぶ．また G_2 を G_1 の縮約グラフとよぶ．

(1) $\forall n \in \mathcal{N}(G_1)\ \alpha(n) \in \mathcal{N}(G_2)$.
(2) $\forall p \in \mathcal{L}(G_1)\ \alpha(p) \in \mathcal{L}(G_2)$.
(3) $\forall p_1, p_2 \in \mathcal{L}(G_1)\ p_1 \neq p_2 \implies \alpha(p_1) \neq \alpha(p_2)$.
(4) $\forall n_1, n_2 \in \mathcal{N}(G_1), \forall p \in \mathcal{L}(G_1)$
$(n_1, p, n_2) \in G_1 \implies (\alpha(n_1), \alpha(p), \alpha(n_2)) \in G_2$.

すなわち縮約関数はノードをノードに，エッジラベルをエッジラベルに対応づける．また，異なるエッジラベルは異なるエッジラベルに対応づける．なお，縮約関数の定義域はノードおよびエッジラベルであるが，自明な方法で部分グラフに対する関数に拡張できる．以降ではこれらを区別せずに扱う．

最後の条件 (4) は，G_1 のエッジに対応するエッジが G_2 に存在することを要請している．これは，縮約グラフを用いてクエリの各変数の値域を求めるアルゴリズムを構築するための基礎となる．

定理 1（縮約グラフの安全性）．G を RDF グラフ，α を G の縮約関数とする．このとき，任意の G の部分グラフ H に対し，$\alpha(H)$ は $\alpha(G)$ に含まれる．

証明. 縮約関数の定義および H のエッジの数に関する帰納法による． □

定理 1 は，クエリグラフ Q にマッチする縮約グラフの部分グラフ集合 A が，Q にマッチする元のグラフの部分グラフ集合 C の安全な近似値になることを意味している．すなわち，任意の $c \in C$ に対し，$\alpha(c) \in A$ がなりたつ．

本節の冒頭で述べたように，縮約グラフを用いることで，クエリの構造に基づきクエリ変数の値域を限定できる．

例 1（縮約グラフによる値域限定効果を示す例）．例として以下の RDF グラフを考える．

$$\begin{array}{llllll} n_1 & \xrightarrow{p} & n_5 & \xrightarrow{q} & n_9 & \xrightarrow{r} & n_{13} \\ n_2 & \xrightarrow{p} & n_6 & & n_{10} & \xrightarrow{r} & n_{14} \\ n_3 & \xrightarrow{p} & n_7 & \xrightarrow{q} & n_{11} & \xrightarrow{r} & n_{15} \\ n_4 & \xrightarrow{p} & n_8 & & n_{12} & \xrightarrow{r} & n_{16} \end{array}$$

たとえば，得られた縮約グラフが以下の場合，

$$\begin{bmatrix} n_1 \\ n_3 \end{bmatrix} \xrightarrow{p} \begin{bmatrix} n_5 \\ n_7 \end{bmatrix} \xrightarrow{q} \begin{bmatrix} n_9 \\ n_{11} \end{bmatrix} \xrightarrow{r} \begin{bmatrix} n_{13} \\ n_{15} \end{bmatrix}$$

$$\begin{bmatrix} n_2 \\ n_4 \end{bmatrix} \xrightarrow{p} \begin{bmatrix} n_6 \\ n_8 \end{bmatrix} \qquad \begin{bmatrix} n_{10} \\ n_{12} \end{bmatrix} \xrightarrow{r} \begin{bmatrix} n_{14} \\ n_{16} \end{bmatrix}$$

以下のクエリ

```
select ?x where { ?x p ?y. ?y q ?z. ?z r ?w. }
```

に対し，クエリの定めるグラフ構造

$$?x \xrightarrow{p} ?y \xrightarrow{q} ?z \xrightarrow{r} ?w$$

から，$?x$ の値域が $\{n_1, n_3\}$ であることが分かる．これは $?x$ の値域を元の RDF グラフのノード集合と推定する場合に比べ大幅に限定されている．

なお，RDF グラフに対し，その縮約グラフは定義からは一意に定まらない．検索範囲を限定するという点では，より正確にクエリ変数の値域が限定できるような縮約グラフが望ましい．たとえば，例 1 に対し，定義 1 からは以下のグラフも縮約グラフとしての条件を満たす．

$$\begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \end{bmatrix} \xrightarrow{p} \begin{bmatrix} n_5 \\ n_6 \\ n_7 \\ n_8 \end{bmatrix} \xrightarrow{q} \begin{bmatrix} n_9 \\ n_{10} \\ n_{11} \\ n_{12} \end{bmatrix} \xrightarrow{r} \begin{bmatrix} n_{13} \\ n_{14} \\ n_{15} \\ n_{16} \end{bmatrix}$$

しかしながら，このような縮約グラフでは，上記のクエリに対して $?x$ の値域を $\{n_1, n_2, n_3, n_4\}$ にしか限定することができない．すなわち，値域の限定効果を高めるためにはノードの接続関係がなるべく保たれるようにノード集合を分割する必要がある．

2.2 生成アルゴリズム

本節では，これまでの議論に基づき，提案手法で用いる縮約グラフの生成アルゴリズムを示す．縮約のポイントとなるのはノード集合の分割方法である．後述するように，ノード集合の分割が決まると，定義 1 の条件を満たす縮約グラフは自明な形で導かれる．

アルゴリズムの基本的なアイデアは，最初に同じ接続構造を持つノード（入力エッジおよび出力エッジのラベル集合が同じノード）をグループ化し，縮約グラフの条件を満たしたうえで，データ並列化効果を高めるために，各グループをサイズが均等になるように分割するというものである．なお，分割数については生成時にパラメータとして指定されるものとする．最適な分割数の選択方法については 4.6 節で議論する．

以降では上記のアイデアに基づく分割伝搬法とよぶヒューリスティックアルゴリズムを示す．これは，指定された起点となるノードグループをもとに，その分割に沿って隣接するノードグループを分割していくものであり，多項式時間で実行可能である．また，後述するように分割によって構造を保存するという性質を備えており，基本的に階層構造であるような規則性の高いグラフに対して特に有効である．

