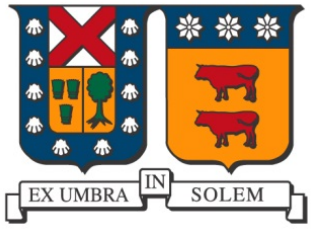
2020-08

RESPUESTAS MAXIMALES EN SPARQL

CRUZ CORVALAN, JUAN PABLO

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“RESPUESTAS MAXIMALES EN SPARQL”

JUAN PABLO CRUZ CORVALÁN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Carlos Buil Aranda
Profesor Correferente: Hernán Vargas

Agosto - 2020

RESUMEN

Resumen— En este trabajo se plantea un algoritmo para obtener un mayor número de resultados y/o consultas alternativas cuando se realiza una consulta SPARQL y se debe afrontar el problema de obtener resultados vacíos, esto con el objetivo de apoyar al usuario cuando este no puede obtener los datos que busca. Para diseñar este algoritmo, se analizaron algunas posibles causas y maneras de solucionarlas por medio de modificaciones realizadas a la consulta original, a la vez que se intenta que las modificaciones no se alejen demasiado de lo esperado por el usuario. Finalmente se implementa parcialmente este algoritmo dentro del software RDFExplorer, agregando la caracteristica de obtener conjuntos de URIs que pueden ser utilizadas para obtener resultados, como para mejorar la comprención de los datos por parte del usuario.

Palabras Clave— Consultas SPARQL; Resultados vacíos; Relajación de consultas; Bases de datos RDF

ABSTRACT

***Abstract*—** In this work, an algorithm is proposed to obtain a greater number of results or alternative queries when a SPARQL query is made and the problem of obtaining empty results must be faced, this with the aim of supporting the user when he cannot obtain the data he is looking for. To design this algorithm, some possible causes and ways of solving them were analyzed by means of modifications made to the original query, while trying to ensure that the modifications do not stray too far from what the user expected. Finally, this algorithm is partially implemented within the RDFExplorer software, adding the characteristic of obtaining sets of URIs that can be used to obtain results, such as to improve the understanding of the data by the user.

***Keywords*—** SPARQL queries; Empty answers; Query relaxation; RDF database

GLOSARIO

BGP: Basic Graph Pattern.
DL: Description Logic.
OWL: Web Ontology Language.
RDF: Resource Description Framework.
RDFS: RDF Schema.
URI: Uniform Resource Identifier.
VQL: Visual Query Language.
VQG: Visual Query Graph.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Las bases de datos resultan ser una parte fundamental en todos los sistemas que hacen uso de datos, por lo que son utilizadas en una cantidad enorme de variados dominios y usadas por todo tipo de usuarios, desde expertos hasta usuarios sin ningún conocimiento del dominio. De igual manera, contar con un mecanismo para poder acceder a los datos almacenados que se deseen resulta de igual o mayor importancia, si no se puede acceder a los datos, estos pierden su utilidad.

Actualmente, en el ámbito de la web semántica se hace uso de bases de datos RDF para almacenar y gestionar la enorme cantidad de data en la web, junto con el uso de SPARQL como método de consulta a estas. Debido a que se puede llegar a tener enormes cantidades de datos almacenados y de dominios totalmente diferentes, esto causa que resulte mucho más facil que ocurra que una consulta SPARQL no obtenga resultados, mas aún cuando es realizada por un usuario con poco conocimiento de los datos. Esto genera que sea algo común que se realizen consultas que no obtienen resultados, provocando que estas consultas no tengan ninguna utilidad y solo resulten ser un gasto de tiempo y recursos.

En este documento se presenta un algoritmo para abarcar el problema de no obtener resultados al consultar una base de datos RDF. En particular se le da un enfoque a la obtención de consultas que sean similares y que sí obtengan resultados. Una vez definido el algoritmo, este se implementa parcialmente en el software RDFExplorer, con la intención de comprobar la efectividad del algoritmo y además de extender las funcionalidades del mismo RDFExplorer, permitiendo obtener consultas alternativas que sirven tanto para incrementar el número de resultados como para permitir comprender un poco la razón del porqué de que no se obtuviesen resultados originalmente.

El presente documento está estructurado de la siguiente forma:

- **Capítulo 1 Definición del problema:** Se identifica y describe el problema general y su situación actual. Finaliza con los objetivos que se abarcan.
- **Capítulo 2 Marco conceptual:** Se presenta el marco conceptual de las tecnologías relacionadas y el estado del arte relacionado al mejorar los resultados de las consultas SPARQL.
- **Capítulo 3 Propuesta de solución:** Se presenta la propuesta de solución, los elementos considerados en esta y el algoritmo final desarrollado para enfrentar el problema.
- **Capítulo 4 Validación de la solución:** Se detalla la implementación de la solución en el entorno de un software específico, seguido de una validación de esta mediante un análisis de los resultados obtenidos.
- **Capítulo 5 Conclusiones:** Se presentan las conclusiones del trabajo finalizado junto al trabajo que queda abierto para mejorar a futuro.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

1.1. Contexto y situación actual

Actualmente proyectos como la web semántica han motivado la construcción de bases de conocimiento que puedan ser entendidas por máquinas para gestionar la data existente en la web. Para hacer esto, se ha propuesto el uso del modelo de metadata RDF como un formato estandar para relacionar y almacenar los datos.

Es por esto que cada vez son más los datos que se almacenan con RDF en la web, lo que provoca una mayor necesidad de hacer consultas a bases de datos RDF con la intención de obtener los datos que se necesitan. La utilización de RDF en muchas y diversas áreas permite que estas se puedan unir debido a la flexibilidad que presenta y a su representación genérica, pudiendo soportar desde datos estructuros hasta semiestructurados.

Pero esta misma flexibilidad puede provocar una dificultad para el usuario de formular correctamente las consultas que retornan los resultados deseados. Por otro lado, para poder obtener los datos deseados, el usuario debe realizar una consulta utilizando SPARQL, lenguaje que puede llegar a ser complicado de utilizar en un principio, más aun considerando la dificultad de tener que entender el grafo RDF.

Por lo anterior, al momento de que un usuario realiza una consulta SPARQL, puede ocurrir que debido a que no se encontraron coincidencias en la base de datos, y a la falta de flexibilidad para considerar resultado que no sean exactos pero que podrían ser útiles, esta consulta finalmente no retorne resultados.

Esto implica, además de una frustración por parte del usuario, que se deberá modificar la consulta manualmente en una forma de prueba y error sin recibir ni la más mínima pista de la causa del problema. Esta modificación se deberá realizar hasta que se logren obtener los resultados esperados, traduciéndoce en una gran cantidad de recursos pedidos como lo es el tiempo. Esto puede llevar a, por un lado, un mal uso de recursos como también puede llegar a ser un desincentivo para el usuario de seguir utilizando SPARQL y las tecnologías relacionadas. Lo previamente mencionado se muestra en la figura 1.

Es por esto que se busca una forma de obtener resultados que no sean vacíos al momento en que una consulta no obtenga coincidencias, de forma tal que la nueva consulta que genere estos resultados se aproxime lo más posible a la consulta original, otorgando así, un mínimo de información que puede resultar de utilidad para el usuario que generó la consulta original.

Además, se tiene que la complejidad computacional de una consulta SPARQL puede variar según sean los operadores utilizados para su construcción. Esto es algo que se debe tener

en cuenta al momento de crear las consultas, por lo que resulta importante tener presente siempre la complejidad de esta, ya que una solución al problema descrito puede generar consultas alternativas con una alta complejidad, llegando a ser incluso ineficiente.

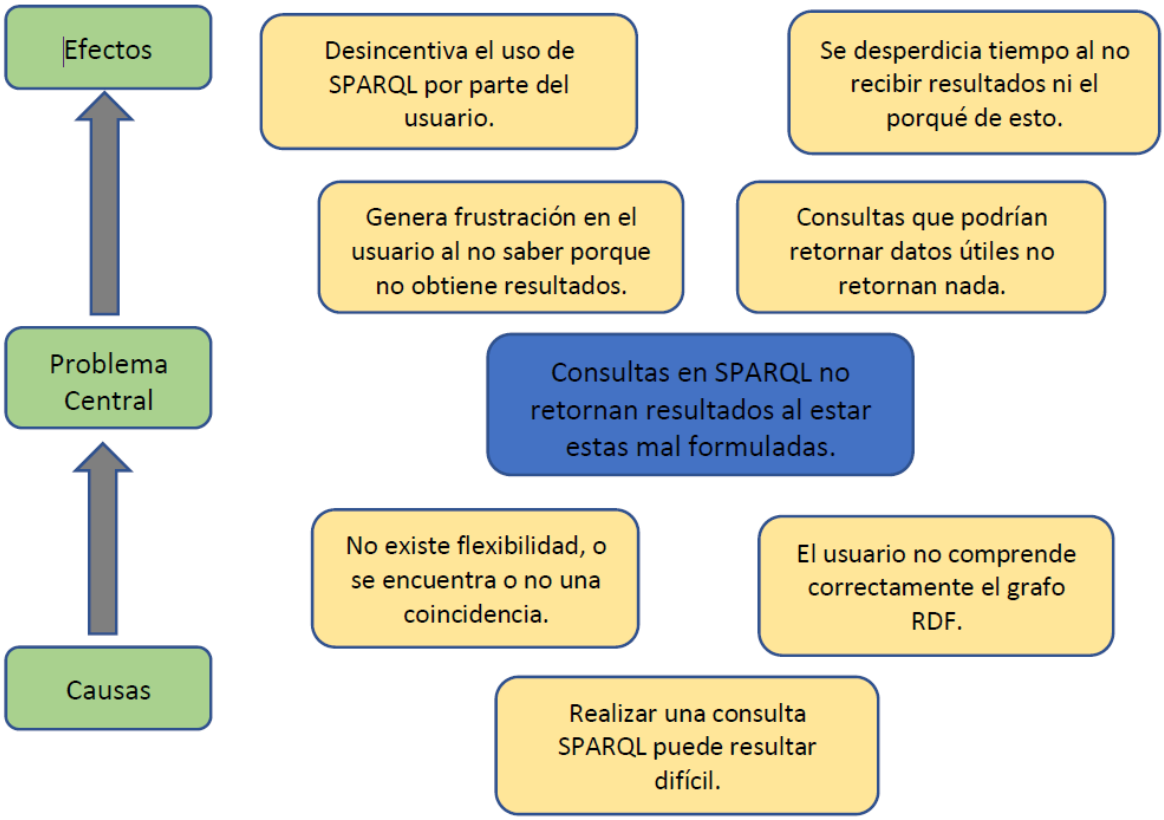


Figura 1: Árbol del problema.
Fuente: Elaboración propia.

En [Saleem *et al.*, 2015] se realiza un estudio con 5.675.204 consultas realizadas en 4 *end-points* distintos en lapsos de tiempos de 8 meses, 6 meses, 1 mes y 2 meses para cada uno de estos. En este estudio se muestra que, de estas 5.675.204 consultas, 374.578 retornan cero resultados, aproximadamente 6,6 % del total, el cual es un porcentaje no menor.

Si se logra solucionar este problema, se espera generar una mejora en los usos de los recursos, en especial del tiempo debido a que se entregaran resultados que, si bien no tienen una coincidencia exacta a la consulta original, sí pueden tener una utilidad para el usuario que obtendrá algo de regreso en lugar de simplemente nada, incluso llegando a ser suficiente para lo que se necesitaban y reduciendo el tiempo que de otro modo, se hubiese debido utilizar para reescribir la consulta. Además, se generaría una mejor sensación por parte del usuario al obtener algún tipo de respuesta, lo que no generaría frustración e incentivaría el uso de SPARQL y, por consiguiente, las tecnologías relacionadas a esta.

1.2. Objetivos

1.2.1. Objetivo General

Construir un algoritmo y heurísticas que permita encontrar el subconjunto máximo de una consulta SPARQL, para que se retorne resultados en el caso en que la consulta original no obtenga resultados.

1.2.2. Obetivos Específicos

- Diseñar un algoritmo para proponer consulta(s) alternativa(s) a la consulta SPARQL original con tal de obtener resultados no nulos.
- Implementar el algoritmo que genere el subconjunto máximo de consultas SPARQL.
- Analizar la complejidad de las consultas-soluciones alternativas generadas del subconjunto máximo de la consulta original.

CAPÍTULO 2
MARCO CONCEPTUAL

2.1. Web semántica

La web actual es una web basada en documentos interconectados. Además, solo es entendible por humanos, dificultando enormemente el procesamiento de información automático. Es por esto por lo que se plantea la web semántica, una extensión de la web actual que le otorga un significado bien definido a la información, permitiendo un mejor trabajo cooperativo entre computadores y personas [Berners-Lee *et al.*, 2001]. Esto a través de un conjunto de estándares y tecnologías que permiten la creación de una web de datos, la que permitiría que fuese entendible tanto por humanos como para máquinas, facilitando la creación de programas que procesen información en la web.

La web semántica se trata sobre dos cosas. Es sobre el formato común para la integración y combinación de datos provenientes de diversas fuentes, mientras la web original se concentra principalmente en el intercambio de documentos. Además, es sobre lenguajes para almacenar el cómo se relaciona la data con los objetos del mundo real. De esta forma, permitiendo a personas o a máquinas, empezar sobre una base de datos y luego moverse a través de un conjunto sinfín de base de datos que están conectadas por tener temas en común. [w3c, 2013]

Para poder llevar esto a cabo, se requiere contar con estándares que sirvan para las diferentes necesidades. Estos estándares se organizan en forma de una pirámide como se muestra en la figura 2.

- URI: "*Uniform Resource Identifier*" es el identificar único que permite localizar un recurso en la web.
- Unicode: Es el estándar para la codificación de los símbolos en los textos.
- XML: Tecnologías utilizadas para dotar de sintaxis a los documentos de forma de generar un formato común para el intercambio de documentos.
- RDF: Modelo de datos utilizado para representar los recursos y las relaciones existentes entre estos.
- RDF-S: Vocabulario sobre RDF que permite tener una semántica claramente definida relacionada a modelos de objetos, a través de propiedades y clases.
- OWL: Lenguaje que permite definir ontologías a través de una descripción más detallada de propiedades y clases, relaciones entre estas, cardinalidad, etc.

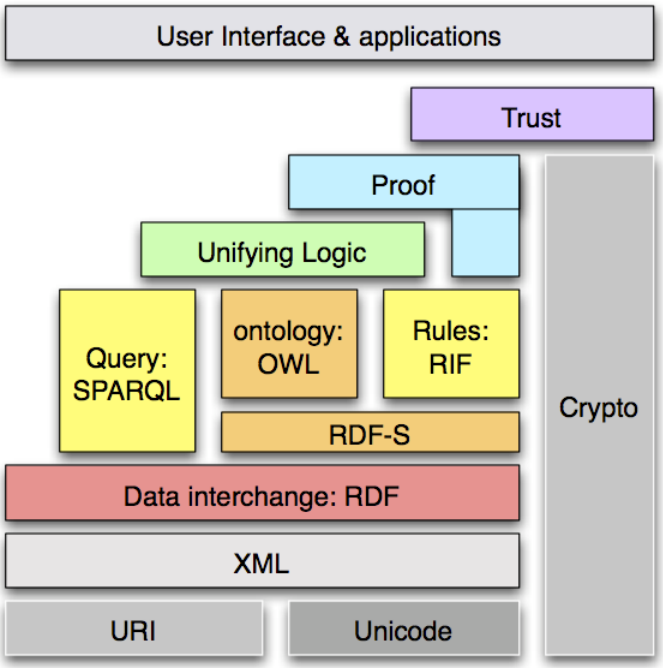


Figura 2: Estándares de la web semántica.
Fuente: W3C.

- SPARQL: Lenguaje para hacer consultas a conjuntos de datos RDF.

A continuación, se entrará en mayor detalle en alguno de estos componentes.

