P2P 環境における三次元ハッシュ索引を用いた分散 RDF データベース問合せ処理

的野 晃整[†] サイド ミルザ パレビ[†] 小島 功[†]

本論文では，P2P 環境における RDF データベースのための効率的な問合せ処理手法を提案する．RDF とは，メタデータ記述のための枠組みで，近年様々な応用分野で広く使われている．現在，大量の RDF データがインターネット上に広く散在しており，P2P に基づく検索手法が注目されている．これまで，P2P 環境上の RDF データに対する検索手法がいくつか提案されているが，関連するデータどうしを結合する演算が考慮されておらず，そのまま問合せ処理を行うとデータ転送量が大幅に増加するという問題がある．分散環境における結合演算を含む RDF 問合せを効率的に処理するために，三次元構造のハッシュ空間に基づく索引を用いた問合せ処理手法を提案する．本手法では，三次元空間に RDF データを構成単位であるトリプルを写像し，一定の区間内にデータが存在するか否かをビットを用いて表現する．結合演算処理の際に事前にビットどうしの演算を行うことで，データの転送前に不要なデータを決定でき，データ転送量を減少できる．本論文では，RDF 問合せを処理するプロトタイプを P2P 基盤上に実装し，分散した RDF データに対して結合演算を含む問合せ処理の効率が向上することを実験によって確認した．

P2P-based Query Processing for Distributed RDF Databases Using a Three-dimensional Hash Index

Akiyoshi Matono,[†] Said Mirza Pahlevi[†] and Isao Kojima[†]

In this paper, we propose a query processing scheme for RDF databases in a P2P environment. RDF is a framework for describing metadata and, today, it is widely used in many fields. RDF data are scattered everywhere and the total size are rapidly increasing. Therefore, RDF query processing in a P2P environment becomes an important issue. So far, several P2P-based query processing approaches for RDF data have already been proposed. These approaches, however, were not designed for RDF queries that include join operations, and thus they suffer from an unnecessary data transfer problem. To deal with this issue, we propose a query processing technique that is based on a three-dimensional hash index. In this approach, we map RDF triples that compose RDF data into the three-dimensional space of the index. To reduce the amount of data transfered between nodes (when processing a join query) we introduce a bit flag in the index that represents the existence of triples. We have implemented the query processing technique in an emulated P2P environment and performed an extensive performance evaluation. Our evaluation shows that the proposed approach can significantly improve query processing performance especially when dealing with join queries.

1. はじめに

今日，Resource Description Framework（RDF）[1] に大きな期待が寄せられている．RDF とは，次世代 Web として期待される Semantic Web を実現するために提案された，メタデータ記述のための枠組みである．RDF は，柔軟で高度な表現能力を有し，さらに任意の利用者が任意の資源に対する情報を任意の場所で記述できる特徴を持っている．そのため，多様な応用分野で広く利用されはじめている．たとえば，Grid [2] のような大規模な分散環境では，多様で複雑な計算資源やデータベースなどの資源を扱う必要があるので，これらのメタデータを RDF に基づいて記述することが提案されている[3)~5)]．またこれにともなって，RDF に基づいて記述されたメタデータ（RDF データ）は大幅に増加しており，その傾向が今後も維持されると予想できる．また RDF データは各地で記述管理されるために，ネットワーク上に広く散在している．そのため，大量の RDF データが分散した環境で，必要とするデータを効率的に検索することが重要となってきた．

分散環境において，大量データの効率的な検索を実

† 産業技術総合研究所グリッド研究センター
Grid Technology Research Center, National Institute of Advanced Industrial Science and Technology

現する手法として，一般的に Peer-to-Peer（P2P）が用いられる．P2P を用いることで，サーバクライアント型と比べ，スケーラビリティや耐故障性，管理不要性が向上する．RDF データを P2P 上で管理するシステムとして Edutella[6] があるが，Gnutella[7] のような非構造型 P2P ネットワークであるため，ネットワークに参加するすべてのノード☆に対して，データを送信する処理（flooding）が発生し，不要なトラフィックが大量に生じる．

この問題を解決する手法として構造型 P2P がある．構造型 P2P 技術の 1 つである分散ハッシュ表（Distributed Hash Table，DHT）を用いることで，信頼性の向上やネットワークコストの低減が期待できる．OntoGrid Project の Atlas[8] は，Grid 上の資源に対するメタデータを RDF に基づいて記述し，分散ハッシュ表上で管理している．

RDF データは，散在したデータ間に意味的な関連がある．そのため，異なる場所の関連するデータを結合するような構造的な問合せが求められる．RDFPeers[9),10)] は，分散ハッシュ表による RDF トリプル☆☆検索を提供しているが，結合演算を考慮した設計になっておらず，結合対象のすべての解候補集合を 1 つのノードに集めた後，結合処理を行うため，不要なデータの転送が発生する．

RDF には，インスタンスとスキーマがあるが，スキーマを持たない RDF データも存在する．そのため，スキーマを持たない RDF データであっても柔軟に管理できる必要がある．P2P を用いた分散 RDF データベースの 1 つである拡張 Edutella[11] では，スキーマに基づいたルーティングと，スーパーピアを導入し，前述した flooding の問題を抑制しているが，前もってスキーマが必要で，スキーマを持たない RDF データを扱うことはできない．同様に，文献 12) も，スキーマグラフの部分グラフから索引を構築する手法であるため，スキーマを持たない RDF データを扱うことはできない．

これらをふまえて，分散環境における RDF データ検索のために，本論文では，RDF データのインスタンスを構成する RDF トリプル構造に基づいた設計によって，結合演算を含む高度な問合せの効率的な処理を提供することを目的とする．結合演算などの関係データベース処理を P2P 上で実現する試みはある[13)] が，RDF が持つトリプル構造という特徴に基づいた設計を行う．

本論文では，RDF トリプルが存在するか否かを表現できる索引を提案し，それを RDFPeers[9),10)] とともに用いる手法を提案する．提案索引は，結合演算時に RDFPeers に格納された RDF トリプルを参照する前に，索引データを参照，演算することで，解候補となる RDF トリプルを絞り込むことができる．これによって，結合対象のすべての解候補集合を 1 つのノードに集める必要がなくなり，不要なデータの転送を軽減できる．

RDFPeers[9),10)] は，結合演算を含まない単純な問合せは低コストで処理できるが，前述した他の手法[6),11),12)] は，スキーマを持たない RDF データを扱えなかったり，flooding による不要なトラフィックが大量に生じたりするなどの本質的な問題があり，本手法との併用が難しい．また，RDFPeers と併用することで，文献 14) であげたホップ数の増加という問題を解決することができる．

提案する手法は，三次元構造のハッシュ空間に基づいた索引を分散ハッシュ表の上で実現する手法である．まず，RDF トリプルの主語，述語，目的語に対応した三次元構造のハッシュ空間 *RDFCube* を導入し，**セル**と呼ばれる一定のサイズの立方体空間に分割する．その後，すべての RDF トリプルを，RDFCube 上に写像する．各セルは，そのセル内に写像された RDF トリプルが存在するか否かを示す 2 値のフラグ（**トリプルビット**）を保持する．この RDFCube 全体のトリプルビットの集合が索引データである．構築した索引データを用いて検索する場合，解候補のトリプルが写像されているセルのビット（**解候補ビット**）を調べることで，事前にそのセルに写像される解が存在しないことが判断できる．さらに，結合演算時には，結合するすべての解候補ビットが 1 でなければならないことを利用して，論理積によって，解候補を大幅に絞り込むことができる．これによって，不要な RDF トリプルの転送を除外することができる．また，RDF データは分散しているため，構築した索引データ自体も分散管理する必要がある．我々は，分散ハッシュ表を用いてこれを実現した．

