SPARQLer必読！

実践で学ぶSPARQLクエリ集

まえがき

本書は、SPARQL[*1]の初学者が実践力を身に着けることを目的に書かれた本です。入門的な内容だけではなく、実践的なクエリがまとまった資料は Web 上でもまだ少ないのが現状です。そこで弊研究室の SPARQLer(SPARQL を操る者) の皆さんの協力の下、様々なクエリをかき集めました。集められたクエリを難易度に応じて分類し、演習としてサンプルクエリを実行しながら読み進められるノック集として構成しました。基本を学んだ後のステップアップのための「初級〜上級」、そして SPARQL を極める「職人級」まで用意しています。楽しんでいただければ幸いです。

本書で扱う SPARQL は、2013 年に W3C 勧告となった SPARQL 1.1[*2]の仕様を基に記述しています。しかしながら、実用上は特定の実装に依存する機能を使ったり、仕様に違反する挙動に対応したりする必要を迫られることがあると思われます。現に、我々は本書で紹介するクエリを出し合う中で、そういった場面に沢山遭遇しました。標準の仕様に準拠していないクエリをいくつか紹介していますが、可能な限り注意書きを入れていますので、お読みになる際はその点もご確認ください。

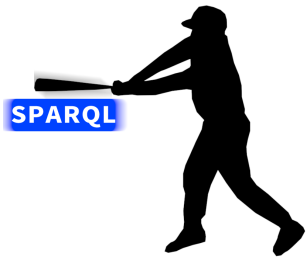
DBpedia Japanese を利用するサンプルクエリについては、2021-12-01 版データセットでクエリの実行を確認しています。

ライセンスについて

[*1] https://www.w3.org/TR/sparql11-overview/
[*2] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/
[*3] https://creativecommons.org/licenses/by-nc-sa/4.0/

目次

第 1 章



ノックを始める前に

まずは準備運動から始めよう！

これから 50 本ノックを始めるにあたり、本章では SPARQL クエリを実行する環境について説明します。

1.1　公開 SPARQL エンドポイントを使う

Web 上には様々な Linked Open Data(LOD) データセットが公開されており、The Linked Open Data Cloud[*1]ではそれらを可視化したグラフで把握することができます。全ての LOD データセットが SPARQL エンドポイント提供しているとは限りませんが、利用可能なエンドポイントをいくつか見つけることができます。

本書の 50 本ノックでは、規模が大きく幅広いデータが存在する DBpedia Japanese の SPARQL エンドポイントを主に利用します。DBpedia とは、Wikipedia に記述された情報から構造化情報を抽出してデータセットを生成するプロジェクトまたはそのデータセットのことです。DBpedia Japanese は日本語版 Wikipedia を対象として生成されたデータセットです。Web ブラウザからアクセス可能なインタフェースがありますので、お手元の PC に特別の準備は不要でノックを楽しむことができます。

それでは早速 DBpedia Japanese(`http://ja.dbpedia.org/`) にアクセスしてみましょう (図 1.1)。

図 1.1　DBpedia Japanese トップページ

ページ上部のナビゲーションバーから「SPARQL Endpoint」のリンクをクリックします。

[*1] `https://lod-cloud.net/`

図 1.2　DBpedia Japanese SPARQL エディタ

図 1.2 の SPARQL クエリエディタが表示されれば、これで準備は完了です。あとは「Query Text」のテキストフォームに SPARQL クエリを入力し「Run Query」ボタンをクリックすればクエリが実行されます。

なお、この Web インタフェースは SPARQL サーバの一つである OpenLink Virtuoso[*2]によって提供される機能です。全ての SPARQL エンドポイントがこのような Web インタフェースを提供しているわけではありません。

1.2　SPARQL クエリエディタ YASGUI

前節では DBpedia Japanese の Web インタフェースの使い方を紹介しました。しかし、クエリを入力し実行するための基本的な機能しかありませんので、より複雑なクエリを効率的に書くためには少々不十分です。ここでは特定のエンドポイントに依存しないクエリエディタ YASGUI を紹介します。

YASGUI(Yet Another Sparql GUI)[*3]は、構文のバリデーションや入力補完など備えたクエリエディタと、クエリ結果を様々なグラフで表示する可視化機能を提供します。特に構文のバリデーション機能は SPARQL 初学者にとって非常に有用ですので、ノックでは基本的に YASGUI を使うことをおすすめします。他にも設定次第で様々な機能を使うことができますので、興味のある方はドキュメント[*4]をご参照ください。

それでは使い方を簡単に説明します。Web ブラウザ上で動作するツールですので、インストール作業などは不要です (CORS プロキシや短縮 URL 作成など一部の機能は別途サーバソフトウェアが必要です)。公式サイトにて使用可能な状態で提供されているので、今回はこちらを使って説明します。

`https://yasgui.triply.cc/` にアクセスすると、次のようなエディタが表示されます（図 1.3)。

エディタ上部のテキストボックスにはエンドポイント URI を指定します。試しに DBpedia

[*2] https://virtuoso.openlinksw.com/
http://vos.openlinksw.com/owiki/wiki/VOS/
[*3] https://yasgui.triply.cc/
[*4] https://triply.cc/docs/yasgui-api

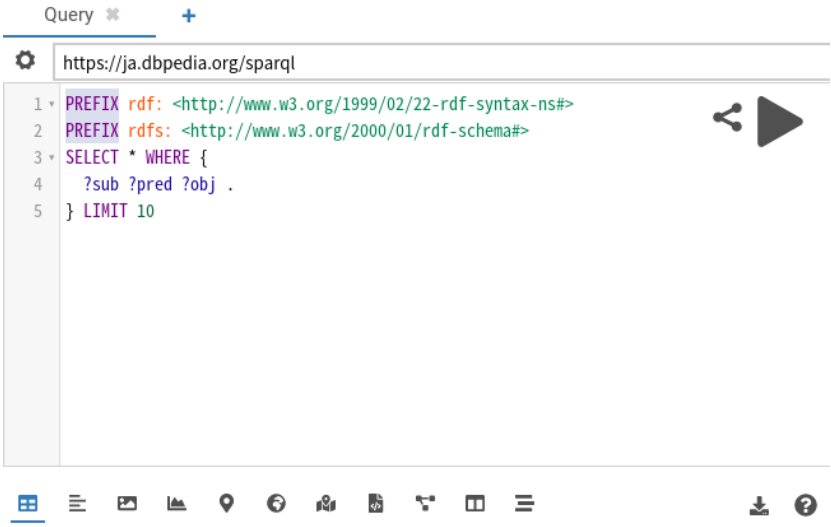


図 1.3　YASGUI

Japanese のエンドポイント URI(`https://ja.dbpedia.org/sparql`) を入力してみます。あとはクエリを入力した後、右上の ▶ ボタンをクリックすればクエリが実行されます。また、 ボタンをクリックすると、そのクエリを共有するためのリンクが生成されます。渾身の一本をみんなに自慢しましょう。

1.3　任意のデータに対して SPARQL クエリを実行する

ここまでは、特定の SPARQL エンドポイントに対してクエリを実行する方法を紹介しました。本節では、自分で用意した RDF データに対してクエリを実行する方法を紹介します。

一般的には、OpenLink Virtuoso や Apache Jena[*5]などの RDF ストアをインストールして SPARQL サーバを構築することが必要です。しかしながら、このような環境構築で読者の皆さんを消耗させることは本書の意図するところではありません。

そこで RDFShape[*6]というツールを紹介します。このツールは RDF データを検証するためのデモとして作成されたようですが、RDF に関連する様々な操作を試すことができる非常に便利なツールです。個々の機能を紹介するだけでもう一冊本が書けそうなので、SPARQL の部分のみを取り上げます。RDFShape はローカルにインストールすることもできますが、こちらも Web 上で公開されているデモを利用することにします。

`http://rdfshape.herokuapp.com/query` にアクセスします。

[*5] `https://jena.apache.org/`
[*6] RDFShape: RDF playground, `http://rdfshape.herokuapp.com/` `https://github.com/labra/rdfshape`

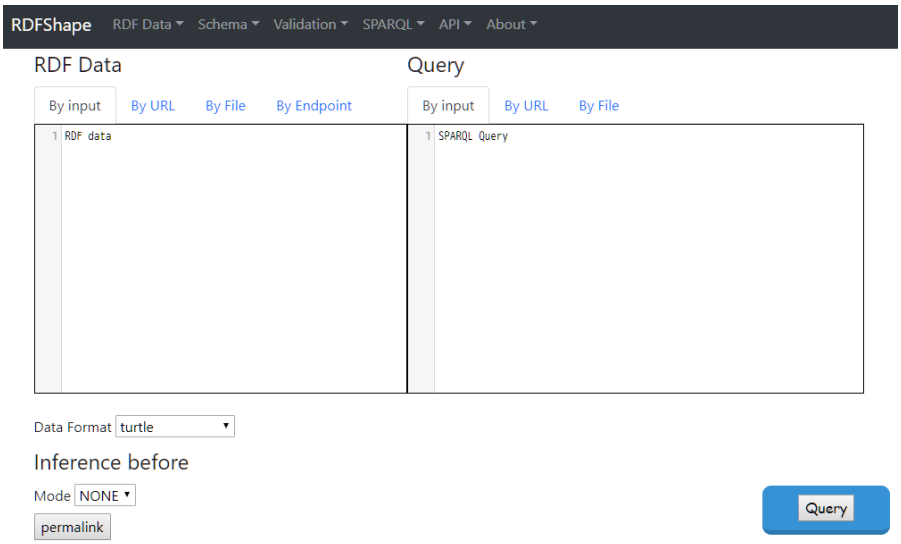


図 1.4　RDFShape SPARQL

図 1.4 のような画面が表示されます。左側に RDF データを、右側に SPARQL クエリを入力し「Query」をクリックすればクエリが実行されます。入力する RDF データは、Turtle や TriG、JSON-LD など主要なフォーマットはサポートされていますので、左下の「Data Format」から適切なものを選択してください。そのままテキストで入力する以外にも、アクセス可能な URI やファイルアップロードで入力することも可能です。