2.2. RDF y RDF Schema

2.2.1. RDF

El modelo de datos *Resource Description Framework* (RDF) es un estándar utilizado para la representación de la información de los recursos existentes en la web. RDF puede ser utilizado para representar, entre otras cosas, información personal, redes sociales, metadata, como también para proveer una forma de integración sobre fuentes de información diferentes, facilitando la unión de data que puede tener diferentes esquemas.

RDF extiende la estructura enlazada de la web a través del uso de URIs para nombrar relaciones entre elementos como también para nombrar a estos elementos. Usando este simple modelo, se permite que data estructurada como semiestructurada se pueda mezclar, exponer, y compartir a través de diferentes aplicaciones. [w3c, 2014b]

Una URI o *Uniform Resource Identifier* es un identificador de un recurso que se encuentra en la web. Un ejemplo de URI es “http://www.example.org/languaje”, que representa al recurso ”*languaje*”. Por otro lado, se tiene el literal, un dato primitivo que es un valor especifico que no cuenta con una URI. Este primer elemento puede ser utilizado para crear conexiones entre dos de estos mismos elementos.[w3c, 2014a]

La conexión entre 2 elementos mencionada anteriormente se conoce como *triple* y resulta ser uno de los conceptos fundamentales de RDF. Un *triple* tiene la forma (s, p, o) o sujeto, predicado, objeto. Esto equivale a tener 2 nodos, que representan al sujeto y al objeto, conectados por 1 arco, el cual representa el predicado. Así, al tener un conjunto de triples, se obtiene un grafo RDF, el que resulta ser un *Edge-labelled graph*, en donde cada arco tiene una etiqueta que representa la relación entre dos elementos. En la figura 3 se puede observar un ejemplo de un *Edge-labelled graph*.

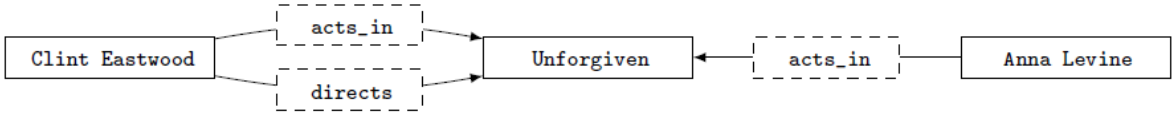


Figura 3: Ejemplo de *Edge-labelled graph*.
Fuente: [Angles *et al.*, 2017].

Más formalmente, un *triple* es una tupla (s, p, o) $\in (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$, donde **I**, **L** y **B** son conjuntos infinitos disjuntos que contienen respectivamente URIs, *blank nodes* y literales. Un *blank node* es un nodo auxiliar que no representa ningún recurso y que es utilizado solamente con una función estructural dentro de un grafo RDF.

Un triple puede establecer la relación existente entre dos recursos, o puede ser utilizado para dar un valor especifico (un literal) a uno de los atributos de un recurso. Un ejemplo de ambos casos se muestra a continuación.

- Relación entre dos recursos:

  En este caso, se cuenta con dos recursos y sus respectivas URIs y se intenta establecer una relación entre ambos recursos a través de un tercer recurso que la representa.

  En la figura 4 se tienen los recursos Alice y Bob, y ambos se relacionan mediante el recurso Amigo, generando la conexión de que Alice es amigo de Bob.

- Valor específico a un atributo de un recurso:

  En este caso, se cuenta con un recurso el cual puede tener atributos, los que resultan ser también recursos, y cada uno de estos atributos debe tener un valor concreto.

  En la figura 5 se tiene el recurso *noticia*, que cuenta con los atributos *título* y *fecha*, donde cada uno de estos tiene un valor definido.

Figura 4: Ejemplo de relación entre recursos.
Fuente: Elaboración propia.

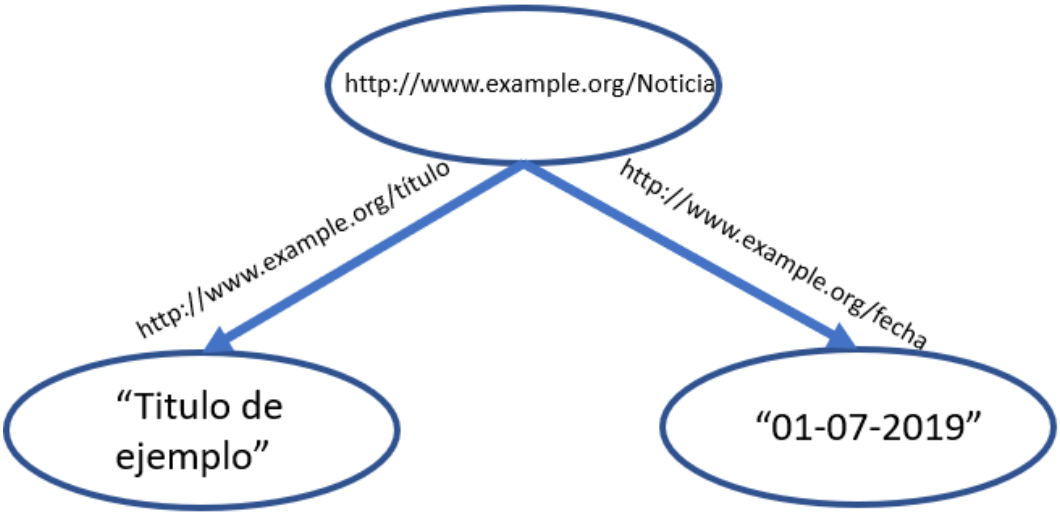


Figura 5: Ejemplo de valor específico a un atributo de un recurso.
Fuente: Elaboración propia.

Para escribir las relaciones del grafo como tuplas, se puede utilizar la notación *turtle*, la que además aporta la posibilidad de usar prefijos para abreviar las URIs. Por ejemplo, el grafo de la figura 5 en formato *turtle* se puede escribir de la siguiente forma:

@prefix ex: <http://www.example.org/>.

ex:Noticia ex:titulo “Título de ejemplo” .

ex:Noticia ex:fecha “01-07-2019”.

Hasta el momento los ejemplos utilizados de literales se han escrito como solamente una cadena de caracteres, pero en RDF estos cuentan con un *datatype*, el cual debe ser señalado de manera explícita. Para señalar el *datatype* de un literal se debe especificar el valor del literal al interior de comillas seguido de $\wedge\wedge$ y finalmente de la URI del *datatype*. La URI de los *datatype* hace uso de XML Schema, por lo que para hacer uso de prefijos para abreviar estas URIs se debe utilizar:

@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

Ejemplos de literales con los tipos más comunes:

- Para cadenas de textos: *"String de ejemplo"*$\wedge\wedge$*xsd:string*
- Para números enteros : *"103231"*$\wedge\wedge$*xsd:integer*
- Para fechas: *"2019-09-09"*$\wedge\wedge$*xsd:date*

Cuando un literal no tiene explícitamente un *datatype*, este es simplemente considerado como un valor sin tipo. En el caso de estos literales, se les puede incluir información del lenguaje utilizando el símbolo @ de la forma siguiente:

"Text in English" @en

De esta forma, se puede contar con diferentes valores dependiendo del lenguaje en que se quiera utilizar. Por ejemplo, el titulo de una noticia se podría tener tanto en español como en ingles de la siguiente forma:

ex:Noticia ex:titulo "Título de ejemplo" @es .

ex:Noticia ex:titulo "Example title" @en

2.2.2. RDF Schema

RDF Schema (RDFS) es una extensión semántica de RDF que provee mecanismos para especificar clases y propiedades, instancias de clases y organizar estas clases de forma jerárquica.

En RDF, se cuenta con la posibilidad de indicar que un recurso es de un tipo determinado utilizando el predicado rdf:type, esto vendría a ser una instancia de una clase en particular, por esto, RDFS permite crear clases utilizando el objeto rdfs:Class e indicando que el sujeto es de este tipo. Permitiendo así representar cualquier categoría que se desee. Por ejemplo, para especificar la clase Documento sería así:

ex:Documento rdf:type rdfs:Class

y para definir una instancia de esta clase, se usa un *triple* de la forma siguiente:

ex:docEjemplo rdf:type ex:Documento

Dentro del vocabulario de RDF y RDFS existen algunas clases predefinidas, entre las que podemos encontrar:

- rdfs:Resource para denotar la clase de todos los recursos.

- rdf:Property para referirse a la clase de todas las propiedades.
- rdfs:Literal para representar la clase de todos los valores de literales.

Junto con las clases, RDFS permite crear subclases. Esta relación se crea entre dos clases mediante el predicado *rdfs:subClassOf*, indicando que la clase en el sujeto es una subclase de la clase en el objeto. De esta forma, se permite crear una jerarquía entre las clases, de esta forma, si un recurso *x* es una instancia de la clase *c1* y a su vez *c1* es una subclase de *c2*, entonces *x* es también una instancia de la clase *c2*. Por ejemplo, para indicar que la clase libro es una subclase de la clase documento:

ex:Libro rdf:type rdfs:Class

ex:Libro rdfs:subClassOf ex:Documento

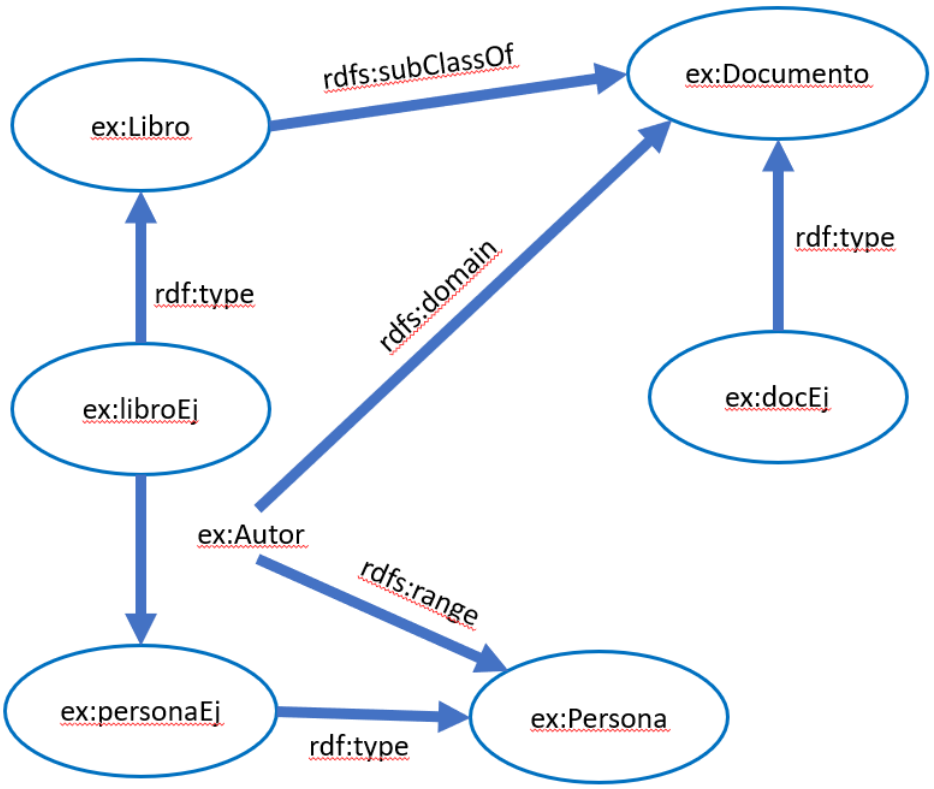


Figura 6: Ejemplo de un grafo utilizando RDFS.
Fuente: Elaboración propia.

Una vez que se cuenta con la posibilidad de crear clases, resulta útil crear e indicar propiedades de estas, para esto, RDFS incluye el predicado *rdfs:Property*, para indicar que un recurso es del tipo propiedad. Esto viene acompañado de poder indicar que los valores de la propiedad deben ser una instancia de alguna clase (objeto) utilizando *rdfs:range* y que la

propiedad puede ser aplicada solamente a instancias de determinadas clases (sujeto) utilizando *rdfs:domain*. Por ejemplo, para crear la propiedad "autor" e indicar que solo puede ser utilizada en instancias de la clase Documento y que solo puede contar con valor de instancias pertenecientes a la clase "Persona":

ex:autor rdf:type rdfs:Property .

rdfs:Property rdfs:domain ex:Documento .

rdfs:Property rdfs:range ex:Persona

Finalmente, con las características indicadas se puede representar un dominio de interés de mejor manera que utilizando solamente RDF. En la figura 6 se muestra un ejemplo de grafo RDF que utiliza RDFS.

2.3. OWL

Web Ontology Language (OWL) es el estandar recomendado para modelar ontologías. Es una extensión de RDF que permite tener elementos avanzados para lograr describir complejos dominios de conocimiento que no serían posible de describir utilizando solamente RDFS, a la vez que provee un sistema de razonamiento para estos. Su diseño está enfocado a encontrar un balance entre la expresividad de un lenguaje y la eficiencia en el razonamiento.

OWL cuenta con 3 sublenguajes de expresión incremental, esto con el objetivo de darle al usuario opciones entre diferentes grados de expresividad. Estos son:

- OWL *Lite*
- OWL *DL*
- OWL *Full*

Una ontología OWL utiliza como elementos básicos las clases y propiedades, las que ya se presentaron en RDFS, e individuos que son instancias de clases en RDF. Las propiedades se denominan roles en OWL.

Para definir una clase en OWL se usa el recurso *owl:Class*, por ejemplo:

ex:Profesor rdf:type owl:Class

Entre las clases predefinidas que trae OWL, se encuentran *owl:Thing* y *owl:Nothing*. La primera es la clase más general y tiene como instancias a todos los individuos. La clase *owl:Nothing* por definición no tiene ninguna instancia.

Las clases se pueden relacionar mediante *rdfs:subClassOf* de igual manera que en RDFS. Además, se pueden declarar que dos clases son disjuntas por medio de *owl:disjointWith*, lo que significa que no comparten ningún individuo. También puede declararse que dos clases son equivalentes por medio de *owl:equivalentClass*.

Además OWL permite indicar que una clase es la unión de dos o más clases por medio de *owl:unionOf*. De igual manera, se puede indicar que una clase es la intersección de dos o más clases por medio de *owl:intersectionOf*, como también se puede utilizar *owl:complementOf* para señalar que una clase es el complemento de otra.

Al igual que en RDF, los individuos pueden ser declarados como instancias de clases. Este proceso se llama asignación de clase y es por medio de la propiedad rdf:type. Para indicar que dos instancias son iguales se utiliza *owl:sameAs*. También se puede indicar que dos instancias son diferentes por medio de *owl:differentFrom*.

Respecto a las propiedades, se cuenta con dos tipos de estas en OWL: para hacer conexiones entre clases se usa *owl:ObjectProperty*, mientras que para indicar valores de datos se utiliza *owl:DatatypeProperty*.

Para definir relaciones entre propiedades se cuenta al igual en RDFS con *rdfs:subPropertyOf*. Además se cuenta con la posibilidad de indicar que dos propiedades son identicas por medio de *owl:equivalentProperty*. Dos propiedades pueden ser inversas, es decir, indican la misma relación pero con los argumentos intercambiados, para hacer esta declaración se utiliza *owl:inverseOf*.

Además OWL permite especificar restricciones existenciales relacionadas a las clases. Esto significa que todas las instancias de una clase estan relacionadas a por lo menos una instancia de otra clase por medio de la propiedad indicada. Para esto se utiliza *owl:someValueFrom*. De igual manera, se cuenta con una restriccion relacionada al cuantificador universal definido por medio de *owl:allValuesFrom*, este se utiliza para indicar que para cada instancia de una clase, todas las relaciones por medio de la propiedad indicada deben ser con instancias de una misma clase.

