Spark SQLによる効率的な問合せ処理のためのワークロードに基づくRDFデータ分割手法

山崎　昂輔†　　天笠　俊之††

† 筑波大学院 理工情報生命学術院 システム情報工学研究群 情報理工学位プログラム　〒305-8573 茨城県つくば市天王台1丁目1-1

†† 筑波大学 計算科学研究センター　〒305–8577 茨城県つくば市天王台1丁目1-1

E-mail: †kosuke.y@kde.cs.tsukuba.ac.jp, ††amagasa@cs.tsukuba.ac.jp

あらまし　近年，機械可読なデータとしてRDF（Resource Description Framework）が注目されている．RDFは主語，述語，目的語のトリプルの組み合わせで多くのデータの表現を可能とする単純で柔軟なフレームワークであり，さまざまな領域でRDFデータが生成され，そのデータ量は急速に増加している．膨大なRDFデータを効率的に処理するために多くの研究が行われており，その中でも並列分散処理による高速化が期待されている．Apache Sparkは，インメモリでデータを処理することで高速化を実現するフレームワークであり，Spark SQLを用いることでSQLを用いたRDFデータの処理が可能となる． 本研究では，ワークロードの共起性を用いたデータ分割手法と，Spark SQLを用いたRDFデータの効率的な問合せ処理を提案する．ワークロード情報を活用することにより問合せ時の関連が強いデータを同一パーティションに配置することが期待でき，処理対象のデータ量の削減を実現できる．実行時間の比較では，多くのクエリで従来手法を上回る結果を示した．

キーワード　RDF, パーティショニング, 分散処理, Apache Spark

1　はじめに

RDF（Resource Description Framework）[1]は，主語，述語，目的語からなるトリプルと呼ばれる基本構造を持ち，これによってデータ間の関係を簡潔に表現できるフレームワークである．このフレームワークはデータへのセマンティクスの付与やグラフ表現が可能なこと，機械可読であるなどの特徴があり，その有用性から近年RDF形式のデータが爆発的に増加している．RDFデータは基本的にSPARQL（SPARQL Protocol and RDF Query Language）という標準的な問合せ言語を用いてデータの問合せを行う．SPARQLでは，問合せたいデータをトリプル構造に基づいて記述することで問合せを実行することができる．問合せにかかるコストはデータ量の増加に伴って大きくなるが，一般に大規模なRDFデータは数百万から数億のトリプルを含んでいる．従って，そのようなデータに対していかに効率的で高速な問合せ処理を実現するかが昨今の課題となっている．

この課題を解決するために，大きく分けて集中型システムと分散型システムの2種類が多数発表されている．集中型システムは，データの保管や問合せ処理を1つのマシンで管理するものであり，Jena [2] やRDF-3X [3] などが挙げられる．一般に集中型システムは通信コストが低いという利点がある一方で，処理効率は単一マシンの性能に依存するためスケーラビリティが低く大規模データに対応することが難しい．一方，分散型システムは，データの処理を複数のマシンで行うため通信のオーバーヘッドが発生するという欠点を持つものの集中型システムの課題であった大規模データへの効率的な処理が可能であり，この特徴に注目した研究が昨今多く検討されている [4], [5].

Apache Spark [6] は大規模並列分散処理を行うフレームワークの1つであり，インメモリデータ構造を使用したデータ処理が可能であることから，分散型システムに活用した研究が増えてきている．SPARQLGX [7]，WORQ [8]，DIAERESIS [9] などがその例であり，これらはデータへの問合せ処理高速化に向けたデータ分割手法を提案している．

データ分割は分散並列処理技術を用いた処理における重要な課題の一つであり，そのデータレイアウトは問合せ時に必要な情報を取得するためにアクセスすべきデータ量を大きく左右する．単純な，あるいはランダムなデータ分割では問合せ時に不要なデータへのアクセスや探索が発生し，それに応じて処理コストが増加し問合せ応答の性能低下へとつながる．

ところで，多くのシステムではユーザが発行する典型的な問合せをワークロード情報として取得可能である．この情報を活用することで，問合せ処理を行う上で関連性の高いデータを同じパーティションに配置でき，データアクセスの抑制とそれによる性能の向上が実現できると期待される．

そこで本研究では，ワークロード中の主語，述語，目的語それぞれの共起性を利用したデータ分割とデータのインデックスを利用した効率的な問合せ処理を実現する手法を提案する．実際のワークロードを基にした共起性の導入により，同時に問合せられるデータを同一のパーティションにまとめられると期待できる．さらに，インデックスを用いることで問合せ時に必要なパーティションのみを選択して読み込むことを可能にし，処理するデータ量を抑制できると考えられる．

我々の提案手法の優位性を確かめるために，評価実験ではデータ分割時間，問合せ実行時間，問合せ時のデータ読み込み時間，読み込んだトリプル数，データ読み込み時間と問合せ実行時間の合計時間の 5 つの観点で既存手法と比較する．

本論文の構成は以下のとおりである．2 章では本研究の前提知識を述べ，3 章で関連研究を紹介する．4 章で提案手法を説明し，5 章で評価実験の結果を示す．最後に 6 章で本研究のまとめと今後の展望を述べる．

2 前提知識

本章では，本研究の基本事項を紹介する．まず 2.1 節で，本研究で扱うデータの対象である RDF について触れ，続く 2.2 節で RDF への標準的な問合せ言語である SPARQL に焦点を当てる．最後に，2.3 節で本研究で並列分散処理のために用いる Apache Spark について説明する．