なお，人手による起点の選択は利便性の点で問題がある．ただし，起点集合の選択はデータ分析の過程で 1 度だけ行えばよい．また，現在人手で選択しているのは最善の効果を得るためであり，起点集合としてどのグループを選んでも後述する並列化処理は可能である．将来的には自動化されるべきものであり，たとえば，すべてのグループについて，それを起点集合として縮約グラフを生成し，そのなかで縮約グラフのノードサイズ（対応する RDF ノードの数）のばらつきが最も小さいものを選択する等の方法を検討している．

2.2.1 分割伝搬アルゴリズム

分割伝搬アルゴリズムは 3 ステップからなる．

ステップ 1：グラフ構造スキーマの生成

最初に各ノードを接続関係に基づいてグループ化する．グループ化は，以下の特徴関数を用いて，ノード集合を同値分割することで行う．

$$cl(n) = (\{p | (\_, p, n) \in G\}, \{p | (n, p, \_) \in G\}).$$

$cl(n)$ は，n の入力エッジおよび出力エッジのラベル集合によって n を特徴づける関数である．

ステップ 2：起点集合の分割

次に，ステップ 1 で生成されるグループから，分割の起点とするグループ（起点集合とよぶ）を 1 つ選び，指定された分割数 k で均等分割する．起点集合の選択は外部から与えられるものとする．以降では，グループを分割してできる部分集合を分割ブロックとよぶ．またノードが属する

分割ブロックを表す識別子を分割ブロック識別子とよぶ．

ステップ 3：分割の伝搬

次に起点集合の分割に基づき，他のグループを分割する．なお，RDF グラフが連結グラフでない場合，各連結成分ごとに起点集合を選択し，ステップ 2 および 3 を行う．

図 3 に伝搬アルゴリズムを示す．入力のうち，B_0 は起点集合，G は対象 RDF グラフ，$\hat{G}$ はステップ 1 で求めたグループをノードとし，グループの要素間に G のエッジが存在するときおよびそのときに限りグループ間のエッジが存在するような無向グラフ，D_0 はステップ 2 で求めた B_0 の各要素と分割ブロック識別子の対応を保持する写像である．

アルゴリズムは最初に G のノードと分割ブロック識別子の対応を保持する写像 D を D_0 で初期化する（2 行目）．以降では $D[n]$ により n の分割ブロック識別子を表す．次に B_0 から幅優先探索 BFS により $\hat{G}$ のノード（グループ）をたどりグループを B_0 に近い方から $B_1, ..., B_l$ に整列する（3 行目）．

次に各 B_i を順に分割する（4–14 行目）．B_i の各要素 n ごとに，その隣接ノードのうち，すでに分割ブロックが決定しているものの集合 C を求める．なお，n の隣接ノードとは n に入るエッジおよび n から出るエッジのいずれかで接続されたノードである．もし C が空でなければ，C から無作為に 1 つノード m を選び，B_i と m の分割ブロック識別子 $D[m]$ の組を n の分割ブロック識別子とする．B_i と組にするのは，別グループの分割ブロックとは異なる識別子とするためである．

C が空の場合，B_i と空集合の組を n の分割ブロック識別子とする．分割ブロック識別子が空集合となることはないため，このような割当てにより n の分割ブロック識別子は C が空でないノードとは異なるものになる．

なおアルゴリズムの中心となる各グループ B_i の分割処理を行うループ（5–13 行目）はループ繰返しにまたがる依存がないため，B_i の均等分割による単純な並列化が可能である．G のノードの次数に大きなばらつきがない限り，各並列タスクの負荷は均衡することが期待できる[*2]．

```
 1: procedure PROPAGATE(B_0, G, Ĝ, D_0)
 2:     D ← D_0
 3:     B_1, ..., B_l ← BFS(B_0, Ĝ)
 4:     for i ← 1, l do
 5:         for all n ∈ B_i do
 6:             C ← {m|(n, _, m) ∈ G ∨ (m, _, n) ∈ G, D[m] ≠ ⊥}
 7:             if C ≠ φ then
 8:                 m ← CHOOSE(C)
 9:                 D[n] ← (B_i, D[m])
10:             else
11:                 D[n] ← (B_i, φ)
12:             end if
13:         end for
14:     end for
15:     return D
16: end procedure
```

図 3 伝搬アルゴリズム
Fig. 3 Propagation algorithm.

上記で得られた分割から縮約グラフを生成するには，ステップ 3 の結果 D に基づき，分割ブロックごとに一意な縮約ノードを対応づけるような縮約関数を定義し，それを RDF グラフの全トリプルに適用するだけでよい．

図 3 の伝搬アルゴリズムの効果を示す例として以下の RDF グラフを考える．

$$
\begin{aligned}
n_1 &\xrightarrow{p} n_5\\
n_2 &\xrightarrow{p} n_6\\
n_3 &\xrightarrow{p} n_7\\
n_4 &\xrightarrow{p} n_8
\end{aligned}
$$

分割伝搬法では，ステップ 1 により n_1, n_2, n_3, n_4 および n_5, n_6, n_7, n_8 がそれぞれ同一グループに分類される．ここでは前者を起点集合とし，ステップ 2 により n_1, n_2 および n_3, n_4 に分割されたとする．このとき，n_5, n_6, n_7, n_8 を n_5, n_7 および n_6, n_8 に分割してしまうと，グループ間の混線が生じ，縮約グラフのエッジ数が増加してしまう．ステップ 3 では，n_1, n_2, n_3, n_4 の分割に沿って n_5, n_6, n_7, n_8 を n_5, n_6 および n_7, n_8 に分割することで，必要なエッジ数を減らし，値域限定効果を向上させている．

なお，元の RDF グラフが同型の小グラフを集めたようなグラフである場合，分割伝搬アルゴリズムで生成される縮約グラフはそれらの同型の小グラフをほぼ同数ずつまとめたものになる．そのため，この縮約グラフをもとに後述するクエリ並列化方法を適用したとき，生成される並列タスクの負荷は均等化される可能性がきわめて高い．RDF はスキーマレスなデータ形式であるが，実用上はデータ生成時に暗黙的なスキーマを想定することは珍しくなく，その場合生成される RDF グラフはこれに近い特性を持つ可能性が高い．

3. 縮約グラフに基づくクエリ並列化

本章では 2 章で述べた縮約グラフを用いてクエリ処理を並列化する方法について述べる．1 章で述べたように，最初にクエリ変数の値域を縮約グラフ上で解析し，その結果に基づきクエリを排反な解集合を持つクエリ群に展開する．

