SIG-SWO-040-06

FROSTにおけるデータストアの圧縮と読み込み手法

Compressing and Loading RDF Data in FROST

香川 俊幸[1*]　兼岩 憲[2]
Toshiyuki Kagawa[1]　Ken Kaneiwa[2]

[1] 電気通信大学大学院 情報理工学研究科 情報・通信工学専攻
[1] Department of Communication Engineering and Informatics, Graduate School of Informatics and Engineering, The University of Electro-Communications
[2] 電気通信大学大学院 情報理工学研究科 情報・ネットワーク工学専攻
[2] Department of Computer and Network Engineering, Graduate School of Informatics and Engineering, The University of Electro-Communications

概要： FROST は RDF データを効率的に処理するためのインメモリで動作する RDF ストアである．メモリは容量が小さいことから，膨大な RDF データに対して検索を行うためには，より効率的なデータの格納が求められる．本研究では，FROST のデータストアに対する圧縮手法と，膨大な RDF データに対応する読み込み手法を提案する．FROST のデータ構造に対し，効率的な ID 付けや冗長な表現を短縮することで，より効率的な圧縮データ構造となる．また，データを分割して読み込み，冗長なデータを同時に保持しないことで，より膨大な RDF データを読み込むことができる．

1　はじめに

セマンティック Web は，リソースに意味を付与することで，WWW の利便性の向上を目指す試みである．RDF（Resource Description Framework）[6] は，セマンティック Web を実現するための技術的な構成要素の 1 つであり，リソース間の関係を機械可読にするための枠組みである．RDF データの利用は近年盛んになっており，Web 上には膨大なデータが存在する．例えば，DBpediaJapanese[2] は約 17GB，DBpedia[1] は約 220GB の RDF データである．この膨大な RDF データに対し，省スペースかつ高速な検索の実現が求められている．

藤原ら [3][4] は，インメモリで動作する RDF ストアである FROST を提案した．FROST は，クエリ実行計画による問い合わせ順序の最適化と，効率的な圧縮と検索性能を両立したデータ構造によって省メモリかつ高速な検索を実現した．これにより，実装メモリサイズ以上の RDF データに対して検索が可能となった．しかし，一般的にメモリ容量はディスク容量より少ないため，インメモリで動作する FROST は全ての膨大な RDF データを読み込んで検索できるわけではない．

本研究では，RDF データの特徴を用いた RDF データの圧縮手法と大規模 RDF データを読み込むための手法を提案する．効率的な ID 付け，型情報の圧縮や RDF グラフの冗長性を削除することによって，従来の FROST におけるデータ構造よりも圧縮効率の良い構造を実現する．また，RDF データを分割して読み込むことにより，現状では読み込み不可能であったデータに対して検索可能になる．これにより，提案手法を FROST に導入し，比較実験による性能評価を行う．

本稿の構成は，以下の通りである．2 章では，本研究で使用する RDF と FROST について述べる．3 章では，より効率的な格納が可能であるデータ構造と大規模な RDF データの読み込みを想定した読み込み方法について述べる．4 章では，提案手法の性能評価のため提案手法を FROST に導入し，ベンチマークである LUBM を用いた実験結果を示す．最後に，5 章では，本稿の結論と今後の課題について述べる．

2　準備

2.1　RDF

RDF は URI（Uniform Resource Identifier）[7] によってリソース間の関係を記述するための枠組みである．RDF では，主語（*subject*），述語（*predicate*），目的語（*object*）の 3 つ組によって情報を記述している．この 3 つ組を RDF トリプルと呼ぶ．以下本稿では

*連絡先： 電気通信大学大学院 情報理工学研究科 情報・通信工学専攻
〒 182-8585 東京都調布市調布ヶ丘 1-5-1
E-mail: kagawa@sw.cei.uec.ac.jp