2.1 RDF（Resource Description Framework）

2.1.1 RDF の基本知識

RDF（Resource Description Framework）は，データを主語 s，述語 p，目的語 o のトリプルと呼ばれる形式を基本パターンとして表現するフレームワークであり，トリプルを組み合わせることで複雑なデータの関係性を簡潔に表現することができる．各データは IRI（International Resource Identifier）を用いて示されており，あらゆる資源を一意に識別し，複数のデータベースにまたがるデータを統合することも可能である．また，リテラルやブランクノードも重要である．リテラルは文字列や数などの具体的なデータ値をそのまま扱うものであり，目的語としてのみ使用可能である．ブランクノードは IRI を持たない一時的なエンティティの表現などに使用されるものであり，主語と目的語としてのみ使用可能である．

以下はトリプルの例である．ただし，⟨_:Brank⟩ は空白ノード，⟨4.20000⟩ は数のリテラルによる表現である．

⟨dbr:University_of_Tsukuba⟩⟨dbo:city⟩⟨dbr:Tsukuba⟩

⟨dbr:Tsukuba⟩⟨db:Temperature⟩⟨_:Brank⟩

⟨_:Brank⟩⟨dbp:febMeanC⟩⟨4.200000⟩

2.1.2 RDF の視覚化

さらに，RDF には視覚化やそれを組み合わせたグラフ表現が可能であるという特徴もある．主語と目的語をノード，述語を有向エッジとすることで視覚化でき，それらを複数組み合わせることでグラフが形成できる．これは複雑なデータ間のネットワークの直感的理解や，グラフアプローチによる処理などに役立つ．図 1 はトリプルの基本パターンを視覚化したものであり，筑波大学がつくば市にあることを示している．そして図 **??** は複数の基本パターンを組み合わせてグラフにしたものである．なお，ここでは楕円が IRI ノード，長方形がリテラル，菱形がブランクノードを表している．

2.1.3 RDF の基本構文

RDF データの表現には，Turtle（Terse RDF Triple Language）や N-Triples などの構文が一般に用いられる．Turtle は，RDF データを簡潔かつ読みやすい形式で書くための構文であり，主に人間が読み書きするシナリオに適している．さらに，図 1 や図 2 で用いているように dbr や dbo などの接頭辞を使用して IRI を短縮し，データの表現を簡略化することが可能である．一方，N-Triples はよりシンプルな形式で，各トリプルを一行で表現する．これは機械による解析などに適しており，その単純さから大規模な RDF データの処理に用いられることも多い．本研究でも使用する RDF データは N-Triples 形式のものである．

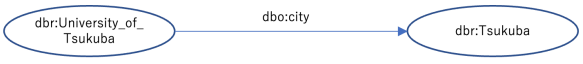

図 1: トリプルの視覚化の例

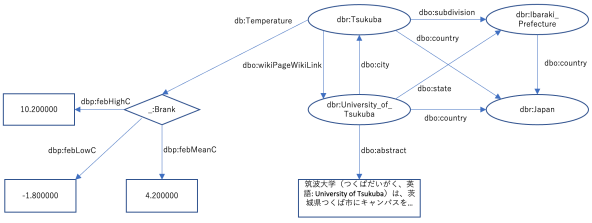

図 2: RDF のグラフ表現の例

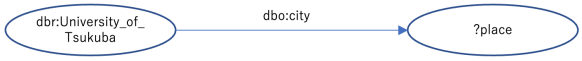

図 3: SPARQL クエリのグラフ表現の例

2.2 SPARQL（SPARQL Protocol and RDF Query Language）

SPARQL（SPARQL Protocol and RDF Query Language）は，RDF データへの標準的な問合せ言語である．基本パターンはリレーショナルデータベースへの標準的な問合せ言語である SQL と類似しており，SELECT 句と WHERE 句から構成される．SELECT 句は，問合せの結果として取得したい情報に対応する変数を列挙し，WHERE 句では問合せたい要素を”?”で始まる変数としたトリプルを記述する．例えば，図 2 の RDF グラフに対して筑波大学がある市町村を問合せたい場合，以下のように記述する．

```
SELECT ?place
WHERE{
  dbr:University_of_Tsukuba dbo:city ?place .
}
```

さらに，SPARQL は RDF と同様にグラフ表現が可能であり，問合せは主にグラフパターンマッチングにより行われる．例示した SPARQL クエリは，図 3 のように視覚化できる．また，これらの問合せ処理は SPARQL Endpoint と呼ばれるインターフェースを介して行われる．

2.3 Apache Spark

2.3.1 Apache Spark の概要

Apache Spark [6] は，大規模データの高速処理を目的とした分散処理フレームワークである．このシステムはスケールアウトアーキテクチャに基づいており，サーバー台数の増加に伴って処理能力を向上させることができる．この特徴により，ビッグデータの急激な増加に対して柔軟かつ効率的に対応することが可能になっている．

Spark 登場前には，Google が開発した GFS（Google File System）[10] と Google MapReduce [11] を基にしたオープンソースである Hadoop [12] が大規模データ処理分野の主流であったが，Hadoop はストレージ管理やデータ処理においていくつかの課題を抱えていた．Spark はこれらの課題を克服し，さらに RAM 上でのデータ処理を行うことで MapReduce よりも高速な処理を実現している．

また，Spark には構造化データや半構造化データの処理を容易にするための豊富なデータアクセスモデルと API が提供されている．これらを利用することで，様々なデータフォーマットを柔軟に扱いながら高速な並列データ処理を実現できる．