3.1 クエリ変数の値域の解析

本節では，クエリ変数の値域を縮約グラフを用いて解析する方法を示す．基本的なアイデアは，元のクエリ q を縮約 RDF グラフ G_A に対するクエリ q_A に変換し，q_A を G_A 上で処理することで，各クエリ変数の値域を縮約グラフのノードの集合として求めるというものである．**図 4** に解析

[*2] ただし 4 章のプロトタイプでは本並列化は実施していない．

```
1: procedure ANALYZEQUERY(q, G_A)
2:     let q = select xs where { ps }
3:     // query translation
4:     ps' ← REPLACECONST(ps)
5:     ps'' ← CONVERTFILTER(ps')
6:     q_A ← select xs where { ps'' }
7:     // query execution
8:     S_A ← EXECUTE(q_A, G_A)
9:     return S_A
10: end procedure
```

図 4 クエリ変数の値域解析アルゴリズム
Fig. 4 Variable range analysis algorithm.

アルゴリズムを示す．解析はクエリ変換および実行の 2 ステップで構成される．

クエリ変換では，最初にクエリ内の定数（RDF グラフのノードを表す値）を対応する縮約グラフのノードを表す値に変換する（4 行目）．これは，縮約時に縮約グラフのノード a と元の RDF グラフのノード r の対応関係を $a\ abs\ r.$ という形のトリプルとして RDF ストアに登録しておくことで容易に行える．次に，縮約グラフ上で正しく判定できないフィルタ節を除去する（5 行目）．フィルタ節の条件式を評価するためには原則として縮約グラフのノードを RDF グラフのノードに変換して評価を行う必要がある．しかしながら，これは通常のクエリ処理では不可能なため，ここでは変換を行わなくても安全に評価できる変数間の等値比較以外については単純に除去する方法を用いている．変数間の等値比較が安全なのは，縮約グラフのノードが等しくなければ，元の値も必ず異なるためである．なお，フィルタ節については縮約グラフの生成に一定の制約を課することで，より高精度な近似も可能であるが，これについては本論文の範囲を超える．

クエリ実行では，クエリ変換の結果得られるクエリを，通常のクエリと同様に縮約グラフに対して処理する．これにより，クエリの解として，各クエリ変数の値域を縮約グラフのノードの集合として得ることができる．一方，解が存在しない場合，縮約グラフの安全性から，元のクエリにも解が存在しないことが保証される．そのため，元データを処理せずに解が存在しないという結果を返すことができる．

なお，図 4 のアルゴリズムではクエリ変換によって生成されるクエリの結果変数は元のクエリの結果変数と同じもの（xs）である．すべてのクエリ変数を結果変数とし，対応する縮約グラフのノードを解として求めることでクエリの解空間をより限定することは可能である．しかしながら，限定の対象とする変数の数が増えると，展開クエリの数が増えるため，並列化オーバヘッド等のデメリットが生じる可能性がある．このため，現在のアルゴリズムは経験的にそれほど多くなることがない結果変数のみを限定の対象にしている．最適な変数群の選択は今後の課題である．

```
1: procedure EXECUTEQUERY(q, G, S_A)
2:     qs ← EXPANDQUERY(q, S_A)
3:     for all q ∈ qs do
4:         // do parallel
5:         S_i ← EXECUTE(q, G)
6:     end for
7:     return S_1 ∪ ... ∪ S_n
8: end procedure
```

```
1: procedure EXPANDQUERY(q, S_A)
2:     let q = select xs where { ps }
3:     for all ρ ∈ S_A do
4:         rps ← ρ(x_1) abs x_1. ... ρ(x_k) abs x_k.
5:         q_i ← select xs where { rps ps }
6:     end for
7:     return { q_1, ..., q_n }
8: end procedure
```

図 5 クエリ並列実行アルゴリズム
Fig. 5 Query parallelization algorithm.

3.2 クエリ並列化

次に 3.1 節の結果を用いて，クエリ処理を並列化する方法を示す．例として，以下のクエリ q を考える．

```
select ?x where { ?x p ?y. }
```

3.1 節のアルゴリズムにより $?x$ について，縮約グラフのノード $a_1, ..., a_n$ が得られたとする．このとき，$?x$ の値が a_i に縮約されるものに限定して q を処理した結果を S_i とすると，縮約グラフの安全性から，q の解 S は $S_1 \cup ... \cup S_n$ に一致する．そのため，q を $?x$ の値域 $a_1, ..., a_n$ に限定されたクエリ群に展開することで，クエリ並列化を，複数クエリの並列処理によって実現することができる．

変数値域の限定は，限定節とよぶ条件節をクエリに追加することで行う．限定節は $a\ abs\ ?x.$ という形式のトリプルパターンである．ここで a はリソースとしてエンコードされた縮約グラフのノード，abs は縮約関係を表す述語，$?x$ は変数である．エンコード方式は結果が元の RDF グラフに含まれない値となる方式を採用しなければならない．また，縮約関係を表す述語は元の RDF グラフに含まれない値を用いなければならない．

以上の考えに基づき，クエリを最適化し実行するアルゴリズムを**図 5** に示す．EXPANDQUERY は，クエリ q および解析結果 S_A を受け取り，S_A に含まれる変数束縛 ρ ごとに q に限定節を加えた展開クエリ群を生成する[*3]．EXECUTEQUERY は，クエリ q，RDF グラフ G，および解析結果 S_A を受け取り，EXPANDQUERY を用いて生成したすべての展開クエリ qs を並列に処理し，結果を統合する．

[*3] なお，クエリ変数の持つ縮約グラフのノードのサイズが 1 の場合，限定節を加えるかわりに，その変数を RDF グラフのノードを表す値で置き換えたクエリを生成した方がよい．これは，RDF ストア側で既存のインデックスを利用した最適化が適用されることが期待できるためである．ただし本論文で評価に用いたプロトタイプでは未実装である．

表 2 縮約グラフの特性
Table 2 Characteristics of contracted graph.