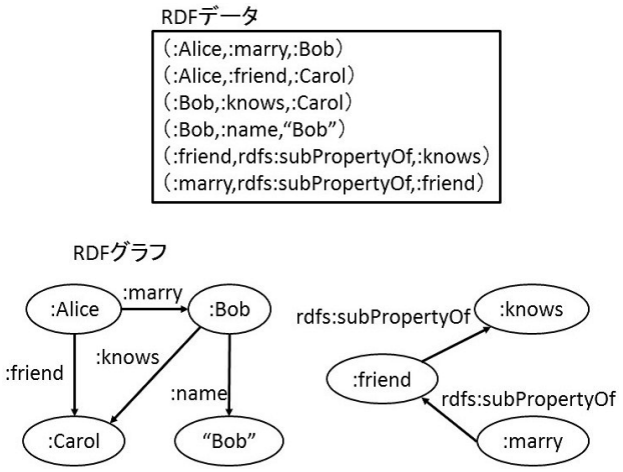

図 1: RDF データと RDF グラフ

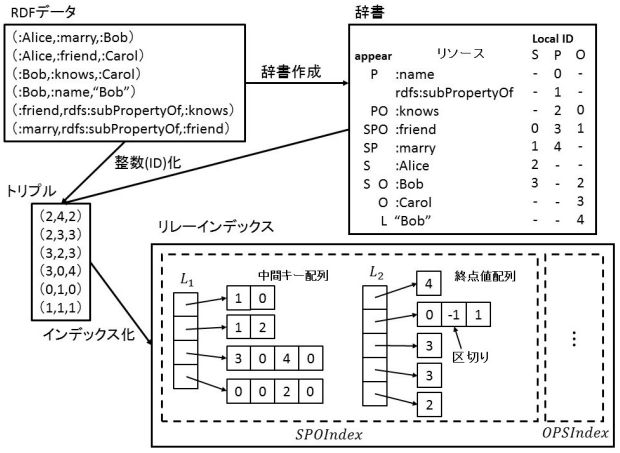

図 2: FROST のデータ構造

RDF トリプルを（s,p,o）と表記する．RDF は，URI 参照の集合，空ノードの集合，およびリテラルの集合から構成される．これらの集合をそれぞれ，URI 参照の集合 U，空ノードの集合 B，およびリテラルの集合 L とすると，RDF トリプルの各要素は，$s \in U \cup B$，$p \in U$，$o \in U \cup B \cup L$ と表される．即ち，(s,p,o) は，$(U \cup B) \times U \times (U \cup B \cup L)$ の要素である．RDF トリプルの集合は $\{(s_1, p_1, o_1), \cdots, (s_n, p_n, o_n)\}$ のように表され，主語と目的語を接点（ノード），述語を有向辺（エッジ）とした RDF グラフ G を構築することが可能である．図 1 に RDF データと RDF グラフの具体例を示す．

2.2 FROST

FROST[3][4] は，大規模 RDF データを高速に処理するためにインメモリで動作する RDF ストアである．効率的なクエリ実行計画の生成と，圧縮データ構造の提案によってこれを実現している．クエリ実行計画では，解決数をより少なく抑え込むことが可能なクエリとなるような順序にトリプルパターンを並べ替える．これによって，部分的なクエリの処理数を削減をしている．大規模なデータをメモリ上で処理するためには，データの圧縮が必要不可欠である．RDF データはリソースが文字列で記述され，かつ同一の文字列が複数回出現するため，冗長性が高い．FROST ではリソースの文字列を整数の ID に対応付けして，RDF データの冗長性を排除する．ID とリソースの対応を記すデータである辞書と，整数化された RDF トリプルに分け，それらに対して圧縮を行うことで省メモリ性を実現している．FROST のデータ構造概要を図 2 に示す．