2.4. SPARQL

SPARQL (*SPARQL Protocol and RDF Query Language*) es un lenguaje estándar para hacer consultas a datos RDF, de forma de contar con una manera de acceder a los datos que se desean. En otras palabras, SPARQL es un lenguaje que busca coincidencias de patrones en grafos RDF. De esta manera, al construir consultas SPARQL, se trata de obtener como respuesta los triples que coincidan con la consulta realizada.[w3c, 2008]

Para indicar que atributos son de interés o cuales son los que se quieren, se utiliza una expresión de *graph pattern*, la que se asemeja a una tupla pero que puede contener variables.

Las variables son representadas con un “?” al principio. Esta expresión va precedida por el operador WHERE y al interior de llaves “”.

La forma en que se define una expresión de *graph pattern* es recursivamente como está a continuación:

1. Una tupla proveniente de $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ es un *graph pattern*, donde:
   - **V** es el conjunto de las variables.
   - **I** es el conjunto de las URIs.
   - **L** es el conjunto de los literales.

   A este tipo de tupla también se le llama *triple pattern*.
2. Si P_1 y P_2 son *graph pattern*, las expresiones (P_1 AND P_2), (P_1 UNION P_2) y (P_1 OPT P_2) también lo son.
3. Si P es un *graph pattern* y R es una condición *build-in* de SPARQL, entonces la expresión (P FILTER R) también es un *graph pattern*.

En el ejemplo de a continuación, se quieren obtener todos los libros de género horror junto con sus autores y fechas de publicación, para esto se utilizan las variables ?autor ?libro ?fecha y ?titulo, además de una expresión compuesta de 4 *triple patterns*:

PREFIX ex: <http://www.example.org/>

SELECT ?titulo ?autor ?fecha

WHERE{

?libro ex:author ?autor .

?libro ex:title ?titulo .

?libro ex:genre ex:horror .

?libro ex:publishDate ?fecha .

}

Si bien lo anterior permitiría obtener los datos que coincidan con lo que se quiere, también es necesario poder hacerles un post procesamiento a estos e indicarles el formato en que deben ser entregados, para esto, SPARQL cuenta con diferentes operadores que lo permiten.

Una consulta SPARQL está compuesta por 3 partes. La primera parte, el *pattern matching* tiene varias características interesantes para la selección de la data como unión, anidación,

partes opcionales, filtros, y la posibilidad de escoger la data fuente. La segunda parte, los modificadores de la solución, una vez fue obtenida la salida del patrón utilizado, se permite modificar estos valores obtenidos aplicando operadores clásicos como proyección, orden, limitar, distintos. Finalmente, la salida de una consulta SPARQL puede ser de diferentes tipos: consulta si/no, selección de valores de las variables, construcción de una nueva data RDF a partir de estos valores, y descripción de recursos. [Pérez *et al.*, 2009]. A continuación, se entra en mayor detalle de cada uno de los elementos mencionados anteriormente.

1. Dentro de la primera parte se encuentran los operadores que permiten establecer de mejor manera los datos que se desean. Estos son:

   - **AND**: Representado por un punto (.), permite concatenar dos patrones.
   - **UNION**: Permite obtener resultados de múltiples patrones.
   - **OPTION**: A diferencia de una consulta normal, en donde todo el patrón debe coincidir para ser una solución, al agregar este operador, el patrón que contenga será considerado solamente si es que se encuentra una coincidencia, en caso contrario simplemente se ignora y se obtendrán las soluciones de la parte que no es opcional.
   - **FILTER**: Permite filtrar resultados basándose en las condiciones *build-in* que se indiquen. Las condiciones *build-in* son construidas con elementos del conjunto $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ y constantes, conexiones lógicas ($\neg$, $\wedge$, $\vee$), símbolos de inecuaciones ($<, \leq, \geq, >$), símbolo de igualdad ($=$), predicados unitarios como bound, isBlank, y isIRI, más otros. [Pérez *et al.*, 2009]. Se considerarán solo las condiciones que son una combinación booleana de terminos construidas usando solamente igualdad(=) y el operador *bound*, es decir:
     a) Si $?X, ?Y \in \mathbf{V}$ y $c \in \mathbf{I} \cup \mathbf{L}$, entonces, $bound(?X)$, $?X = c$ y $?X = ?Y$ son condiciones *build-in*.
     b) Si R_1 y R_2 son condiciones *build-in*, entonces ($\neg R_1$), ($R_1 \vee R_2$) y ($R_1 \wedge R_2$) son condiciones *build-in*.

2. Dentro de esta segunda parte se encuentran los operadores de modificación, estos son:

   - **Order**: Establece el orden en que se entregan las soluciones encontradas, pueden ser ordenaras ascendentemente (por defecto) o utilizando descendentemente con el operador DESC.

     Por ejemplo, ORDER BY DESC(?name) ordenará los resultados de forma descendente según su valor en la variable ?name.
   - **Distinct**:Es utilizado para indicar que se desean solamente soluciones únicas, eliminando duplicados de estas. Debe colocarse seguido del operador SELECT.
   - **Limit**:Permite indicar la cantidad máxima de soluciones que se quieren obtener.

3. Finalmente, en la parte 3 se encuentran los siguientes tipos de salidas:

- **SELECT**:Permite indicar cuales son las variables de las que se desea obtener sus valores.
- **CONSTRUCT**:Al utilizarlo, la consulta retornará un grafo RDF que es generado tomando cada una de las soluciones obtenidas y reemplazándolas por las variables según el *template* indicado.
- **ASK**:Al utilizar este operador, la consulta responderá solamente con sí o no en función de si se encontraron soluciones (se obtuvo un resultado no nulo) o si no hubo coincidencias. No se obtendrá ninguna información de las posibles soluciones, solamente si existen o no.

De esta forma, al combinar los elementos anteriores se permite construir consultas SPARQL mucho más complejas que buscarán patrones en el grafo de mucha mayor complejidad.

2.4.1. Semántica de SPARQL

El núcleo de la semántica de SPARQL es la llamada SPARQL algebra, en la que se definen un conjunto de operaciones que se pueden utilizar para calcular el resultado de una consulta en SPARQL. Para definir estas operaciones, previamente se debe definir lo que es un *mapping* y algunas de sus características.

Un *mapping* μ es una función de la forma $\mu : \mathbf{V} \rightarrow \mathbf{U}$. Si t es un *triple pattern*, $\mu(t)$ es el *triple* obtenido al reemplazar las variables en t según μ, a esto también se le conoce como una solución.

El dominio de μ, denotado por $dom(\mu)$, es el subconjunto de **V** con un μ definido.

Sean μ_1 y μ_2 dos *mappings*, se dice que ambos son compatibles cuando para toda variable $?X \in dom(\mu_1) \cap dom(\mu_2)$ se cumple que $\mu_1(?X) = \mu_2(?X)$.

Hasta el momento se ha utilizado μ, el que es una posible solución, ahora también se utilizará Ω, que representará un conjunto de *mappings*. Se definen las siguientes operaciones a utilizar sobre conjuntos de *mappings*.

- Join: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \wedge \mu, \mu$ son compatibles$\}$
- Union: $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
- Difference: $\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2 \wedge \mu$ y μ' no son compatibles$\}$
- Left outer-join: $\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

Se define la semántica de una expresión de un *graph pattern* como una función $[\![\cdot]\!]_D$, la cual toma la expresión P y retorna un conjunto de *mappings* Ω encontrados sobre el conjunto de datos D.

Sea $[\![P]\!]_D$ la evaluación del *graph pattern* P en el conjunto de datos RDF D. Este se define recursivamente como:

- Si P es un *triple patter* t, entonces $[\![P]\!]_D = \{\mu \mid dom(\mu) = var(t) \wedge \mu(t) \in D\}$
- Si P es (P_1 AND P_2), entonces $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$
- Si P es (P_1 OPT P_2), entonces $[\![P]\!]_D = [\![P_1]\!]_D \mathbin{⟕} [\![P_2]\!]_D$
- Si P es (P_1 UNION P_2), entonces $[\![P]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$
- $[\![P$ FILTER $R]\!]_D = \{\mu \in [\![P]\!]_D \mid \mu \models R\}$. Donde R es una condicion *build-in* y $\mu \models R$ denota que μ satisface R si:
  1. R es $bound(?X)$ y $?X \in dom(\mu)$.
  2. R es $?X = c$, $?X \in dom(\mu)$ y $\mu(?X) = c$.
  3. R es $?X = ?Y$, $?X \in dom(\mu)$, $?Y \in dom(\mu)$ y $\mu(?X) = \mu(?Y)$.
  4. R es ($\neg R_1$), R_1 es una condicion *build-in* y no se cumple que $\mu \models R$.
  5. R es ($R_1 \vee R_2$), R_1 y R_2 son condiciones *build-in*, y $\mu \models R_1$ o $\mu \models R_2$.
  6. R es ($R_1 \wedge R_2$), R_1 y R_2 son condiciones *build-in*, y $\mu \models R_1$ y $\mu \models R_2$.

2.5. RDFExplorer

RDFExplorer [Vargas *et al.*, 2019] es un sistema para realizar consultas SPARQL y explorar grafos RDF utilizando para esto una interfaz gráfica, en donde la consulta es representada como un grafo. Actualmente está implementado de manera que el usuario interactúa con su interfaz mediante el uso de una aplicación web, en la que se va paulatinamente construyendo un grafo mediante la creación de nodos y conectando estos a través de enlaces o arcos.

Para poder contar con la capacidad de representar las consultas SPARQL a través de grafos, el sistema cuenta con un grafo de consulta visual o *visual query graph* (VQG), el que permite expresar las consultas por medio de grafos que se construyen mediante operadores que pertenecen a un lenguaje de consulta visual o *visual query language* (VQL) y que se utilizan mediante simples interacciones en la interfaz. Este punto se verá en más detalle más adelante.

La interfaz cuenta con un lienzo principal en que se puede construir el grafo funcionando como un editor del VQG, pero además cuenta con otros 2 componentes principales: un panel (a la izquierda) de busqueda y un panel (a la derecha) para ver detalles del nodo seleccionado. Además, cuenta con otros 3 componentes en forma de botones que al seleccionarlos modifican el panel de la derecha, mostrando respectivamente: un editor de nodos (que permite agregar restricciones), un editor de consultas SPARQL (que muestra la consulta actual), y un

panel de ayuda. En la figura 7 se muestra un ejemplo de un grafo realizado en la interfaz, en este se muestra a la izquierda el panel de busqueda, al centro el grafo que se tiene y a la derecha se muestra el detalle de la variable seleccionada "?movie".

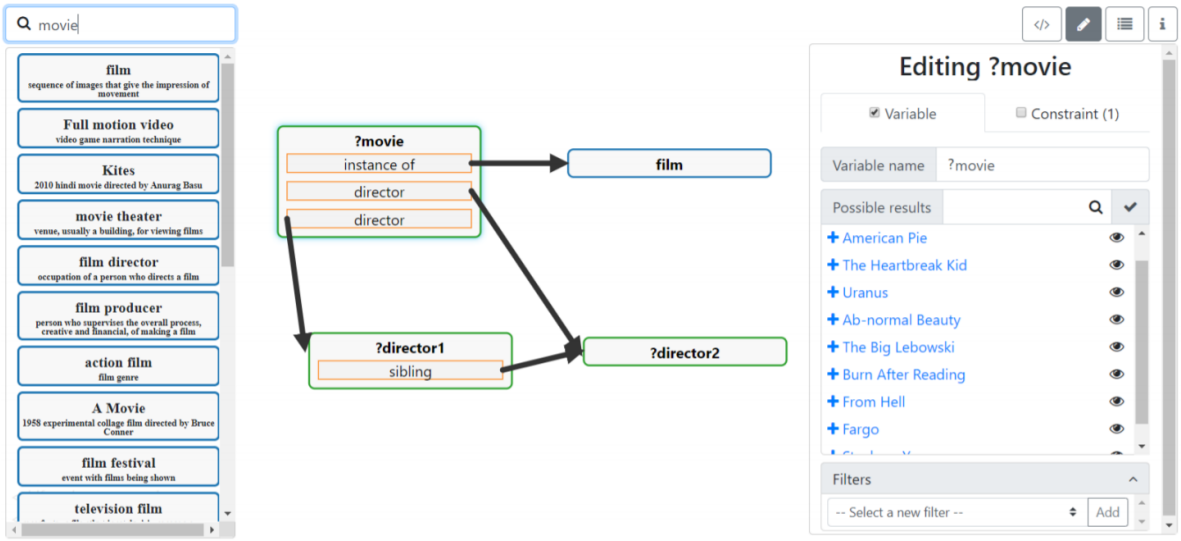


Figura 7: Ejemplo de la interfaz de RDFExplorer.
Fuente: [Vargas *et al.*, 2019].

Como se ve en la figura 7, si se selecciona un nodo variable, en el panel de la derecha se mostrará una muestra de resultados obtenidos para esa variable.

Si bien, los enlaces y nodos pueden ser variables o URIs, en el caso de ser esta última, en la interfaz se muestra la etiqueta, si es que existe, en lugar de mostrar directamente la URI. De igual manera, en todas las secciones de RDFExplorer se hace uso de las etiquetas de cada uno de los recursos, permitiendo que al usuario le resulte más natural y facilitando la comprensión de todos los resultados y elementos con los que se interactúa.

A medida que se va construyendo el grafo, RDFExplorer traduce este grafo a una consulta SPARQL que se realiza hacia la base de datos, lo que permite tener resultados parciales a la vez que se va armando y modificando el grafo. Cuando una de estas consultas obtiene resultados vacíos, lo único que se obtiene es la información de que no se obtuvieron soluciones. Esta información se muestra en la pestaña de posibles resultados. Un ejemplo de esto se muestra en la figura 8.

2.5.1. Lenguaje de RDFExplorer

Como se mencionó previamente, RDFExplorer consta de un lenguaje denominado *visual query language* (VQL), para construir el grafo con el que se trabaja.

Un grafo de consulta visual, en ingles llamado *visual query graph* (VQG), es un grafo dirigido

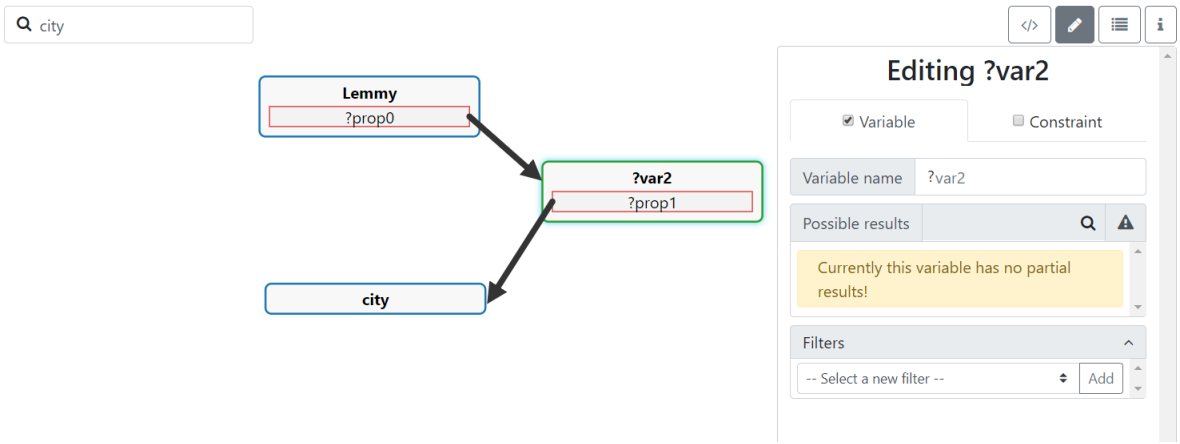


Figura 8: Vista de los resultados cuando estos son vacíos.
Fuente: Elaboración propia.

y etiquetado $\mathsf{G} = (\mathsf{N}, \mathsf{E})$, con nodos N y arcos E. Los nodos del N son un conjunto finito de URIs, literales o variables: $\mathsf{N} \subset \mathbf{I} \cup \mathbf{L} \cup \mathbf{V}$. Los enlaces del E son un conjunto finito de triples en donde cada uno de estos indica un arco dirigido entre dos nodos con una etiqueta que pertenece al conjunto de URIs o de variables: $\mathsf{E} \subset \mathsf{N} \times (\mathbf{I} \cup \mathbf{V}) \times \mathsf{N}$.