次の図 1.5 はクエリを実行した例です。結果は上部にテーブル形式で表示されます。

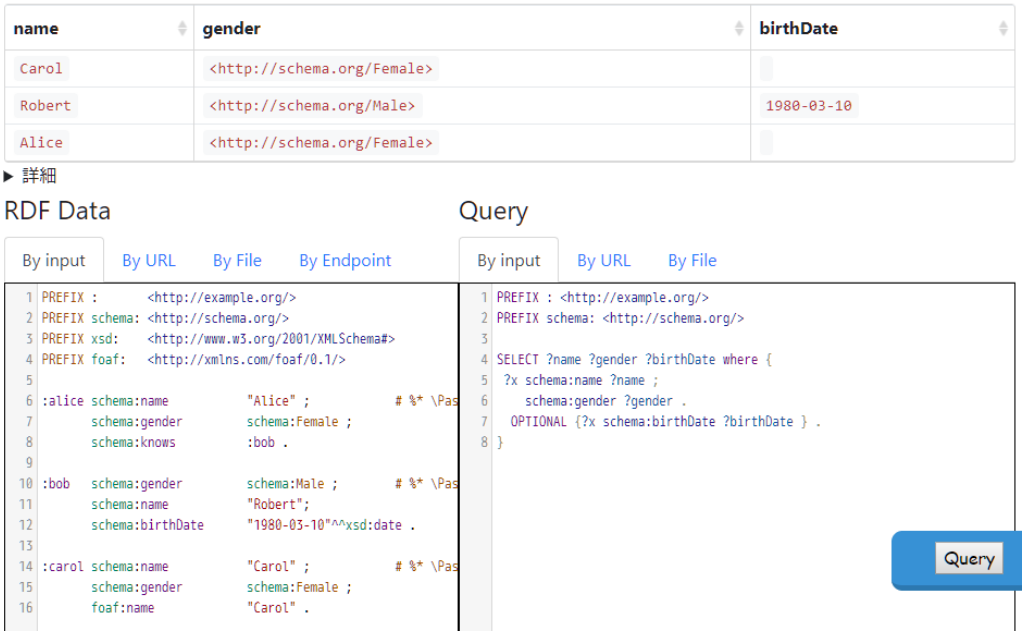


図 1.5　RDFShape SPARQL クエリ実行例

第 2 章

初級

基本の構文や関数を確認しながら軽く打っていこう！

それではノックを始めましょう。各ノックに表示している Endpoint は、そのクエリ対象の SPARQL エンドポイント URI を示します。「汎用」と表示されているものは、特定のエンドポイントに依存しないクエリです。特定の SPARQL サーバの実装に依存クエリについては、その製品名称 (「OpenLink Virtuoso」など) を表示しています。

1 本目　クラス IRI のリストを取得する

Endpoint 汎用

```
SELECT DISTINCT ?class
WHERE {
  ?s a ?class .
}
```

そのままです。なお、キーワード「a」は rdf:type と等価です (大文字小文字を区別します)。

2 本目　グラフ IRI のリストを取得する

Endpoint 汎用

```
SELECT DISTINCT ?g
WHERE {
  GRAPH ?g {
    ?s ?p ?o .
  }
}
```

名前付きグラフの IRI をすべて取得します。GRAPH キーワードを明示的に指定しない場合は、デフォルトグラフがクエリの対象となります (詳細は 👍35 本目)。

3 本目　ASK でクエリパターンにマッチするか否かを調べる

Endpoint 汎用

```
ASK {
  <http://example.com/foo> ?p ?o .
}
```

ASK を使うと、クエリパターンにマッチする結果の有無をブール値の true/false で得ることができます。

4 本目　DESCRIBE で指定した URI に関するデータを取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
1 DESCRIBE ?s
2 WHERE {
3   ?s <http://www.w3.org/2000/01/rdf-schema#label> "ペタンク"@ja .
4 }
```

DESCRIBE は、指定した IRI のリソースに関する情報を含んだ RDF グラフを取得するクエリです。サンプルクエリでは、WHERE で指定したパターンに従って、rdfs:label の目的語に`"ペタンク"@ja` を持つ主語のリソースに関する情報が得られます。

結果の内容は仕様上定義されておらず、SPARQL サーバの実装に依存します。例えば Virtuoso の場合、デフォルトの設定では、指定した IRI を主語または目的語に持つトリプルが結果として返却されます。`sql:describe-mode` pragma[*1]を用いて、いくつかの異なる形式で結果を得ることができます。

5 本目　FILTER で指定した範囲の値にマッチするリソースを取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
1 SELECT *
2 WHERE {
3   ?s <http://ja.dbpedia.org/property/生年> ?date .
4   FILTER (?date > "1950-01-01"^^xsd:date && ?date < "1959-12-31"^^xsd:date)
5 }
6 ORDER BY (?date)
```

FILTER とその条件に比較演算子「>」「<」を使って、1950 年 1 月 1 日〜1959 年 12 月 31 日の間に生まれた人物を取得しています。`"1959-12-31"^^xsd:date` は型付きリテラルを表しており、キャスト関数を使って `xsd:date("1959-12-31")` と書くこともできます。

また、論理和の演算子「&&」は Virtuoso では「and」も使うことができますが、標準ではないので注意してください。

6 本目　リテラルの言語タグを取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
1 SELECT DISTINCT (LANG(?label) AS ?langtag)
2 WHERE {
3   ?s <http://www.w3.org/2000/01/rdf-schema#label> ?label .
4 }
```

LANG 関数を使ってリテラルの言語タグを取得しています。言語タグがない場合は`""`が返されます。

[*1] `http://docs.openlinksw.com/virtuoso/rdfsqlfromsparqldescribe/`

7本目　IF で条件分岐する

Endpoint `https://ja.dbpedia.org/sparql`

```
SELECT DISTINCT ?name
  (IF(isIRI(?kamon), STRAFTER(STR(?kamon), "resource/"), STR(?kamon)) AS ?kamonStr)
WHERE {
  ?s <http://ja.dbpedia.org/property/家紋> ?page ;
     <http://www.w3.org/2000/01/rdf-schema#label> ?name ;
     <http://ja.dbpedia.org/property/家紋名称> ?kamon .
}
```

ある値で条件分岐したいときは、IF 関数を使います。IF 関数は、第一引数をブール値[*2]として評価し、結果が true であれば第二引数を返し、違えば第三引数を返します。

このクエリでは家紋の名称を取得したいのですが、`<http://ja.dbpedia.org/property/家紋名称>`の目的語 (?kamon) は、IRI とリテラルが混在しています。isIRI 関数で?kamon が IRI であるかを判定し、IRI であれば STRAFTER 関数で"resource/"以降の文字列を切り出します (STRAFTER 関数は文字列リテラルを引数とするので、?kamon を STR 関数で変換していることに注意)。IRI でなければ `STR(?kamon)` を返します。

8本目　BIND で値を変数にバインドする

Endpoint `https://ja.dbpedia.org/sparql`

```
SELECT DISTINCT ?name ?kamonStr
WHERE {
  ?s <http://ja.dbpedia.org/property/家紋> ?page ;
     <http://www.w3.org/2000/01/rdf-schema#label> ?name ;
     <http://ja.dbpedia.org/property/家紋名称> ?kamon .
  BIND (IF(isIRI(?kamon), STRAFTER(STR(?kamon), "resource/"), STR(?kamon))
    AS ?kamonStr)
}
```

7 本目と同じ結果が得られますが、IF を SELECT 句に置くのではなく、WHERE 句の中で BIND を使って書くこともできます。複雑な条件分岐を書く場合は SELECT 句に置くと読みにくくなるので、BIND を使って整理して書くと良いでしょう。

9本目　REPLACE で文字列を正規表現で置換する

Endpoint `https://ja.dbpedia.org/sparql`

```
SELECT DISTINCT ?name ?kamonStr
WHERE {
  ?s <http://ja.dbpedia.org/property/家紋> ?page ;
     <http://www.w3.org/2000/01/rdf-schema#label> ?name ;
     <http://ja.dbpedia.org/property/家紋名称> ?kamon .
```

[*2] 17.2.2 Effective Boolean Value (EBV), `https://www.w3.org/TR/sparql11-query/#ebv`

```
6   BIND (IF(isIRI(?kamon), REPLACE(STR(?kamon), ".+/resource/([^/]+)$", "$1"), STR(?
    kamon)) AS ?kamonStr)
7 }
```

このクエリも 7 本目、8 本目と同じ結果が得られます。STRAFTER 関数を使って IRI から家紋名称を取り出していましたが、このクエリでは REPLACE 関数を使って同等の処理を実現しています。REPLACE 関数では第一引数の文字列を、第二引数を正規表現パターン、第三引数を置換文字列として置換します。

正規表現をパターンをクエリ内に記述するときは、エスケープが必要な文字に注意してください。特にバックスラッシュ (\) には注意が必要です。文字「\」にマッチさせたい場合は、まず SPARQL クエリ内の文字列に対するエスケープシーケンスで「\\」となり、さらに正規表現に対するエスケープシーケンスで「\\\\」とする必要があります。

10 本目　REGEX で文字列を正規表現でマッチする

Endpoint `https://ja.dbpedia.org/sparql`