2.3 辞書

RDF データは文字列情報を多く含んでいるため，省メモリかつ高い検索性を持つデータ構造の辞書が求められる．FROST は，Front Coding を用いて辞書を圧縮することにより，辞書の省メモリ性を実現している．Front Coding とは，1 つ前の文字列と比較した際に，共通する接頭辞を省略する手法である．前方一致の文字列が連続することで大きな効果を得られるため，辞書の文字列は昇順にソートする．Front Coding 後のデータは，共通部分の長さ，非省略文字列の長さ，非省略文字列で表される．このうち，共通部分の長さと非省略文字列の長さを可変長整数によって表現することで，より小さい表現となる．

RDF トリプルにおいて，リソースが主語，述語，目的語のいずれかに出現するとき，それぞれの役割を持つという．FROST では，データの利便性を向上させるために役割によって ID 空間の分割を行っている．リソースを SPO，SP，PO，SO，S，P，O，Literal の 8 つの役割をもつリソースの集合に分割し，各集合をできるだけ同じ役割の集合が隣接するように並べ，その順にグローバル ID を振る．このグローバル ID と各集合の要素数から，主語，述語，目的語のそれぞれのローカル ID の計算ができる．後述するリレーインデックスの ID をローカル ID とすることによって，インデックスの効率化が可能である．

2.4 リレーインデックス

リレーインデックスとは，データ圧縮と検索高速化を目的としたデータ構造である．ID 化された RDF トリプルは全てリレーインデックスとして保持される．FROST では SPO インデックス，OPS インデックスと呼ばれる 2 つのリレーインデックスを持つ．SPO イ

ンデックスでは，主語 s，述語 p，目的語 o の順でリソースを検索する．同様に OPS インデックスでは，o，p，s の順に検索を行う．トリプルの各要素を辿る順に起点キー，中間キー，終点値と呼ぶ．

リレーインデックスは，2 つのポインタ配列 L_1 と L_2 によって構成される．L_2 には中間キーごとに終点値配列へのポインタが格納される．$L_2[k]$ は中間キー k に対する終点値配列を表し，起点キーごとの終点値が連続して格納される．L_1 には起点キーごとに中間キー配列へのポインタが格納される．$L_1[k]$ は起点キー k に対する中間キー配列を表し，ある中間キー k_i と終点値配列 $L_2[k_i]$ の位置 n_i が交互に格納される．

インデックスの圧縮のため，中間キー配列は固定長整数を用いて圧縮を行う．これにより，圧縮を実現しながらも，二分探索が可能であるため，検索性も兼ね備えている．さらに，終点値配列は区切りで示された塊ごとに終点値が昇順に格納されている．終点値に対し塊ごとに差分圧縮を行い，更に可変長整数を用いて表現することで，省メモリなインデックスとなる．

3 RDF データの圧縮と読み込み

3.1 ID の局部最適化

RDF データはリソースの文字列に対し，整数の ID を付けることでトリプルを短く表現できる．RDF データに出現するリソースには頻繁に出現するものがある．頻出するリソースに小さい ID を振ることでトリプルの表現を小さくすることができるが，大きい ID を振ってしまうと表現が冗長になってしまう．どのリソースにどの ID を付けるのかといった問題は主に文字列の圧縮手法に依存することが多い．例えば FROST では，Front Coding の圧縮効果を引き出すためにアルファベット昇順に ID が付けられる．

トリプルを保持する際の表現を考えた場合，最も効率の良い ID の付け方はリソースの出現回数順（降順）であることが自明である．しかし，Front Coding による圧縮効率の良い並び方と効率の良い ID 化のための出現回数順は必ずしも一致しない．圧縮効率を保ったまま，2 通りの異なる順序を同時に実現することは難しい．そこで，ID の局部最適化を提案する．RDF データに頻繁に出現するものに対してのみ，小さい ID をつける．索引を使用することで，異なる順序の同居を実現させる．一部のみの索引で充分であるため，大きな容量を使うことなく，トリプルの表現を短くできる．