2.3.2 DataFrame

DataFrame は Spark で提供されている API の一つであり，大規模構造化データのための表形式データモデルとして分散処理に特化している．各列には名前とデータ型を保持することができ，SQL のような操作を容易にする設計となっている．また，JSON, CSV, Parquet など様々なデータフォーマットとの互換性があり，多くのデータソースからの読み込みが可能である．

2.3.3 Spark SQL

Spark SQL も Spark で提供されている API の一つであり，構造化データの処理に特化している．ユーザは SQL クエリを用いて DataFrame のデータを処理することが可能であり，様々なデータセットに対して統合的な解析操作を実現する．ただし，SPARQL には対応していないため，本研究では SPARQL から SQL への変換処理を行ったのち，DataFrame で読み込んだ RDF データに対して Spark SQL による問合せ処理を実行した．

3 関 連 研 究

本章では，本研究の関連研究を紹介する．3.1 節では，ワークロードを利用したデータ分割を行っている手法を説明し，3.2 節ではワークロードを使用しない手法として媒介中心性とインデックスを利用した手法を説明する．

3.1 ワークロードを用いた手法

ワークロードを利用したデータ分割手法として，Adnan らの研究 [13] がある．この研究では，SPARQL クエリのワークロードから共起している述語をカウントし，貪欲なクラスタリングを適用することで同時に問合せられることが多い述語ペアが同じクラスになるようにデータを分割する．Adnan らは，分割したデータを複数の SPARQL Endpoint 上に一つずつ格納し，フェデレーションエンジンを利用して処理しており，Spark 処理系を対象としている本研究とはこの点で異なっている．

Madkour らの研究 [8] もワークロードに基づいたデータ分割手法を提案している．このアプローチでは，複数クエリで頻繁に繰り返される結合パターンに着目し，ディスク I/O やネットワークシャッフリングのオーバーヘッドを最小化することを目指している．そのために，複数のテーブル間の結合が必要なクエリに対して，ブルームフィルタによる要素の削減を行う．そして，得られたデータに対して主語もしくは目的語に基づいて垂直分割を行い，その結果をキャッシュすることで処理対象のデータ量を抑えている．

Guo らの研究 [14] では，ワークロードクエリのグラフパターンに着目し，各サブグラフから頻出パターンを抽出することでそれに基づいたデータレイアウトを構築している．データ分割は動的に行われ，一定数のワークロードクエリが入力されるたびにデータレイアウトの再構築とそれに伴う移動コストを計算し，それを基に実際に再配置する．なお，この手法も Spark 系を対象とはしていない．

3.2 ワークロードを用いない手法

Georgia らの研究 [9] では，二段階のデータ分割を行う．第一段階では，スキーマグラフに対して媒介中心性を計算し，中心性が高いノードを重要なノードとして特定する．その後，残りのノードに対してスキーマノード同士の依存関係を計算する Dependence を定義し，重要なノードへ割り当てることでデータの分割を行う．第一段階の分割で得られた結果に対して，第二段階では述語ベースで垂直方向のサブパーティショニングを行う．各垂直パーティションには，一つの述語に対する主語と目的語が含まれており，各パーティションのデータサイズを小さくしている．さらに，これらのパーティションに対してクエリ実行時に必要なサブパーティションを直接探せるようにインデックスを生成している．この二段階の分割とインデックスによって，述語が束縛されていないクエリに対してその述語を含む第一段階のパーティションに含まれるデータすべてにアクセスする必要がなくなり，必要なパーティションのみのロードを可能にしている．

4 提 案 手 法

本章では，提案手法について説明する．本研究は大規模な RDF データに対する問合せを高速に行うことを目的としている．これを実現するために，ワークロード中の主語，述語，目的語それぞれに対する共起性を利用した二段階のデータ分割と，それぞれに対するデータへのインデックスを利用する手法を提案する．4.1 節では提案手法の概要を導入し，4.2 節ではデータ分割処理について，4.3 節では問合せ処理についてそれぞれ説明する．