Se define como $\text{var}(\mathsf{G})$ al conjunto de variables en $\mathsf{G} = (\mathsf{N}, \mathsf{E})$, como nodos o arcos etiquetados:

$$\text{var}(\mathsf{G}) := \{v \in \mathbf{V} \mid v \in \mathsf{N} \text{ or } \exists n_1, n_2 : (n_1, v, n_2) \in\}$$

Inicialmente el VQG está vacío: $G_0 = (\emptyset, \emptyset)$, pero se va construyendo utilizando el VQL, el cual consiste en cuatro operadores algebraicos que corresponden a interacciones atómicas por parte del usuario. Estas operaciones atómicas son las siguientes:

- Agregar nodo de variable:
  $\eta(\mathsf{G}) := (\mathsf{N} \cup \{v\}, \mathsf{E})$ donde $v \notin \text{var}(\mathsf{G})$
- Agregar nodo de constante:
  $\eta(\mathsf{G}, x) := (\mathsf{N} \cup \{x\}, \mathsf{E})$ donde $x \in (\mathbf{I} \cup \mathbf{L})$
- Agregar arco entre dos nodos con una etiqueta de variable:
  $\varepsilon(\mathsf{G}, n_1, n_2) := (\mathsf{N}, \mathsf{E} \cup \{(n_1, v, n_2)\})$ donde $\{n_1, n_2\} \subseteq \mathsf{N}$ y $v \notin \text{var}(\mathsf{G})$
- Agregar arco entre dos nodos con una etiqueta de constante:
  $\varepsilon(\mathsf{G}, n_1, x, n_2) := (\mathsf{N}, \mathsf{E} \cup \{(n_1, x, n_2)\})$ donde $\{n_1, n_2\} \subseteq \mathsf{N}$ y $x \in \mathbf{I}$

El VQG representa un SPARQL *basic graph pattern* (BGP), pero no es exactamente uno. Por lo que se debe traducir a un BGP para hacer uso de esta y realizar la consulta SPARQL equivalente. Esta traducción es casi directa y natural. Dado un VQG $\mathsf{G} = (\mathsf{N}, \mathsf{E})$, por diseño, el

conjunto E ya es un BGP, por lo que ya está. Cabe resaltar que esta traducción no considera a los nodos huerfanos (nodos sin arcos incidentes).

2.6. Estado del arte

En la literatura se describen variadas técnicas para incrementar el número de resultados obtenidos de una consulta que se realiza a una base de datos RDF. A continuación se mencionarán algunas de estas.

En [Huang *et al.*, 2008] se busca hacer más flexibles a las consultas por medio de relajaciones, utilizando para esto la ontología de RDFS y algunas reglas de inferencias utilizadas en *entailment* RDFS. Para relajar las consultas, se define un algoritmo que genera un grafo de relajaciones en donde cada nodo representa una consulta que apunta a las que se obtiene de relajar un *triple* de esta. En la figura 9 se muestra un ejemplo de este grafo. Para relajar un *triple*, este se reemplaza por el *triple* obtenido según 4 inferencias relacionadas a subclases y subpropiedades . Junto a lo anterior, se define un sistema para puntuar las consultas relajadas, permitiendo ordenarlas según que tan similares son a la consulta original, para luego ejecutar solamente una cantidad fija de las mejor puntuadas. Para calcular el puntaje de una consulta respecto a la original, se tiene que obtener el promedio de los puntajes de cada *triple* de la consulta, el cual a su vez se calcula por la similitud de los recursos que tienen la misma posición en ambas consultas (la original y la que se esté comparando), y para calcular esta similitud se utiliza una formula que considera el largo de su parentesco en la ontología de RDFS.

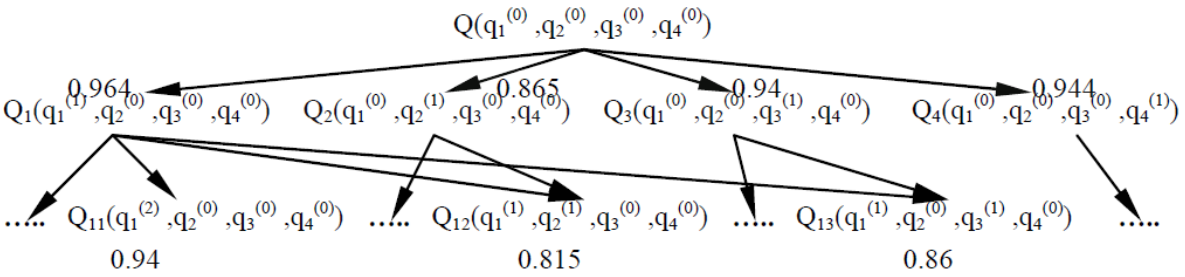


Figura 9: Grafo de relajaciones de una consulta.
Fuente: [Huang *et al.*, 2008].

Un proceso similar al momento de relajar una consulta se hace en [Hurtado *et al.*, 2006]. Aquí la consulta se relaja reemplazando un *triple* de la misma manera que en [Huang *et al.*, 2008], pero con el añadido de dos inferencias de RDFS más, las que incluyen el uso de las caracteristicas del dominio y del rango incluidas en RDFS. En la figura 10 se presentan dos ejemplos de los grafos obtenidos al relajar multiples veces un *triple*.

Un método *fuzzy* se presenta en [Hogan *et al.*, 2012] limitado a consultas con *graph patterns* que tengan una variable común en la posición del sujeto. Aquí se define una función *matcher*

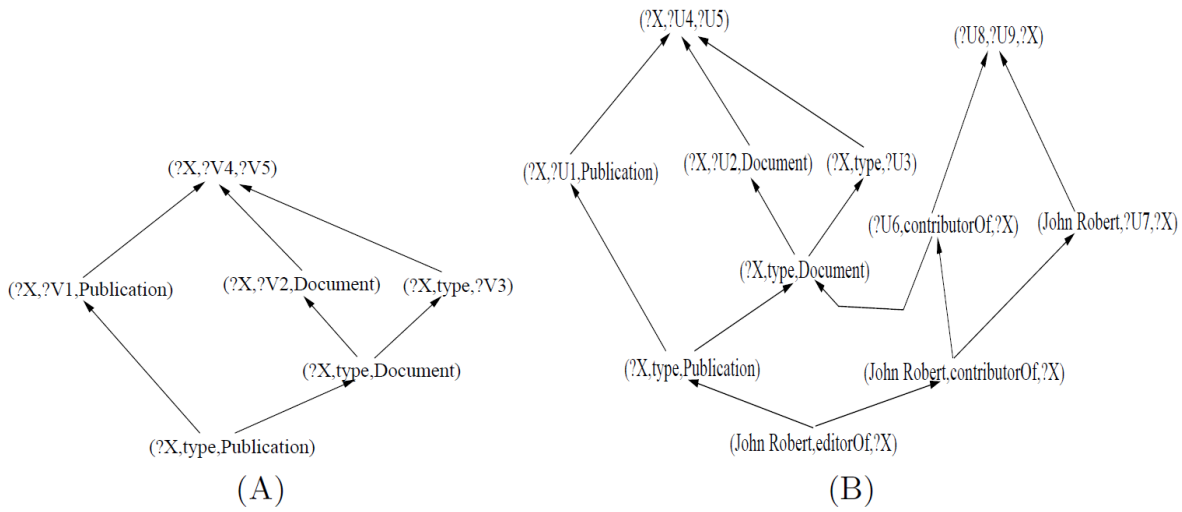


Figura 10: Dos ejemplos de los grafos de la relajación de un *triple pattern*.
Fuente: [Hurtado *et al.*, 2006].

que entrega un valor entre 0 y 1 para representar que tan intercambiable son 2 términos, usando para esto diferentes funciones según sean literales, variables o URIs para calcular la distancia entre ambos términos.

En [Elbassuoni *et al.*, 2011] también se realiza un proceso para relajar o modificar las consultas pero con un enfoque en el uso de estadísticas en lugar de ontologías. Se presentan 3 tipos de relajación: reemplazar una entidad por otra, reemplazar una entidad por una variable y eliminar un *triple*. Para el primer caso, se buscan nuevas entidades que tengan la mayor similitud posible, y esta similitud se obtiene al comparar las distribuciones de probabilidad de ambas entidades, que a su vez se obtienen mediante un calculo que utiliza toda la base de datos RDF relacionada.

En [Andreşel *et al.*, 2018] se utilizan ontologias tanto para relajar y obtener un mayor número de resultados como para restringir y obtener un número reducido de resultados en las consultas. Para esto, se definen un conjunto de reglas para cada uno de estos 2 casos que son utilizadas para reformular partes de la consulta original. Además, en cada uno de estos casos se hace la distinción de reglas basadas en una ontología, es decir, *TBox* y las que se basan en el *dataset*.

En [Fokou *et al.*, 2015] se enfoca en encontrar las partes de la consulta que son responsable de que no se obtengan resultados. Para hacer esto, se desarrolla un algoritmo que busca las subconsultas más pequeñas que son causantes del problema(denominadas *mfs*), esto se logra mediante la realización de nuevas consultas que prueban todos los subconjuntos posibles del *graph pattern*. Además, presentan como método de relajación el poder encontrar los subconjunto de *triples* máximos que sí obtienen resultados (denominados *xss*). En la figura 11 se muestra un ejemplo de las subconsultas realizadas y los *triples* que cada una utiliza. De igual manera este algoritmo se utiliza en [Dellal *et al.*, 2019], con la diferencia que en esta

se añade la característica de que se realiza sobre *knowlegde bases* que cuentan con un valor de credibilidad asociado a cada *triple* almacenado, llamadas *uncertain knowledge bases*.

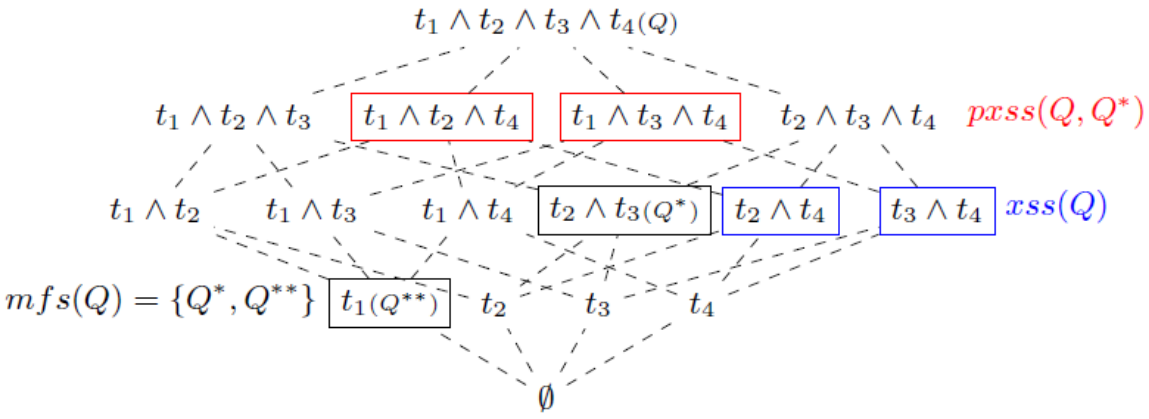


Figura 11: Grafo de subconsultas generadas a partir de una consulta inicial con 4 *triples*. Fuente: [Fokou *et al.*, 2015].

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

Como se ha visto en los capítulos anteriores, el problema es que al realizar una consulta SPARQL, no se obtengan resultados ni nada que pueda ser utilizado por parte del usuario para comprender la causa de esto. Por lo que se propone obtener un conjunto de consultas SPARQL alternativas que sí tienen resultados, a la vez que se intenta mantener la semejanza con la consulta original. Para esto, se debe diseñar un algoritmo que permita obtener estas consultas, por lo que primero se analizarán las causas a considerar a las que se debe la aparición de este problema. Una vez que se tenga una comprensión general de estas causas, se dará paso a buscar cómo solucionarlas de manera tal que finalmente se obtengan consultas alternativas a la original y para esto, se deberá encontrar un conjunto de valores que representarán a cada una de las consultas alternativas. Finalmente, se realizará un análisis de la complejidad del algoritmo planteado.

3.1. Consultas alternativas

Inicialmente se cuenta con una consulta Q_{ini} que al ser realizada a una base de datos D, se obtienen resultados vacíos, es decir, $[\![Q_{ini}]\!]_D = \emptyset$.

Una posible solución a este problema es que al momento de detectar que no se encuentran soluciones en D, se realizen modificaciones a Q_{ini} para incrementar el número de resultados entregados, o sea, $[\![Q_{ini}]\!]_D \neq \emptyset$. Esto si bien soluciona el problema, no entrega información que pudiese ser utilizada para reconocer el error o entender la causa de este por parte del usuario. Por esto, se propone obtener un conjunto K de consultas alternativas que sí retornan resultados, es decir:

$K = \{Q \mid [\![Q]\!]_D \neq \emptyset\}$

De esta manera, el usuario puede compararlas y comprender cual es la razón del problema. Además, mientras mayor sea el tamaño del conjunto K, más alternativas tendrá el usuario de donde escoger.

Para generar el conjunto K se utilizarán consultas Q' obtenidas luego de reemplazar un recurso representado por una URI u presente en el *graph pattern* de Q_{ini} por otra URI. De esta forma, las consultas alternativas mantendrán una similitud con Q_{ini}, lo que por un lado permite que se obtengan soluciones que se asemejen a lo esperado inicialmente, a la vez que facilita al usuario poder comparar las diferencias entre cada consulta.

Otra manera de obtener nuevas consultas para K es a traves de modificaciones a Q_{ini} reemplazando una o más URIs por variables o directamente eliminar uno o más *triples*. El pro-

blema de esto es que mientras más veces se realize una de estas modificaciones, la consulta tendrá menos restricciones y por ende, obtendrá soluciones que muy probablemente se alejen mucho de lo que originalmente se deseaba obtener por medio de Q_{ini}.

Por otro lado, solo se considerarán consultas alternativas que cuenten con una URI diferente a la vez. Esto debido a que mientras más URIs diferentes se tenga al mismo tiempo, más probable es que estas consultas se alejen más del significado original que se tenía.

Por ejemplo, si se tiene una consulta Q_{ini} que tienen el siguiente *triple* t:

?x ex:trabajo ex:profesor

Entonces se podría esperar obtener un conjunto K con 3 consultas, que en lugar de t cada una tengan uno de los siguientes *triples*:

?x ex:trabajo ex:catedrático

?x ex:trabajo ex:maestro

?x ex:ocupación ex:profesor

Así, se desarrollará un algoritmo para obtener la máxima cantidad posible de consultas en K, manteniendo una relación de semejanza lo más posible. Como solo se considerarán consultas con una URI distinto, el algoritmo solo debe encontrar el conjunto de URIs para cada una de las URIs u presentes en el *graph pattern* de Q_{ini}.

Como estas consultas deben guardar cierta relación con la original, a continuación se analizarán las posibles causas del problema principal, con la intención de hacer uso de esta información para definir la forma que se utilizará para obtener consultas que estén relacionadas a Q_{ini}.

3.2. Causas del problema a considerar

Nos centramos en dos causas del problema: la no existencia de los datos exactos en la base de datos, y que la consulta no refleja correctamente lo que el usuario desea.

1. Los datos exactos no existen: Esto ocurre cuando lo que el usuario busca se ve correctamente reflejada en SPARQL pero no existen soluciones en la base de datos.
2. El planteamiento del *graph pattern* no corresponde con la estructura del grafo RDF:

   Esto se debe principalmente a que el usuario no conozca o comprenda una gran cantidad de los contenidos y la estructura de la base de datos, la que puede resultar ser enorme y con una gran complejidad. También puede deberse a la mala suposición, por

parte del usuario, relacionada a la estructura del grafo, generando errores de conceptos en el dominio de la base de datos, dando por resultado un *graph pattern* que no coincide con lo que se busca.