本論文の構成を次に示す．2 章では，RDF と分散ハッシュ表，RDFPeers について概説する．3 章で本手法を述べ，4 章で提案手法の性能を実験によって評価し，本手法の有用性を検証する．最後に 5 章でまとめと今後に課題について述べる．

☆ P2P ネットワークに参加する計算機のことを，ノードやピアと呼ぶ．

☆☆ 主語，述語，目的語から構成されるため，このように呼ばれる．また，RDF ステートメントや RDF 文，3 つ組とも呼ばれる．

2. 背　　景

2.1 Resource Description Framework

Resource Description Framework（RDF）[1] は，資源に対する構造的なメタデータの記述を提供した枠組みである．RDF に基づいて記述されたメタデータは，資源間の二項関係を表現できる RDF トリプルと呼ばれる基本単位の集合によって構成されている．RDF トリプルは，主語（subject）と述語（predicate），目的語（object）で構成されており，主語は URI，述語は URI，目的語は URI あるいはリテラルによって表現されており，主語と目的語とが述語の関係で成り立っている．トリプルの集合は，主語と目的語を頂点，述語を有向辺とした有向辺にラベルを持つ有向グラフ構造を表現している．

図 1 に単純な RDF データの例を示す．これは，以下の RDF トリプルによって構成されている☆．

aist:rdfcube *dc*:creator *aist*:matono .
aist:matono *foaf*:name "Matono" .
aist:matono *foaf*:age "28" .

一方，RDF データに対する問合せは，1 つ以上の問合せトリプルで構成されている．問合せトリプルとは，1 つ以上の変数を含んだトリプルのことで，定数である要素を基に一致するトリプルの集合から変数に束縛される要素を検索する．以下に，問合せトリプルの例を示す．この例では，*aist*:rdfcube という資源の作者（*dc*:creator）を求める問合せで，目的語の?x が変数，主語の *aist*:rdfcube と述語の *dc*:creator が定数である．

aist:rdfcube *dc*:creator ?x .

前述の RDF データに上記問合せを発行した場合，この問合せトリプルに一致する RDF トリプルは，⟨*aist*:rdfcube *dc*:creator *aist*:matono⟩ である．すなわち，変数?x に束縛される要素は *aist*:matono である．

2.2 Peer-to-Peer と分散ハッシュ表

Peer-to-Peer（P2P）とは，非集中管理型のネットワークで，任意のノードどうしが直接通信する技術である．一般的に，オーバレイネットワークと呼ばれる，既存のネットワークを利用した仮想的なネットワーク上でリンクを形成して通信が行われる．P2P によって構築されるオーバレイネットワークは，構造型と非構造型に分けられる．分散ハッシュ表は，構造型の P2P 技術の 1 つで，目的の情報を保持するノードの検索を効率的に行うための技術である．

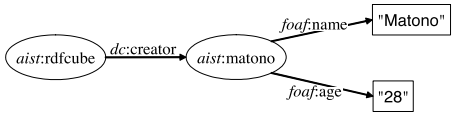

図 1 単純な RDF データ
Fig. 1 A simple RDF data.

☆ 記述構文は Notation 3 http://www.w3.org/DesignIssues/Notation3.html である．本論文では，各資源の接頭辞の名前空間および，リテラルの型は省略する．

分散ハッシュ表は，キーと値のペアを分散して格納する技術で，ノードとキーを同一のハッシュ空間上に写像する．各ノードは，それ自身が写像されたハッシュ値に近い範囲を担当し，その担当する範囲内に写像されたキーとその値を管理する．利用者は，与えられたキーのハッシュ値を求め，そのハッシュ値を担当するノードを探索し，アクセスする．この一連の処理を lookup と呼ぶ．lookup によって，キーとその値の格納や検索を行う．すなわち，一般的に完全一致検索のみが可能である．

最も単純で有名な Chord [15] をあげて，具体的に説明する（**図 2**）．Chord では，n 個のノードから構成されるハッシュ空間が仮想的なリングを形成し，各ノードは前のノード（predecessor）から自ノードまでの範囲のハッシュ値を担当し，次のノード（successor）へのリンクを保持している．加えて，各ノードは finger テーブルを保持している．finger テーブルには，自ノードからハッシュ値が 2^l $(0 \leq l < m)$ の距離にあるノード集合の位置情報（location，一般的に IP アドレス）を保持している（m はハッシュ関数の最大基数．すなわち，ハッシュ値の最大値は $2^m = 2^6 = 64$）．すなわち，近いノードの位置情報は密に，遠いノードは疎に管理される．任意のノードからノードまでのホップ数☆☆は，$O(\log n)$ である．

たとえば，ノード N42 からキー 28 の値を得る場合を考える．まず，ノード N42 の finger テーブル内でキー 28 が含まれる範囲を担当するノードは N11 であることを確認し，N11 にアクセスする．同様に，N11 の finger テーブル内でキー 28 を担当する N27 にアクセスする．このようにルーティングしていき，最終的に，キー 28 の担当ノード N33 にアクセスし，キー 28 の値を得る．

2.3 RDFPeers

RDFPeers [9),10)] は，彼らが提案している分散ハッシュ表 Multi-Attribute Addressable Network（MAAN）[16] を用いた，RDF トリプルのための効率

☆☆ ノードにアクセスする回数．

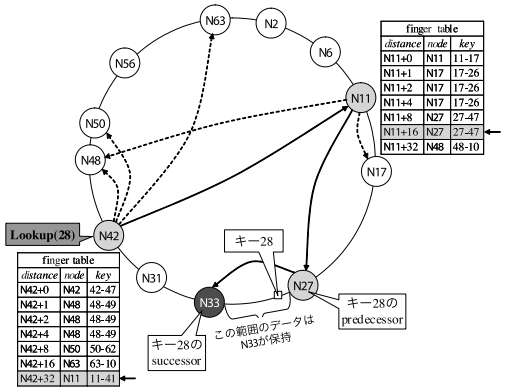

図 2 Chord リング
Fig. 2 Chord ring.

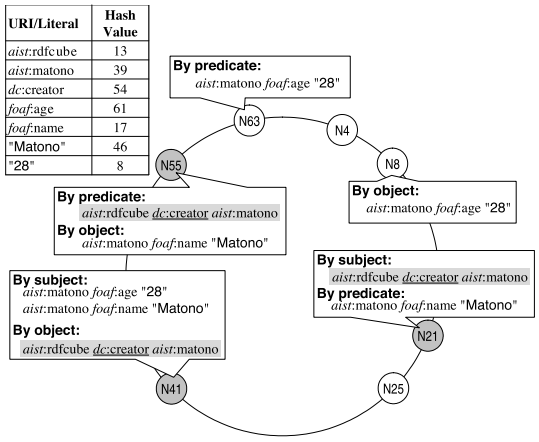

図 3 図 1 の RDF データを RDFPeers に格納
Fig. 3 Storing RDF triples in Fig. 1 using RDFPeers.