データセット	縮約ノード数		縮約エッジ数		縮約率
LUBM10	499	(314,868)	4,909	(1,272,953)	3.92×10^{-3}
LUBM100	516	(3,301,733)	6,338	(13,409,395)	4.73×10^{-4}
LUBM1000	556	(32,885,166)	6,533	(133,613,894)	4.90×10^{-5}

4. 評価

本章では提案手法のプロトタイプを実装し評価した結果を示す．

4.1 評価環境

評価に用いた RDF ストアは Jena [13] である．Jena は Sesame [3] と並び初期の頃から存在する RDF ストアであり，標準的な SPARQL インタフェースに加え，RDF データ処理に便利な様々な拡張機能を提供している．一方で，クエリ処理性能の点では RDF-3X [18] 等の先端的な RDF ストアに比べ十分とはいえないことが報告されている [16]．そのため，Jena の持つ豊富な機能を維持したまま，クエリ処理性能を向上させることができれば，多くの利用者にとって有益である．

評価に用いた計算機環境を以下に示す．

- CPU：QuadCore Opteron 8350（2.3 GHz）× 4
- Memory：112 GB [*4]
- OS：CentOS 5
- RDF ストア：Jena 2.6.4 + TDB 0.8.10 + ARQ 2.8.8

評価に用いたデータセットは標準的な RDF ベンチマークの 1 つである LUBM [20] である．本ベンチマークは大学数をパラメータとして与えることで，その大きさを自由に変えることができる．

縮約グラフの生成は起点集合として以下のエッジを持つノード集合を用いて行った．

- 入力エッジラベル
  `:worksFor`, `:subOrganizationOf`, `:headOf`, `:memberOf`
- 出力エッジラベル
  `:subOrganizationOf`, `rdf:type`

この起点集合は大学における各学部を表すノード集合に対応している．

LUBM では 14 のクエリが提供されているが，ここでは最も負荷の高いクエリの 1 つであるクエリ Q2 を代表として用いた．クエリ Q2 の内容を以下に示す．

```
select ?x ?y ?z where {
  ?x rdf:type :GraduateStudent.
  ?y rdf:type :University.
  ?z rdf:type :Department.
  ?x :memberOf ?z.
  ?z :subOrganizationOf ?y.
  ?x :undergraduateDegreeFrom ?y.
}
```

[*4] なお，すべてのクエリ評価において OS によるディスクキャッシュを除くメモリ使用量は 1 GB 以下である．

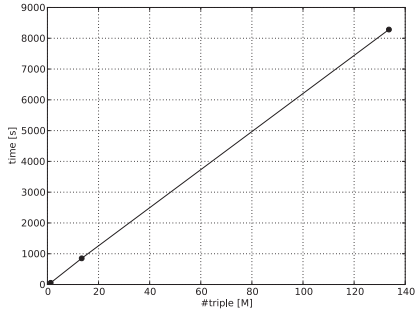

図 6 RDF グラフサイズに対する縮約グラフの生成時間
Fig. 6 Contracted graph generation time for RDF graph size.

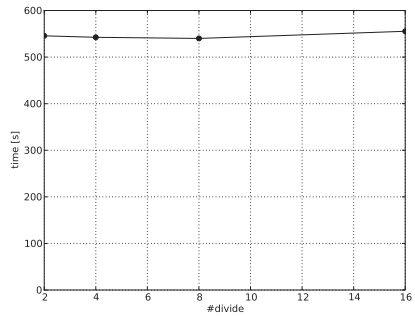

図 7 分割数に対する縮約グラフの生成時間
Fig. 7 Contracted graph generation time for the number of division.

4.2 縮約グラフの特性および生成時間

生成した縮約グラフの特性を**表 2** に示す．縮約ノード数およびエッジ数については，元の RDF グラフの値をかっこ内にあわせて示している．縮約率は元の RDF グラフに対する縮約グラフの大きさの比率である．分割数はいずれも 8 である．

表から分かるように，縮約グラフのサイズは元のグラフの 1%以下であり，元のグラフのサイズが大きくなっても，その増加は緩やかである．

縮約グラフの生成時間を**図 6** および**図 7** に示す．図 6 は RDF グラフサイズ（トリプル数）に対する縮約グラフの生成時間を表している．評価に用いた用いたデータは LUBM10，100 および 1000 であり，分割数はいずれも

8 である．LUBM10 の場合に対する生成時間の増加率は LUBM100 で約 15 倍，LUBM1000 で約 150 倍であり，ほぼ線形である．

図 7 は LUBM100 に対し分割数を 2 から 16 まで変化させたときの縮約グラフの生成時間を表している．図から分かるように分割数は縮約グラフの生成時間に大きな影響を与えない．

4.3 クエリ解析時間

次に，提案手法で並列化のための必要となるクエリ解析のオーバヘッドを評価する．

クエリ Q2 の解析時間を**図 8** に示す．parse はクエリの構文解析時間，translation はクエリ変換時間，cg search は縮約グラフの検索時間，expansion はクエリ展開時間を表す．縮約グラフのサイズから予想できるように，解析時間は総計で 400 ms 以下であり，負荷の高いクエリにおいては，そのオーバヘッドは許容範囲であると考えられる．

4.4 展開クエリの負荷分布

提案手法を適用することで，入力クエリは複数のクエリ群に展開される．ここでは，生成される展開クエリ群の負荷の大きさおよびばらつきを評価する．すべての展開クエリを同時に処理できるだけの計算資源がある場合，この結果から性能向上の上限値が得られる．対象データセットとしては LUBM100 を用いた．提案手法非適用時の処理時間は 7,270 ms である．

図 9 の dp は，クエリ Q2 から提案手法によって生成された各展開クエリを個別に処理した場合の処理時間の分布である．処理時間の平均値は 819 ms，標準偏差は 81.0 であり，各クエリの負荷はおおむね均等化されていることが分かる．また，平均値は提案手法非適用時の処理時間の約 8 分の 1 であり，分割数に応じた負荷の削減が達成できている．

図 9 の hash は，ハッシュ分割に基づく並列化 [9], [19] と同様の効果を持つ並列化を縮約グラフを用いて実現したものである．具体的には，RDF グラフのノードのうち，述語として利用されているものを除いたノードの集合をランダムに均等分割してグループ化し，各グループをノードとする縮約グラフを生成し，これを用いて並列化を行っている．なお，述語として用いられているノードは縮約グラフの定義 1 の条件を満たすようにそれぞれ独立なグループとしている．提案手法と同様，縮約グラフの生成時には RDF グラフのノードと縮約グラフのノードの対応関係を表すトリプルを追加し限定節で利用できるようにしている．