Ejemplo del primer caso se da cuando, considerando la pequeña base de datos en la tabla 1, se quiere obtener todos los títulos de los libros de género ciencia ficción escritos por W.H. Hogdson. Una consulta SPARQL que representa correctamente esto es la siguiente:

```
PREFIX ex: <http://www.example.org/>

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

SELECT ?titulo WHERE

{

   ?libro rdf:type ex:Libro .

   ?libro ex:title ?titulo .

   ?libro ex:genre ex:CienciaFiccion .

   ?libro ex:author ex:WHHogdson .

}
```

Tabla 1: Ejemplo de base de dato RDF.
Fuente: Elaboración Propia.

Resource1	type	libro
Resource2	type	libro
Resource1	genre	Horror
Resource2	genre	Horror
Resource1	title	”The Dream of X”
Resource2	title	”The House on the Borderland”
Resource1	author	WHHogdson
Resource2	author	WHHogdson

Al realizar esta consulta no encontrará ninguna solución, ya que si bien, se puede ver que sí existen recursos del tipo *libro* y también algunos cuentan con la propiedad *title*, no se tienen registros de algúno que sea del género de ciencia ficción, por lo que se obtendrá un resultado nulo.

Para este tipo de problemas, se busca incrementar el número de soluciones y para esto, se puede usar *entailment*, que transforma conocimiento implícito en explicito que puede ser

una solución a la consulta, es decir, a partir de *triples* ya existentes se infieren nuevos *triples* que pueden sí ser datos como los que se están buscando.

Por otra parte, se puede aumentar el número de soluciones incrementado lo general que es una consulta. Para hacer esto ya se han desarrollado algunas soluciones utilizando *query relaxation* para generalizar la consulta, permitiendo considerar un mayor número de soluciones similares a la deseada utilizando por ejemplo la jerarquía de clases.

Un ejemplo del segundo caso sería, volviendo al ejemplo del caso anterior, cuando se quiere obtener todos los títulos de los libros de genero de horror escritos por W.H.Hogdson pero se realiza la siguiente consulta:

```
PREFIX ex: <http://www.example.org/>

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

SELECT ?titulo WHERE

{

   ?libro rdf:type ex:Libro .

   ?libro ex:title ?titulo .

   ?libro ex:genre ex:Horror .

   ex:WHHodgson ex:author ?libro .

}
```

A diferencia del caso anterior ahora sí existen registros de libros de genero de horror, pero aun así la consulta obtendrá resultados vacíos. Esto se debe a que el *triple* que indica al autor del libro se colocó erróneamente, invirtiendo el sentido del *triple* e indicando que una persona tiene como autor a un libro. Esto puede deberse a que el usuario no entiende correctamente el significado de la propiedad *author*.

Este problema puede verse enormemente amplificado si se considera una base de datos con millones de *triples* y con una gran cantidad de propiedades que el usuario desconoce o no comprende del todo su significado. En estas situaciones el problema es llevar la consulta de lenguaje natural a SPARQL utilizando el vocabulario de la base de datos, el cual no es comprendido del todo por el usuario.

Otro ejemplo de esta última causa, considerando el mismo contexto de libros, sería que el usuario diera por supuesto previamente que todos los recursos en los que está interesado son libros y por ende, en la base de datos todos deben ser del tipo Libro pero en la realidad existen diferentes tipos como novelas, documentos, etc. En este caso, el problema se genera en un principio por parte del usuario para luego verse traslado el error a la consulta SPARQL.

3.3. Técnicas utilizadas para afrontar las causas

Con ambas causas ya analizadas, se pasará a indicar algunas maneras de como solucionarlas, de manera que con esta información se puedan definir las relaciones que se considerarán para obtener las URIs que generarán las nuevas consultas.

A continuación se describirán las maneras que se considerarán utilizar para afrontar cada una de estas causas. Para el problema de la no existencia de datos exactos se utilizará:

- *SPARQL Entailment*
- *Query relaxation*

Mientras que para el caso en que el planteamiento del *graph pattern* no coincide con el dominio se usará una ontología como guía para modificar la consulta.

3.3.1. SPARQL Entailment

Cuando los datos exactos no existen, se puede utilizar el conocimiento implicito para ver si este contiene alguna solución a la consulta original y en caso de tenerla, retornar el valor como solución. Ahora bien, como el objetivo descrito es obtener un conjunto de consultas alternativas, se debe modificar la consulta inicial para que esta consulte específicamente por los datos que se obtendrían del conocimiento implícito.

Como solamente se consideran las consulta con una URI diferente, solo se pueden usar las inferencias que generen *triples* con un valor diferente a uno de los *triples* involucrados en la inferencia. Estas, en el régimen RDFS son las reglas de inferencia relacionadas a las relaciones de subclase y de subpropiedad:

1. $\frac{(\text{a type b}) \wedge (\text{b subclase c})}{\text{a type c}}$
2. $\frac{(\text{a subclase b}) \wedge (\text{b subclase c})}{\text{a subclase c}}$
3. $\frac{(\text{a subproperty b}) \wedge (\text{b subproperty c})}{\text{a subproperty c}}$

Por ejemplo, si se tiene una consulta con el *graph pattern*:

```
{
   ?v1 rdf:subClass ex:Documento .
   ?v1 ex:author ex:Person1 .
```

}

Y si se considera la regla de inferencia 2 aplicada a ex:Documento, se tiene:

$$\frac{(\text{a subclase b}) \wedge (\text{b subclase ex:Documento})}{\text{a subclase ex:Documento}}$$

Para utilizar esta inferencia se tiene que hacer uso de los resultados parciales obtenidos de:

{ ?v1 rdf:subClass ex:Documento . }

Y luego reemplazar estos resultados en la consulta inicial, obteniendo el siguiente *graph pattern*. Sea ex:result1 una subclase de Documento:

{

?v1 rdf:subClass ex:result1 .

?v1 ex:author ex:Person1 .

}

De esta forma, la consulta obtendría resultados que cumplen con la regla de inferencia.

En este caso, como se hace uso de las subclases y subpropiedades, se estaría haciendo uso de una especialización en la jerarquía existente de las clases y propiedades.

3.3.2. Query relaxation

De igual manera, para el primer causante se puede hacer uso de la técnica de *query relaxation* que busca modificar una consulta de manera tal que sea más general y por ende, abarcar una mayor cantidad de valores y finalmente aumentar las posibilidades de obtener soluciones. Por ejemplo, se podrían considerar todos los libros sin importar su género o, si se estuviese solicitando un conjunto de libros, consultar en su lugar por un conjunto de documentos reemplazando el *triple*:

$?x\ rdf{:}type\ ex{:}Libro$

Por el *triple*:

$?x\ rdf{:}type\ ex{:}Documento$

Lo anterior se puede lograr mediante el uso de la técnica *query relaxation* en RDF como se mencionó anteriormente, utilizando para esto la ontología RDFS, la cual produce una jerarquía de clases y de propiedades.

Una consulta se puede relaja mediante el proceso de relajar uno o más de sus *triples*. Las

maneras que se considerarán para hacer esto a un *triple* son con:

- subclase: relajando uno o más de sus elementos (sujeto, objeto), reemplazandolo por su superclase.
- subpropiedad: relajando su predicado, reemplazandolo por su superpropiedad.

También para relajar un *triple* se puede hacer uso del dominio y rango del predicado de este de la siguiente manera:

- dominio: el *triple* se reemplaza por uno que consulta por todos los recursos que sean del tipo indicado en el dominio.
- rango: el *triple* se reemplaza por uno que consulta por todos los recursos que sean del tipo indicado en el rango.

Pero ambos casos no se considerarán debido a que al hacer uso del dominio y del rango, el *triple* obtenido se diferencia en más de un elemento a la vez.

Junto a lo anterior, se considerará el uso del tipo, reemplazando un valor por la clase a la que pertenece. Con esto, se cuenta con tres formas de relajar un recurso de un *triple*, esto según sea el tipo de recurso a reemplazar:

- Sujeto u objeto:
  - Instancia: Se reemplaza por la clase a la cual pertenece.
  - Clase: Se reemplaza por una superclase de esta, es decir, una clase que la contiene.
- Propiedad: Se reemplaza por una superpropiedad, es decir, una propiedad que la contiene.

A diferencia del caso anterior (SPARQL *entailment*), aquí se hace uso de una generalización en la jerarquía de clases y propiedades, por lo que al hacer uso de ambas técnicas se está utilizando en su totalidad la jerarquía de estas.

3.3.3. Uso de una ontología

Para el caso en que el planteamiento del *graph pattern* no coincide con el dominio, como la causa está relacionada al mal entendimiento del grafo o del dominio de los datos, se considerará modificar la consulta intentando cambiar levemente su significado y para esto, usar

otros recursos que mantengan una relación que sea indicada por una ontología que represente el dominio de los datos.

Para hacer lo anterior se podría utilizar la jerarquía de clases para modificar la consulta. Esto sirve para algunos casos, pero en las ocasiones en que se deba a la mala comprensión del grafo se tienen que realizar modificaciones de distinta naturaleza, con la intención de modificar la consulta para obtener una que coincida con los datos.

Idealmente, se debería utilizar una ontología que describe el dominio de los datos que se están utilizando pero tomaría bastante tiempo en diseñar una, además se deberia cambiar cada vez que se utilize otro conjunto de datos que sean de un dominio diferente. Por esto, se tendrá que hacer uso de una ontología mucho más general y que ya esté diseñada como OWL.

En específico se utilizará OWL 2 QL, el que cuenta con las siguientes caracteristicas:

1. Subclase: Propiedad utilizada para crear una jerarquía de clases.
2. Clase equivalente: Propiedad que indica que dos clases son equivalentes, es decir, tienen las mismas instancias.
3. Clases disjuntas: Propiedad que puede ser utilizada para indicar que dos clases son disjuntas, es decir, no pueden compartir instancias.
4. Propiedad inversa: Propiedad que indica que es la propiedad inversa de otra propiedad.
5. Inclusión de propiedad: Propiedad utilizada para crear una jerarquía de propiedades.
6. Propiedad equivalente: Propiedad que indica que dos propiedades son equivalentes, es decir, tienen las mismas instancias.
7. Dominio de la propiedad: Propiedad que puede ser utilizada para indicar que una propiedad solo puede ser aplicada a algún tipo específico de individuos.
8. Rango de la propiedad: Propiedad que puede ser utilizada para indicar que una propiedad solo puede tener como valor algún tipo específico de individuos.
9. Propiedades disjuntas: Propiedad que puede ser utilizada para indicar que dos propiedades son disjuntas, es decir, no pueden compartir instancias.
10. Propiedades simétricas: Propiedad que puede ser utilizada para indicar que si una propiedad tiene una instancia (x,y), entonces la instancia (y,x) también pertenece a la propiedad.
11. Propiedades reflexivas: Puede ser utilizada para indicar que un individuo esta relacionado con sigo mismo por medio de la propiedad que es reflexiva.

12. Propiedades irreflexivas: Puede ser utilizada para indicar que un individuo no puede estar relacionado con sigo mismo por medio de la propiedad que es irreflexiva.

13. Propiedades asimétricas: Propiedad que puede ser utilizada para indicar que si una propiedad tiene una instancia (x,y), entonces la instancia (y,x) no puede pertenece a la propiedad.

De estas caracteristicas, las 1-2-3-4-5-6-9 pueden ser utilizados directamente para modificar un valor.

La caracteristica de subclases e inclusión de propiedad puede ser utilizada de una manera similar a las ya señalas en *entailment* y *query relaxation*.

Las caracteristicas 10-11-12-13 son para dar restricciones o caracteristicas a pares de instancias de una propiedad específica, por lo que no pueden ser utilizadas para obtener nuevas propiedades alternativas. De igual manera, el dominio y el rango de una propiedad indican restricciones sobre las instancias que pueden existir en el sujeto y objeto respectivamente del *triple*, por lo que tampoco pueden ser utilizadas de la misma manera que las otras caracteristicas.

En la siguiente sección se indicará como unir las tres técnicas en una sola que tiene el objetivo de obtener el conjunto de consultas alternativas.

3.4. Diseño del algoritmo

Ya se han visto previamente las causas que se considerarán y una manera general de afrontar cada una de estas. Por lo que ahora se tiene que desarrollar un algoritmo que permita solucionar el problema considerando lo visto. Para esto, primero se definirá la salida que se desea obtener del algoritmo, luego cual será la manera de obtención de esta y finalmente el algoritmo completo.

3.4.1. Salida deseada del algoritmo

Como se vio anteriormente, se desea obtener el conjunto K de consultas nuevas que se asemejen lo más posible a la consulta original. Como solo se consideran las que tengan solamente una URI diferente, en lugar de obtener consultas completas, se puede considerar el obtener un conjunto de URIs por cada uno de los recursos presentes en Q_{ini}, de forma que al reemplazar estos últimos con los primeros se obtenga una consulta Q' que pertenecería a K.

Sea $Alt(a)$ el conjunto de URIs alternativos de la URI a presente en el *graph pattern* de Q_{ini}. Inicialmente se tiene que $Alt(a)$ está vacío. Se espera que finalizado el algoritmo, cada uno

de los elementos en $Alt(a)$ al reemplazarlos por a en Q_{ini}, genere una consulta Q' que retorna resultados no vacíos.

Finalmente, el algoritmo deberá entregar un conjunto $Alt(a_i)$ por cada URI única a_i presente en Q_{ini}. Por ejemplo, si se tiene la consulta:

SELECT ?v1 ?v2 ?v3 WHERE:

{

ex:Persona ?v1 ?v2 .

?v2 ?v3 ex:Ciudad .

}

Entonces las URIs de Q son $\{ex{:}Persona, ex{:}Ciudad\}$, por lo que se espera obtener como resultado los conjuntos: $Alt(ex{:}Persona)$ y $Alt(ex{:}Ciudad)$.

A continuación se describirá el proceso para obtener este conjunto para una URI a cualquiera.

3.4.2. Obtención del conjunto de recursos relacionados

Para obtener un conjunto $Alt(a)$ donde a es una URI, se hará uso de las relaciones indicadas previamente de *SPARQL entailment* y *query relaxation*, es decir, se utilizará la jerarquía de clases para modificar a traves de generalización y especialización aplicado directamente en a.

La obtención de las URIs relacionados con a dependerá de como es utilizado en el *triple* en que aparece, las relaciones dependerán de si es utilizada como un predicado o no. A continuación se muestran los dos casos y las reglas de relación que se considerarán.

Subclase: subClassOf

Instancia: instanceOf

Subpropiedad: subPropertyOf

- Caso 1: Sea a el recurso original que es utilizado como objeto o sujeto en un triple, entonces, todos los posibles valores de b que cumplen con que existe alguna de las siguientes relaciones es utilizado como modificación y es agregado al conjunto $Alt(a)$:
  - $a\ subClassOf\ b$
  - $b\ subClassOf\ a$

Las dos modificaciones anteriores reflejan la jerarquía de clases, pero para aumentar un poco más la búsqueda de otros recursos dentro del vecindario, se agregan las dos siguientes para generalizar una instancia a su clase y además, considerar las clases que son vecinas:

- $a\ instanceOf\ b$
- $a\ subClassOf\ c\ \wedge\ b\ subClassOf\ c$

- Caso 2: Para el caso en que a es una propiedad, entonces, todos los valores de b que cumplan con alguna de las siguientes relaciones es utilizado para modificar y es agregado al conjunto $Alt(a)$:

  - $a\ subPropertyOf\ b$
  - $b\ subPropertyOf\ a$

  Las dos relajaciones anteriores reflejan la jerarquía entre propiedades, pero para aumentar un poco más la búsqueda de otros recursos dentro del vecindario, se agrega la siguiente para considerar las propiedades que son vecinas:

  $a\ subPropertyOf\ c\ \wedge\ b\ subPropertyOf\ c$

Para aumentar más aún la búsqueda en el vecindario de a, se hace uso de las siguientes reglas de inferencia:

ins := instanceOf

sc := subClassOf

sp := subPropertyOf

$$\frac{(\text{a ins b}) \wedge (\text{b sc c})}{\text{a ins c}}$$

$$\frac{(\text{a sc b}) \wedge (\text{b sc c})}{\text{a sc c}}$$

$$\frac{(\text{a sp b}) \wedge (\text{b sp c})}{\text{a sp c}}$$

Así se pueden considerar, por ejemplo, si se usa la segunda regla de inferencia, las superclases de segundo grado como una superclase más de a y de esta forma, agregar un número mayor de URIs al conjunto $Alt(a)$.