4.1 提案手法の概要

本手法は SPARQL クエリのワークロード情報が与えられることを前提としており，ワークロードクエリ中のトリプルにお

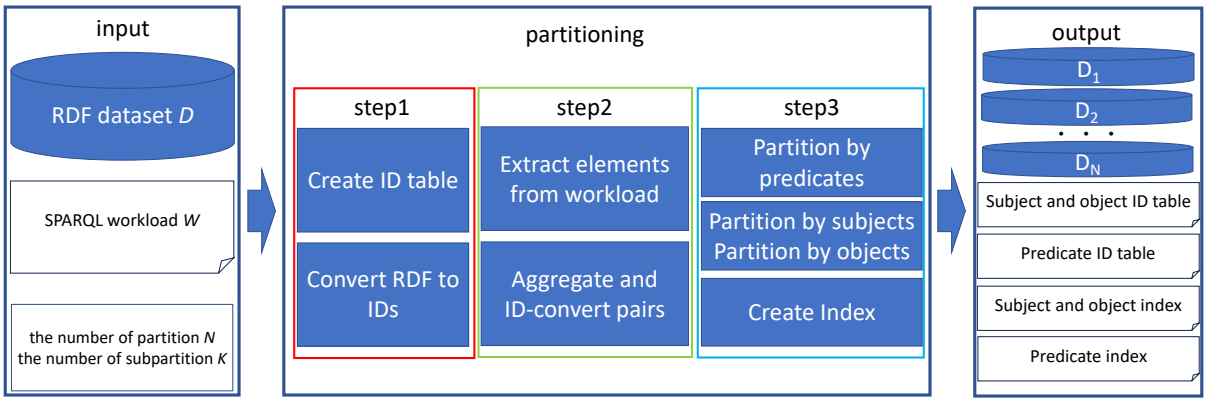

図 4: 提案手法のデータ分割処理の流れ

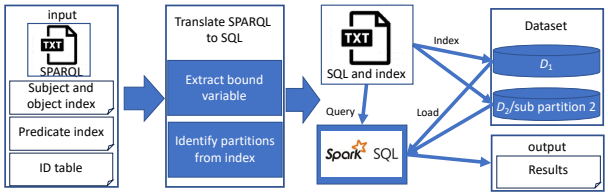

図 5: 提案手法の問合せ処理の流れ

いて共起性の高いトリプルを同一パーティションに配置することで SQL での効率的なクエリ処理を実現するという考えに基づく．図 4 は，提案手法のデータ分割処理の概要を示している．入力は，1) トリプルの集合からなる RDF データ D，2) ワークロードクエリ $W = q_1, \ldots, q_{|W|}$，3) パーティション数 N，およびサブパーティション数 K である．出力は RDF データのパーティション $D_1, \ldots, D_N$ （$D_1 \cap \cdots \cap D_N = \emptyset$），データと ID との関係を記録した ID テーブル，および各データのパーティションへのパスを記録するインデックスである．データ分割処理は大きく三つのステップで構成される．ステップ 1 では，データの冗長性を軽減するための ID 化を行う．ステップ 2 では，ワークロードから主語，述語，目的語をそれぞれ抽出し，共起する要素を ID に変換してペアを作成する．その後，ペアの出現回数を測定し，出現頻度の順にソートする．ステップ 3 では，述語の共起回数を基にしたクラスタリングによる第一段階の分割を行い，その結果に対して第二段階の分割として主語の共起回数に基づくクラスタリングと目的語の共起回数に基づくクラスタリングの二種類を行い，データを分割して出力する．同時に，主語，述語，目的語それぞれのパーティションへのインデックスも出力する．

図 5 は，提案手法の問合せ処理の概要を示している．問合せ処理は大きく二つのステップで構成される．ステップ 1 では，SPARQL の SQL への変換処理を行う．これは，通常 RDF データへの問合せは SPARQL によって行うが，Spark SQL は SPARQL に対応していないためである．また，その際 ID テーブルを利用した束縛変数の ID 化とインデックスを利用したパーティションの特定を同時に行う．ステップ 2 では，実際に SQL による問合せ処理を行う．変換されたクエリを受け取ると，ステップ 1 で特定していたパーティションを読み込み，DataFrame によりテーブルを作成し，それに対して Spark SQL による問合せを行うことで結果を取得する．

Algorithm 1 Workload Analysis.

Require: SPARQL workload W,Subject-object ID table ID_{so}, Predicate ID table ID_p
Ensure: Subject, predicate, and object co-occurrence counts denoted as $C_{sub}, C_{pre}, and C_{obj}$

1: Load ID_{so} and ID_p as HashMap
2: $L_{sub} := \emptyset; L_{pre} := \emptyset; L_{obj} := \emptyset$
3: **for all** $query \leftarrow W$ **do**
4: $where$ = extractInWhere($query$) ▷ Extract contents inside WHERE clause
5: **for all** $(s,t) \leftarrow$ extractBoundSubject($where$) **do** ▷ Create subject pairs from WHERE clause
6: $L_{sub} \leftarrow (\mathrm{ID}_{so}(s), \mathrm{ID}_{so}(t))$
7: **end for**
8: **for all** $(s,t) \leftarrow$ extractBoundPredicate($where$) **do** ▷ Create predicate pairs from WHERE clause
9: $L_{pre} \leftarrow (\mathrm{ID}_p(s), \mathrm{ID}_p(t))$
10: **end for**
11: **for all** $(s,t) \leftarrow$ extractBoundObject($where$) **do** ▷ Create object pairs from WHERE clause
12: $L_{obj} \leftarrow (\mathrm{ID}_{so}(s), \mathrm{ID}_{so}(t))$
13: **end for**
14: **end for**
15: Convert all elements in L_{sub}, L_{pre}, and L_{obj} into ID using HashMap
16: C_{sub} =countAllPairsByMapReduce(L_{sub})
17: C_{pre} =countAllPairsByMapReduce(L_{pre})
18: C_{obj} =countAllPairsByMapReduce(L_{obj})
19: Output C_{sub}, C_{pre}, and C_{obj}

4.2 データ分割処理

4.2.1 Step1: ID の割り当て

RDF データ中の IRI やリテラルは一般に長い文字列であり，問合せ処理における文字列マッチングのコストが高くなる．したがって，これらを事前に唯一性が保証された整数値の ID に変換しておくことで，冗長性を軽減する．

4.2.2 Step2: ワークロードの分析

続いて，ワークロードクエリを分析して主語，述語，目的語それぞれに対して共起しているペアを集計する．入力は SPARQL クエリのワークロード W と，ステップ 1 で作成したデータと ID の対応テーブルであり，出力は各ペアとその共起回数をまとめたテキストである．Algorithm 1 はこのステップのアルゴリズムを示している．まず，与えられたワークロード $W = q_1, \ldots, q_{|W|}$ に対して `WHERE` 句内を抽出し，そこから主語，述語，目的語それぞれの束縛変数をすべて取り出す．各束縛変数を ID に変換し，対応する共起リスト $L_{sub}, L_{pre}, Lobj$ に追加する (2-14 行目)．このリストの要素はタプル $p = \langle P_1, P_2 \rangle$ であり，ここから共起回数を集計する．