```
1 SELECT ?s ?label
2 WHERE {
3   ?s <http://www.w3.org/2000/01/rdf-schema#label> ?label .
4   FILTER REGEX(?label, "QL$", "i")
5 }
```

FILTER と REGEX 関数を使って、指定した正規表現にマッチする?label を持つリソースを取得します。REGEX 関数の第三引数には"i"フラグが指定されていますので、大文字小文字問わず末尾が「QL」である?label を持つリソースが抽出されます。

グラフパターンではなく、文字列リテラルを評価した条件で FILTER するときに多用する関数なので、よく覚えておきましょう。

11 本目　DBpedia のカテゴリのリストを取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
1 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
2 SELECT ?s
3 WHERE {
4   ?s a skos:Concept.
5   FILTER CONTAINS(STR(?s), "resource/Category:")
6 }
```

`skos:Concept` のインスタンスであるという条件のみだとカテゴリ以外のリソースも混じってしまうので、CONTAINS 関数を使ってカテゴリを示す IRI かどうか判定します。CONTAINS 関数は、REGEX 関数のような正規表現パターンではなく単純に部分文字列として含まれるかどうかを判定します。この場合は REGEX 関数でも代替可能です。

12 本目　カテゴリと同名の記事を取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?article ?category
WHERE {
  ?article rdfs:label ?labelArticle .
  ?category a skos:Concept ;
    rdfs:label ?labelConcept .
  FILTER (CONTAINS(STR(?category), "resource/Category:"))
  FILTER (?article != ?category)
  FILTER (?labelArticle = ?labelConcept)
}
ORDER BY ?labelArticle
```

カテゴリを取得する方法は 11 本目と同様です。記事とカテゴリのラベルをそれぞれ取得し、それらが同じであるという条件を 10 行目の FILTER で指定しています。2 つが同じリソースである場合を排除するために、9 行目の FILTER が必要です。

13 本目　都道府県のリストを取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
PREFIX dcterms: <http://purl.org/dc/terms/>
SELECT ?pref ?code
WHERE {
  ?pref dcterms:subject <http://ja.dbpedia.org/resource/Category:日本の都道府県> ;
        <http://ja.dbpedia.org/property/区分> ?category ;
        <http://dbpedia.org/ontology/areaCode> ?code .
  FILTER REGEX(?category, "[都道府県]")
  FILTER REGEX(?code, "^JP")
}
ORDER BY (xsd:integer(STRAFTER(?code, "JP-")))
```

DBpedia では記事のカテゴリを `dcterms:subject` の目的語として取得できます。このクエリでは、「日本の都道府県」カテゴリに属し `prop-ja:区分`が「都道府県」のいずれかである、というグラフパターンを指定することで、都道府県を取得しています。さらに、全国地方公共団体コードを?code に取り出してソートしています。

14 本目　バレーボールを題材としたアニメ作品を取得する

Endpoint https://ja.dbpedia.org/sparql

```
1 PREFIX dcterms: <http://purl.org/dc/terms/>
2 PREFIX category-ja: <http://ja.dbpedia.org/resource/Category:>
3 SELECT DISTINCT ?s WHERE {
4   ?s dcterms:subject ?volley ;
5      dcterms:subject ?cat .
6   VALUES ?volley {category-ja:バレーボールを題材とした作品 category-ja:女子バレーボール
    を題材とした作品 category-ja:バレーボール漫画}
7   FILTER CONTAINS(STR(?cat), "アニメ作品")
8 }
```

このクエリも DBpedia のカテゴリをうまく利用しています。6 行目で、バレーボールを題材とした作品に付与されるであろうカテゴリを列挙して、?volley にバインドします。VALUES は BIND とは異なり、複数の値を変数にバインドすることができます。4 行目のパターンを評価すると、「バレーボールを題材とした作品」「女子バレーボールを題材とした作品」「バレーボール漫画」のいずれかのカテゴリに属する、という条件になります。このクエリでは、VALUES を FILTER と IN を用いて以下のように書き換えても同じ結果が得られます。

```
6 FILTER (?volley IN (category-ja:バレーボールを題材とした作品, category-ja:女子バレーボ
  ールを題材とした作品, category-ja:バレーボール漫画))
```

7 行目の FILTER は、「アニメ作品」という文字列を含むカテゴリでフィルタすることでアニメ作品だけに絞り込んでいます。

15 本目　プログラミング言語の一覧を取得する

Endpoint https://ja.dbpedia.org/sparql

```
1 SELECT *
2 WHERE {
3   ?s a <http://dbpedia.org/ontology/ProgrammingLanguage> ;
4      <http://www.w3.org/2000/01/rdf-schema#label> ?label .
5 }
6 ORDER BY ?label
```

dbpedia-owl:ProgrammingLanguage のインスタンスであるリソースを取得しています。

16 本目　プログラミング言語の一覧を頭文字で集計して降順に表示する

Endpoint https://ja.dbpedia.org/sparql

```
1 SELECT ?initial (COUNT(?s) AS ?count) (GROUP_CONCAT(?label; SEPARATOR=", ") AS ?list)
2 WHERE {
3   ?s a <http://dbpedia.org/ontology/ProgrammingLanguage> ;
4      <http://www.w3.org/2000/01/rdf-schema#label> ?label .
5   BIND (SUBSTR(?label, 1, 1) AS ?initial)
```

```
}
GROUP BY ?initial
ORDER BY DESC(COUNT(?initial))
```

15 本目に加えて、頭文字毎に集計しています。SUBSTR で先頭 1 文字を切り出し、GROUP BY でそれを集約します。GROUP_CONCAT は、集約された値を SEPARATOR で連結した文字列を返します。このクエリでは、頭文字が同じプログラミング言語のラベルが「, 」で連結されて表示されます。

17 本目　特定のプロパティを N 個以上持つリソースを取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
SELECT ?s (COUNT(?class) AS ?count)
WHERE {
  ?s a ?class .
}
GROUP BY ?s
HAVING (COUNT(?class) > 14)
ORDER BY DESC(COUNT(?class))
```

GROUP BY で集約を作成し、集約された結果に対する条件は FILTER ではなく HAVING を使って指定します。このクエリでは、クラスを 15 個以上持つリソースを取得しています。

18 本目　今日が誕生日の人を取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
SELECT *
WHERE {
 ?s <http://ja.dbpedia.org/property/生月> ?month;
    <http://ja.dbpedia.org/property/生日> ?day .
 FILTER (?month = MONTH(NOW()) && ?day = DAY(NOW()))
}
```

NOW や MONTH などの日時に関する関数を使って、今日が誕生日の人物を取得しています。比較を行うときは型に注意しましょう。?month、?day は整数 (`xsd:integer`) の値を持ち、MONTH 関数と DAY 関数も整数が返り値です。

19 本目　東京都出身の 50 歳以下の俳優を取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX category-ja:     <http://ja.dbpedia.org/resource/Category:>
PREFIX prop-ja: <http://ja.dbpedia.org/property/>
PREFIX dbpedia-ja: <http://ja.dbpedia.org/resource/>
SELECT * WHERE {
  ?actor prop-ja:職業 dbpedia-ja:俳優 ;
```

```
8     dcterms:subject category-ja:東京都出身の人物 ;
9     prop-ja:生年 ?year.
10  OPTIONAL {?actor prop-ja:出生地 ?birthplace}
11  FILTER (?year >= (YEAR(NOW()) - 50))
12  FILTER (DATATYPE(?year) = xsd:integer)
13 }
14 ORDER BY ?year
```

「東京都出身の人物」カテゴリと prop-ja:職業を使うことで、東京出身の俳優を取得できます。prop-ja:生年で得られる生年をもとに 50 歳以下という条件で絞り込みます。また、?year が文字列で「不詳」となっているデータもあるため、12 行目で型が xsd:integer でないものを除外しています。

20 本目　職業に「声優」を含む人名典拠を取得する

Endpoint http://id.ndl.go.jp/auth/ndla/sparql

```
1  PREFIX rda: <http://RDVocab.info/ElementsGr2/>
2  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3
4  SELECT *
5  WHERE {
6    ?concept foaf:primaryTopic ?topic .
7    ?topic a foaf:Person ;
8       foaf:name ?name ;
9       rda:biographicalInformation ?work .
10   FILTER CONTAINS(?work, "声優")
11 }
```

エンドポイントは DBpedia Japanese ではなく Web NDL Authorities(NDLA) です。クエリ自体は簡単ですが、NDLA の典拠データのモデルを理解していないと書きづらいため紹介します。

NDLA の個人名・家族名・団体名の典拠データは、典拠情報とその記述対象である名称実体で構成されます[*3]。6 行目の?concept が典拠情報にあたり、7 行目の?topic が名称実体にあたります。rda:biographicalInformation から職業を取得し、「声優」を含むもののみを抽出しています (9, 10 行目)。

21 本目　プロパティパス InversePath を使う

Endpoint https://ja.dbpedia.org/sparql