的な検索を提供した手法である．MAAN は，Chord を拡張した分散ハッシュ表で，複数の属性と範囲問合せ☆を利用できる．これらの拡張は，既存の分散ハッシュ表を用いても実現できる．複数の属性は，複数回の lookup を行うことで対応でき，範囲問合せは採用するハッシュ関数を変更することで実現できる．

RDFPeers は，RDF トリプルを単位として格納する．与えられたトリプルに対し，主語，述語，目的語のそれぞれをキーとして，値にトリプル自体を格納する．すなわち，1 つのトリプルを格納する過程で，3 回の lookup によって，1 つのトリプルが 3 カ所に格納される．一方，検索では，トリプルが格納されている 3 カ所のいずれか 1 つにアクセスすればよい．与えられた問合せトリプルの定数である要素をキーとして，そのキーのハッシュ値を担当するノードを lookup によって探索する．これによって，定数要素が一致するトリプル集合を取得でき，変数に束縛される要素集合を決定できる．

ノード数が n 個からなる RDFPeers に対する 1 回の lookup に要するホップ数は，通常の分散ハッシュ表と同様に $\log n$ である．したがって，t 個のトリプルの格納時のホップ数は $3t \log n$，q 個の問合せトリプルからなる問合せを処理するためのホップ数は $q \log n$ である．

図 3 に図 1 で示した RDF データを RDFPeers に格納した例を示す．左上の表は，各要素とそのハッシュ値を示している．たとえば，トリプル ⟨*aist*:`rdfcube` *dc*:`creator` *aist*:`matono`⟩ の場合，主語 *aist*:`rdfcube` のハッシュ値は 13 であるため，ハッシュ値 13 の位置を担当するノード N21 が，主語をキーとしてトリプルを保持する．同様に，ノード N55 とノード N41 が，述語 *dc*:`creator` と目的語 *aist*:`matono` をそれぞれキーとして，トリプルを格納する．

☆ 範囲問合せとは $0 < a < 1$ などのような，値の範囲を用いた問合せのことである．

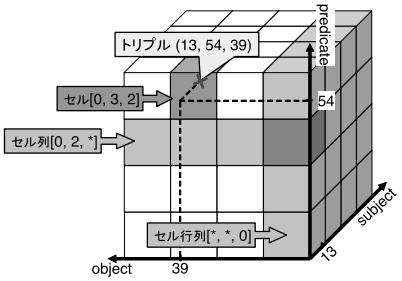

図 4 三次元ハッシュ空間（RDFCube）
Fig. 4 A three-dimensional hash space (RDFCube).

3. 提 案 手 法

3.1 三次元ハッシュ空間 RDFCube

我々は，分散した RDF データの効率的な結合演算を実現するために，*RDFCube* と呼ばれる三次元ハッシュ空間を提案する．RDFCube のモデルを**図 4** に示す．RDFCube は，主語，述語，目的語をそれぞれの次元に対応させた，立方体構造をしており，各次元はハッシュ空間を表現している．

ハッシュ関数の最小値を 0，最大値を 2^m とし，RDFCube の主語，述語，目的語の空間を，それぞれ 2^c 個に分割する（$0 < c \leq m$; m はハッシュ関数の最大基数）．このとき，主語，述語，目的語のそれぞれに対応したハッシュ空間上の区間，$[2^{m-c}i, 2^{m-c}(i+1))$，$[2^{m-c}j, 2^{m-c}(j+1))$，$[2^{m-c}k, 2^{m-c}(k+1))$ で囲まれた 3 次元空間を**セル**と呼び，セル座標 $[i, j, k]$ で一

意に識別する ($0 \le i < 2^c$, $0 \le j < 2^c$, $0 \le k < 2^c$). i, j, k を各空間上のセル値と呼ぶ.

いずれかの軸に平行な直線上に存在するすべてのセルの集合を**セル列**と呼ぶ. たとえば, 主語軸に平行なセル列は $[\{0, \ldots, 2^c - 1\}, j, k]$ と記述し, $[*, j, k]$ と略記する. 同様に, いずれかの軸に直交する平面上のすべてのセルの集合を**セル行列**と呼ぶ. たとえば, 目的語軸に垂直なセル行列は $[\{0, \ldots, 2^c - 1\}, \{0, \ldots, 2^c - 1\}, k]$ と記述し, $[*, *, k]$ と略記する. 図 4 にセル $[0, 3, 2]$, セル列 $[0, 2, *]$, セル行列 $[*, *, 0]$ を例示する.

セル行列は, 3 セル値のうち 1 セル値が決定しているため, その行列上のセルは, 2 次元のセル座標 $[u, v]$ によって, 識別できる. このとき, セル値の記述優先順位は, 主語, 述語, 目的語の順に高いものとする. たとえば, セル $[i, j, k]$ はセル行列 $[i, *, *]$ 上ではセル座標 $[j, k]$ で識別され, セル列 $[i, *, k]$ はセル行列 $[i, *, *]$ 上ではセル座標集合 $[*, k]$ で識別される.

RDF トリプルは, 主語, 述語, 目的語のそれぞれのハッシュ値に基づいて, RDFCube 内の座標に写像され, そのトリプルを担当するセルが決定される. たとえば, トリプル $\langle aist{:}\texttt{rdfcube}\ dc{:}\texttt{creator}\ aist{:}\texttt{matono}\rangle$ を RDFCube 上に写像する場合, RDFCube の各空間の区間を $[0, 64)$, 分割数を 4 とし, トリプルのそれぞれの要素のハッシュ値が以下であったと仮定すると,

$$hash(aist{:}\texttt{rdfcube}) = 13$$
$$hash(dc{:}\texttt{creator}) = 54$$
$$hash(aist{:}\texttt{matono}) = 39$$

与えられた RDF トリプルは, 図 4 に示すように RDFCube の $(13, 54, 39)$ の座標に写像される. したがって, このトリプルを担当するセルは $[0, 3, 2]$ となる.

RDFCube の各セルは, 自セル内に写像される RDF トリプルが存在するか否かを示す 2 値の状態を表す**トリプルビット**と呼ぶフラグを保持する. セルの集合 $\{c_1, c_2, \ldots, c_l\}$ のビットの状態を示す関数を $bits(\{c_1, c_2, \ldots, c_l\})$ とする. たとえば, $bits(\{c_1, c_2\}) = \{1, 0\}$ は, セル c_1 の空間内に写像される RDF トリプルが存在し, セル c_2 の空間内に写像される RDF トリプルが存在しないことを示す. $bits$ 関数に与えられたセル集合がセル列であるとき, その値を**ビット列**と呼び, $bits$ 関数に与えられたセル集合がセル行列であるとき, その値を**ビット行列**と呼ぶ. また, あるビット集合中のトリプルビットが 1 である割合を**ビット濃度**と呼ぶ.

問合せ RDF トリプルも RDFCube 内に写像される. 問合せトリプルは, 変数を含むため, 定数要素によって, それが写像される座標集合を決定する. すなわち, 2 つが変数である場合は 1 平面上に写像され, 1 つが変数である場合は, 1 直線上に写像される. ある問合せトリプルの写像先の座標集合は, 問合せトリプルの解集合である RDF トリプル集合が写像されている. すなわち, 写像先の座標集合を担当するセルの集合が, その問合せトリプルを担当するセル集合である. これらの問合せトリプルが写像されるセル集合を**解候補セル**集合と呼び, それらが保持するビット集合を**解候補ビット**集合と呼ぶ.

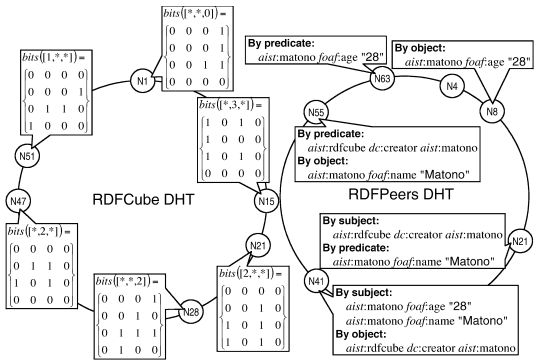

図 5 2 つの分散ハッシュ表
Fig. 5 Two distributed hash tables.

3.2 RDFCube のための分散ハッシュ表

我々は, 分散環境上で RDFCube を実現するために分散ハッシュ表を用いる手法を提案する. これにより, ビット情報を保持した RDFCube のための分散ハッシュ表と, RDF トリプル情報を保持した RDFPeers のための分散ハッシュ表の 2 つが構築される. **図 5** に 2 つの分散ハッシュ表を示す. 右側が RDFPeers のための, 左側が RDFCube のための分散ハッシュ表である.