縮約グラフの各ノードはハッシュ分割におけるハッシュ値の役割をはたす．縮約グラフのノードサイズを均等にしているため，hash の結果はハッシュ分割に基づく並列化において，各パーティションのサイズが均等になるという理想状況における処理時間を表している．

処理時間の平均値は 2,212 ms，標準偏差は 159.0 であり，提案手法に比べ負荷の大きさは約 2.7 倍，ばらつきは約 2.0 倍で，いずれも倍以上大きくなっている．

さらに，負荷の分布状況をより統計的に把握するために，分割数を 100 に増やしたときの度数分布を**図 10** に示す．横軸が処理時間，縦軸がその区間に属するクエリ数である．平均値は 69.2 ms，標準偏差は 6.7 である．また，ハッシュ分割を用いた場合の度数分布を**図 11** に示す．平均値は 177.5 ms，標準偏差は 13.4 である．

なお，図 10 と図 11 の縦軸の度数の差に注意されたい．図 10 では度数 22 の分布の中心が存在するのに対し，図 11 の度数は最大でも 13 であり，分布の中心といえる値が存在していない．

これらの結果から，縮約グラフを用いてクエリの構造に

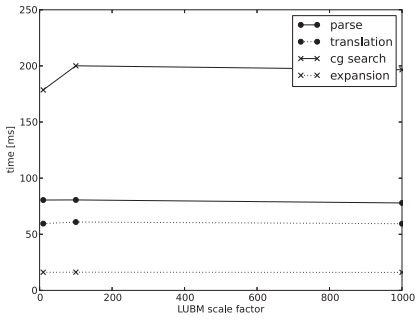

図 8 クエリ解析時間
Fig. 8 Query analysis time.

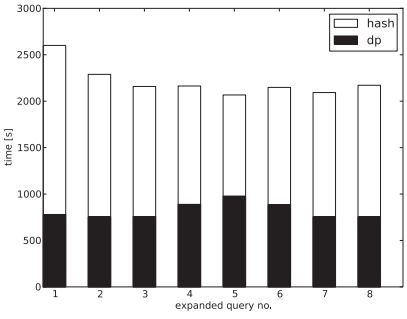

図 9 展開クエリの負荷
Fig. 9 Workload of expanded queries.

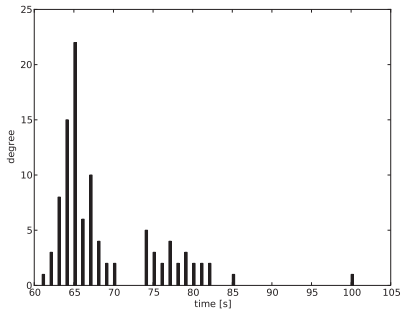

図 10 負荷分布（分割伝播）
Fig. 10 Distribution of workload (DP).

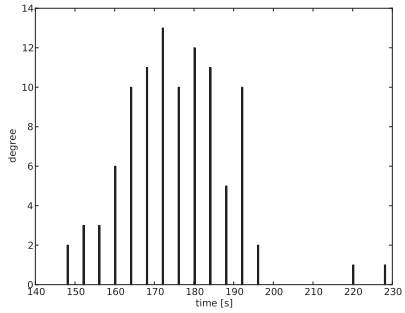

図 11 負荷分布（ハッシュ分割）
Fig. 11 Distribution of workload (Hash).

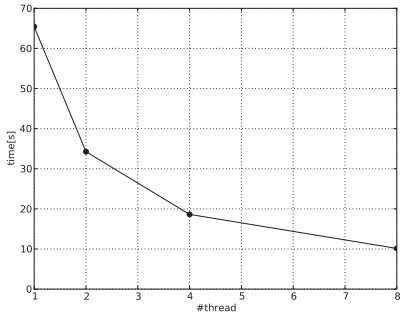

図 12 展開クエリの並列処理時間
Fig. 12 Parallel execution time of expanded queries.

沿って解空間（変数の値域）を限定し，並列処理することで，負荷の大きさおよびばらつきが大きく改善されることが分かる．なお，ハッシュ分割による方法では，並列化の際に提案手法で行うクエリ解析が不要であり，関連するオーバヘッドが発生しない．これは，クエリの総処理時間の点で有利である．ただし，図 8 からも分かるように，解析時間はデータサイズが増加してもほぼ一定であり，大規模データにおいてはこのオーバヘッドは相対的に無視できる．

また，標準的なグラフ分割ライブラリでの 1 つである METIS [14] を用いた場合についても評価を試みた．METIS はエッジラベルのない無向グラフを対象に，分割をまたぐエッジができる限り少なくなるような分割を求めるものである．評価は，エッジラベルを除いた RDF グラフから METIS を用いてノード集合の分割を生成し，それを用いて縮約グラフを生成することで行った．しかしながら，クエリ解析を行った結果，対象クエリの変数の値域がまったく分割されず，並列化を行うことができなかった．この結果はエッジラベルに基づく分割の重要性を示唆するものといえる．

4.5 並列化の効果

次に，提案手法によって生成される展開クエリ群を並列処理したときの時間を評価する．対象データセットは LUBM1000 を用いた．提案手法非適用時の処理時間は約 65 s である．用いた分割数，スレッド数および利用したコア数はいずれも 8 である．

すべての展開クエリを並列処理した場合の処理時間を**図 12** に示す．横軸は生成スレッド数であり，並列処理されるクエリ数を表す．提案手法非適用時に比べ，8 並列時に約 6.5 倍の性能向上が得られている．完全に線形な性能向上が得られない理由としては，最も遅い展開クエリに律速されること，およびメモリバンド幅等の計算資源の競合が考えられる．このうち，後者については複数計算機を用いて分散並列処理した場合には生じない．なお，本結果については，展開クエリを個別に実行したときの結果は 7.9 s から 8.3 s の範囲に収まっており，要因としては後者の影響が大きいと考えられる．

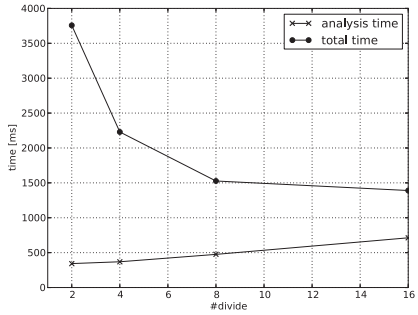

図 13 分割数が処理時間に及ぼす影響
Fig. 13 Effects of the number of division on execution time.