例えば，図 6(左) の 4 つのクエリに対して述語の共起回数

```
SELECT * WHERE          SELECT * WHERE          SELECT * WHERE          SELECT * WHERE
{                       {                       {                       {
 ?s p1 o1 .              s1 p1 ?o .              ?s p1 o1 .              ?s p1 ?o .
 ?s p2 o2                ?o p2 ?o2               ?s p3 o3                ?s p2 ?o .
}                       }                       }                       ?s p3 ?o .
                                                                        ?s p4 ?o .
                                                                        }
```

P1	P2	count
p1	p2	3
p1	p3	2
p1	p4	1
p2	p3	1
p2	p4	1
p3	p4	1

図 6: クエリの例 (左) とそれに対する述語の共起回数の例 (右)

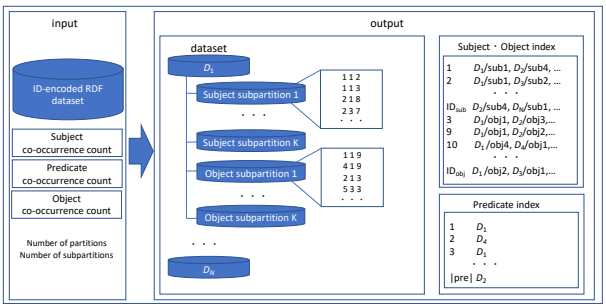

図 7: Step3 のデータ構成

を数える場合，$L_{pre} = \langle\langle p1, p2\rangle, \ldots, \langle p2, p4\rangle, \langle p3, p4\rangle\rangle$ となる．そして，これに対して集計処理を施すことで図 6(左) の結果が得られる．

4.2.3 Step3: データ分割

最後に，ステップ 2 で集計した情報を基にデータセットを分割する．図 7 は，ステップ 3 で入出力されるデータの具体的な構成を示したものである．

提案手法では，データは二段階に分割する．クラスタリングの手法は両段階で同じであり，そのアルゴリズムは Algorithm 2 に示す通りである．このアルゴリズムでは，まず，各クラスタに属する述語を記録する cluster， 述語が属するクラスタを記録する Index の二つのハッシュマップを用意する．ただし，Index は最初は空である．また，述語の種類数をパーティション数 N で割った値を一つのクラスタのサイズとしてあらかじめ計算しておく（1-4 行目）．共起している述語ペアを取り出し，ペアの一方がクラスタに属していない場合はそれをもう一方と同じクラスタに追加した場合のクラスタに含まれる述語数と t を比較する．サイズが t 以下である場合には同じクラスタに追加し，Index を更新する（9-15 行目）．述語ペアの両方がまだクラスタに属していない場合には，クラスターに含まれる述語数が最も小さいクラスタに両方の述語を追加し，Index を更新する（17-22）．最後に，9-15 行目の処理で追加できなかった述語やワークロードに出現していなかった述語を，述語数が最小のクラスタに順に追加していく．

Algorithm 3 にステップ 3 全体のアルゴリズムを示す．Index_s と Index_o は ID をキー，その ID が含まれるパーティションの番号の集合を値とする HashMap である（1-2 行目）．第一段

Algorithm 2 Clustering(D_{ID}, Co-occurrences$elem$, N)

Require: ID-encoded RDF dataset DID, Co-occurrence counts Co-occurrences$_{elem}$, ▷ elem is either subject, predicate, or object Number of partitions N
Ensure: Partitioned datasets $D_1, \ldots, D_N$, Index
1: cluster = $((0, \emptyset), \ldots, (N-1, \emptyset))$: HashMap; ▷ Record elements contained in each cluster
2: Index: HashMap; ▷ Record the cluster to which an element belongs
3: t = |distinct($elem$)|/N ▷ The number of types of elem in one cluster
4: **for all** $(p_1, p_2) \leftarrow$Co-occurrences **do**
5: **if** p_1 in Index AND p_2 in Index **then**
6: continue
7: **end if**
8: **if** One of them already belongs to a cluster **then**
9: Let p_1 already belong to a cluster
10: clusterNo = Index(p_1)
11: **if** |cluster(clusterNo)| + 1 ≤ t **then** ▷ Compare the number of elem types after addition with t
12: cluster(clusterNo) ← p_2
13: Index(p_2) ← clusterNo
14: **end if**
15: **end if**
16: **if** Neither belongs to a cluster **then**
17: clusterNo = cluster with the fewest number of elem
18: cluster(clusterNo) ← p_1, p_2
19: Index(p_1) ← clusterNo
20: Index(p_2) ← clusterNo
21: **end if**
22: **end for**
23: **for all** $p \leftarrow$ remaining elements **do**
24: clusterNo = cluster with the fewest number of elem
25: cluster(clusterNo) ← p
26: Index$_p$(p) ← clusterNo
27: **end for**
28: Output Index, cluster

階では ID 化された RDF データ全体と述語の共起回数，パーティション数を clustering 関数の入力として述語の共起に基づいたクラスタリングを行う（3 行目）．続いて第二段階では，その結果得られたクラスタを入力データセットとして，主語，目的語それぞれに対して clustering 関数を適用してクラスタリングを行う．得られたパーティションは parquet 形式で逐次出力し，Index を更新する（4-15 行目）．すべてのデータの分割が完了すると，Index を出力して処理が終了する．

4.3 問合せ処理

4.3.1 SPARQL から SQL への変換

SPARQL は Spark SQL での処理ができないため，SPARQL が与えられた時，まず SQL への変換を行う．変換処理は以下の流れになる．

1. SPARQL クエリ Q から `WHERE` 句内のトリプルを抽出
2. 各トリプル（$q_1, \ldots q_Q$）に対して，それぞれ次の処理を行う．
   (a) トリプル q_i から束縛変数を抽出する．
   (b) インデックスを基に読み込むべきファイルへのパス

Algorithm 3 Data Partitioning Algorithm.