提案手法では両方を用いて問合せ処理を行う. まず左の分散ハッシュ表からトリプルビット情報を取得し, ビット演算を行い, 演算したビット情報を基に, 右の分散ハッシュ表から RDF トリプルを取得する. すなわち, RDFCube は RDFPeers の検索処理前に効率の向上を目的として利用する点で, RDFPeers のための索引であるといえる.

RDFCube の分散ハッシュ表は, セル行列の座標をキーとし, そのセル行列が保持するビット行列を値として格納する. これにより定数を少なくとも 1 つ以上持つ問合せトリプルであれば, セル行列の座標を決定でき, それをキーとして lookup 探索することで, その解候補ビット集合を取得できる.

3.3 RDFCube の構築

本節では, RDFCube を構築する手法を述べる. RD-

Algorithm 1：RDFCube の構築

Input: triple t

```
1 cell [i,j,k] ← cells(t)
2 if onBit([i,*,*], [j,k]) is 0 then
3 |   onBit([*,j,*], [i,k])
4 |   onBit([*,*,k], [i,j])
```

FCube の構築とは，与えられたトリプル t が写像されるセルのビットを 1 にすることである．このれらは分散ハッシュ表上に格納されるため，構築処理には lookup によるネットワーク通信をともなう．

Algorithm 1 に RDFCube 構築の手続きを示す．まず，1 行目で与えられたトリプルが写像されるセル $[i,j,k]$ を `cells` 関数によって決定し，2–4 行目で `onBit` 関数によって $bits([i,j,k])$ を 1 に更新する．`onBit` 関数は，分散ハッシュ表に対してセル行列 $[i,*,*]$，$[*,j,*]$，$[*,*,k]$ のそれぞれをキーに lookup を行い，それぞれのセル行列上の座標 $[j,k]$，$[i,k]$，$[i,j]$ のビットを 1 に更新する．また，返り値として更新前のトリプルビットの値を返す．これによって，2 行目の結果から，対象のセルにトリプルがすでに写像されているかどうかを判断でき，リモートノードへの不要なアクセスを減少できる．

2 行目で返されるビットが 1 のときは 1 回の lookup，0 のときは 3 回の lookup が行われる．すなわち，ノード数が n である分散ハッシュ表に対して，t 個のトリプルの索引を構築するホップ数は，たかだか $3t\log n$ となる．同様に，2.3 節に示したように t 個のトリプルを RDFPeers に格納するホップ数は $3t\log n$ である．そのため，RDFCube 構築と RDFPeers 格納の合計のホップ数は，たかだか $6t\log n$ となる．

3.4 問合せ処理

提案索引は，結合演算処理のための索引である．結合演算とは，複数の問合せトリプルを含む RDF 問合せにおいて，異なる問合せトリプルどうしで同じ変数が指定されている場合に行われる演算である．結合演算が行われない問合せでは提案索引を用いず，RDFPeers のみによる問合せ処理を行う．

分散ハッシュ表のノード数が n のとき，1 つの問合せトリプルを RDFPeers から取得するホップ数は $\log n$ で，RDFCube からそのビット情報を取得するホップ数も $\log n$ である．そのため，q 個の問合せトリプルからなる検索処理に提案索引を用いた場合のホップ数は $2q\log n$ となる．

Algorithm 2：問合せ処理

Input: set of query triples Q
Output: set of triples T

```
1  set of triples T ← ∅
2  map of query triples and sets of bits M ← ∅
3  foreach q ∈ Q do
4  |   set of cells cells ← cells(q)
5  |   set of bits bits ← getBits(cells)
6  |   M.put(q, bits)
7  M ← andOperation(M, Q)
8  foreach q ∈ Q do
9  |   set of bits bits ← M.get(q)
10 |   T ← T ∪ getFilteredTriples(q, bits)
11 T ← apply(Q, T)
12 return T
```

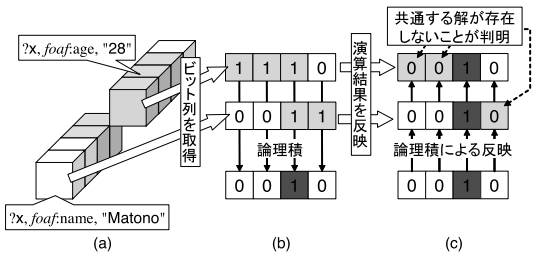

図 6 結合演算の例
Fig. 6 An example of join operation.

3.4.1 結合演算を含む問合せ処理

提案索引を用いた問合せ処理の手続きを Algorithm 2 に示す．まず，i) 1–2 行目で初期化処理を行う．ii) 問合せトリプル q からその解候補ビット集合 $bits$ を RDFCube の分散ハッシュ表から取得し，マップ M に格納し（3–6 行目），iii) 不要なビットを除去するよう論理積演算を行い，M を変更する（7 行目）．その後，iv) 変更後の M を基に，ビットが 1 であるセルに写像されたトリプル T のみを RDFPeers の分散ハッシュ表から取得する（8–10 行目）．最後に，v) 得られた解候補のトリプル集合に対して，従来の RDF 問合せ処理を行い，正しい解を得る（11 行目）．

以下の例を用いながら，Algorithm 2 を説明する．**図 6** にこの例を処理する流れを示す．

```
?x  foaf:name  "Matono" .
?x  foaf:age   "28" .
```

ii) 4 行目で `cells` 関数によって，各問合せトリプルの解候補セル集合 $[\{0,1,2,3\},1,2]$ と $[\{0,1,2,3\},3,0]$ を得る（図 6 (a)）．5 行目でこのセル列を `getBits` 関数に渡し，解候補ビット集合 {0 0 1 1} と {1 1 1 0} を得る．`getBits` 関数は，入力のセル

集合が保持するビット集合を RDFCube 分散ハッシュ表から取得する関数で，詳細は 3.4.2 項に示す．6 行目では，問合せトリプル q をキーとして，解候補ビット集合 $bits$ を M に格納する．M は問合せトリプルとその解候補ビット集合のペアが格納されるマップである．

iii) 7 行目で `andOperation` 関数によって図 6 (b) のように解候補ビット列どうしの論理積演算を行い ($bits([*,1,2]) \wedge bits([*,3,0]) = \{0\ 0\ 1\ 0\}$)，その結果を図 6 (c) のように解候補ビット集合 $bits([*,1,2])$ と $bits([*,3,0])$ に反映させ，M を更新する．この処理の詳細は 3.4.3 項に示す．この時点で M 内には，$\langle$?x *foaf*:name "Matono"$\rangle$ の値として $bits([*,1,2]) = \{0\ 0\ 1\ 0\}$ が，$\langle$?x *foaf*:age "28"$\rangle$ の値として $bits([*,3,0]) = \{0\ 0\ 1\ 0\}$ が格納されている．したがって，前者の問合せトリプルの解候補セル集合は，$[2,1,2]$ で，後者の解候補セル集合は $[2,3,0]$ になる．(図 6 (c) の色の濃いセル)．反対に，セル $[3,1,2]$ と $[\{0,1\},3,0]$ は，ii) の段階では，解候補セルであったが，iii) のビット演算によって解候補セルではなくなる（図 6 (c) の色の薄いセル)．

iv) その後，`getFilteredTriples` 関数を用いて，問合せトリプルの解候補セル集合 $[2,1,2]$ あるいは $[2,3,0]$ に写像されるトリプルのみになるようフィルタリングしながら，RDFPeers の分散ハッシュ表から解候補トリプル集合を取得する．この処理はリモートノード上で行われる．詳細は 3.4.4 項に示す．

v) 11 行目で，得られた解候補のトリプル集合に対して，通常の RDF 問合せ処理を発行し，正しい解を得る．この処理は，RDFPeers における結合処理と同じであるが，提案索引を用いた場合は，すでに，iv) でのフィルタリングによって不要なトリプルを除去してあるため，RDFPeers のみの場合と比べ，入力の集合は小さい．