```
1  SELECT ?s
2  WHERE {
3     ?s ^<http://dbpedia.org/ontology/wikiPageWikiLink> <http://ja.dbpedia.org/resource/
     日本の山一覧> .
4  }
```

プロパティパス式「^」は逆向き (目的語から主語) の向きに述語を辿ります。つまり、以下のグラフパターンと等価です。

[*3] RDF モデルについて, http://id.ndl.go.jp/information/model/

```
<http://ja.dbpedia.org/resource/日本の山一覧> <http://dbpedia.org/ontology/
wikiPageWikiLink> ?s .
```

dbpedia-owl:wikiPageWikiLink は、Wikipedia 記事中の他の記事へのリンクを示します。従って、「日本の山一覧」からリンクを張られている記事のリソース IRI が?s として得られます。

22 本目　乱数を生成する

Endpoint 汎用

```
1 SELECT (RAND() AS ?rand)
2 WHERE {
3   ?s ?p ?o .
4 }
5 LIMIT 10
```

RAND 関数を実行すると $[0, 1)$ の一様乱数が生成されます。しかし、乱数生成の挙動は実装に依存することが多いので注意が必要です。

例えば、DBpedia で使われている Virtuoso でこのクエリを実行すると、同じ数値が 10 個出力されてしまいます。原因の解説については初級の範囲を超えてしまいますので、後述することにします (☞48 本目)。これを回避するためには RAND 関数ではなく、Virtuoso の独自関数 bif:rnd を使う方法があります (☞38 本目)。

第 3 章

中級

より複雑なクエリで理解を深めよう！

初級のノックを踏まえて、GROUP BY による集約やプロパティパスを使った複雑なグラフパターンに挑戦します。

23 本目　DBpedia から上位カテゴリを持たないカテゴリを取得する

Endpoint https://ja.dbpedia.org/sparql

```
1 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
2 SELECT distinct ?s WHERE {
3   ?s a skos:Concept .
4   FILTER NOT EXISTS {?s skos:broader ?bc}
5   FILTER REGEX(STR(?s), 'dbpedia.org/resource/Category')
6 }
```

11 本目をベースに、上位カテゴリを持つカテゴリのパターンを NOT EXISTS をに指定して結果から除外します。

24 本目　クラスの IRI のローカル名っぽい部分をリストにする

Endpoint 汎用

```
1  SELECT DISTINCT
2    ?prefix (COUNT(DISTINCT(?local)) AS ?num)
3    (GROUP_CONCAT(DISTINCT ?local; SEPARATOR=", ") AS ?localNames)
4  WHERE {
5    ?s a ?o .
6    BIND (STR(?o) AS ?oStr)
7    BIND (REPLACE(?oStr, ".+[^a-zA-Z_0-9-]([A-Za-z0-9_-]*)$", "$1") AS ?local)
8    BIND (STRBEFORE(?oStr, ?local) AS ?prefix)
9  }
10 GROUP BY ?prefix
11 ORDER BY DESC(?num)
```

1 本目ではすべてのクラス IRI を取得しましたが、このクエリでは IRI からローカル名[*1]を分離し、接頭辞毎に集計します。また、GROUP BY と GROUP_CONCAT を使って集約されたローカル名を表示します。

7 行目の REPLACE で正規表現を使ってローカル名を抽出していますが、この正規表現は SPARQL 仕様のローカル名と厳密には対応していません。詳細は定義[*2]を参照してください。

[*1] 実際には接頭辞宣言によるため、あくまでそれっぽい部分です

[*2] https://www.w3.org/TR/sparql11-query/#rPN_LOCAL

25 本目　特定のプロパティを N 個以上持つリソースを取得する (サブクエリ版)

Endpoint https://ja.dbpedia.org/sparql

```
SELECT ?s ?count
WHERE {
  FILTER (?count > 14)
  {
    SELECT ?s (COUNT(?type) AS ?count)
    WHERE {
      ?s a ?type .
    }
    GROUP BY ?s
  }
}
ORDER BY DESC(?count)
```

17 本目と同じことをサブクエリを使って書いています。このクエリのように単純な条件であれば、HAVING を使ったほうが短く簡潔で良いでしょう。

26 本目　夏季冬季オリンピック金メダリスト出身地別リストを取得する

Endpoint https://ja.dbpedia.org/sparql

```
SELECT ?pref (count(?name) AS ?count) (GROUP_CONCAT(?name; separator=',') AS ?people)
WHERE {
  ?person <http://purl.org/dc/terms/subject> <http://ja.dbpedia.org/resource/Category:
  日本のオリンピック金メダリスト> ;
          <http://purl.org/dc/terms/subject> ?bplace ;
          <http://www.w3.org/2000/01/rdf-schema#label> ?name .
  ?bplace <http://www.w3.org/2000/01/rdf-schema#label> ?pref .
  FILTER REGEX(?pref, "[都道府県州国]出身の人物", "i")
}
GROUP BY (?pref)
ORDER BY DESC (count(?person))
```

こちらも GROUP BY と GROUP_CONCAT を活用した集計です。

27 本目　日本人の姓（っぽい箇所）を取り出して集計する

Endpoint https://ja.dbpedia.org/sparql

```
SELECT ?family (COUNT(?family) AS ?count)
WHERE {
  ?s a <http://dbpedia.org/ontology/Person>;
     <http://dbpedia.org/ontology/nationality> <http://ja.dbpedia.org/resource/日本> ;
     <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
  BIND (REPLACE(?comment, "([ ・  ( (]|は、|は日本).*$", "") AS ?family)
  FILTER (STRLEN(?family) > 0)
}
GROUP BY ?family
```

```
10 ORDER BY DESC(COUNT(?family))
```

DBpedia で姓のみを取得できるプロパティがあればよいのですが、元の Wikipedia 記事の記述からうまく切り出すしかありません。記事冒頭の書き出しには一定のパターンがあるので、正規表現を使って姓と思われる箇所を取り出しています (6 行目)。姓名が一緒に取り出されてしまったり多少のゴミが含まれていたりするので、もう少し工夫の余地はありそうです。

28 本目　世界遺産のリストを取得する

Endpoint https://ja.dbpedia.org/sparql

```
1  SELECT ?heritage ?name ?country ?lat ?long
2  WHERE {
3    ?heritage <http://purl.org/dc/terms/subject> ?category ;
4              <http://www.w3.org/2000/01/rdf-schema#label> ?name ;
5              <http://ja.dbpedia.org/property/country> ?country .
6    ?category <http://www.w3.org/2004/02/skos/core#broader> <http://ja.dbpedia.org/
     resource/Category:五十音順の世界遺産> .
7    OPTIONAL {
8      ?heritage <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat ;
9                <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long .
10   }
11 }
12 ORDER BY (?country)
```

カテゴリ「五十音順の世界遺産」は下位カテゴリに「世界遺産 あ行」〜「世界遺産 わ行」という下位カテゴリを持ち、そのカテゴリから個別の世界遺産のページを辿ることができます。従って、3, 6 行目で「五十音順の世界遺産」が上位カテゴリであるカテゴリを持つリソースを取得するパターンを指定しています。また、緯度・経度を持たないリソースも存在するため、OPTIONAL 内のパターンとして指定しておきます。

29 本目　借用語のリストを取得する

Endpoint https://ja.dbpedia.org/sparql

```
1  PREFIX dcterms: <http://purl.org/dc/terms/>
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
4
5  SELECT ?category (COUNT(?word) AS ?count)
6    (GROUP_CONCAT(?word; SEPARATOR=', ') AS ?wordList)
7  WHERE {
8    ?s dcterms:subject ?category ;
9       rdfs:label ?word .
10   ?category skos:broader <http://ja.dbpedia.org/resource/Category:借用語> ;
11             rdfs:label ?country .
12   FILTER NOT EXISTS {
13     ?s dcterms:subject <http://ja.dbpedia.org/resource/Category:日本語における借用語> .
14   }
```

```
}
GROUP BY (?category)
ORDER BY DESC(COUNT(?s))
```

カテゴリ「借用語」の下位カテゴリから借用語を取得し、カテゴリ毎に GROUP BY で集計しています。「日本語における借用語」カテゴリのリソースは借用語そのものを扱ったものではないので、NOT EXISTS のパターンで除外しています (13 行目)。

30 本目　政党のイデオロギーと政治的思想・立場を取得する

Endpoint https://ja.dbpedia.org/sparql

```
SELECT ?s ?name (COUNT(DISTINCT(?ideology)) AS ?count)
  (GROUP_CONCAT(DISTINCT ?ideology; separator=", ") AS ?ideologies)
WHERE {
   ?s <http://dbpedia.org/ontology/ideology>|<http://ja.dbpedia.org/property/政治的思想
   ・立場> ?i ;
      <http://www.w3.org/2000/01/rdf-schema#label> ?name .
   ?i <http://www.w3.org/2000/01/rdf-schema#label> ?ideology .
}
ORDER BY DESC(COUNT(DISTINCT(?ideology)))
```

プロパティパスを使って、述語が dbpedia-owl:ideology または prop-ja:政治的思想・立場の目的語を取得しています。両者の目的語は重複した値を持ち得るので、COUNT や GROUP_CONCAT では DISTINCT を忘れないようにします。

31 本目　複数の都道府県にまたがる日本の山のリストを取得する

Endpoint https://ja.dbpedia.org/sparql