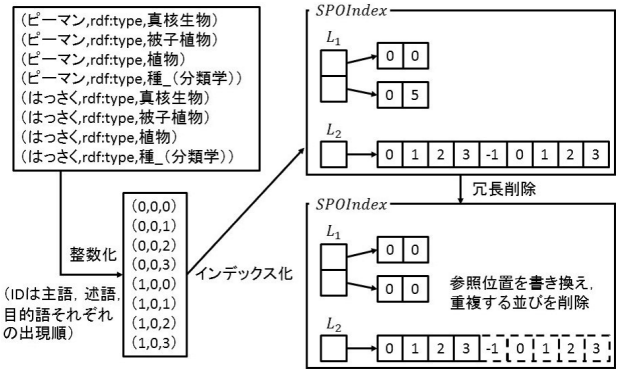

図 3: 同型 RDF グラフの圧縮

3.2 型情報置換

リソースの中には数値を示すものがある．リテラル値として数値を示しているが，それ単体では単位等がわからないため正確な情報を取得できない．この問題を解決するために，型情報 URI はリテラル値の後ろに記述される．

型情報は文字列の後方に出現することから前方一致の文字列を省略する Front Coding では圧縮されない．そのため，同じ文字列が出現するにも関わらず，圧縮されずに残っている．型情報を他の短い文字に置換することによって，短い表現となる．置換後の文字列は「”＋型情報 ID」とし，この「”」によって型情報であることを判定する．また，型情報の中で使用頻度の高いデータ型はあらかじめ基本データ型として用意されている．基本データ型の URI は同様の接頭辞である．接頭辞を ID に置き換えることでより小さい容量で保持できる．

3.3 同型 RDF グラフの圧縮

RDF データには同型の RDF グラフが存在する．本来同じカテゴリに属する要素は，それぞれ同様のトリプルを有するからである．同型の RDF グラフを表すトリプル構造は同型であることから，同じ数値データが保持される．この冗長表現を削除することによって，より小さい容量で保持できる．

同型の RDF グラフが存在するとき，FROST では終点値配列に同様の数値の並びが存在する．これを省略するために，同様の並びを全て 1 つにまとめる．中間キー配列に格納されている終点値配列へのアクセス位置をまとめられた並びとする．これにより，冗長な表現を削除できる．この削除によって，データ構造を変化させるが，参照する際の値が変化しないため，クエリ解決時間は増加しない．SPO インデックスの圧縮を図 3 に示す．OPS インデックスも同様に圧縮する．

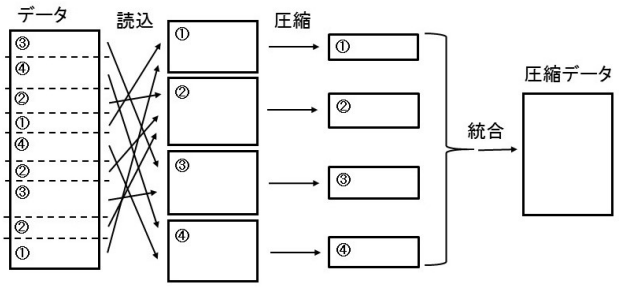

図 4: 分割読み込み概要

3.4 分割読み込み

大規模な RDF データを読み込むためには，圧縮データ構造を作成する際の最大使用メモリ量が関係する．RDF データの圧縮率が高くても，最大使用メモリ量が高ければそもそもデータを読み込めない．RDF データを読み込み，データ構造を構築する際に最も使用メモリ量が高い時点は圧縮をかける前のデータを全て保持している時点だと考えられる．例えば，FROST ではリソースに対し，Front Coding を行う前の文字列を保持している時点や，RDF トリプルをリレーインデックス構造に変換する前の時点だと考えられる．