Con estas tres reglas de inferencias se pueden hacer otras que son un poco más complejas, permitiendo hacer una búsqueda un poco mayor al considerar otro tipo de relaciones como pueden ser:

$$\frac{(\text{a sc b}) \wedge (\text{b sc c}) \wedge (\text{c sc d})}{\text{a sc d}}$$

$$\frac{(\text{a ins b}) \wedge (\text{b sc c}) \wedge (\text{c sc d}) \wedge (\text{d sc f})}{\text{a ins f}}$$

$$\frac{(\text{a sp b}) \wedge (\text{b sp c}) \wedge (\text{c sp d})}{\text{a sp d}}$$

Con las relaciones y reglas de inferencia indicadas ya se abarca una parte de la obtención de las URIs para el conjunto $Alt(a)$ relacionadas a la primera causante, por lo que a continuación se indican las relaciones utilizadas para obtener URIs que modifiquen el sentido de la consulta original. Como se mencionó en la sección anterior, para esto, se hará uso de OWL, en específico OWL 2 QL. Esta ontología provee un conjunto de caracteristicas de las cuales se pueden hacer uso para modificar la consulta las siguientes:

1. Si a es utilizado como objeto o sujeto en un *triple*, entonces se usan las relaciones:
   - $a\ rdfs{:}subClassOf\ b$
   - $a\ owl{:}equivalentClass\ b$
   - $a\ owl{:}disjointClass\ b$
2. Si a es utilizado como predicado en un *triple*, entonces se usan las relaciones:
   - $a\ rdfs{:}subPropertOf\ b$
   - $a\ owl{:}inverseOf\ b$
   - $a\ owl{:}equivalentProperty\ b$
   - $a\ owl{:}disjointProperty\ b$

De esta manera, se cuenta con un mayor abanico de maneras de obtener recursos, con algunos variando más el sentido de la consulta que otros. Al igual que se hizo inicialmente con la jerarquía de clases, se puede razonar para considerar conocimientos implícitos. Como subpropiedades de la inversa, etc. El problema de hacer esto es que se podría llegar a recursos que se alejan bastante del recurso original, por lo que esta opción se omitirá.

El método que se utilizará para obtener todos los recursos relacionados sin que estos se alejen del original es generando niveles de generalización y de especialización, en donde a cada

uno de los recursos obtenidos por nivel se le aplican las reglas de modificación según la ontología. En la figura 12 se muestra parte de un ejemplo del arbol de las relaciones realizadas en cada nivel.

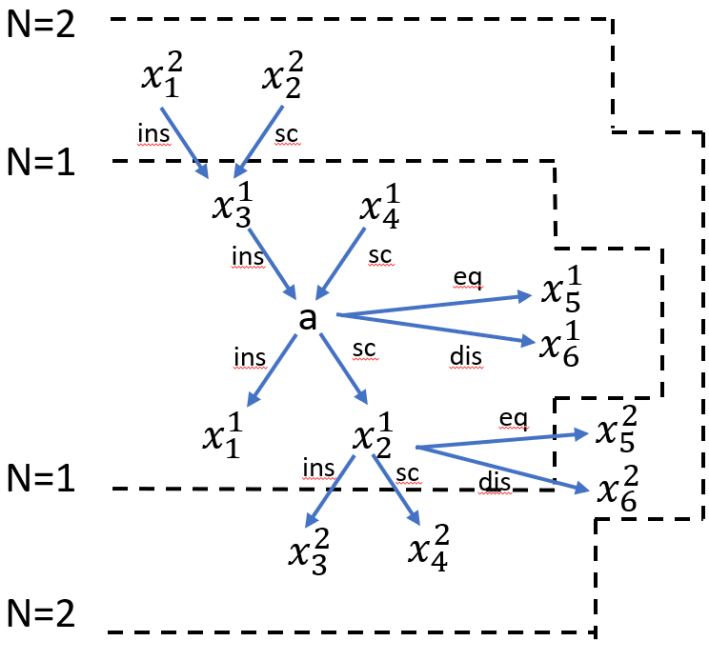


Figura 12: Ejemplo de las relaciones generadas y utilizadas en los niveles 1 y 2, en donde a cada recurso se le obtienen los recursos que cumplan con las relaciones definidas, los cuales pertenerán al nivel siguiente (subClase = sp, instancia = ins, equivalentClass = eq, disjointClass = dis).

Fuente: Elaboración propia.

Empezando con la URI inicial a, se le aplican todas las reglas de relacion de jerarquía que tienen solamente un largo de uno y las reglas de ontología. Seguido a esto, se aplican las reglas de jerarquía, de especialización o de generalización de largo dos, a la vez que se aplican las reglas de jerarquía del nivel anterior con la adición de las relaciones de ontología. Este proceso se realiza sucesivamente hasta el nivel n deseado, en donde las URIs obtenidos se encontrarán a una distancia n de a.

De esta manera se obtendrán todas las URIs de los recursos que potencialmente podrían ser utilizados como reemplazo de a en la consulta Q_{ini}. Como paso final, se deben descartar todas aquellas que al ser reemplazadas en Q_{ini} generen una consulta Q' que no obtenga resultados.

3.4.3. Algoritmo

Ya se vio cómo obtener un conjunto de alternativas para cada uno de las URIs de Q_{ini}. También se describió la salida deseada y las formas de obtener estas salidas, por lo que a conti-

nuación se describirá el algoritmo propuesto en su totalidad.

Como entrada del algoritmo se cuenta con una consulta inicial Q_{ini} que obtiene resultados vacíos, una base de datos RDF D, un máximo M de alternativas obtenidas y un limite N de niveles en las relaciones usadas.

Se denotará $rec(Q)$ al conjunto de URIs presentes en el *graph pattern* de Q.

Sea $generateRelations(u, i)$ la función que genera el conjunto de URIs que se relacionan con la URI u por medio de un camino de largo i y que cumplen con las relaciones indicadas en la sección anterior.

Por cada una de las URIs $a_i \in rec(Q_{ini})$ se deben obtener todas las URIs que cumplan las relaciones y que estén a una distancia d de 1, es decir, $generateRelations(a_i, 1)$. Una vez obtenido este conjunto de URIs, por cada $b_j \in generateRelations(a_i, 1)$ se reemplaza b_j por a_i en Q_{ini} y si esta nueva consulta Q' obtiene resultados no vacíos, entonces se agrega la URI b_j al conjunto $Alt(a_i)$. Luego se incrementa d en 1 y se realiza una nueva iteración que considerará URIs que se encuentren un poco más lejos.

El algoritmo finaliza una vez se alcance la cantidad máxima determinada de alternativas M o si se realiza una cantidad N de iteraciones.

Algorithm 1: Obtención de los conjuntos de recursos alternativos

Initial: Q_{ini}; N; M; D ;
Result: Conjuntos de URIs alternativos
$Alt(a) = \{\emptyset\} \qquad \forall a \in rec(Q_{ini})$;
$it = 1$;
while $(\sum_{a \in rec(Q_{ini})} |Alt(a)| < M) \wedge (it < N)$ **do**
  for $a \in rec(Q_{ini})$ **do**
    for $b \in generateRelations(a, it)$ **do**
      $Q' \leftarrow Q_{ini}.replace(a, b)$;
      if $[\![Q']\!]_D \neq \emptyset$ **then**
        $Alt(a) \leftarrow Alt(a) \cup b$;
      end
    end
  end
  $it++$;
end

3.5. Complejidad del algoritmo

Lo que se busca realizar con el algoritmo es razonar utilizando todas las características descritas en la sección 3.4.2, esto con la intención de encontrar caminos que permitan obtener nuevas URIs que tengan determinadas relaciones. Entonces, para determinar la complejidad se considerará la *description logic* (DL) a la que pertenece el conjunto de características usadas.

Como la ontología que se indicó para utilizar fue la OWL 2 QL, entonces la DL que se está utilizando pertenece a la familia *DL-Lite* y, el razonamiento de estas DL está en *P* para su complejidad combinada (Considerando toda la *Knowlegde base*) y en *LogSpace* para la complejidad de datos (Considerando solo ABox) [Calvanese *et al.*, 2007].

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

Para realizar la validación de la solución propuesta en el capítulo anterior, se implementará el algoritmo en el software RDFExplorer [Vargas *et al.*, 2019], el cual es un sistema para realizar consultas SPARQL mediante una interfaz gráfica. Para esto, primero se describirá el entorno en el que se trabajará seguido de una explicación sobre cómo se extenderá RDFExplorer para incluir la solución. Finalmente se explicará su implementación, de la cual se obtendrán algunos resultados que luego serán comentados.

4.1. Contexto y entorno

La solución propuesta en el capítulo anterior permite obtener consultas alternativas para una consulta inicial realizada en SPARQL, por lo que para validar lo previamente propuesto, se debe desarrollar un software que permita realizar consultar SPARQL junto con el algoritmo descrito, o se puede implementar en un sistema ya existente que realiza esta función. Se considerará la segunda opción, tomando como base el software RDFExplorer y extendiéndolo para que pueda hacer uso de la solución propuesta.

RDFExplorer ya se describió en detalle en la sección 2.5, en específico su funcionamiento junto al lenguaje que utiliza. A continuación se describirá brevemente el conjunto de datos con el que se trabajará.

4.1.1. Conjunto de datos utilizado

La versión de RDFExplorer que se utilizará trabaja utilizando solamente la base de datos de *wikidata* [Vrandečić y Krötzsch, 2014], la cual es una base de conocimiento (base de datos para la gestión del conocimiento) abierta y libre que puede ser leída y editaba tanto por humanos como por máquinas. Cuenta con aproximadamente 80.000.000 URIs.

En específico, las URIs que se utilizarán más adelante para realizar la implementación se presentan en la tabla 2 para ser nombradas de manera simple. Además se definen los siguientes prefijos[1]:

Prefix wd: <http://www.wikidata.org/entity/>

Prefix wdt: <http://www.wikidata.org/prop/direct/>

[1] http://www.wikidata.org

Tabla 2: URIs utilizadas en la implementación con sus respectivas *labels*.
Fuente: Elaboración Propia.

URI	*label*
wdt:P31	instanceOf
wdt:P279	subClassOf
wdt:P1647	subPropertyOf
wdt:P1696	inverseProperty

4.2. Extensión de RDFExplorer

Para implementar la obtención de alternativas, el lenguaje de RDFExplorer se extenderá de manera que cada constante del grafo tenga un conjunto de otras constantes que al reemplazarlas, el grafo resultante obtenga soluciones. Para hacer esto, primero se debe definir una nueva operación que modifica un grafo $\mathsf{G} = (\mathsf{N}, \mathsf{E})$. Esta modificación se realiza reemplazando:

- Un nodo constante x: En este caso, se debe eliminar este nodo, agregar un nuevo nodo constante y modificar los arcos entrantes o salientes a x.
- Un arco etiquetado con una constante x: En este caso, se debe eliminar el arco etiquetado con x y agregar en su lugar uno con una etiqueta diferente.

Antes de definir esta operación, se denota al conjunto $Re(\mathsf{G}, x)$, como el conjunto de arcos del grafo G que deberán ser modificados, i.e. los arcos que contienen la constante x:

$$Re(\mathsf{G}, x) \coloneqq \{e \in \mathsf{E} \mid \exists\, n_1, l, n_2 : e = (x, l, n_2) \vee e = (n_1, x, n_2) \vee e = (n_1, l, x)\}$$

Con lo anterior definido, pasamos a definir la siguiente operación:

- Modificar (reemplazar) una constante x por otra constante c en el grafo G:

$$M(\mathsf{G}, x, c) \coloneqq \begin{cases} Mn(\mathsf{G}, x, c) \text{ si } x \in \mathsf{N} \\ Me(\mathsf{G}, x, c) \text{ si } x \notin \mathsf{N} \end{cases}$$

Donde:

$$\begin{aligned} Mn(\mathsf{G}, x, c) \coloneqq (&\mathsf{N} - \{x\} \cup \{c\}, \mathsf{E} - Re(\mathsf{G}, x) \\ &\cup \{(c, l, n_2) \mid (x, l, n_2) \in Re(\mathsf{G}, x)\} \\ &\cup \{(n_1, l, c) \mid (n_1, l, x) \in Re(\mathsf{G}, x)\}) \qquad where \{n_1, n_2\} \subseteq \mathsf{N} \text{ and } l \in (\mathsf{I} \cup var(\mathsf{G})) \end{aligned}$$

$$Me(\mathsf{G}, x, c) \coloneqq (\mathsf{N}, \mathsf{E} - Re(\mathsf{G}, x) \\ \cup \{(n_1, c, n_2) \mid (n_1, x, n_2) \in Re(\mathsf{G}, x)\}) \quad where\{n_1, n_2\} \subseteq \mathsf{N}$$

Se define $Alt(x)$ como el conjunto de IRIs (**I**) que al ser reemplazas en G en el lugar de x, la consulta resultante retornará resultados no nulos:

$$Alt(x) \coloneqq \{c \in \mathbf{I} \mid [\![M(\mathsf{G}, x, c)]\!]_D \neq \emptyset\} \tag{1}$$

Inicialmente este conjunto se encuentra vacío y es definido para todas las constantes pertenecientes al conjunto **I** de IRIs y que se encuentren en el grafo G.

$$Alt_0(x) = \{\emptyset\} \qquad \forall\{x \in \mathbf{I} \mid x \in \mathsf{N} \vee \exists n_1, n_2 : (n_1, x, n_2) \in \mathsf{E}\}$$

4.3. Interfaz de la extensión

Para representar los conjuntos $Alt(i)$ en la interfaz gráfica, se hará uso de la pestaña de edición que tienen todas las constantes, en donde se entregará el listado de URIs pertenecientes al elemento i bajo la lista llamada $Recommends$, en la figura 13 se muestra un ejemplo de cómo se ve el conjunto $Alt(\text{“}\textit{personal name}\text{”})$ en donde se tienen 5 valores.

Al momento de seleccionar uno de estos valores(mediante el símbolo +), se agrega en la lista *“Constraints”* del elemento (esto es por el momento, luego se modificará para que se reemplace directamente), reflejándose instantáneamente en el grafo.

Debido a que al reemplazar alguno de estos valores, por definición, el grafo nuevo sí retornará resultado, no se requiere recalcular el conjunto, por lo que este no se modificará hasta que sea necesario, permitiendo que el usuario pueda ir cambiando entre las alternativas entregadas en caso de que alguna seleccionada no corresponda con lo que busca. En el ejemplo mostrado en la figura, el usuario podría en un principio escoger *“religión”*, para luego darse cuenta de que los resultados no se relacionan a lo esperado, por lo que luego puede decidir seleccionar otra alternativa como *“given name”* o, por mera curiosidad, probar con todos los valores de la lista.

Para este caso, se hace uso de la característica de RDFExplorer de que al momento de seleccionar un elemento, se abre una pestaña con herramientas e información relacionada a este. Si se quisiera implementar una herramienta que genere alternativas a la consulta que contenga 2 o más elementos diferentes al mismo tiempo, por el momento no hay una manera de mostrar claramente esta información, ya que no existe una forma de mostrar información relacionada a, por ejemplo, un *triple*.