```
SELECT ?s (COUNT(DISTINCT(?pref)) AS ?count)
  (GROUP_CONCAT(DISTINCT ?pref; SEPARATOR=', ') AS ?prefList)
WHERE {
  ?s a <http://dbpedia.org/ontology/Mountain> ;
     <http://dbpedia.org/ontology/address> ?address ;
     <http://www.w3.org/2000/01/rdf-schema#label> ?name ;
     ^<http://dbpedia.org/ontology/wikiPageWikiLink> <http://ja.dbpedia.org/resource/日
     本の山一覧> .
  BIND (REPLACE(STR(REPLACE(?address, "([ (） ]|藤津郡太良町・)", "")), "(京都府|[都道府
  県]).*$", "$1") AS ?pref)
  FILTER (REGEX(?pref, "[都道府県]"))
  FILTER (!REGEX(?pref, "(同県|小県)"))
}
GROUP BY ?s
HAVING (COUNT(DISTINCT(?pref)) > 1)
ORDER BY DESC (COUNT(DISTINCT(?pref)))
```

プロパティパス式「^dbpedia-owl:wikiPageWikiLink」を使って、「日本の山一覧」からリンクされている山のリストを抽出しています。山の所在地の都道府県名のみを取得できるプロパティが存在しないため、8〜10 行目では、?address から正規表現を使って都道府県名を取り出しています。

32 本目　コレクション (rdf:List) からメンバ (rdf:first) を取り出す

Endpoint 汎用

```
PREFIX ex: <http://example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT *
WHERE {
  ?s ex:member/rdf:rest*/rdf:first ?o .
}
```

RDF コレクションはいわゆる線形リストのデータ構造で、リンクを辿って順にノードを参照します。SAPRQL ではプロパティパスを使うことで、簡単に各メンバを取り出すことができます。「/」は指定の述語を順にたどり、「*」は述語の 0 回以上繰り返しを意味します。

例えば、以下のようなデータが与えられたときを考えましょう。

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://example.org/group/1> ex:member _:bn1 .
_:bn1 rdf:first <http://example.org/member/foo> .
_:bn1 rdf:rest _:bn2 .
_:bn2 rdf:first <http://example.org/member/bar> .
_:bn2 rdf:rest _:bn3 .
_:bn3 rdf:first <http://example.org/member/baz> .
_:bn3 rdf:rest rdf:nil .
```

サンプルクエリをこのデータに対して実行すると、?o には<http://example.org/member/foo>、<http://example.org/member/bar>、 <http://example.org/member/baz>が得られます。ただし、得られる結果の順序は保証されないことに注意してください。

33 本目　コレクションから任意の位置のメンバを取り出す

Endpoint 汎用

```
PREFIX ex: <http://example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT *
WHERE {
  ?s ex:member/rdf:rest{2}/rdf:first ?o .
}
```

プロパティパス式「rdf:rest{n}」で繰り返し回数を指定することで、任意の位置のメンバを取り出すことができます。このクエリの場合は rdf:rest を 2 回辿ることになりますので、32 本目と同じデータを与えたとすると、?o は<http://example.org/member/baz>になります。

ただし、このプロパティパス式は現在の W3C 勧告の仕様には存在しません。ドラフトの仕様[*3]で

[*3] SPARQL 1.1 Property Paths - W3C Working Draft 26 January 2010,

は存在していたようで、Virtuoso や Apache Jena[*4]など主要な実装では拡張としてサポートされています。

34 本目　あるカテゴリの下位カテゴリに属する記事を取得する

Endpoint `https://ja.dbpedia.org/sparql`

```
1 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
2 PREFIX dcterms: <http://purl.org/dc/terms/>
3
4 SELECT DISTINCT ?s ?category
5 WHERE {
6   ?s dcterms:subject ?category .
7   <http://ja.dbpedia.org/resource/Category:アニメ> ^skos:broader+ ?category .
8 }
```

プロパティパス式「`^skos:broader+`」は、`skos:broader` の 1 回以上の繰り返しのパスを目的語から主語の方向に辿るパスを意味します。7 行目で「アニメ」カテゴリの下位カテゴリを取得し、6 行目でそのカテゴリを持つ記事を取得します。

このように一つの述語に複数の演算子を適用することができますが、優先順位に注意してください (この例ではどちらを先に評価しても結果は同じです)。

35 本目　複数のグラフをマージしたグラフに問い合わせる

Endpoint 汎用

```
1 SELECT ?s ?o
2 FROM <http://example.org/foo>
3 FROM <http://example.org/bar>
4 FROM <http://example.org/baz>
5 WHERE {
6   ?s <http://example.org/term> ?o .
7 }
```

FROM 句はデフォルトグラフとするグラフを示す IRI を指定します。複数の FROM 句を指定した場合は、それらがマージされたグラフがデフォルトグラフとなります。このクエリの例では 3 つの IRI が FROM 句で指定されており、これらのグラフがマージされたグラフを対象にクエリが実行されることになります。

一方で、FROM 句を指定しない場合のクエリ対象は、SPARQL サーバによって異なる場合があるので注意してください。Virtuoso ではすべての名前付きグラフがマージされたグラフが対象となります。[*5]。Fuseki ではデータセットで定義されているデフォルトグラフが対象となります[*6](TDB の設定 `tdb:unionDefaultGraph`[*7]ですべての名前付きグラフをデフォルトグラフとするように変更可能)。

https://www.w3.org/TR/2010/WD-sparql11-property-paths-20100126/#path-language

[*4] ARQ - Property Paths, https://jena.apache.org/documentation/query/property_paths.html

[*5] https://github.com/openlink/virtuoso-opensource/issues/616#issuecomment-268542782

[*6] TDB Datasets, http://jena.apache.org/documentation/tdb/datasets.html

[*7] TDB Configuration, https://jena.apache.org/documentation/tdb/configuration.html

さらに Fuseki では、デフォルトグラフを示す<urn:x-arq:DefaultGraph>と全ての名前付きグラフがマージされたグラフを示す<urn:x-arq:UnionGraph>という特別なグラフ IRI が与えられています。次のクエリのように FROM 句で指定すれば、データセット内の全てのトリプルに問い合わせすることが可能です。

```
SELECT *
FROM <urn:x-arq:DefaultGraph>
FROM <urn:x-arq:UnionGraph>
WHERE {
  ?s <http://example.org/term> ?o .
}
```

名前付きグラフがアプリケーションなどで動的に生成されるケースなどで役に立つかもしれません。

36 本目　ジャニーズ事務所所属タレントと 4 回以上共演した人を取得する

Endpoint https://ja.dbpedia.org/sparql

```
SELECT DISTINCT ?p1 ?p2
WHERE {
  ?work1 <http://dbpedia.org/ontology/starring> ?p1 ;
         <http://dbpedia.org/ontology/starring> ?p2 .
  ?work2 <http://dbpedia.org/ontology/starring> ?p1 ;
         <http://dbpedia.org/ontology/starring> ?p2 .
  ?work3 <http://dbpedia.org/ontology/starring> ?p1 ;
         <http://dbpedia.org/ontology/starring> ?p2 .
  ?work4 <http://dbpedia.org/ontology/starring> ?p1 ;
         <http://dbpedia.org/ontology/starring> ?p2 .
  ?p1 <http://ja.dbpedia.org/property/production> <http://ja.dbpedia.org/resource/ジャニーズ事務所> .
  FILTER (?work1 != ?work2)
  FILTER (?work1 != ?work3)
  FILTER (?work1 != ?work4)
  FILTER (?work2 != ?work3)
  FILTER (?work2 != ?work4)
  FILTER (?work3 != ?work4)
  FILTER (?p1 != ?p2)
}
```

少し長めなので順に説明します。

- 3〜10 行目 出演を表す述語 dbpedia-owl:starring を使って、共演関係にある 2 人の人物 (?p1, ?p2) を抽出する。出演作品?work1 から?work4 を 4 つ並べて、4 回以上共演している人物とする。
- 11 行目 片方の人物 (?p1) はジャニーズ事務所所属である
- 12〜17 行目 4 つの出演作品は重複がないという条件
- 18 行目 ?p1 と?p2 は同一人物ではない

このクエリを実行すると、執筆時点では、18 組の結果が得られました。うまく結果が得られたと思われるかもしれませんが、一つ落とし穴があるのです（人物だけでなく TOKIO や嵐などのグループ

が含まれているという問題もありますが、ここでは目を瞑ります)。

結果をよく見ると、次のような重複する組が存在しているのです。

p1	p2
http://ja.dbpedia.org/resource/城島茂	http://ja.dbpedia.org/resource/松岡昌宏
http://ja.dbpedia.org/resource/松岡昌宏	http://ja.dbpedia.org/resource/城島茂

SPARQL ではグラフパターンにマッチするデータが全て返されるので、このクエリの条件のみでは?p1 と?p2 の値が逆になった結果が含まれます。なお、SELECT DISTINCT で重複排除されるのは、あくまで結果の組に対してです。

```
FILTER NOT EXISTS {
  ?p2 <http://ja.dbpedia.org/property/production> <http://ja.dbpedia.org/resource/ジャニーズ事務所>
}
```

という条件を追加すればよいと思われるかもしれませんが、これではジャニーズ事務所所属タレント同士が共演する場合も取り除かれてしまいます。つまり、「ジャニーズ事務所所属タレントと 4 回以上共演したジャニーズ事務所所属タレントでない人」というお題になってしまいます。