4.6 分割数の影響

分割数を 2 から 16 まで変化させたときのクエリ処理時間の変化を**図 13** に示す．対象クエリ Q2，データセットは LUBM100，コア数およびスレッド数はいずれも 16 である．図の analysis time はクエリ解析時間，total time はクエリ解析および実行時間を含む総処理時間である．生成される展開クエリ数は分割数に等しい．

分割数が増加するにつれて展開クエリ数が増え個々の展開クエリの負荷が小さくなるため，展開クエリの並列実行時間は短くなる．一方で縮約グラフのサイズが大きくなるためクエリ解析時間は増加する．図 13 においては，分割数が 2 の場合，解析時間の割合は全体の 1 割以下であるが，分割数が 16 の場合は総処理時間の半分以上を占めている．そのため，分割数を 8 から 16 に増やしても性能向上はほとんどみられない．

最適な分割数は，直接にはクエリ解析時間と展開クエリ実行時間のトレードオフで決まるが，これはデータセットのサイズにも間接的に影響を受ける．表 2 から分かるように，データセットのサイズの増加に対する縮約グラフのサイズの増加は緩やかである．たとえば，データセットのサイズが約 100 倍になっても，縮約グラフのサイズの増加は約 1.1 倍にとどまっている．そのため，クエリ解析時間の影響はデータセットのサイズが増加するにつれて相対的に小さくなる．一方，分割数を大きくしすぎると展開クエリ

表 3 クエリ処理時間
Table 3 Query processing time.

クエリ	従来のクエリ処理 [s]	提案手法 [s]
Q1	0.2	2.3
Q3	0.2	5.2
Q14	768.5	282.1

の数が同時実行可能な数（実行環境の計算資源数）を超えるため，並列化効果が失われる．

縮約グラフ生成時の分割数は，これらの要因を考慮した上で，計算資源数を基準に，データセットのサイズに応じて小さくするのがよいと考えられる．

4.7 その他のクエリに関する評価

ここまで，LUBM のクエリの中から高負荷クエリの代表として Q2 を対象に提案手法の詳細な評価を行ってきた．ここでは，LUBM のその他のクエリに関する簡単な評価結果を示す．比較対象は，Jena が提供する標準のクエリ処理（`com.hp.hpl.jena.query.QueryExecutionFactory` により生成されるクエリ処理エンジンを利用するもの）である．ただし，現在推論を用いるクエリ処理に未対応のため，推論の不要な以下のクエリ（Q1，Q3，Q14）を用いて評価を行った．なお用いた分割数，スレッド数および利用したコア数はいずれも 8 である．

結果を**表 3** に示す．クエリ Q1，Q3 については提案手法の適用によって性能が低下している．これらはいずれもトリプルパターン 2 つからなる単純なもので，結果の数も 10 件以下である．このようなきわめて負荷の軽いクエリについては，並列化のオーバヘッドに見合う効果が得られない．一方高負荷なクエリ Q14 については，約 2.7 倍の性能向上が得られている．Q2 のような十分な並列化効果が得られていない理由は，結果の数がきわめて多く（7,924,765 件），検索よりも結果集合生成処理に多くの時間が費やされているためであると考えられる．

提案手法をより汎用的な最適化技術にするためには，クエリの特性および縮約グラフに基づく最適化効果の推定等が必要であり，今後の課題である．

4.8 縮約グラフに基づく絞り込みの効果

1 章で述べたように現状の RDF ストアは解が存在しない場合にしばしば処理に長時間を要する．ここでは，本論文のテーマである並列化と直接関係するものではないが，縮約グラフによって解が存在しないことが分かる場合の効果を示す．

以下のクエリは同じ大学の同じ学部に属する教授と学生で，同じ名前を持つ組を探すものである．

```
select ?s ?p where {
  ?u rdf:type :University.
  ?d rdf:type :Department.
  ?d :subOrganizationOf ?u.
  ?s :memberOf ?d.
  { ?s rdf:type :GraduateStudent. }
  union
  { ?s rdf:type :UndergraduateStudent. }
  ?s :name ?n1.
  ?p :worksFor ?d.
  ?p rdf:type :FullProfessor.
  ?p :name ?n2.
  filter (?n1 = ?n2).
}
```

このクエリは解を持たないが，単純なインデックスによるチェックではそれが分からない．処理時間は LUBM1000 の場合で約 2,650 s であり，大幅に待たされることになる．これに対して，分割伝播法で生成される縮約グラフを用いると，解がないことがただちに分かる．処理時間は LUBM1000 の場合で約 1 s であり，数千倍の性能向上が得られている．

5. 関連研究

本章では提案手法と関連の深い SPARQL クエリ処理の並列化に関する研究を中心に先行研究を概観する．

RDF データを分割し，クラスタ上の各ノードに配置することで，クエリ処理性能の向上をはかるというのは自然な発想であり，Jena Clustered TDB [19]，YARS2 [11]，Bigdata [21]，Virtuoso [8]，4store [10] 等の RDF ストアにおいてクラスタ化機能が実装されている．これらは基本的にトリプル集合を各ノードに分割配置したうえで，主にクエリ処理における I/O や結合演算を必要に応じて各ノードに割り当てることで負荷分散を行う．どのような並列化戦略が用いられるかは RDF ストアのクエリ実行計画器に依存し，利用者がデータ並列化を指示するのは難しい．また結合演算の並列化は実行時にデータの分割やソートが必要となるため，特に結合演算の結果集合が小さい場合にはしばしば性能劣化が生じることが報告されている [9]．

Rohloff らは，トリプル集合を主語リソースを基準に適当なハッシュ関数を用いて分割し，MapReduce フレームワークの実装の 1 つである Hadoop を用いてクエリを各ノードで並列実行する方法を提案している [22]．トリプル集合は，同じ主語を持つトリプルを各行に並べたテキストファイルとして HDFS 上に配置される．クエリ処理は各条件節ごとに上記のトリプルファイルに対する Hadoop ジョブを繰り返し実行することで行う．この方法は単純なクエリでは効果がある一方，長い関連を含むクエリではノード間のデータ通信によるオーバヘッドのため，従来の単一ノードでの処理に比べて大幅に性能が劣化することが指摘されている [12]．