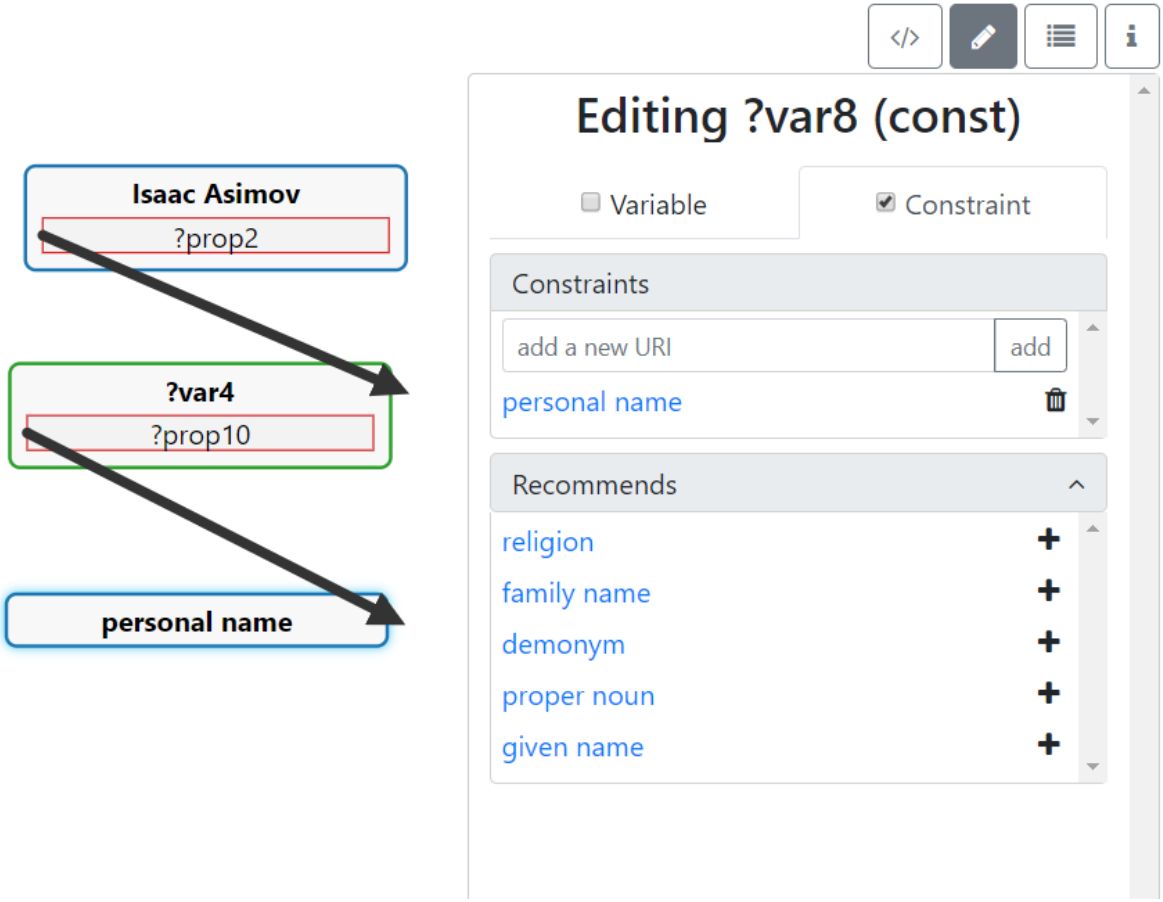


Figura 13: Ejemplo de la interfaz extendida para el nodo seleccionado *"personal name"*. Fuente: Elaboración propia.

4.4. Obtención de los conjuntos

Como se mencionó anteriormente, RDFExplorer es una aplicación que no cuenta con un acceso directo a los datos, por lo que se deben realizar constantes consultas a un servidor mediante un endpoint SPARQL para obtenerlos. Debido a esto, cuando se realiza una consulta, se debe considerar el tiempo extra de envío y respuesta del servidor. Lo anterior implica que la implementación de algún algoritmo de inferencia o un régimen de *entailment* resulte ser muy costoso en tiempo. Es por esto por lo que para la obtención de los conjuntos $Alt(i)$, se utilizará una cantidad fija y previamente definida de reglas de modificación.

Las reglas de modificación describen la relación que debe existir entre el recurso inicial y el que tomará su lugar dentro de la consulta una vez realizada la modificación.

Sea a un recurso, su conjunto $Alt(a)$ contendrá todos los recursos r que cumplan con la definición 1 y alguna de las siguientes relaciones:

- Relacionadas a la relajación de la consulta:

    1. SuperClases de primer orden:
       (a subClassOf r)
    2. SuperClases de segundo orden:
       (a subClassOf b) . (b subClassOf r)
    3. SuperClases de tercer orden:
       (a subClassOf b) . (b subClassOf c) . (c subClassOf r)
    4. SuperClases de cuarto orden:
       (a subClassOf b) . (b subClassOf c) . (c subClassOf d) . (c subClassOf r)
    5. Vecinos de clase:
       (a subClassOf c) . (r subClassOf c)
    6. Vecinos de propiedad:
       (a subPropertyOf c) . (r subPropertyOf c)
    7. Clase a la que pertenece:
       (a instanceOf r)
    8. Clase de segundo orden a la que pertenece:
       (a instanceOf b) . (b subClassOf r)
    9. Clase de tercer orden a la que pertenece:
       (a instanceOf b) . (b subClassOf c) . (c subClassOf r)
    10. Clase de cuarto orden a la que pertenece:
        (a instanceOf b) . (b subClassOf c) . (c subClassOf d) . (d subClassOf r)
    11. SuperPropiedad de primer orden:
        (a subPropertyOf r)
    12. SuperPropiedad de segundo orden:
        (a subPropertyOf b) . (b subPropertyOf r)
    13. SuperPropiedad de tercer orden:
        (a subPropertyOf b) . (b subPropertyOf c) . (c subPropertyOf r)

- Relacionadas a la ontología OWL 2 QL:

    14. Axiomas de subclases: (Ya se encuentra abordada parcialmente en las anteriores.)
        (r subClassOf a)
    15. Inclusión de propiedades: (Ya se encuentra abordada parcialmente en las anteriores.)
        (r subPropertyOf a)

16. Propiedades inversas:
    (a inverseProperty r)

    El resto de caracteristicas presentes en OWL 2 QL no cuentan con un equivalente en el vocabulario de *wikidata*, a excepción de “Clase equivalente” y “propiedad equivalente”, las que no resultan de utilidad en este caso, debido a que solo se esta utilizando una base de datos.

Para referirse más rápido a las relaciones utilizadas para nodos o arcos, se hará uso de la siguiente referencia:

- **Caso 1:** Modificaciones 1 - 5, 7 – 10, 14 son utilizadas para recursos que son utilizados como nodo.
- **Caso 2:** Modificaciones 6,11,12,13,15,16 son utilizadas para recursos que son utilizados como etiqueta de un arco.

Las reformulaciones o modificaciones realizadas a un *triple* dependerán de la forma que tenga este. A continuación se indica cuales se deben realizar, para simplificar la notación, se utilizará la expresión XXX para indicar la forma de un *triple* en donde X puede ser V (variable) o C (constante):

- VVV:

  Como no es posible hacer ninguna modificación que entregue más o diferentes resultados, no se realiza ninguna modificación.

- VVC_1:

  Se le aplican todas las modificaciones del caso 1 a la constante C_1.

- VC_1V:

  Se le aplican todas las modificaciones del caso 2 a la constante C_1.

- VC_1C_2:

  Por cada una de las constantes se deja una intacta y se le aplican a la otra todas las modificaciones previamente indicada según corresponda, es decir, para C_1, se considera la implementación de VC_1V.

- C_1VV:

  Se le aplican todas las modificaciones del caso 1 a la constante C_1.

- C_1VC_2:

  Por cada una de las constantes se deja una intacta y se le aplican a la otra todas las modificaciones previamente indicada según corresponda, es decir, para C_1, se considera la implementación de C_1VV.

- C_1C_2V:

  Por cada una de las constantes se deja una intacta y se le aplican a la otra todas las modificaciones previamente indicada según corresponda, es decir, para C_1, se considera la implementación de C_1VV.

- CCC:

  En este caso no se realizan modificaciones, debido a que, como se explicará más adelante, estas modificaciones se realizarán al último *triple* agregado que genere problemas, lo que no puede darse con un *triple* de la forma CCC.

4.5. Implementación

Como se indicó, en RDFExplorer se construye el grafo de manera gradual, agregando secuencialmente los nodos y arcos en donde, al momento de agregar uno de estos, se recalculan las soluciones. Por esto, es una buena suposición que cuando se agrega un arco *(triple)* y los resultados pasan a ser nulos, es este último *triple* el causante del problema. Por esto, la implementación se enfocará en implementar el algoritmo solamente para este último *triple* agregado.

De manera general, cada vez que se agrega un *triple* que contiene alguna variable, se realiza la consulta relacionada al grafo obtenido. Luego de que se obtiene la respuesta de esta consulta, si no contiene soluciones entonces se buscan recomendaciones para modificar este *triple* de forma tal que las consultas obtenidas por estas modificaciones den resultados no vacíos, i.e. calcular los conjuntos $Alt(a)$ para los recursos de este *triple*.

Para implementar esto, primero se crea una copia del grafo G llamado G′. Como la traducción en RDFExplorer del grafo G a SPARQL solo depende del conjunto de arcos E, la obtención de los conjuntos $Alt(a)$ trabajará sobre el conjunto E′ de arcos del grafo G′. Como la implementación se realizará sobre el último triple agregado t, lo primero que se debe hacer es identificar a este en el conjunto E′.

Para obtener los conjuntos $Alt(t_i)$ de los recursos t_i presentes en t, se deben obtener los recursos r que cumplan con las reglas de modificación descritas previamente a la vez que cumplen con su definición (1). Para asegurarse que cumplan con alguna regla de modificación, se realiza una consulta utilizando como *pattern graph* una de las relaciones descritas con la diferencia que el recurso r se reemplaza por una variable $?r$. La manera de la obtención de los conjuntos $Alt(t_i)$ dependerá de la forma que tenga t:

- VVC:

  Sea el triple $t = ?s\ ?p\ o$. Se hacen las consultas relacionadas a las modificaciones del caso 1, algunos ejemplos de los *pattern graph* de estas son:

1. superClases de o:

$$\{o \textit{ subClassOf } ?r.\}$$

2. subClases de o:

$$\{?r \textit{ subClassOf } o.\}$$

3. vecinos de clase de o:

$$\{?r \textit{ subClassOf } ?aux.$$
$$o \textit{ subClassOf } ?aux.\}$$

Al tener cada uno de los posibles recursos en las soluciones de la variable $?r$, se hace una consulta con cada una de estas en lugar de o para filtrar los resultados que sí entregarán resultados, i.e. el *triple* t se modifica por $?s\ ?p\ r$ donde r es un recurso obtenido de estas consultas. Las que retornan soluciones se agregan a el conjunto $Alt(o)$.

Ejemplo:

Se tiene la consulta inicial con el *pattern graph* $\{a\ ?x\ ?y\ .\ ?y\ ?z\ b\}$, la cual no obtiene soluciones. Se utiliza la modificación $\{b\ subClassOf\ r\}$, por lo que se realiza la consulta:

SELECT $?r$ WHERE

$\{$

$b\ wdt{:}P279\ ?r.$

$\}$

Obteniendo las siguientes soluciones:

$?r$
l
m
n

Por cada una de las soluciones obtenidas se modifica la consulta original y se realizan las consultas con los siguientes *pattern graph*:

1.

$$\{$$
$$a\ ?x\ ?y.$$
$$?y\ ?z\ l.$$
$$\}$$

2.
$$\{ \; a \; ?x \; ?y. \; ?y \; ?z \; m. \; \}$$

3.
$$\{ \; a \; ?x \; ?y. \; ?y \; ?z \; n. \; \}$$

Si una de estas consultas retorna resultados, entonces se agrega el valor utilizado para hacer la modificación en el conjunto $Alt(o)$

- VCV:

  Sea el *triple* $t = ?s \; p \; ?o$. Se hacen las consultas relacionadas a las modificaciones del caso 2, algunos ejemplos de los *pattern graph* de estas son:

  1. superPropiedades de p:
  $$\{p \; \textit{subPropertyOf} \; ?r.\}$$
  2. subPropiedades de p:
  $$\{?r \; \textit{subPropertyOf} \; p.\}$$

  Al tener cada uno de los posibles recursos en las soluciones de $?r$, se hace una consulta con cada una de estas en lugar de p, i.e. el *triple* t se modifica por $?s \; r \; ?o$ donde r es una propiedad obtenida de estas consultas. Las que retornan soluciones se agregan al conjunto $Alt(p)$.

- VCC:

  Sea el *triple* $t = ?s \; p \; o$. Se hacen los 2 casos anteriores para tener alternativas tanto para p como para o.

- De igual manera se realiza el proceso para CVV, CVC y CCV.

En los casos restantes: CCC y VVV, no se realizan modificaciones, ya que en el primer caso, cuando se agrega un *triple* con esta forma, en RDFExplorer no se actualizan las soluciones del grafo. Para el segundo caso, como solo contiene variables no es posible hacer alguna modificación más que eliminar el triple completo.

4.6. Optimización de las consultas

En la sección anterior se vio y describió la implementación para obtener los conjuntos de recursos alternativos, i.e. los conjuntos $Alt(t_i)$. Este proceso básicamente se puede describir en dos pasos:

1. Obtener todos los recursos que cumplan con alguna de las relaciones.
2. Filtrar los resultados del paso anterior para quedarse solamente con los que cumplen con la definición (1).

Más detalládamente, el paso 2 se realiza mediante la ejecución de una consulta por cada recurso obtenido del paso 1. Esto puede provocar que se deba realizar una gran cantidad de consultas y, a su vez esto genera un importante problema en el tiempo de obtención de los resultados.

Este problema de tiempo se debe a que RDFExplorer tiene un parámetro de *delay*, el cual es utilizado para agregar un tiempo de espera entre las ejecuciones de las consultas. Este parámetro es usado debido a que RDFExplorer cuenta con la caracteristica de que realiza todas las consultas SPARQL de manera asíncrona, lo que en ocasiones puede provocar que se genere una enorme cantidad de consultas en una ventana de tiempo muy pequeña, lo que puede generar inconvenientes, ya que si el servidor al que se les realiza las peticiones recibe una gran cantidad de peticiones en un pequeño espacio de tiempo, no aceptará a la mayoría de estas, generando que existan consultas que no obtengan respuesta.

Para solucionar este problema se busca reducir la cantidad de consultas realizadas al servidor, con la intención de disminuir lo más posible el porcentaje del tiempo total que es utilizado solamente por el tiempo de *delay*. Se consideraron dos implementaciones altenativas para esto, cada una con sus ventajas y desventajas:

1. Reducir la cantidad total de consultas del paso 2, realizando multiples consultas en una única consulta mediante el uso del operador UNION.
2. Unir el paso 1 con el paso 2, integrando las relaciones del paso 1 a la consulta inicial mediante nuevos *triples* utilizando el operador JOIN.

En las siguientes subsecciones se describirán cada una de las implementaciones, junto con sus ventajas y desventajas.

4.6.1. Múltiples consultas en una única

En esta implementación se busca hacer lo mismo, pero al momento de realizar el paso 2 en lugar de hacer una consulta por cada resultado del paso 1, multiples consultas se unen en

una sola mediante el uso del operador UNION de SPARQL. Este método se denominará como implementación *union*.

Para esto se selecciona una variable y se le da un identificador por cada consulta, por ejemplo $?x$ pasa a ser $?x1$ $?x2$ y $?x3$ en el caso de unir tres consultas. De esta forma se podrá identificar cual de las consultas obtiene resultados. A continuación se presenta cual sería la consulta a utilizar en el paso 2 del ejemplo mostrado para el caso *VVC*.