これを排除するにはかなり工夫が必要で、詳しくは 37 本目をご覧ください。

37 本目　ジャニーズ事務所所属タレントと 4 回以上共演した人を取得する (重複排除)

Endpoint　https://ja.dbpedia.org/sparql

```
1  SELECT DISTINCT ?pp1 ?pp2
2  WHERE {
3    ?work1 <http://dbpedia.org/ontology/starring> ?p1 ;
4           <http://dbpedia.org/ontology/starring> ?p2 .
5    ?work2 <http://dbpedia.org/ontology/starring> ?p1 ;
6           <http://dbpedia.org/ontology/starring> ?p2 .
7    ?work3 <http://dbpedia.org/ontology/starring> ?p1 ;
8           <http://dbpedia.org/ontology/starring> ?p2 .
9    ?work4 <http://dbpedia.org/ontology/starring> ?p1 ;
10          <http://dbpedia.org/ontology/starring> ?p2 .
11   ?p1 <http://ja.dbpedia.org/property/production> <http://ja.dbpedia.org/resource/ジャニーズ事務所> .
12   FILTER (?work1 NOT IN (?work2, ?work3, ?work4))
13   FILTER (?work2 NOT IN (?work3, ?work4))
14   FILTER (?work3 != ?work4)
15   FILTER (?p1 != ?p2)
16   BIND (IF(STR(?p1) <= STR(?p2), ?p1, ?p2) AS ?pp1)
17   BIND (IF(STR(?p1) > STR(?p2), ?p1, ?p2) AS ?pp2)
18 }
```

36 本目の結果に存在していた重複する組を排除する条件を追加したクエリです。16,17 行目の条件がポイントです。この IF で?p1, ?p2 の順序を比較することで、同じ値は必ず同じ変数にバインドされるということを保証しています。`{?p1 := foo, ?p2 := bar}`あるいは`{?p1 := bar, ?p2 := foo}`が与えられたとき、必ず`{?pp1 := foo, ?pp2 := bar}`になるということです。このようにすれば、SELECT DISTINCT で重複が排除されます。

本題ではありませんが、?work1, ?work2, ?work3, ?work4 間で重複がないという条件をより短く書くことにも挑戦してみたのですが、12〜14 行目のように NOT IN を使う以上にうまい方法は思い付きませんでした。これでは 36 本目のように「!=」で全通り羅列しているのと大差ないですね。

第 4 章

様々な構文や関数を使って自在にクエリを組み立てよう！ 上級

これまでのノックを踏まえて応用的なテクニックを紹介します。対象とするデータセットの構造に合わせて、柔軟にクエリを組み立てる能力を養います。

38 本目　乱数を生成する (Virtuoso 用)

Endpoint 汎用

```
SELECT (bif:rnd(10, ?s, ?p, ?o) AS ?rand)
WHERE {
  ?s ?p ?o .
}
LIMIT 10
```

Virtuoso の独自関数 bif:rnd()[*1]を使って乱数を生成します。22 本目で扱った RAND 関数とは仕様が異なり、第一引数に指定した整数未満 0 以上の範囲で乱数を生成します。第一引数を指定しただけでは、22 本目と同様に同じ数値が出力されてしまいます。第二引数以降に?s, ?p, ?o を渡すことで、毎回異なる乱数が生成されるようになります。

どうしても RAND 関数を使いたい場合は、以下のように少しダーティなハックが必要です[*2]。

```
SELECT (RAND(1 + strlen(str(?s))*0) AS ?rand)
WHERE {
  ?s ?p ?o .
}
LIMIT 10
```

ただし、標準の RAND 関数は引数を取らないためこのクエリでも仕様に違反します。Virtuoso の実装では、RAND 関数の第一引数は double 型の数値をとり、その数値未満の乱数を生成するようです。Virtuoso は文書化されていない仕様が多くて困りますね…。

39 本目　SPARQL どうでしょう「サイコロの旅」

Endpoint https://ja.dbpedia.org/sparql

```
SELECT ?line ?stations
WHERE {
  <http://ja.dbpedia.org/resource/秋葉原駅> <http://dbpedia.org/ontology/
  servingRailwayLine> ?line .
```

[*1] http://docs.openlinksw.com/virtuoso/rndsalltr/

[*2] How to select random DBPedia nodes from SPARQL?, https://stackoverflow.com/questions/5677340/how-to-select-random-dbpedia-nodes-from-sparql

```
  ?stations <http://dbpedia.org/ontology/servingRailwayLine> ?line .
}
ORDER BY (RAND(1 + STRLEN(?stations)*0))
LIMIT 1
```

このクエリでは乱数を生成する RAND 関数を活用して、秋葉原駅から乗る路線とその路線での降車駅をランダムに選択する「サイコロの旅」を SPARQL で実現します。38 本目で紹介した方法で乱数を生成しています。

40 本目　パスの深さをカウントする

Endpoint https://ja.dbpedia.org/sparql

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?gchild (COUNT(?child)-1 AS ?depth)
WHERE {
  <http://ja.dbpedia.org/resource/Category:アニメ> ^skos:broader* ?child .
  ?child ^skos:broader* ?gchild .
}
GROUP BY ?gchild
ORDER BY ?depth
```

プロパティパスと GROUP BY をうまく使って述語をいくつ辿ったかをカウントすることができます。このクエリでは、カテゴリ「アニメ」以下のカテゴリとその深さを集計しています。下位カテゴリを取得する方法は 34 本目と同じです。

より簡単なデータで考えてみましょう。以下のグラフは、:baz →:bar →:foo と skos:broader でリンクしている構造です。

```
@prefix : <http://example.com/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

:baz skos:broader :bar .
:bar skos:broader :foo .
```

このとき、<http://example.com/foo>を起点に skos:broader の逆を辿ってみましょう。

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT *
WHERE {
  <http://example.com/foo> ^skos:broader* ?child .
  ?child ^skos:broader* ?gchild .
}
```

?child と?gchild を出力してみると以下のようになります。「*」は 0 以上の繰り返しですので、?child と?gchild には foo から終点のノードまでのノードの組み合わせが出力されます。これを?gchild で GROUP BY して COUNT すれば、foo からの深さとなります。

chilld	gchild
<http://example.com/foo>	<http://example.com/foo>
<http://example.com/foo>	<http://example.com/bar>
<http://example.com/foo>	<http://example.com/baz>
<http://example.com/bar>	<http://example.com/bar>
<http://example.com/bar>	<http://example.com/baz>
<http://example.com/baz>	<http://example.com/baz>

41 本目　31 本目のクエリ結果を都道府県毎にリスト化する

Endpoint https://ja.dbpedia.org/sparql

```
1  SELECT ?pref (COUNT(?mount) AS ?count)
2    (GROUP_CONCAT(?name; SEPARATOR=', ') AS ?mountList)
3  WHERE {
4    ?s a <http://dbpedia.org/ontology/AdministrativeRegion> ;
5       <http://www.w3.org/2000/01/rdf-schema#label> ?pref .
6    FILTER CONTAINS(?mountPref, STR(?pref))
7    {
8      SELECT ?mount ?name (GROUP_CONCAT(DISTINCT(?pref); SEPARATOR=', ') AS ?mountPref)
9      WHERE {
10       ?mount a <http://dbpedia.org/ontology/Mountain> ;
11          <http://dbpedia.org/ontology/address> ?address ;
12          <http://www.w3.org/2000/01/rdf-schema#label> ?name ;
13          ^<http://dbpedia.org/ontology/wikiPageWikiLink> <http://ja.dbpedia.org/
            resource/日本の山一覧> .
14       BIND (REPLACE(STR(REPLACE(?address, "([ (） ]|藤津郡太良町・)", "")), "(京都府|[都
         道府県]).*$", "$1") AS ?pref)
15       FILTER (REGEX(?pref, "[都道府県]"))
16       FILTER (!REGEX(?pref, "(同県|小県)"))
17     }
18     GROUP BY ?mount ?name
19     HAVING (COUNT(DISTINCT(?pref)) > 1)
20   }
21 }
22 GROUP BY ?pref
23 ORDER BY DESC(COUNT(?mount))
```

31 本目では、山ごとに集約を作って都道府県のリストを作っていました。このクエリは都道府県毎に集約し、その都道府県にまたがる山をリストにして出力します。

7 行目からのサブクエリでは 31 本目と同様に山毎の集約を作ります。外側のクエリは、

- 4,5 行目 都道府県を取得するパターン
- 6 行目 サブクエリで得られた山の所在都道府県のリスト (?mountPref) 中に都道府県名が存在するかを判定
- 18,19 行目 都道府県名で集約

という構成になっていて、サブクエリと外側のクエリで二重に集約を作っているところがポイント

です。

42 本目　動的に IRI を生成する

Endpoint https://ja.dbpedia.org/sparql

```
SELECT DISTINCT ?nameProp ?name
WHERE {
  ?location ?nameProp ?name .
  {
    SELECT DISTINCT (IRI(?prop) AS ?nameProp)
    WHERE {
      ?s <http://ja.dbpedia.org/property/subdivisionType> ?o .
      BIND (CONCAT(REPLACE(STR(?o), "resource", "property"), "名"^^xsd:string) AS ?prop
      )
    }
  }
}
```

動的に IRI を生成するというテクニックを紹介します。一般的にはどのような語彙があるか知った上でクエリを書きますが、存在するか分からないが存在するものだけ取得したい場合には、このテクニックが活用できます。