3.4.2 分散ハッシュ表上のビット集合の取得

本項では，RDFCube のため分散ハッシュ表からビット集合を取得する手順，すなわち Algorithm 2 の `getBits` 関数について述べる．Algorithm 3 にその手順を示す．この手順はセル列あるいはセル行列が入力として与えられ，それのビット列あるいはビット行列が返される．入力の型に応じて場合分けされており，セル列の場合は 1–7 行目の `getBitSeq` 関数が，セル行列の場合は 8–14 行目の `getBitMatrix` 関数が呼ばれる．`getBitMatrix` 関数は，与えられたセル行列に対応するビット行列をそのまま返すが，`getBitSeq` 関数は行列のうち必要な列のみを転送する．

Algorithm 3：getBits

Input: cell sequence or cell matrix $[i,j,k]$
Output: bit sequence or bit matrix

```
1  if [i,j,k] is cell sequence then
2      if i is variable then
3          return getBitSeq([*,j,*], [*,k])
4      else if j is variable then
5          return getBitSeq([i,*,*], [*,k])
6      else if k is variable then
7          return getBitSeq([i,*,*], [j,*])
8  else if [i,j,k] is cell matrix then
9      if i is constant then
10         return getBitMatrix([i,*,*])
11     else if j is constant then
12         return getBitMatrix([*,j,*])
13     else if k is constant then
14         return getBitMatrix([*,*,k])
```

たとえば，セル列 $[*,3,0]$ のビット列を取得する場合，3 行目で `getBitSeq`($[*,3,*],[*,0]$) が呼ばれる．この関数は，転送量を減少させるために，セル行列全体ではなく，必要なセル列のみを転送する．たとえば，セル行列 $[*,3,*]$ をキーとして，次のビット行列が分散ハッシュ表に格納されていたとする．

$$bits([*,3,*]) = \left\{ \begin{array}{c} 1\ 0\ 1\ 0 \\ 1\ 0\ 0\ 0 \\ 1\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0 \end{array} \right\}$$

この行列は行が主語，列が目的語を示しているため，第 2 引数の座標集合 $[*,0]$ から 0 列目である以下のビット列のみが結果として返送される．

$$bits([*,3,0]) = \left\{ \begin{array}{c} 1 \\ 1 \\ 1 \\ 0 \end{array} \right\}$$

3.4.3 解候補ビット間の論理積

本項では，解候補ビットどうしの論理積演算処理，すなわち Algorithm 2 の `andOperation` 関数について述べる．その手順を Algorithm 4 に示す．この手順は，結合するビット集合の構造に関係なく，いったん一次元の列を抽出して，演算した後，元のビット集合を再構築するため，列どうし，行列どうし，行列と列などのあらゆる構造の結合演算を処理できる．

以下の例を用いて，Algorithm 4 を説明する．前提

Algorithm 4：andOperation

Input: map of query triples and sets of bits M, set of query triples Q
Output: map of query triples and sets of bits M

```
1  initialize map vMap
2  foreach qTriple ∈ Q do
3      set of bits bits ← M.get(qTriple)
4      foreach var ∈ variables in qTriple do
5          bit sequence mask ← vMap.get(var)
6          bit sequence seq ← project(bits, var)
7          mask ← and(mask, seq)
8          vMap.put(var, mask)
9  foreach qTriple ∈ Q do
10     set of bits bits ← M.get(qTriple)
11     set of variables vars ← varibables in qTriple
12     set of bit sequences seqs ← vMap.get(vars)
13     set of bits tmp ← backProject(seqs)
14     bits ← and(tmp, bits)
15     M.put(qTriple, bits)
16 return M
```

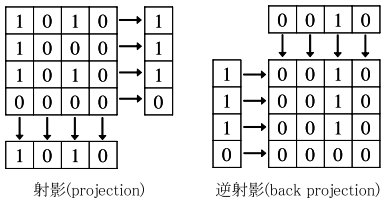

図 7 射影と逆射影
Fig. 7 Projection and back projection.

として，1 行目の問合せトリプルはセル行列 $[*, 3, *]$ に，2 行目はセル列 $[*, 1, 2]$ に写像されるとする．

?x *dc*:creator ?y .
?y *foaf*:name "Matono" .

$vMap$ は変数とその解候補ビット列のペアを格納するマップで，1 行目では，Q に出現するすべての変数をキーに，全ビットが 1 に初期化されたビット列を格納する．まず，3 行目で M から得られた解候補ビット集合が以下であるとする．

$$bits([*,3,*]) = \left\{ \begin{array}{l} 1\ 0\ 1\ 0 \\ 1\ 0\ 0\ 0 \\ 1\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0 \end{array} \right\}$$

$$bits([*,1,2]) = \left\{ 0\ 0\ 1\ 1 \right\}$$

6 行目で，project 関数の論理和による射影によって，各変数の解候補ビット列を取得する．射影とは，**図 7** の左のように，b 次元構造のビット集合から論理和によって b 個のビット列を生成する手続きである．1 行目の問合せトリプルの場合，ビット行列 $bits([*, 3, *])$ は行が主語，列が目的語に対応している．そのため，変数?x は各行で，変数?y は各列で論理和を行い，以下のような結果を得る．

$$bits(?x \in \langle ?x\ dc{:}creator\ ?y \rangle) = \left\{ 1\ 1\ 1\ 0 \right\}$$

$$bits(?y \in \langle ?x\ dc{:}creator\ ?y \rangle) = \left\{ 1\ 0\ 1\ 0 \right\}$$

一方，2 行目の問合せトリプルの場合，変数は?y のみであるため，変数?y に対応するビット列は，以下のように射影後も変化ない．

$$bits(?y \in \langle ?y\ foaf{:}name\ \text{"Matono"} \rangle) = \left\{ 0\ 0\ 1\ 1 \right\}$$

これらのビット列は，各変数に束縛されるトリプル集合が写像されるハッシュ値の範囲を表す解候補ビット列である．

その後，7 行目の and 関数ですべての同一変数の解候補ビット列と $vMap$ に格納されているビット列とで論理積を行い，8 行目でその結果を $vMap$ に再格納する．?x は，結合を行う対象がないため，解候補ビット列がそのまま $vMap$ に格納される．一方?y は，$bits(?y \in \langle ?x\ dc{:}creator\ ?y \rangle)$ と $bits(?y \in \langle ?y\ foaf{:}name\ \text{"Matono"} \rangle)$ との論理積の結果が $vMap$ に格納される．以下の 2 つがそれぞれ $vMap$ に格納される．

$$bits(?x) = \left\{ 1\ 1\ 1\ 0 \right\}$$

$$bits(?y) = \left\{ 0\ 0\ 1\ 0 \right\}$$

これらのビット列は，異なる問合せトリプルで共通して，各変数に束縛されるトリプル集合が写像されるハッシュ値の範囲を表す解候補ビット列である．

次に，11 行目で，$qTriple$ に出現するすべての変数 $vars$ を取得し，12 行目で $vars$ に対応するビット列の集合 $seqs$ を $vMap$ から取得する．この $seqs$ とは，1 行目の問合せトリプルの場合，上の $bits(?x)$ と $bits(?y)$ で，2 行目の場合 $bits(?y)$ のみである．