Huang らは，トリプル集合を排反なパーティションに分割するのではなく，一部のトリプルを複数のパーティションに重複して含めることで，通信コストを削減する方法を提案している [12]．クエリ処理は Hadoop および既存の RDF ストアの 1 つである RDF-3X を用いて行われる．これにより LUBM ベンチマークのクエリでは計算機間のデータ転送が完全に不要となり，並列化による大幅に性能向上が得られている．ただし，これは LUBM ベンチマークのクエリの単純さによるところが大きい．たとえば `?x p ?y. ?y q ?z. ?z r ?w.` のような長い関連を問うクエリの場合，クエリを分割したうえで，Hadoop ジョブの繰返し実行により計算機間のデータ転送，および結果の統合を行う必要がある．また，トリプルの重複配置によるデータ量の指数的増大を防ぐために高次数のノード（多くのリソースやリテラルと関連を持つノード）については，重複配置をせずクエリ処理時に特別に扱うようにしている．そのため LUBM データセットにおける name 述語や DBpedia データセット [2] における wikiPageWikiLink 述語のように高次数の値を持つ述語を含むクエリに対しては，やはりデータ転送のオーバヘッドが問題となる可能性がある．

Weaver らは Blue Gene/L クラスタ上で RDF データを各ノードに分割配置し，DeWitt らの並列結合演算アルゴリズム [7] を用いてクエリを並列処理するシステムを実装し評価している [26]．クエリはバッチ処理することを想定しており，データの各ノードへのロードもクエリ処理時に行われる．彼らの特徴はクエリ処理とロード処理をあわせて並列処理する点にある．扱えるクエリは単純な基本グラフパターンに限定されており，フィルタ式等は扱えない．LUBM および Barton データセットを用いて評価を行い，ロード時間が計算機台数に比例して短縮できるという結果を得ている．一方で，計算機間の通信コストのためクエリ処理時間は計算機台数を増やしてもあまり短縮されない．特に LUBM クエリ Q2 のように多くの結合演算を必要とするクエリでは，8 台を超えるとかえって処理時間が増加することが報告されている．

Myung らは SPARQL クエリを Pig Latin 等による MapReduce フレームワーク上の処理に変換して分散並列化する方法を提案している [17]．彼らの方法は現在のところ，通信オーバヘッドのために十分な性能が得られていない．また，SPARQL の機能の一部しか扱うことができない．

提案手法は，上記のクラスタ化アプローチと同様，複数ノードを用いて複製を配置することでスケーラビリティの確保が可能である．また，クラスタ化アプローチと異なり，任意のクエリに対しデータ通信は不要である．一方，データの物理的な分割を行わないため，トリプル数が数十億を超える場合，単一 RDF ストアでの処理に問題が生じる可能性がある．ただし，各クラスタノードが担当する縮約グラフの範囲を固定することで，そのワーキングセットを小さくすることが可能であると考えており，今後研究を進める予定である．

RDF データの各トリプルは 1 つの事実の言明を表しているが，これに対する推論規則集合として RDFS，OWL 等が提案されており，利用者が概念の階層関係に基づいた検索を行う機能を提供している．現在の RDF ストアの多くは，推論規則の閉包を求めることでこのような推論処理を実現している．閉包計算は高コストな処理であり，その並列化方法が提案されている [25]．ただし，これらの研究では，検索処理の並列化については扱われていない．

Abadi はコラム指向 RDBMS を用いて，RDF データを垂直分割することで，SPARQL クエリの処理効率を向上させる方法を提案している [1]．提案方法は，バックエンドとして用いる RDF ストアの実装に依存していないため，このような先進的な RDF ストアと自由に組み合わせることが可能である．

Cai らは分散ハッシュテーブルの手法を用いて，各トリプルを主語，述語，目的語のハッシュ値に基づき分割配置する方法を提案している [4]．また，Liarou らの連続 RDF クエリ処理 [15] や Cudre-Mauroux らの GridVine [5] においても分散ハッシュテーブルがデータ分割に用いられている．ただし，いずれもクエリ処理の並列化については扱われていない．

6. おわりに

6.1 結論

本論文では，RDF グラフの構造を要約した縮約グラフとよぶ概念を導入し，これを用いて効率的にクエリ処理を並列化する方法を示した．提案手法は任意の RDF ストアに適用可能であり，適用にあたって RDF ストアの内部処理の修正は不要である．そのため，MapReduce フレームワークを用いるいくつかの並列化アプローチにみられるような利用可能クエリの制限がなく，利用者は既存の RDF ストアが提供するすべての機能をそのまま利用することができる[*5]．

また，データを物理的に分割しないため，計算機間にまたがる処理はなく，通信オーバヘッドの問題が生じないという利点を持つ．データ量の増加に対しては，新しく計算機を追加し，同じデータを複製配置することで，計算機台数の増加に応じた性能向上を得ることができる．

一方で，物理的な分割を行わないということは，大規模なデータを単一の RDF ストアで管理することを意味する[*6]．そのため，性能を最大限に発揮するためには，特に

[*5] ただし，後述するように推論規則を利用する場合を除く．
[*6] 複製を用いる場合でも，各複製が全 RDF グラフを保持する点で同じである．

十分なディスクキャッシュ領域を確保できるだけの大容量メモリを備えた計算機が望ましい．この点で，低性能なコモディティ PC を多数並べて大規模データに対処する MapReduce アプローチとは効果的な適用範囲に差があり，利用者の状況に応じて使い分けるのが適当であると考えられる．

6.2 今後の課題

本論文で提案した縮約グラフに基づくクエリ最適化手法に関する研究は端緒についたばかりであり，多くの課題が残されている．本論文中にあげた問題点のほか，以下のような研究課題があり，今後取り組む予定である．