このクエリは、DBpedia から地名を取得したいが地名を目的語とする述語がよく分からない、というケースです。6〜9 行目のサブクエリで、地名を目的語に持ち得る述語の IRI を動的に生成しています (?nameProp)。3 行目ではそれを述語としたパターンを記述しています。

43 本目　スポーツを題材にしたアニメ作品を取得する

Endpoint https://ja.dbpedia.org/sparql

```
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpedia-owl:     <http://dbpedia.org/ontology/>
PREFIX dbpedia-ja:      <http://ja.dbpedia.org/resource/>

SELECT DISTINCT ?title ?category
WHERE {
  ?s dcterms:subject ?category, ?categories ;
     rdfs:label ?title .
  FILTER CONTAINS(STR(?categories), "アニメ")
  {
    SELECT ?category
    WHERE {
      {
        dbpedia-ja:スポーツ競技一覧 dbpedia-owl:wikiPageWikiLink/rdfs:label ?label .
        BIND (IRI(CONCAT("http://ja.dbpedia.org/resource/Category:"^^xsd:string, ?label
        , "アニメ"^^xsd:string)) AS ?category)
      } UNION {
        dbpedia-ja:スポーツ競技一覧 dbpedia-owl:wikiPageWikiLink/rdfs:label ?label .
        BIND (IRI(CONCAT("http://ja.dbpedia.org/resource/Category:男子"^^xsd:string, ?
```

```
        label, "アニメ"^^xsd:string)) AS ?category)
      } UNION {
        dbpedia-ja:スポーツ競技一覧 dbpedia-owl:wikiPageWikiLink/rdfs:label ?label .
        BIND (IRI(CONCAT("http://ja.dbpedia.org/resource/Category:女子"^^xsd:string, ?
        label, "アニメ"^^xsd:string)) AS ?category)
      } UNION {
        dbpedia-ja:スポーツ競技一覧 dbpedia-owl:wikiPageWikiLink/rdfs:label ?label .
        BIND (IRI(CONCAT("http://ja.dbpedia.org/resource/Category:"^^xsd:string, ?label
        , "を題材とした作品"^^xsd:string)) AS ?category)
      } UNION {
        dbpedia-ja:スポーツ競技一覧 dbpedia-owl:wikiPageWikiLink/rdfs:label ?label .
        BIND (IRI(CONCAT("http://ja.dbpedia.org/resource/Category:男子"^^xsd:string, ?
        label, "を題材とした作品"^^xsd:string)) AS ?category)
      } UNION {
        dbpedia-ja:スポーツ競技一覧 dbpedia-owl:wikiPageWikiLink/rdfs:label ?label .
        BIND (IRI(CONCAT("http://ja.dbpedia.org/resource/Category:女子"^^xsd:string, ?
        label, "を題材とした作品"^^xsd:string)) AS ?category)
      }
    }
  }
}
```

長いクエリですが、同じようなグラフパターンを UNION で繋げているだけなので、そこまで難しいクエリではありません。12 行目からのサブクエリの WHERE でやっていることは、42 本目のテクニックを使って、スポーツを題材にしたアニメ作品に該当しそうなカテゴリの IRI を動的に生成しています。以下のようなテンプレートで (6× スポーツ競技の名称 (?label) の数) 通りのカテゴリの IRI が作成されます。

```
http://ja.dbpedia.org/resource/Category:(|男子|女子){?label}(アニメ|を題材とした作品)
```

ちなみに CONCAT 関数の引数のリテラルをそれぞれ^^xsd:string の型付きリテラルにしていますが、通常のリテラルにすると正しく動きません。原因は調査中なのですが、Virtuoso の文字列操作関数の実装に何らかの問題があると思われます。

そして、8〜10 行目では生成した IRI を使ってアニメ作品のリソースを絞り込んでいます。この点は 14 本目と同じやり方です。

44 本目　複数のパターンの直積によって IRI を生成する

Endpoint `https://ja.dbpedia.org/sparql`

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>

SELECT (IRI(
    CONCAT("http://ja.dbpedia.org/resource/Category:", ?x, ?label, ?y)
  ) AS ?category)
WHERE {
  <http://ja.dbpedia.org/resource/スポーツ競技一覧> dbpedia-owl:wikiPageWikiLink/rdfs:
  label ?label .
  VALUES ?x {"" "男子" "女子"}
```

```
10   VALUES ?y {"アニメ" "を題材とした作品"}
11 }
```

43 本目を改良して、長ったらしい UNION を使わずに様々な IRI のパターンを生成する方法を考えます。43 本目で 6 通りのグラフパターンを UNION していたのは、スポーツ競技名称の先頭に「」「男子」「女子」のいずれかが付き、末尾に「アニメ」「を題材とした作品」が付くカテゴリ名を生成することが目的でした。これらのパターンは{"", "男子", "女子"}と{"アニメ", "を題材とした作品"}という 2 つの集合の直積を考えれば、より短く書くことができます。

9 行目の?x は生成するカテゴリ名のプレフィックスにあたり、10 行目の y はカテゴリ名のサフィックスにあたる文字列をバインドしています。直積を作り出すために 2 つの VALUES で定義していることに注意してください。単に 2 つの変数の組に値をバインドする `VALUES (?x ?y) {("foo" "bar") ...}` の構文とは異なります。そして 5 行目で?x, ?label, ?y を結合してカテゴリの IRI を動的に生成しています。

次のノックでは、この方法で生成したカテゴリの IRI を使ってアニメ作品を抽出しましょう (☞45 本目)。

45 本目　スポーツを題材にしたアニメ作品を取得する (44 本目の方法を利用)

Endpoint `https://ja.dbpedia.org/sparql`

```
1  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2  PREFIX dbpedia-owl:     <http://dbpedia.org/ontology/>
3  SELECT DISTINCT ?title ?category
4  WHERE {
5    ?s <http://purl.org/dc/terms/subject> ?category, ?categories ;
6       rdfs:label ?title .
7       FILTER CONTAINS(STR(?categories), "アニメ")
8    {
9      SELECT (IRI(
10        CONCAT("http://ja.dbpedia.org/resource/Category:", ?x, ?label, ?y)
11      ) AS ?category)
12     WHERE {
13       <http://ja.dbpedia.org/resource/スポーツ競技一覧> dbpedia-owl:wikiPageWikiLink/
         rdfs:label ?label .
14       VALUES ?x {"" "男子" "女子"}
15       VALUES ?y {"アニメ" "を題材とした作品"}
16     }
17   }
18 }
```

サブクエリの中身は 44 本目と同じです。それでは実行してみましょう。

…ところが、このクエリを実行しようとすると DBpedia Japanese のエンドポイントでは、

```
Virtuoso 42000 Error
The estimated execution time 76656 (sec) exceeds the limit of 500 (sec).
```

と怒られてしまいます。

推定実行時間が制限 (500 秒) を超えてしまうため実行してもらえませんでした。動的に生成される IRI は 1,620 個ほどなので、実行時間の制限を引き上げたエンドポイントを用意すれば、現実的な時間

で結果は得られると思われます。実行計画を分析して、推定実行時間が収まるようにクエリを最適化していく方法もありますが、このノックの範囲を超えるので割愛いたします。

第 5 章

職人級

君はこの魔クエリを受け止められるかな？
これで君も SPARQL 職人だ！

実用性度外視のクエリも含む番外編です。

46 本目　Virtuoso Server の情報を取得する

Endpoint　OpenLink Virtuoso

```
SELECT
  (bif:sys_stat('st_dbms_name')          AS ?name)
  (bif:sys_stat('st_dbms_ver')           AS ?version)
  (bif:sys_stat('st_build_date')         AS ?date)
  (bif:sys_stat('st_build_thread_model') AS ?thread)
  (bif:sys_stat('st_build_opsys_id')     AS ?opsys)
WHERE {?s ?p ?o}
LIMIT 1
```

エンドポイントの Virtuoso Server に関する情報を bif:sys_stat 関数[*1]で取得します。クエリ自体は全く難しいものではありませんが、ユースケースを考えると職人向けかなと思い、ここで紹介しておきました。エンドポイントの外形監視のためにこのクエリを送信して情報取得したり、Virtuoso のバージョンなどによって条件分岐が必要なクエリで使ったりすることが考えられます。

47 本目　Fizz Buzz

Endpoint　https://ja.dbpedia.org/sparql

```
SELECT ?out
WHERE {
  ?s <http://www.w3.org/2000/01/rdf-schema#label> ?label .
  BIND (xsd:integer(?label) AS ?num)
  FILTER (REGEX(?label, "^\\d+$") && ?num > 0)
  BIND (CONCAT(IF(?num / 3 * 3 = ?num, "Fizz", ""), IF(?num / 5 * 5 = ?num, "Buzz",
  "")) AS ?str)
  BIND (IF(STRLEN(?str) = 0, ?num, ?str) AS ?out)
}
```

[*1] http://docs.openlinksw.com/virtuoso/fn_sys_stat/
Interrogating attributes of a Virtuoso Server instance via SPARQL,
http://vos.openlinksw.com/owiki/wiki/VOS/VirtCheckSvrVersionViaSparql