13 行目で backProject 関数によって逆射影を行い，ビット集合 tmp を生成する．逆射影とは，図 7 の右側のように，b 個のビット列から論理積によって b 次元構造のビット集合を生成する手続きである．その後，14 行目で tmp と，M から取得したビット集合 $bits$ との論理積を行い，$bits$ を変更する．なお tmp と $bits$ は同次元構造のビット集合である．具体的には，変数ごとの解候補ビット列 $bits(?x)$ と $bits(?y)$ の逆射影の結果を $bits([*, 3, *])$ に反映させ，$bits(?y)$ を $bits([*, 1, 2])$ に反映させる．その結果は次のようになる．

Algorithm 5：filterTriples

Input: candidate triples C, set of bits $bits$
Output: a set of triples T

```
1 a set of triples T ← ∅
2 foreach triple ∈ C do
3     a cell [i,j,k] ← cells(triple)
4     a bit bit ← bits.getBit([i,j,k])
5     if bit = 1 then
6         T ← T ∪ triple
7 return T
```

$$bits'([*,3,*]) = \left\{ \begin{matrix} 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0 \end{matrix} \right\}$$

$$bits'([*,1,2]) = \left\{ 0\ 0\ 1\ 0 \right\}$$

結果的に，演算前と比べると，不要なセルのビットは 0 に変更されている．このように，すべての問合せトリプルで共通して解が存在するセルのビットのみが 1 を保つことになる．

3.4.4 解候補のフィルタリング

本項では，トリプルの集合をフィルタリングする手順，すなわち Algorithm 2 の `getFilteredTriples` 関数の過程で行う処理について述べる．その手順を Algorithm 5 に示す．この処理は，転送される解候補集合を小さくするためのフィルタリング処理で，lookup によってアクセスしたリモートノード上で実行され，この処理の返り値がローカルノードへ転送される．すなわち，この処理が実行されるときは，すでに解候補のトリプル集合が決定している．入力は問合せトリプルの解候補集合 C と問合せトリプルの解候補ビット集合 $bits$ で，出力はフィルタリングされた結果のトリプル集合である．

`cells` 関数は，ある解候補トリプル $triple$ が写像されるセル $[i,j,k]$ を決定し，`getBit` 関数によって，解候補ビット集合 $bits$ から，セル $[i,j,k]$ が保持するビット $bits([i,j,k])$ を返す．$bits([i,j,k]) = 0$ のとき，セル $[i,j,k]$ に写像されたトリプル $triple$ は不要な解であることを意味している．

3.5 転送索引サイズ抑制のための拡張

RDFCube によって転送する RDF トリプルを減少させることができる．しかし，提案索引を利用するためには，ビット集合を RDFCube の分散ハッシュ表から受信し，演算した結果をフィルタリングで利用するために，ビット集合を RDFPeers の分散ハッシュ表へ送信する必要がある．したがって，RDFCube の各次元の分割数を c としたときの索引データの転送量は，ビット列の場合 $2c$ となり，ビット行列の場合 $2c^2$ となる．前者の場合は分割数に比例する程度であるため，RDF データに比べ十分小さいが，後者の場合分割数の累乗に比例するため，けっして無視できない大きさになる．

そこで，ビット行列を転送するのではなく，ビット行列から射影した 2 つのビット列を転送する拡張手法を提案する．これによって，2 つのビット列が 2 回転送されるため，転送量を $4c$ に抑制できる．

2 列を転送する手法は，行列を転送する手法とは異なり，ビット行列を取得していないため，Algorithm 4 の 14 行目で $bits$ を生成できず，代わりに tmp によってフィルタリングを行う．tmp は，逆射影したのみで，ビット行列との論理演算を行っていないため，トリプルが写像されていないセルのビットまで 1 になっている可能性がある．しかし，たとえそうであったとしても，そのセルに写像されているトリプルが存在しないために，そのトリプルは転送されない．

ただし，両手法でトリプルの転送量が異なる場合もある．行列を転送する手法の場合，行列全体を取得しているために，変数 1 つずつに対して射影し，論理積を行い，1 つずつ逆射影する手順を採用できる．すなわち，これによって，ある変数の演算の結果が，他の変数の結果に影響を与える．一方，2 列を転送する場合，変数 1 つずつで処理できないため，すべてまとめて射影，論理積，逆射影を行わなければならない．そのため，他の変数の演算結果は影響しない．この違いによって，転送量が増加する場合がある．しかしながら，これによって発生する差より，ビット集合の差の方が大きい場合がほとんどであるため，理論的には 2 列を転送する手法の方が効率が良い．

4. 性能評価

4.1 実験環境

本章では，提案した手法の性能を実験的に評価する．実験では，索引として RDFCube を用いた場合と用いない場合との性能を比較する．本実験ではそれぞれを rdfcube と rdfpeers と呼ぶ．

実験に用いたデータは，DBLP で配布している XML 文書☆を RDF データに変換し，いくつかトリプルの数が異なるようにランダムに抽出したものを用いた．抽出した RDF データの統計を**表 1** に示す．DBLP の

☆ `http://dblp.uni-trier.de/xml/`

表 1 実験に用いた RDF データの統計値
Table 1 Statistics of RDF data used in experiment.

	size (KB)	#triples	#主語	#述語	#目的語
data 1	627	13,761	1,427	20	9,240
data 2	1,288	26,413	2,726	20	16,737
data 3	2,580	52,926	5,443	21	31,119
data 4	5,016	103,076	10,771	22	56,080

データは，文献情報を列挙した構造をしており，著者や年，タイトルなど基本要素が類似したエントリを多く含むため，RDF の述語の種類は少ない．この構造は，RSS や Dublin Core と類似した構造であり，一般的な構造の RDF データであるといえる．

実験に用いた問合せは，次の 3 つである．Query 1 はビット列どうしを並列に結合する問合せで，Query 2 はビット行列とビット列を直列に結合する問合せで，Query 3 はビット行列どうしの並列な結合と，列と行列の直列な結合が行われる問合せである．

Query 1

```
?x  rdf:type  dblp:Article.
?x  dblp:author "Jim Gray".
?x  dblp:year "1998".
?x  dblp:journal "CoRR".
```

Query 2

```
?x  dblp:series ?y. ?y  dblp:title "LNCS".
```

Query 3

```
?y  dblp:crossref ?x. ?x  dblp:title "VLDB2004".
?y  dblp:title ?z.
```

評価実験は以下の 4 つを行う．それぞれの評価結果は 4.2.1 項から 4.2.4 項で述べる．

(1) 転送索引サイズ抑制のための拡張手法の評価．行列を転送する手法と，2 つの列を転送する手法との比較を行う．データセットを data 2，各次元の分割数を 32 から 256 へ増加させ，問合せは Query 2 と 3 を用いて，それぞれの転送量の比率を示す．

(2) 格納性能比較．以下 2 つの条件において，ホップ数と転送データ量の比率を計測する．a) データセットを data 1–4 へ変化，分割数を 128 とする．b) データセットを data 2，分割数を 32 から 256 へ増加させる．

(3) 検索性能比較．以下 2 つの条件において，ホップ数と転送データ量の比率を計測する．a) データセットを data 1–4 へ変化，分割数を 128，問合せは Query 1–3 を用いる．b) データセットを data 2，分割数を 32 から 256 へ増加，問合せは Query 1–3 を用いる．

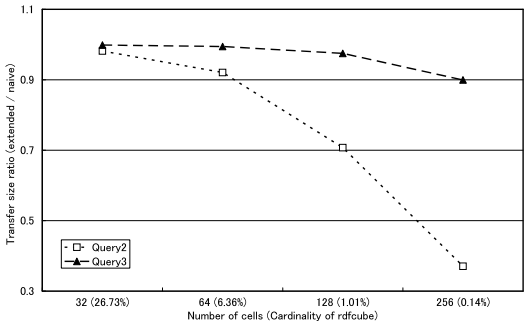

図 8 転送索引サイズの抑制（横軸は分割数，縦軸は naive に対する extended の転送量比率）
Fig. 8 Decreasing the amount of transfer index data (the horizontal axis shows #cells while the vertical axis shows the ratio of extended to naive).