Require: ID-encoded RDF dataset D_{ID}, Subject co-occurrence counts Co-occurrences$_{sub}$, Predicate co-occurrence counts Co-occurrences$_{pre}$, Object co-occurrence counts Co-occurrences$_{obj}$, Number of partitions N, Number of sub-partitions K
Ensure: N partitions $D_1, \ldots, D_N$,
Subject index Index$_s$, Predicate index Index$_p$, Object index Index$_o$

```
Index_s := ((subject_1, ∅), ..., (subject_|subject|, ∅)): HashMap;
Index_o := ((object_1, ∅), ..., (object_|object|, ∅)): HashMap;
firstPartition, Index_p = clustering(D_ID, Co-occurrences_pre, N)
for all partition ← firstPartition do
    secondPartition, Index = clustering(partition, Co-occurrences_sub, K)
    Output secondPartition
    for all (e, p) ← Index do
        Index_s(e) ← p
    end for
    secondPartition, Index = clustering(partition, Co-occurrences_obj, K)
    Output secondPartition
    for all (e, p) ← Index do
        Index_o(e) ← p
    end for
end for
Output Index_s, Index_p, Index_o
```

を特定し，テーブル名を決定する．

(c) 決定したテーブル名を `FROM` 句に，束縛変数を `WHERE` 句に，非束縛変数を `SELECT` 句に指定し，その結果を `AS` 句によって table_i とする SQL サブクエリを作成する．

3. 得られた $|Q|$ 個の SQL サブクエリを `FROM` 句に指定し，SPARQL クエリの `SELECT` 句に指定されていた変数を SQL の変数に指定する．SPARQL クエリの `WHERE` 句内の非束縛変数の結合を基に SQL の `WHERE` 句内を作成し，得られた SQL と読み込むファイルのパス，テーブル名をまとめて出力する．

上記の手順 2(b) のテーブル名はファイルへのパスから指定され，パスが同じ場合常にテーブル名は同じになる．なお，重複したパスとテーブル名は 1 つにまとめられる．したがって，複数の SQL サブクエリに同じパスの組み合わせが存在する場合，それらすべてのサブクエリに対して 1 つのテーブルを作成するだけでよくなり，データの読み込み時間と読み込むデータ量の削減が期待できる．一方で，クエリの `WHERE` 句が `s1 ?p1 ?o1` や `s2 ?p2 ?o2` のようなパターンで構成されている場合，各サブクエリの結果は複数のファイルに分散している可能性がある．あるサブクエリパターンの結果がファイル 1, 2, 4 に，別のサブクエリパターンの結果がファイル 4, 5, 6 にある場合，変換プロセスではファイル 1, 2, 4 を 1 つのテーブルに，ファイル 4, 5, 6 を別のテーブルにグループ化する．これにより，ファイル 4 からのデータが重複する可能性があり，この場合は読み込むデータ量が増加することになる．さらに，データセット内のすべてのデータにインデックスを作成しているため，サブクエリの束縛変数にインデックスがない場合は一致するデータが存在しないと判断できる．評価実験における速度評価ではこのような場合，あらかじめ用意しておいた空のテーブルを読み込むファイルとして指定して問合せを行った．

5 評価実験

本章では，提案手法の性能を確認するために合成データセットと実世界データセットを 1 つずつ使用し，複数の指標によって従来手法との比較を行った結果について述べる．従来手法としては従来手法を上回る性能を示している DIAERESIS を使用した．その際のパーティション数は，DIAERESIS の論文中のものと同様に 4 に設定した．なお，提案手法のパーティション数は第一段階で 5, 第二段階で２０とした．また，ID 化の優位性を確認するために，提案手法で ID 化を行わない場合の性能も検証した．

5.1 実験環境

実験は，Apache Spark（３．３．１）を Spark Standalone モードで動作させた 1 台の物理マシン上で実施した．提案手法では基本的にドライバーメモリとエグゼキューターメモリに 32GB を割り当てた．ただし，ID 化を行う時のみ 50GB を割り当てている．従来手法は，ドライバーメモリとエグゼキューターメモリのいずれも 10GB を割り当てている．マシンのスペックは表 1 の通りである．さらに，AWS EC2 の EMR クラスター

表 1: 実行マシンのスペック

OS	Ubuntu 18.04.5 LTS
CPU	AMD EPYC 7502P 32-Core Processor
メモリ	128GB

(emr-7.0.0) でインスタンスタイプ m5.xlarge，インスタンスサイズ 10 での実験も行った．提案手法は scala(2.12.7) で実装し，sbt（1.8.0）でコンパイルした．比較手法は scala(2.12.10) で実装されており，JDK1.6 でコンパイルした．

5.2 データセット

本実験では以下の 2 つのデータセットを使用した．なお，AWS 上では，比較手法において LUBM を用いた際にエラーが発生したため SWDF のみ実験を実施している．