読み込み時の使用メモリ量を減少させる方法として，分割読み込みを提案する．読み込むデータを複数に分割して読み込むことにより，冗長なデータを同時に保持することを避ける．さらに，各読み込み終了後のデータに対して圧縮を適用することで，最大使用メモリ量を減少させることができる．分割読み込みの概要を図 4 に示す．各回に読み込むデータは閾値を設けることで，どの回にどのデータを読み込むかを判定する．FROST にこの手法を導入する場合，単純にデータの行数でデータを分割してしまうと辞書データが正しい順序にならない等の問題がある．

3.4.1 辞書分割読み込み

辞書を構築する前準備として，全てのリソースをそのままの文字列で保持する必要がある．これは，Front Coding の効果を最大限発揮するために，昇順にソートするからである．辞書構築において，この部分が瞬間的なメモリ使用量が最も高いと予想される．よって，この部分に対して分割読み込みの手法を適用する．

辞書内のリソースは昇順ソートするため，読み込む RDF データをリソースの辞書的な順番を基準に分割を行う．各回の読み込みにおいて，どのリソースを読み込むかを判定する閾値となるリソースを，サンプリングによって決定する．RDF データの先頭から読み込みを行い，分割数に応じた一定数以上のリソースを読み込んだ段階でこれをサンプルとする．分割数を d，昇順にソートされたサンプルリソースを r'，サンプルリソースの総数を N とすると，閾値となるリソースは $r'_{N/d}, r'_{2N/d}, \cdots, r'_{(d-1)N/d}$ である．つまり，i 回目の読み込みでは，$r'_{(i-1)N/d} \leq r < r'_{iN/d}$（この不等号は文字列の辞書式順序を表す）を満たすリソース r のみを読み込む．

このような分割を行うことによって，各読み込みでのリソースは全体のリソースを昇順ソートした順番と完全に一致する．そのため，Front Coding を行った際のデータも一致するため，分割読み込みによって圧縮率が低下することはない．

3.4.2 トリプル分割読み込み

辞書と同様に，リレーインデックスを構築する前準備として，ID 化されている全てのトリプルを保持する．この部分のメモリ使用量が最も高いと予想される．リレーインデックスを効率良く構築するためには，中間キー配列に対して二分探索を用いることから，中間キー（SPO インデックス，OPS インデックスにおける述語 P）の昇順に構築していく必要がある．

2 つのリレーインデックスは共に述語の昇順に構築することから，述語の順番を基準に分割を行う．各回の読み込みにおいて，RDF トリプルの述語がどの述語ならば読み込むのかを判定する閾値となる述語 ID を決定する．トリプルを読み込む段階で，辞書は完成しているため，データ内の述語の ID の最大値 p_{max} が判明している．そのため，閾値の述語 ID は $p_{max}/d, 2p_{max}/d, \cdots, (d-1)p_{max}/d$ である．i 回目の読み込みでは，述語 ID が p とすると，$(i-1)p_{max}/d \leq p < ip_{max}/d$（この不等号は自然数の大小を表す）の RDF トリプルのみを読み込む．

中間キー配列は固定長整数によって二分探索による探索を実現している．固定長整数はランダムアクセスが可能だが，圧縮率が低下するという欠点がある．そのため，一度可変長整数を用いて圧縮を行う．これにより，分割読み込みの効果が得られ，最大使用メモリ量が減少する．

3.4.3 分割読み込みによる読み込み速度

分割読み込みでは，分割数の回数 RDF データを読み込むこととなる．通常の読み込みに比べ，ファイルの参照回数が増えることや，閾値による判定を行うことから，読み込みにかかる時間が増加する．

読み込み時間増加への対策として，圧縮後のデータをディスクへ書き出し，次回以降の読み込み時間を短縮する．次回以降の読み込み時では既に圧縮されたデータを高速に読み込める．