(4) スケーラビリティ評価．トリプル数の増加に比例するように分割数も増加させ，問合せは Query 1–3 を用いて，rdfcube と rdfpeers の転送量の比率が一定であるかどうかを確認する．

なお，ノード数に対するスケーラビリティは，提案手法で利用する分散ハッシュ表のアルゴリズムや基盤システムに依存し，提案手法の性能とは独立である．そのため，本論文での評価実験では，ノード数はつねに 100 として行う．

実験は分散ハッシュ表環境を提供する Overley Weaver [17] 上で RDFPeers と提案索引手法を実装し，エミュレートして計測した．実験では，分散ハッシュ表に Chord [15] を用い，最初にすべてのノードの finger テーブルを完全に構築し，ノードの離脱や追加は行わないものとした．

4.2 実験結果

4.2.1 転送索引サイズ抑制手法の評価

索引データのサイズを抑えるための拡張手法の評価実験を行った．**図 8** に行列を転送する手法（naive）と，2 つのビット列を転送する手法（extended）での転送量の比率を示す．

この図から，分割数の増加にともなって extended の方が naive と比べ転送量の面で効率的であることが分かる．特に，各次元の分割数の増加とともに，naive と extended の差が拡大している．この結果から，extended が優れていることが確認できる．残りの実験は，すべて extended を用いて行った結果である．

4.2.2 格納性能評価

索引構築時の rdfcube と rdfpeers の性能を比較した．パラメータとして，トリプル数と分割数を用い，それぞれを横軸として**図 9** と**図 10** に示す．

これらのグラフから，データ格納と同時に索引を構

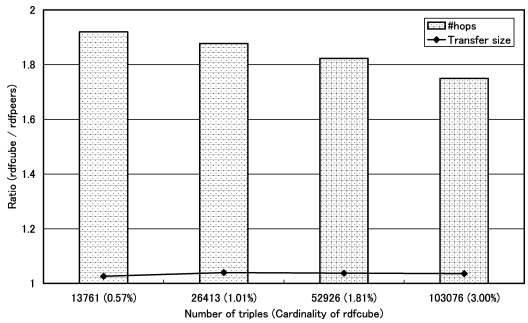

図 9 格納性能 1（横軸はデータのトリプル数，縦軸は rdfpeers に対する rdfcube の比率）

Fig. 9 Storing performance 1 (the horizontal axis shows #triples while the vertical axis shows the ratio of rdfcube to rdfpeers).

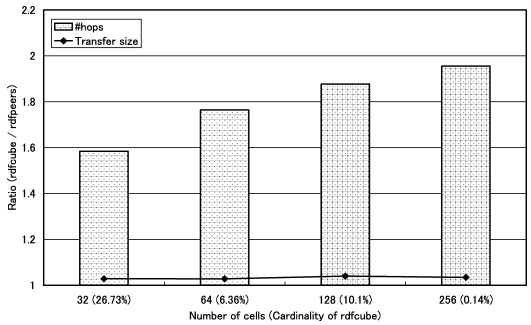

図 10 格納性能 2（横軸は分割数，縦軸は rdfpeers に対する rdfcube の比率）

Fig. 10 Storing performance 2 (the horizontal axis shows #cells while the vertical axis shows the ratio of rdfcube to rdfpeers).

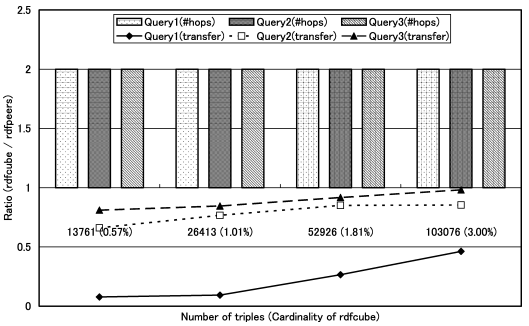

図 11 検索性能 1（横軸はデータのトリプル数，縦軸は rdfpeers に対する rdfcube の比率）

Fig. 11 Retrieval performance 1 (the horizontal axis shows #triples while the vertical axis shows the ratio of rdfcube to rdfpeers).

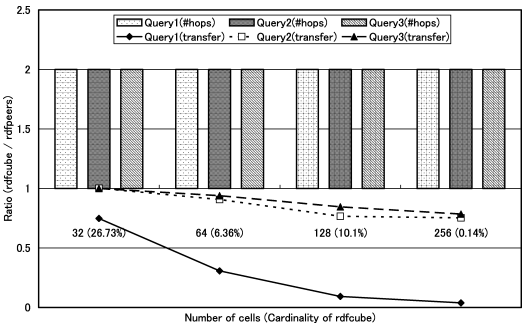

図 12 検索性能 2（横軸は RDFCube の分割数，縦軸は rdfpeers に対する rdfcube の比率）

Fig. 12 Retrieval performance 2 (the horizontal axis shows #cells while the vertical axis shows the ratio of rdfcube to rdfpeers).

築したとしても，ホップ数がおよそ 2 倍以下になることが分かる．これは，3.3 節で述べたように，索引構築時には 3 回あるいは 1 回の lookup が行われるためで，索引構築に要するホップ数が，データ格納に要するホップ数よりつねに少ないことが確認できる．また，トリプル数の増加あるいは分割数の減少にともなって，ビット濃度が高くなり，ホップ数の割合が低下している．これは 1 つのセルに複数のトリプルが写像される割合が増加するためである．

一方，データの転送量は，索引を構築する場合でもほぼ増加しないといえる．この理由は，データ格納ではトリプルを転送するのに対し，索引構築では，セル座標のみを転送するためで，RDF トリプルに比べ，セル座標のサイズが十分小さいことが確認できる．

4.2.3 検索性能評価

索引構築時の性能を評価する．rdfpeers と rdfcube の転送量とホップ数で比較した．パラメータとして，トリプル数と分割数を用い，それぞれを横軸として**図 11** と**図 12** に示す．

図 11 と図 12 から，rdfcube は rdfpeers に比べ，2 倍のホップ数を要することが分かる．これは，RDF トリプル取得と索引データ取得に要するホップ数が等しいことを意味しており，文献 14) の結果を改善することができている．rdfpeers では，問合せトリプル数だけ lookup を行うため，もともとホップ数は非常に小さいため，2 倍になったとしても，十分実用に耐えられる結果であると考えている．

一方，転送量の側面では最大およそ 50 分の 1 まで効率化できている．並列結合演算かつビット列どうしの演算である Query 1 は，?x が 4 回出現しているため，絞り込みの精度が高い．また，Query 2–3 では，同一変数が問合せ中に出現する回数が 1 あるいは 2 しかないため，大幅な絞り込みができていない．図 12 の分割数が 32 のときは，rdfcube がわずかに rdfpeers に劣っている．これは，ビット濃度が 26.73%と高くなっており，絞り込みがまったくできず，索引データ

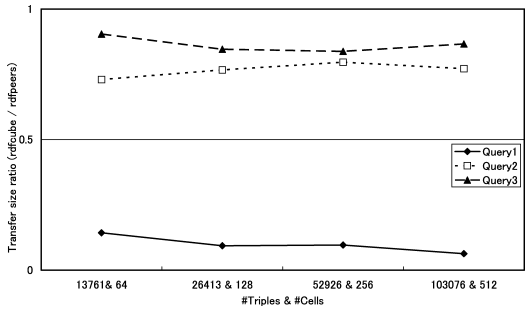

図 13 スケーラビリティ実験の結果（横軸はデータのトリプル数と分割数，縦軸は rdfpeers に対する rdfcube の転送量比率）
Fig. 13 Result of scalability test (the horizontal axis shows #triples and #cells while the vertical axis shows the ratio of rdfcube to rdfpeers).