- 提案手法の問題の 1 つは，適切な縮約グラフの生成基準を人手で選択する必要がある点である．実用にあたっては，RDF データ管理者が自らの管理するデータ特性について十分に把握できていない場合でも効果的に並列化が行えるように，適切な縮約基準を自動的に選択する手法が望まれる．
- 現在の提案手法は OWL 等の推論規則を用いるクエリに対しては適用できない．このようなクエリを扱うためには，RDF データを事前に推論展開しておく必要がある．推論展開によるデータ量の爆発を避けるためには，縮約グラフ上で推論を行えるように推論規則を変換する方法が必要となる．
- トリプル数が T オーダ以上となる大規模 RDF データは，単一 RDF ストアで扱うのが難しくなる．提案手法で示した縮約グラフは，物理的にデータを分割しクエリを分散並列処理する場合においても，負荷分散・通信量削減効果の大きい分割基準の選択に有用であると考えている．この場合，縮約グラフはデータの大域的な分散状況を表すことになるが，それを効果的に用いてクエリを分散並列処理する方法について多くの検討の余地が残されている．
- 大規模データに対処するためのもう 1 つの方法として，分散共有メモリ上に単一 RDF ストアを構築し，縮約グラフが定める分割に沿って，RDF データを物理的に分散させるという方法が考えられる．このアプローチは，物理的な分割にともなう複雑さが生じないという利点がある．一方で単一 RDF ストアの制約は残るため，どの程度スケーラビリティがあるのかについては未知数であり，検討が必要である．

謝辞 本研究の遂行にご尽力いただいた日立製作所中央研究所の牛嶋一浩氏，安田知弘氏，および横浜研究所コンパイラグループの方々に感謝する．

参考文献

[1] Abadi, D.: Query execution in column-oriented database systems, Ph.D. Thesis, Massachusetts Institute of Technology (2008).
[2] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R. and Hellmann, S.: DBpedia – A Crystallization Point for the Web of Data, *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, Vol.7, pp.154–165 (2009).
[3] Broekstra, J., Kampman, A. and van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, *Proc. International Semantic Web Conference* (2002).
[4] Cai, M. and Frank, M.R.: RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network, *Proc. International World Wide Web Conference* (2004).
[5] Cudre-Mauroux, P., Agrawal, S. and Aberer, K.: GridVine: An infrastructure for peer information management, *IEEE Internet Computing*, Vol.11, No.5, pp.36–44 (2007).
[6] Cyganiak, R. and Jentzsch, A.: The Linking Open Data Cloud Diagram, available from ⟨http://www4.wiwiss.fu-berlin.de/lodcloud/state/⟩.
[7] DeWitt, D.J. and Gerber, R.H.: Multiprocessor hash-based join algorithms, *Proc. International Conference on Very Large Data Bases* (1985).
[8] Erling, O. and Mikhailov, I.: Towards web scale RDF, *Proc. International Workshop on Scalable Semantic Web Knowledge* (2008).
[9] Groppe, J. and Groppe, S.: Parallelizing Join Computations of SPARQL Queries for Large Semantic Web Databases, *Proc. Symposium on Applied Computing* (2011).
[10] Harris, S., Lamb, N. and Shadbol, N.: 4store: The design and implementation of a clustered RDF store, *Proc. International Workshop on Scalable Semantic Web Knowledge Base Systems*, pp.94–109 (2009).
[11] Harth, A., Umbrich, J., Hogan, A. and Decker, S.: YARS2: A federated repository for querying graph structured data from the web, *Proc. International Semantic Web Conference* (2007).
[12] Huang, J., Abadi, D.J. and Ren, K.: Scalable SPARQL Querying of Large RDF Graphs, *Proc. International Conference on Very Large Data Bases* (2011).
[13] Jena: A Semantic Web Framework for Java, available from ⟨http://jena.sourceforge.net⟩.
[14] Karypis, G.: METIS – Serial Graph Partitioning and Fill-reducing Matrix Ordering, available from ⟨http://glaros.dtc.umn.edu/gkhome/views/metis⟩.
[15] Liarou, E., Idreos, S. and Koubarakis, M.: Continuous RDF query processing over DHTs, *Proc. International Semantic Web Conference* (2007).
[16] Morsey, M., Lehmann, J., Auer, S. and Ngomo, A.-C.N.: DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data, *Proc. International Semantic Web Conference* (2011).
[17] Myung, J., Yeon, J. and Lee, S.: SPARQL Basic Graph Pattern Processing with Iterative MapReduce, *International Workshop on Massive Data Analytics over the Cloud* (2010).
[18] Neumann, T. and Weikum, G.: RDF-3X: A RISC-style engine for RDF, *Proc. VLDB Endowment*, Vol.1, No.1 (2008).
[19] Owens, A., Seaborne, A., Gibbins, N. and Schraefel, M.: Clustered TDB: A clustered triple store for Jena (2008), available from ⟨http://eprints.ecs.soton.ac.uk/16974/1/www2009fixedref.pdf⟩.

[20] Pan, Z., Guo, Y. and Heflin, J.: LUBM: A benchmark for OWL knowledge base systems, *Journal of Web Semantics*, Vol.3, pp.158–182 (2005).
[21] Personick, M.: Bigdata: Approaching web scale for the semantic web (2009), available from ⟨http://www.bigdata.com/whitepapers/bigdata_whitepaper_07-08-2009.pdf⟩.
[22] Rohloff, K. and Schantz, R.: High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store, *Proc. International Workshop on Programming Support Innovations for Emerging Distributed Applications* (2010).
[23] W3C: RDF, available from ⟨http://www.w3.org/TR/rdf-primer⟩.
[24] W3C: SPARQL Query Language for RDF, available from ⟨http://www.w3.org/TR/rdf-sparql-query⟩.
[25] Weaver, J. and Hendler, J.A.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples, *Proc. International Semantic Web Conference* (2009).
[26] Weaver, J. and Williams, G.T.: Scalable RDF query processing on clusters and supercomputers, *Proc. International Workshop on Scalable Semantic Web Knowledge Base Systems*, pp.81–93 (2009).

千代 英一郎 （正会員）

2009 年東京大学大学院情報理工学系研究科博士課程修了．博士（情報理工学）．（株）日立製作所横浜研究所研究員．コンパイラ，プログラム解析・検証，グラフデータベース等の研究開発に従事．

宮田 康志

2005 年京都大学工学部電気電子工学科卒業．2007 年同大学大学院工学研究科修士課程修了．現在，（株）日立製作所横浜研究所研究員．ミドルウェア分野の研究に従事．

西山 博泰 （正会員）

1993 年筑波大学大学院工学研究科博士課程修了．工学博士．（株）日立製作所横浜研究所主管研究員．最適化コンパイラ，Java 実行環境等言語処理系の研究に従事．ACM，IEEE 各会員．