5.2.1 LUBM（The Lehigh University Benchmark）

LUBM（The Lehigh University Benchmark）[15] は大学の数をパラメータとした合成データ生成ツールであり，RDF データベースシステムのベンチマークとして広く利用されている．実験では，パラメータを 100 としたデータセットを作成しており，これはデータサイズ 2.37GB，トリプル数 13.8M で構成されている．ワークロードと性能評価には，ベンチマークから提供されたクエリから 13 個を使用した．

5.2.2 SWDF（The Semantic Web Dog Food）

SWDF（The Semantic Web Dog Food）は，人，論文，講演に関するセマンティックウェブ会議のメタデータを含む実世界の RDF データセットである．このデータセットはデータサイズが 50MB であり，304,583 トリプルが含まれている．ワークロードと性能評価には，比較手法である DIAERESIS で使用されていたクエリから 207 個を使用した．なお，このクエリは実際のクエリログに基づいて FEASIBLE ベンチマークジェネレータ [16] によって生成されたものである．

5.3 評価指標

本実験では，性能評価に以下の指標を導入した．なお，比較手法の実験の条件に合わせるために，SPARQL から SQL への

表 2: 前処理時間の比較（括弧内は ID 化時間（左）と分割時間（右））

Method	Processing Time	Data Size
LUBM (13.4M Triples)		
DIAERESIS	16.38 ± 0.30 min	762MB
Proposed Method (with ID)	28.31 ± 0.39 min (14.46 + 13.85)	199MB
Proposed Method (without ID)	38.44 ± 0.87 min	406MB
SWDF (30.4K Triples)		
DIAERESIS	1.80 ± 0.02 min	14MB
Proposed Method (with ID)	1.27 ± 0.03 min (0.20 + 1.07)	6.6MB
Proposed Method (without ID)	3.06 ± 0.07 min	64MB

表 3: 比較手法を 1 とした時の問合せ実行時間の比の平均

Dataset	DIAERESIS	Proposed method (with ID)	Proposed method (without ID)
LUBM	1	0.80	0.95
SWDF	1	0.51	0.96

変換にかかる時間は計測していない．報告する時間は 5 回の実行の平均値である．

- 前処理時間: 分割したデータを出力するまでの時間
- 問合せ実行時間: SQL を実行してから結果を得るまでの時間
- テーブル作成コスト: 読み込んだデータ中のトリプル数と，テーブル作成までの時間
- 合計問合せ処理時間: 問合せ実行時間とテーブル作成時間の合計

5.4 実験結果（単一マシン）

5.4.1 前処理時間

前処理時間は元の RDF データセットを入力してから分割データが出力されるまでの時間を計測したものであり，その結果を表 2 に示す．提案手法の ID 化の有無による処理時間では，ID 化を行う場合の方が各データセットに対して大きく優れた結果を示しており，ID 化の優位性が確認できた．提案手法と比較手法では，LUBM では比較手法が，SWDF では提案手法が高速であり，ID 化にかかる時間がボトルネックであると考えられる．

5.4.2 問合せ実行時間

表 3 は SQL 実行時間の比較結果であり，いずれのデータセットにおいても提案手法が比較手法を上回る結果となっている．個々のクエリでの比較では，LUBM では 13 個中 12 個のクエリで，SWDF では 207 個中 155 個のクエリで提案手法が優れていた．特に SWDF では，平均 73%，最大 90%の処理時間削減となっており，一方で結果が劣っていたクエリについてもその処理時間増加量の平均は 23%，最大 46%であった．よって，一部劣る処理はあるもののほとんどの問合せ処理で性能向上が実証できていると考える．

5.4.3 テーブル作成コスト

提案手法と比較手法はいずれもインデックスを利用して必要なデータを事前に特定し，それらによって短時間で最低限のデータへアクセスしている． 表 4 は問合せ時に読み込んだトリプル数の比較であるが，いずれのデータセットにおいても提案手法は比較手法よりも平均的に多くのトリプルを読み込んでいる．これについて詳細な検証を行うと，特に LUBM のコンプレックスクエリで特に多くのトリプルを読み込んでいることが確認された．また，SWDF では半数以上の問合せでは読み込んだトリプル数は比較手法より少なかった．したがって，特定のパターンを含むクエリに対してはより多くのトリプルを読み込む傾向にあると考えられる．

表 4: 比較手法を 1 とした時のロードトリプル数の比の平均

Dataset	DIAERESIS	Proposed Method (with ID)	Proposed Method (without ID)
LUBM	1	2.01	2.59
SWDF	1	1.82	2.03

表 5: 比較手法を 1 とした時のテーブル作成時間の比の平均

Dataset	DIAERESIS	Proposed method (with ID)	Proposed method (without ID)
LUBM	1	0.66	1.48
SWDF	1	0.52	0.68

表 6: 比較手法を 1 とした時の合計時間の比の平均

Dataset	DIAERESIS	Proposed Method (with ID)	Proposed Method (without ID)
LUBM	1	0.71	1.26
SWDF	1	0.47	0.84

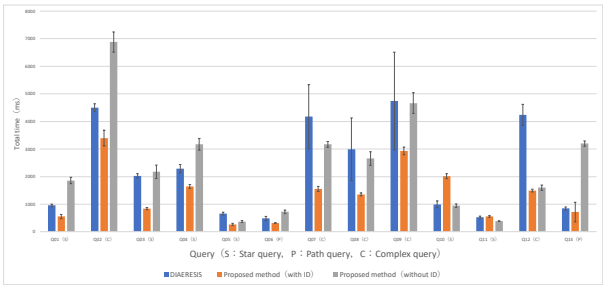

図 8: LUBM における合計時間の比較

一方，テーブルを作成する時間は表 5 に示す通りであり，提案手法はいずれのデータセットに対しても比較手法より 30%以上高速にテーブル作成を完了している．個別に LUBM のクエリを分析したところ，標準偏差も多くのクエリで比較手法より大幅に小さく，安定していることが確認できた．この傾向は SWDF でも同様であり，207 個中 205 個のクエリで比較手法より高速であった．これはワークロードの導入により複数のサブクエリ間でテーブルの共有が可能になり，作成すべきテーブル数が削減できたためではないかと考えられる．