```
GROUP BY ?num
ORDER BY ASC(?num)
LIMIT 100
```

Fizz Buzz はご存知の通り、1 から順に数を数え、3 の倍数なら「Fizz」、5 の倍数なら「Buzz」、3 と 5 の倍数なら「FizzBuzz」と言う言葉遊びです。これをプログラムで実装することで、プログラマを試験する方法としてもよく知られています。この本を読んだあなたなら、就活の選考で「君、SPARQL で Fizz Buzz 書いてみてよ」と言われても焦らずに済むでしょう！

クエリ自体は大して難しくないのですが、連番を生成する方法が SPARQL 標準の機能では思い付かなかったため (Virtuoso の独自関数を使うと実現できます ☞48 本目)、DBpedia から整数のラベルを引っ張ってきています。また、SPARQL は剰余演算子がないので、同じ数で割って掛けた値が元の値と等しいかどうかで判定しています。

48 本目　連番を生成する

Endpoint　OpenLink Virtuoso

```
SELECT (bif:sequence_next('my_unique_key', 1, ?s, ?p, ?o) AS ?id) # 毎回実行
       ?call_once
WHERE {
  ?s ?p ?o .
  BIND (bif:sequence_set('my_unique_key', 1, 0) AS ?call_once) # 一度だけ実行
}
LIMIT 100
```

Virtuoso の独自関数 sequence_set[*2]、sequence_next[*3]を使うことで、連番を生成できます。sequence_set 関数でシーケンスオブジェクトを初期化し、sequence_next で値を取得&インクリメントします。しかし、Virtuoso のオプティマイザは、静的な値しか引数に持たない関数は一度しか実行せず、すべて同じ値を返すような挙動をします。そのため、オプティマイザをうまく騙す必要があります。

sequence_set 関数は一度実行されればよく、WHERE 句内で実行して変数に結果をバインドしておきます（5 行目）。ただし、バインドした変数?call_once をどこかで参照しておかないと最適化されて sequence_set 関数は一度も実行されないので、SELECT に含めるようにします。次に、sequence_next 関数は本来 2 つの引数をとり、`bif:sequence_next('my_unique_key', 1)` と書くことができます。LIMIT で指定した 100 回実行してほしいのですが、これでは最適化されて一度しか実行されません。そこで、Virtuoso の独自関数は余分な引数を無視することを利用し、?s、?p、?o を引数に渡すことで必ず毎回実行されるようにします。

しかしながら、このシーケンスオブジェクトはサーバでグローバルに共有されますし、単一のクエリ内で連番を生成することを意図した機能ではないと思われます。オプティマイザ回避術もバッドノウハウの塊なので、あまり使おうと思わないほうが良いでしょう…。

[*2] http://docs.openlinksw.com/virtuoso/fn_sequence_set/
[*3] http://docs.openlinksw.com/virtuoso/fn_sequence_next/

49 本目　クエリを実行するクエリ

Endpoint OpenLink Virtuoso

```
1  SELECT ?sparql ?sql ?exec ?state ?message (bif:length(?rows) AS ?length)
2    (bif:aref(bif:aref(?rows, 0), 0) AS ?concept0) (bif:aref(bif:aref(?rows, 0), 1) AS ?
     label0)
3    (bif:aref(bif:aref(?rows, 1), 0) AS ?concept1) (bif:aref(bif:aref(?rows, 1), 1) AS ?
     label1)
4    (bif:aref(bif:aref(?rows, 2), 0) AS ?concept2) (bif:aref(bif:aref(?rows, 2), 1) AS ?
     label2)
5  WHERE {
6    ?s ?p ?o .
7    BIND ('SELECT * WHERE {?concept a skos:Concept; rdfs:label ?label}' AS ?sparql)
8    BIND (STR(bif:sparql_to_sql_text(?sparql)) AS ?sql)
9    BIND ("" AS ?state)
10   BIND ("no error" AS ?message)
11   BIND (bif:vector() AS ?meta)
12   BIND (bif:vector() AS ?rows)
13   BIND (bif:exec(?sql, ?state, ?message, bif:vector(), 3, ?meta, ?rows) AS ?exec)
14 } LIMIT 1
```

クエリ内の文字列リテラルを SPARQL クエリとして評価し、その結果を取得します。Virtuoso の独自関数を多用していますので、順に説明します。

- 7 行目 SPARQL クエリとして実行する文字列を?sparql にバインド
- 8 行目 bif:sparql_to_sql_text 関数[*4]を使い SPARQL を SQL へ変換し?sql へバインド
  (補足) Virtuoso は RDBMS であり RDF データも RDB にストアされています[*5]。SPARQL クエリは内部的には SQL に変換することで実行しています。
- 9 行目〜12 行目 bif:exec 関数[*6]に渡す引数のお膳立てです（詳細はドキュメントを参照）。
- 13 行目 bif:exec 関数で?sql を実行する。実行結果は?rows に格納される。
- 2〜4 行目 検索結果の配列から?concept と?label の値を取り出す。

?rows は Virtuoso の vector 型なので、bif:aref[*7]などの配列操作関数を使う必要があります。

50 本目　クワイン (Quine)

Endpoint https://ja.dbpedia.org/sparql

```
1  SELECT (REPLACE(REPLACE(?s, SUBSTR(?etc, 216, 1), SUBSTR(?etc, 245, 1)), SUBSTR(?etc,
   74, 1), ?s) AS ?query) WHERE { <http://ja.dbpedia.org/resource/D.C.P.S.
   _〜ダ・カーポ〜_プラスシチュエーション> <http://ja.dbpedia.org/property/etc> ?etc . }
   GROUP BY ('SELECT (REPLACE(REPLACE(?s, SUBSTR(?etc, 216, 1), SUBSTR(?etc, 245, 1)),
```

[*4] http://docs.openlinksw.com/virtuoso/fn_sparql_to_sql_text/
[*5] 16.1.3. RDF_QUAD and other tables - OpenLink Documentation, http://docs.openlinksw.com/virtuoso/rdfquadtables/
[*6] http://docs.openlinksw.com/virtuoso/fn_exec/
[*7] http://docs.openlinksw.com/virtuoso/fn_aref/

```
SUBSTR(?etc, 74, 1), ?s) AS ?query) WHERE { <http://ja.dbpedia.org/resource/D.C.P.S.
_～ダ・カーポ～_プラスシチュエーション> <http://ja.dbpedia.org/property/etc> ?etc . }
GROUP BY ("&" AS ?s)' AS ?s)
```

一見してクエリの整形ミスを疑われるかもしれませんが、そうではないのでご安心を。まず、クワイン (Quine) とは何かご存知でしょうか。簡単に説明すると、クワインとは自身のソースコードと全く同じ文字列を出力するプログラムのことです。様々なプログラミング言語で一種の娯楽としてクワインが書かれています。SPARQL のクワインについては、私が Google などで検索する限りでは見つかりませんでしたので、挑戦してみた結果がこのクエリです。なお、SPARQL は問い合わせ言語ですので、クエリ文字列とその実行結果であるソリューションの文字列が完全に一致すればクワインであることにします。

参考として、SQL によるクワインの例を見てみましょう (英語版 Wikipedia より[*8])。

```
1 SELECT REPLACE(REPLACE('SELECT REPLACE(REPLACE("$",CHAR(34),CHAR(39)),CHAR(36),"$") AS
Quine',CHAR(34),CHAR(39)),CHAR(36),'SELECT REPLACE(REPLACE("$",CHAR(34),CHAR(39)),CHAR
(36),"$") AS Quine') AS Quine
```

REPLACE がいくつも書かれていると思いますが、これはリテラルを囲うシングルクォートがリテラル内に出現しないようにしながら、自分自身の文字列となるようにうまく置換をしているのです。ちなみに、`CHAR(34)`、`CHAR(36)`、`CHAR(39)` はそれぞれダブルクォート (")、ドル記号 ($)、シングルクォート (') です。

基本的な戦略はこの SQL のクワインと全く同じです。SPARQL のリテラルはダブルクォートまたはシングルクォートで囲まれますので、これをいかにうまく置換してやるかが要となります。SQL の CHAR 関数と同等の関数が SPARQL にあるならば簡単なのですが、別の方法でエンコードすることを考えます。サンプルクエリでは、DBpedia Japanese からシングル/ダブルクォートを含む適当な文字列 (?etc) を取ってきて、それを参照して REPLACE することで実現しています。SUBSTR で必要な文字を取り出していて、`SUBSTR(?etc, 216, 1)`、`SUBSTR(?etc, 245, 1)`、`SUBSTR(?etc, 74, 1)` はそれぞれダブルクォート (")、アンパサンド (&) です。なお、GROUP BY を使っていますが、変数?s へ文字列をバインドするためで、集約としての意味はありません。あとはこれらを使ってうまく REPLACE していけば完成です。

しかし、このサンプルクエリには重大な欠陥があることを指摘しておかねばなりません。外部リソースを参照するという禁じ手を使ってしまっているので厳密にはルール違反です。また、誰かが当該箇所を書き換えてしまうと、クワインとして成立しなくなるかもしれません。ただ、Web 上のリソースを活用するというのはいかにも SPARQL らしくて良いのではないでしょうか（言い訳）。もっと良いクワインを考案された方はぜひ教えてください。

[*8] “Quine (computing)”, `https://en.wikipedia.org/wiki/Quine_%28computing%29`

SPARQL 50 本ノック

2018 年 12 月 31 日　初版
2019 年　7 月　7 日　第 2 版
2022 年　9 月 23 日　第 3 版

著　　者　ばびぶべぼん
協　　力　TKB 大学 M 研究室の SPARQLer
発　　行　ばびぶべぼ研究室
lab@babibubebo.org
https://babibubebo.org/lab/

ISDN 278-4-663011-08-2

2784663011082

ばびぶべぼ研究室