分だけ転送量が増加したためである．

これらのことから，提案索引の性能を引き出すためには，ビット濃度が低い状況で，同一変数が多く出現する問合せを処理する際に利用されたい．

4.2.4 スケーラビリティ評価

最後に，RDF トリプル数の増加に対するスケーラビリティを確認する．**図 13** は，トリプル数の増加に比例して分割数も増加させた場合の rdfcube と rdfpeers の転送量の比率を示している．

図 13 が示すように rdfpeers に対する rdfcube の転送量の比率は，各問合せでほぼ一定であると判断できる．これらから提案索引を用いた場合の転送量の抑制性能は，分割数を比例して増加させることで，トリプル数の増加に対するスケーラビリティを確保できる．

5. まとめと今後の課題

本論文では，分散環境における RDF データを問合せ処理を効率化することを目的とし，RDFPeers [9),10)] 上で，結合演算を処理するための索引手法を提案した．提案手法は，ビットの論理積を行うことで，不要なデータを転送前に決定でき，転送量を減少させることができる．文献 14) で提案した手法は，検索時のホップ数の増加という問題を残していたが，本提案手法では 2 倍程度に維持できた．提案手法は，トリプル数の増加に対して，転送量抑制の割合を一定に維持でき，また，ビット濃度が低く，同一変数を多く含む問合せで，より効率的であることを確認した．

今後の課題としてトリプルの削除や，動的なセル分割がある．提案手法では，トリプルの削除を考慮できていない．素朴な解決策として，ビット型ではなく，正の整数型を採用することで，その値を増減させ RDF トリプル追加，削除に対応できる．しかしながら，この手法では，データサイズの増加という問題や，索引構築時のホップ数，転送量増加という問題が生じる．また，動的なセル分割はスケーラビリティを確保するうえで重要な課題である．単純に分割時に近隣セルのビットをコピーする手法では，ビット状態が不正確になるため，それを修正する手法が必要になる．さらには，データの再分散を不要とする手法の提案や，RDF スキーマを利用した問合せ処理の効率化も今後考察する必要がある．

謝辞 実験に際し，当グリッド研究センターの首藤一幸氏（現ウタゴエ）に，多大なご助力を賜りました．ここに記して謝意を表します．

参 考 文 献

1) World Wide Web Consortium: Resource Description Framework (RDF), `http://www.w3.org/RDF/` (2004). W3C Recommendation 10 February 2004.
2) Foster, I. and Kesselman, C. (Eds.): *The Grid: Blueprint for a New Computing Infrastructure*, 1st edition, Morgan-Kaufman, San Francisco, CA, USA (1998).
3) Said, M.P. and Kojima, I.: Ontology-Based Grid Index Service for Advanced Resource Discovery and Monitoring., *EGC*, Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A. and Bubak, M. (Eds.), Lecture Notes in Computer Science, Vol.3470, pp.144–153, Springer (2005).
4) Heine, F., Hovestadt, M. and Kao, O.: Towards Ontology-Driven P2P Grid Resource Discovery, *GRID*, Buyya, R. (Ed.), pp.76–83, IEEE Computer Society (2004).
5) Tangmunarunkit, H., Decker, S. and Kesselman, C.: Ontology-Based Resource Matching in the Grid — The Grid Meets the Semantic Web, *International Semantic Web Conference*, Fensel, D., Sycara, K.P. and Mylopoulos, J. (Eds.), Lecture Notes in Computer Science, Vol.2870, pp.706–721, Springer (2003).
6) Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmér, M. and Risch, T.: EDUTELLA: A P2P networking infrastructure based on RDF, *WWW*, pp.604–615 (2002).
7) Ripeanu, M., Iamnitchi, A. and Foster, I.T.: Mapping the Gnutella Network, *IEEE Internet Computing*, Vol.6, No.1, pp.50–57 (2002).
8) Koubarakis, M., Kaoudi, Z., Miliaraki, I., Magiridou, M. and Papadakis-Pesaresi, A.: Semantic Grid Resource Discovery using DHTs in Atlas, *3rd GGF Semantic Grid Workshop, Co-located with GGF16*, Athens, Greece, February

13–16, 2006 (2006).
9) Cai, M. and Frank, M.R.: RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network, *WWW*, Feldman, S.I., Uretsky, M., Najork, M. and Wills, C.E. (Eds.), pp.650–657, ACM (2004).
10) Cai, M., Frank, M.R., Yan, B. and MacGregor, R.M.: A subscribable peer-to-peer RDF repository for distributed metadata management, *J. Web Sem.*, Vol.2, No.2, pp.109–130 (2004).
11) Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M.T., Brunkhorst, I. and Löser, A.: Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks, *Proc. 12th International World Wide Web Conference*, WWW2003, Budapest, Hungary, 20–24 May 2003, pp.536–543, ACM (2003).
12) Sidirourgos, L., Kokkinidis, G. and Dalamagas, T.: Efficient Query Routing in RDF/S schema-based P2P Systems, *4th Hellenic Data Management Symposium* (*HDMS'05*) (2005).
13) Boncz, P.A. and Treijtel, C.: AmbientDB: Relational Query Processing in a P2P Network, *DBISP2P*, Aberer, K., Kalogeraki, V. and Koubarakis, M. (Eds.), Lecture Notes in Computer Science, Vol.2944, pp.153–168, Springer (2003).
14) 的野晃整, サイドミルザパレビ, 小島 功：P2P 環境における RDF データのための三次元索引に基づいた検索，データベースと Web 情報システムに関するシンポジウム（DBWeb2005）論文集, 情報処理学会シンポジウムシリーズ，Vol.2005, No.16, pp.9–16,（社）情報処理学会 (2005).
15) Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F. and Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications, *SIGCOMM*, pp.149–160 (2001).
16) Cai, M., Frank, M.R., Chen, J. and Szekely, P.A.: MAAN: A Multi-Attribute Addressable Network for Grid Information Services, *GRID*, Stockinger, H. (Ed.), pp.184–191, IEEE Computer Society (2003).
17) 首藤一幸，田中良夫，関口智嗣：オーバレイ構築ツールキット Overlay Weaver，第 9 回プログラミングおよび応用のシステムに関するワークショップ（SPA2006）(2006).

(平成 17 年 12 月 21 日受付)
(平成 18 年 4 月 9 日採録)

(担当編集委員 浦本 直彦)

的野 晃整（正会員）

2000 年岡山県立大学情報通信工学科卒業．2002 年同大学大学院情報系工学研究科博士前期課程修了．2005 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（工学）．同年産業技術総合研究所グリッド研究センター特別研究員，現在に至る．RDF データ検索の研究に従事．電子情報通信学会，ACM，日本データベース学会各会員．

Said Mirza Pahlevi

1992 年山形大学工学部電子情報工学科卒業．1996 年山形大学大学院工学研究科博士前期課程修了．2003 年筑波大学大学院工学研究科電子・情報工学専攻博士課程修了．博士（工学）．同年産業技術総合研究所グリッド研究センター特別研究員，現在に至る．データグリッドおよびセマンティックグリッドの研究に従事．ACM 会員．

小島 功（正会員）

1958 年生まれ．1984 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年電子技術総合研究所入所．現在，産業技術総合研究所グリッド研究センターデータグリッドチームチーム長．データグリッドの研究に従事．ACM 会員．GGF（Global Grid Forum）および OASIS メンバ．