5.4.4 合 計 時 間

最後にテーブル作成時間と問合せ実行時間の合計時間を問合せ全体にかかる時間とみなして比較を行う． 表 6 を見ると，提案手法は両データセットで比較手法を大幅に凌駕していることが確認できる．詳細な分析を行ったところ，LUBM においては図 8 に示すように，比較手法より 2 倍程度遅いクエリも 1 つ存在したものの，その他はほとんどのクエリで比較手法を大きく上回っており，最大約 75%の高速化を達成した．SWDF においても，198 のクエリで比較手法を上回っており，その差は最

Method	Processing Time
DIAERESIS	2.84 ± 0.06 min
Proposed Method（with ID）	20.18 ± 0.81 min（0.70+19.48）

図 9: 比較手法と提案手法の前処理時間（括弧内は ID 化時間（左）と分割時間（右））

method	Load Time	Query Time	Total Time
DIAERESIS	1	1	1
Proposed Method（with ID）	0.80	0.61	0.59

図 10: 比較手法を 1 とした時の各処理時間の比の平均

大約 95%であった．一方，残りの 9 個のクエリで比較手法に劣る結果となっていたが，その差は最大約 15%であり，許容できるものと考えられる．

5.5 実験結果（AWS）

5.5.1 前処理時間

AWS 上で RDF データを分割する際にかかった時間を計測した結果を表 9 に示す．両手法とも単一マシンでの実験よりも処理が遅くなっており，これは複数ノード間での通信にかかるコストによるものだと考える．また，提案手法の実装における分散処理の最適化が十分ではなく，その影響もより顕著に出たと考えている．

5.5.2 実 行 時 間

各処理にかかる時間を表 10 に示す．いずれも比較手法を大幅に上回っており，個別のクエリでも約 70%のクエリで提案手法が高速に処理されていた．さらに，標準偏差も多くのクエリで小さく安定した処理ができていると考えられる．

6 ま と め

本論文では，RDF データの効率的な分割と処理のための Spark ベースの方法を提案した．入力ワークロードクエリの共起を分析することにより関連するデータを同じパーティションに配置し，データ取得の効率向上を達成した．また，各データとそれが属するパーティションを記録したインデックスの導入によりさらなる効率化も可能にした．評価実験では，提案手法が単一ノードにおいて既存手法を上回ることを示した．これは主にデータアクセスの向上や共起性の活用による読み込むトリプル数の削減，ID 化による冗長性排除によるものであると考えられる．一方で，クエリの構成によっては重複したデータの読み込みが発生することがあり，それに伴って読み込むトリプルの数が極端に多くなる場合も確認した．したがって，今後の課題として重複したデータ読み込みの抑制が挙げられる．また，ワークロードの量や質による影響の検証，統計データ等の利用によるクエリ効率化，より大規模なデータセットにおける検証等も行う予定である．

謝　　辞

この成果は，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の 「ポスト 5G 情報通信システム基盤強化研究開発事業」（JPNP20017）の委託事業，JST CREST（JPMJCR22M2），科学研究費補助金（JP22H03694）に支援によるものです．

文　　献

[1] Graham KLYNE. Resource description framework (rdf) : Concepts and abstract syntax. *http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/*, 2004.
[2] B. McBride. Jena: a semantic web toolkit. *IEEE Internet Computing*, Vol. 6, No. 6, pp. 55–59, 2002.
[3] Thomas Neumann and Gerhard Weikum. Rdf-3x: A risc-style engine for rdf. *Proc. VLDB Endow.*, Vol. 1, No. 1, p. 647–659, aug 2008.
[4] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. Dream: Distributed rdf engine with adaptive query planner and minimal communication. Vol. 8, No. 6, p. 654–665, feb 2015.
[5] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2rdf: Rdf querying with sparql on spark. *arXiv preprint arXiv:1512.07021*, 2015.
[6] http://spark.apache.org.
[7] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. Sparqlgx: Efficient distributed evaluation of sparql with apache spark. In *The Semantic Web–ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part II 15*, pp. 80–87. Springer, 2016.
[8] Amgad Madkour, Ahmed M Aly, and Walid G Aref. Worq: Workload-driven rdf query processing. In *International Semantic Web Conference*, pp. 583–599. Springer, 2018.
[9] Georgia Troullinou, Giannis Agathangelos, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. Diaeresis: Rdf data partitioning and query processing on spark.
[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 29–43, 2003.
[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.
[12] http://hadoop.apache.org.
[13] Adnan Akhter, Muhammad Saleem, Alexander Bigerl, and Axel-Cyrille Ngonga Ngomo. Efficient rdf knowledge graph partitioning using querying workload. K-CAP '21, 2021.
[14] Xintong Guo, Hong Gao, and Zhaonian Zou. Wise: Workload-aware partitioning for rdf systems. *Big Data Research*, Vol. 22, p. 100161, 12 2020.
[15] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Semant.*, Vol. 3, No. 2-3, pp. 158–182, 2005.
[16] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. Feasible: A feature-based sparql benchmark generation framework. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14*, pp. 52–69. Springer, 2015.