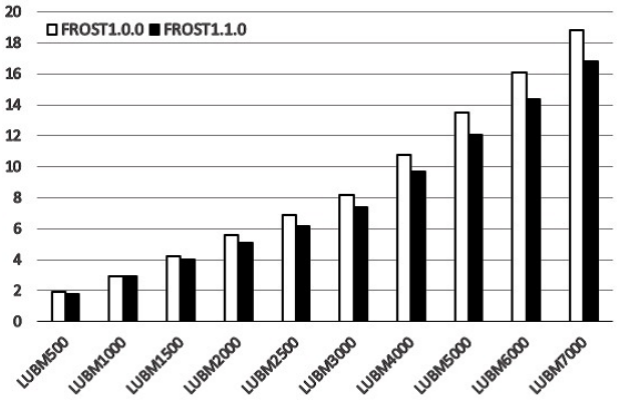

図 5: メモリ使用量 (GB) の比較

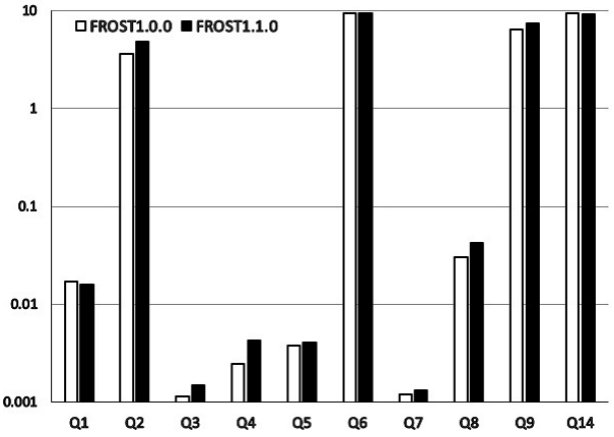

図 6: クエリ解決時間 (s) の比較

4 実験

Lehigh University Benchmark（LUBM）[5] を用いた性能比較実験を行う．LUBM は，大学数をパラメータとして生成される RDF データであり，入力パラメータによる様々なデータサイズのデータを柔軟に生成することができる．実験環境は OS：Windows 8.1 Enterprise(64 bit)，CPU：Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20 GHz(16 CPUs)，実装メモリ：128 GB である．以下比較対象である FROST1.0.0 に対し，Java を用いて本研究の手法を導入したものを FROST1.1.0 と表記する．

4.1 圧縮手法を用いたメモリ使用量の比較

圧縮手法の効果を比較するために，LUBM500〜LUBM3000，LUBM4000〜LUBM7000（LUBM の後の数値はパラメータである大学数）の 10 個のデータを用いて読み込みを行った．各データのサイズはパラメータの大きさに依存する．FROST1.1.0 の読み込み方法は FROST1.0.0 と同様である．

読み込み後のメモリ使用量の結果を図 5 に示す．最も大きいもので FROST1.0.0 に比べ，約 2GB の圧縮に成功している．データサイズが小さいものは差が出ていないが，大きくなるにつれて差が大きくなっている．これは，データサイズが大きくなるにつれて同型の RDF グラフが増加するため，冗長削除を行った際の減少率が増加していくためだと考えられる．

4.2 クエリ実行時間の比較

圧縮手法を適用したことによるクエリ実行時間の変化を観測するために，LUBM1000 に対するクエリ解決時間の測定を行った．LUBM では全部で 14 個のクエリ Q_1〜Q_{14} が提供されている．Q_{10} はクラス間の暗黙的な推論が必要であり，Q_{11}〜Q_{13} までは OWL による推論が必要となっている．FROST では，RDFS 推論のみの実装となっているため，Q_{10}〜Q_{13} に必要な推論には対応していない．よって，Q_{10}〜Q_{13} を除いた Q_1〜Q_9, Q_{14} のみを用いる．

クエリ解決時間の結果を図 6 に示す．測定は 3 回実行した結果の平均をとった．なお，クエリごとに実行速度が異なるため，対数グラフを用いている．本研究の手法を導入しても，FROST1.0.0 に比べ，クエリ解決速度が大きく変化することはなかった．解決速度が減少しているクエリが存在するが，これは誤差によるものだと考えている．クエリ解決時間には，リソースの文字列を復元する時間が含まれるため，圧縮手法を導入した分復元も必要となる．そのため，わずかであっても時間は増加するはずである．

4.3 分割読み込みによる読み込み実験

分割読み込みの効果を検証するために，4.1 節よりも大規模な RDF データ（LUBM8000〜LUBM10000, LUBM12000〜LUBM16000 の計 6 個）を用いて読み込みを行った．分割読み込みにおける分割数は，辞書，トリプル共に 4 分割に統一し，3.1〜3.3 節の圧縮手法は適用していない．

ディスク上の RDF データサイズと読み込み後のメモリ使用量を図 7 に示す．メモリ不足のため，FROST1.0.0 では LUBM9000 までしか読み込むことができなかった．LUBM9000 のデータサイズは 201.2GB であり，実装メモリサイズの約 1.6 倍のデータである．それと比較し，分割読み込みでは，LUBM16000 まで読み込むことが可能であった．LUBM16000 のデータサイズは 358.9GB であり，実装メモリサイズの約 2.8 倍のデータである．FROST1.0.0 と比較して約 1.8 倍までのデータサイズのデータを読み込める．

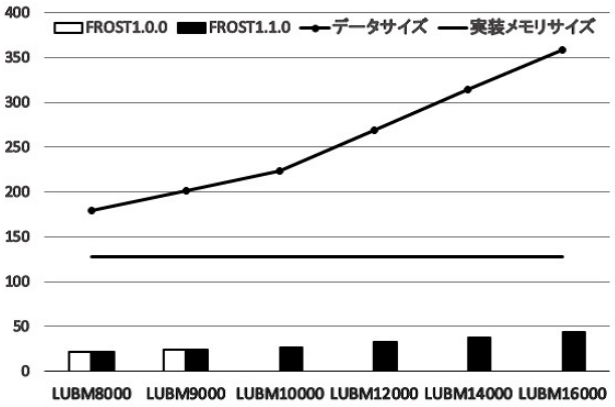

図 7: LUBM のデータサイズ (GB) と分割読み込みによる読み込み後のメモリ使用量 (GB) の比較

5 まとめ

本研究では，FROST における圧縮データ構造に対し，さらに冗長性の排除等による効率的な格納方法を提案した．また，大規模な RDF データの読み込みを想定し，データを分割して読み込む手法の提案を行った．FROST との比較実験により，提案手法がより効率よくデータを格納していること，従来では読み込むことができなかった RDF データの読み込みが可能であることを示した．

今後の課題として，圧縮率の更なる向上が挙げられる．リソースに対する主な圧縮手法は Front Coding であり，リテラルは共通の接頭辞が少ないことから圧縮されずに多くの部分が残っていると考えられる．このリテラルに対して効果的な圧縮を行うことにより，今まで以上に小さなデータ構造が期待できる．

参考文献

[1] DBpedia, http://dbpedia.org/

[2] DBpediaJapanese, http://ja.dbpedia.org/

[3] 藤原 浩司, 兼岩 憲 : 大規模 RDF グラフに対する高速検索とデータ圧縮の両立, セマンティック Web とオントロジー研究会, SIG-SWO-A1402-08 (2014)

[4] FROST : SPARQL 検索エンジン, http://www.sw.cei.uec.ac.jp/frost/index-j.html

[5] Y. Guo, Z. Pan, and J. Heflin : LUBM: A Benchmark for OWL Knowledge Base Systems, *Journal of Web Semantics*, Vol. 3, pp. 158–182 (2005)

[6] 兼岩 憲 : RDF と RDF スキーマの推論, 人工知能学会誌, Vol. 26, No. 5, pp. 473–481 (2011)

[7] T. Berners-Lee, R. Fielding, and L. Masinter : Uniform Resource Identifier(URI):Generic Syntax, Technical report, Network Working Group(2005), http://tools.ietf.org/html/rfc3986