SELECT $?x1$ $?x2$ $?x3$ WHERE {

{ a $?x1$ $?y$.

$?y$ $?z$ l . } UNION

{ a $?x2$ $?y$.

$?y$ $?z$ m . } UNION

{ a $?x3$ $?y$.

$?y$ $?z$ n . }

}

Luego se debe revisar cuales de las variables consultadas obtuvieron resultados, y según corresponda se agregarán los recursos al conjunto $Alt(b)$. Por ejemplo, si $?x1$ tiene resultados entonces se agrega l al conjunto, si $?x2$ tiene resultados entonces se agrega m, etc.

De esta manera se pueden realizar una cantidad de n consultas en solamente una. Esto permite reducir el tiempo generado por *delay* t_d del paso 2 a $\frac{t_d}{n}$. Por lo que en el caso más simple, es decir con n igual a 1, el tiempo será igual al de la implementación anterior, por lo que resulta conveniente utilizar esta en su lugar.

Por otro lado existe un problema con esta implementación, las consultas SPARQL realizadas al servidor tienen un límite de tamaño, por lo que se debe tener en consideración la cantidad de consultas que se están intentando unir. Mas aún, si la consulta inicial tiene una cantidad de *triples* mayor, entonces el limite de uniones se ve reducido.

4.6.2. Integrar restricciones a la consulta inicial

En esta implementación, la que se denominará como implementación *join*, se intenta incluir todas las consultas del paso 2 en las del paso 1. Para esto se modifica la consulta inicial para incluir las relaciones que se deseen utilizar, agregando los *triples* de esta relacion y reemplazando el recurso objetivo con una nueva variable. A continuación se muestra cual sería la consulta a utilizar considerando el ejemplo del caso VVC.

SELECT $?r$ WHERE {

$a\ ?x\ ?y$.

$?y\ ?z\ ?r$.

$b\ wdt{:}P279\ ?r$.

}

Finalmente los resultados obtenidos de la variable $?r$ son agregados a $Alt(b)$.

De esta manera se reduce en su totalidad el tiempo del paso 2, ya que no es necesario realizarlo. Con esto, el principal factor a considerar para comparar esta implementación con las anteriores es el tiempo que tomará al servidor procesar las consultas.

El problema de esta implementación es que se incrementa la cantidad de *triples* y variables de las consultas del paso 1, lo que puede generar que el tiempo necesario para que el servidor obtenga los resultados sea mucho mayor que en las otras implementaciones.

4.6.3. Comparación de las implementaciones

Como se vio, ambas implementaciones reducen el tiempo relacionado al *delay*, por lo que se deben comparar para ver cual resulta ser mejor para utilizar.

Como la implementación por *join* tiene problemas relacionados al aumento de la complejidad de la consulta inicial, las comparaciones entre ambas implementaciones se realizarán según la cantidad de *triples* de la consulta original, de esta manera se pueden utilizar ambas implementaciones en el caso de que cada una resulte ser mejor según el número de *triples*.

Cuando la consulta inicial solamente presenta un *triple* en el *graph pattern*, la implementación con *join* resulta ser más eficiente. Un resumen de los tiempos obtenidos de un pequeño conjunto de consultas de prueba se muestra en la tabla 3. Esto resulta ser algo esperable, debido a que la consulta resulta ser bastante pequeña, por lo que el servidor no tarda en procesarla.

Tabla 3: Tiempos promedios de las implementaciones aplicadas a consultas de solo un *triple*. Fuente: Elaboración Propia.

	tiempo[ms]		
	promedio	mínimo	máximo
join	**808**	**780**	**900**
union	1637	1193	2661

Con 2 triples, se obtienen resultados muy variantes, dando en algunos casos tiempos mucho

menores para la implementación *union* mientras que en otros casos resulta ser más rápida la implementación *join*. Al hacer un breve análisis en mayor detalle, se encuentra que estas diferencias de tiempo se deben a la estructura que presenta el *graph pattern* y a la posición en este del recurso al que se le estan buscando alternativas. Con esto, se podría entrar a analizar en mayor detalle cuando conviene utilizar una u otra implementación al subdividir los casos más allá de solamente por el número de *triples*, pero por el momento simplemente se considerará el uso de la implementación *union* para este caso, la cual no presenta tiempos relativamente altos ni grandes variaciones.

Tabla 4: Tiempos promedios de las implementaciones aplicadas a consultas de dos *triple*.
Fuente: Elaboración Propia.

	tiempo[ms]		
	promedio	mínimo	máximo
join	7030	**500**	40000
union	**1446**	893	**2933**

Desde 3 *triples* en adelante, el tiempo de la implementación por *union* resulta ser mucho menor. Esto se debe principalmente a que al utilizar *join*, en varios casos no se obtienen resultados por parte del servidor debido a un *timeout* al procesar la consulta.

Finalmente, se utilizará la implementación *join* solamente para consultas de tamaño 1, mientras que para el resto de casos se utilizará *union*.

Tabla 5: Implementación a utilizar según cantidad de *triples*.
Fuente: Elaboración Propia.

Cantidad de *triples*	Implementación
1	*join*
2	*union*
3+	*union*

4.7. Resultados

Finalmente, como última parte de la validación se analizarán los resultados de la implementación realizada. Estos resultados son obtenidos luego de realizar un pequeño conjunto de pruebas en RDFExplorer, de las que resultan importantes los siguientes valores:

- La cantidad de resultados que se obtuvo por parte del algoritmo implementado, e.g. consultas alternativas.
- El tiempo que se tarda en completar el algoritmo una vez que se identifica que se obtuvieron resultados vacíos.

Para realizar estas pruebas se utilizaron las consultas que se muestran en la tabla 6. Estas consultas obtienen resultados vacíos y su número de *triples* varía desde 1 hasta 3, además de considerar algunas de patrón tipo estrella, de tipo cadena y mixto.

Tabla 6: Consultas utilizadas para realizar pruebas de resultados.
Fuente: Elaboración Propia.

Nombre consulta	SPARQL
Q1	SELECT ?var1 ?prop1 ?prop2 WHERE{ ?var1 ?prop1 wd:Q457793 . ?var1 ?prop2 wd:Q37226 . }
Q2	SELECT ?var1 ?prop1 ? prop2 WHERE{ wd:Q34981 ?prop1 ?var1 . ?var1 ?prop2 wd:Q16575965 . }
Q3	SELECT ?var1 ?prop1 ?prop2 ?prop3 WHERE{ ?var1 ?prop1 wd:Q24925 . ?var1 ?prop2 wd:Q8261 . ?var1 ?prop3 wd:Q21 . }
Q4	SELECT ?var1 ?var2 ?prop1 WHERE{ ?var1 ?prop1 wd:Q571 . ?var1 wdt:P2561 ?var2 . }
Q5	SELECT ?var1 ?var2 ?prop1 ?prop2 WHERE{ ?var1 wdt:P50 ?var2 . ?var2 ?prop1 wd:Q121594 . ?var2 ?prop2 wd:Q16575965 . }
Q6	SELECT ?var1 ?prop1 ?prop2 ?prop3 WHERE{ wd:Q127367 ?prop1 ?var1 . ?var1 ?prop2 wd:Q33999 . ?var1 ?prop3 wd:Q467 . }
Q7	SELECT ?prop1 WHERE{ wd:Q11424 ?prop1 wd:Q571 . }

Como se indicó anteriormente, la implementación solamente aplica el algoritmo al último *triple* agregado, por lo que cada consulta se realiza un número de veces igual a la cantidad de *triples*, variando en cada iteración al último *triple* agregado. Esto con la intención de aplicar el algoritmo a toda la consulta, permitiendo comprobar la cantidad de consultas alternativas obtenidas en total.

De esta manera, primero se analizará la cantidad de alternativas obtenidas por cada *triple* presente en cada consulta y el tiempo obtenido en cada una. Al realizar estas consultas en RDFExplorer, se obtuvieron los tiempos y cantidad de alternativas presente en la tabla 7.

De estos resultados se puede ver que respecto al tiempo, se tarda principalmente entre 1 y 2 segundos en cada *triple* en obtener resultados. Por el lado de la cantidad de alternativas obtenidas, en general se obtienen entre 0 y 3. La mayoría de los recursos que no obtieron

Tabla 7: Tiempos y cantidad de resultados alternativos al realizar las consultas de prueba. Fuente: Elaboración Propia.

		Tiempo[ms]	Cantidad alternativas
Q1	t_1	1742	3
	t_2	1599	2
Q2	t_1	479	0
	t_2	1165	3
Q3	t_1	2713	2
	t_2	3146	2
	t_3	1802	0
Q4	t_1	4249	14
	t_2	1165	11
Q5	t_1	668	1
	t_2	2344	1
	t_3	2612	1
Q6	t_1	845	0
	t_2	1476	0
	t_3	1654	1
Q7	t_1	887	2

alternativas son instancias y no clases, lo que tiene sentido al ver que las relaciones que se utilizaron en su mayoría obtienen recursos que son clases.

Para hacerse una idea de como sería si el algoritmo se aplicara a toda la consulta en lugar de solo al último *triple*, en el peor de los casos el tiempo sería igual a la sumatoria de los tiempos individuales obtenidos por cada *triple* y la cantidad de consultas alternativas sería igual a la sumatoria de las alternativas de todos los recursos. En la tabla 8 se muestran estos valores.

Analizando estos resultados se puede ver que el principal problema es el tiempo que se tardaría en realizar el algoritmo a toda la consulta, por lo que se debe buscar una mejor implementación si se espera aplicar el algoritmo a toda la consulta. Mas aún, si el número de consultas alternativas es muy bajo, también se deberá incrementar el alcance de las relaciones que se están utilizando, con la intención de considerar recursos que por el momento no lo están, y esto implica un aumento en el tiempo de respuesta.

Tabla 8: Tiempos y cantidad de resultados alternativos total por consulta realizada. Fuente: Elaboración Propia.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Tiempo[ms]	3341	1644	7661	5414	5624	3975	887
Cantidad alternativas	5	3	4	25	3	1	2

Se observa que se obtienen por lo general almenos una alternativa, lo cual puede ser sufi-

ciente para el objetivo de incrementar el número de resultados obtenidos, pero esto puede no ser suficiente si se quiere utilizar a estas consultas como *feedback*. Por lo que se requiere a futuro mejorar la implementación para mejorar la cantidad de alternativas, cuidando que a la vez que no se vea afectado el tiempo de manera negativa.

Finalmente, en la tabla 9 se muestran la cantidad total de resultados que se obtienen al realizar todas las consultas alternativas generadas.

Tabla 9: Cantidad de resultados totales obtenidos por medio de todas las recomendaciones. Fuente: Elaboración Propia.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Cantidad resultados	26	6	7680	186	25	37	2

CAPÍTULO 5
CONCLUSIONES

5.1. Conclusiones generales

En el desarrollo de este trabajo se buscó dar solución al problema que se genera cuando se realizan consultas SPARQL y no se obtienen resultados o algún *feedback*. Esto se realizó, en primera instancia, a través del desarrollo de un algoritmo que permite obtener consultas alternativas a la realizada por el usuario, intentando que estas sean lo más parecidas posibles y que sí tengan resultados. Para diseñar este algoritmo, se analizarón algunas posibles causas y la manera de enfrentarlas, generando un algoritmo que reemplaza URIs de la consulta original por nuevas URIs que tienen relaciones fijas. Este algoritmo se implementó en el software RDFExplorer, añadiendo la característica de que cuando una consulta obtiene resultados vacíos, se calcula y entregan al usuario un conjunto de URIs alternativas para cada URI del último *triple* agregado.

De los resultados obtenidos en la implementación, se puede concluir que efectivamente el algoritmo permite obtener consultas alternativas que resulten de utilidad. Pero aun así, la cantidad de alternativas obtenidas puede resultar ser baja, esto debido a la poca cantidad de relaciones que se estan utilizado y que se usa principalmente la jerarquia de clases e inferencias relacionadas a estas, por lo que para incrementar el número de alternativas obtenidas, se deberá aumentar el número de relaciones utilizadas.

Como el algoritmo planteado no solamente incrementa el número de resultados, si no que también entrega el conjunto de consultas alternativas, se puede da la opción al usuario de elegir cuales de estas desea considerar, lo que le da un cierto grado de control en caso de que se tenga conocimiento del dominio de los datos, permitiendole filtrar entre las alternativas que crea le serán más útiles.

Por el lado de RDFExplorer, esto permite no solo obtener resultados, también le añade una nueva manera de navegar por los datos al sugerir recursos relacionados con los que seguir construyendo el grafo y explorando los datos RDF.

Todo lo anterior permite que si el usuario no tiene muchos conocimientos de los datos y su dominio, cuando realiza una consulta que no obtiene resultados, no solo puede optar a obtener resultados si no que además su conocimiento del dominio se ve levemente incrementado, lo que puede ser un *feedback* positivo para el usuario y le ayudará en futuras consultas que pueda llegar a realizar.

Un elemento que influye enormemente en el tiempo de respuesta en la implementación es la cantidad de consultas que se realizan al *endpoint* objetivo en un determinado periodo de tiempo, lo que es una limitación sobre la cual no se tiene control ya que se aplica por el lado

del servidor, por lo que se debió optimizar este apartado reduciendo la cantidad de consultas realizadas y, por ende, reducir los tiempos.

Aun así, la implementación realizada no resulta ser muy eficiente si se consideran los tiempos obtenidos, más aun considerando que desde un principio esta fue limitada en la cantidad de relaciones utilizadas y en los *triples* en que se aplica. Por lo que este es un punto importante que debe ser mejorado.

5.2. Trabajo futuro

Si bien se propuso e implemento una solución, existen varios puntos del trabajo que pueden ser mejorados y desarrollados. Estas mejoras se relacionan tanto al algoritmo como a la implementación en RDFExplorer.

Como se indica en 3.1, el algoritmo desarrollado solamente modifica una URI del *graph pattern* a la vez. Una posible mejora a realizar es agregar la caracteristica de realizar múltiples modificaciones a la vez. Con esto se generarían más consultas alternativas, permitiendo considerar una mayor cantidad de variaciones que incrementaría el número de resultados obtenidos, pero se debe tener bastante cuidado de que las consultas generadas no se alejen demasiado de la original.

En la sección 3.2 se indica que una de las causas del problema se relaciona a la mala comprensión del grafo RDF y que el uso de una ontología puede ser de utilidad para solucionar esto. Durante este trabajo se utilizó para este punto OWL 2 QL como una manera de realizar modificaciones que intenten mitigar esto, pero lo ideal sería utilizar una ontología propia del dominio de los datos que se están utilizando. Por lo que una notable mejora sería en la sección 3.3.3 considerar usar una ontologia propia de los datos para utilizarla como una guía de modificaciones, permitiendo reemplazar URIs por otras que mantendrían una relación propia del dominio, lo que permitiría generar y sugerir consultas útiles a los usuarios sin la necesidad de que deban comprender el dominio de los datos.

Respecto a la implementación en RDFExplorer, la mejora más importante y evidente se ve en la sección 4.7, en donde se analizaron los resultados. Como trabajo futuro, se debe mejorar los tiempos que se tarda en obtener los resultados, para lo que se debe mejorar la implementación realizada. Además, si se logra reducir considerablemente el tiempo, la implementación se podría aplicar a todos los *triples* de la consulta y no solo al último *triple* agregado.

También se podría analizar la opción de calcular los conjuntos de alternativas sin la necesidad de que la consulta no obtenga resultados, ya sea de manera automatica o como una opción activable por parte del usuario. Esto permitiría al usuario ver valores que se aproximen a los que está utilizando, sirviendo tanto como sugerencias y como una manera de explorar el grafo RDF. Lógicamente, antes de implementar esta opción se debe analizar los tiempos

extras que se agregarían debido a esto, por lo que primero se deben lograr reducir estos.

Finalmente, un detalle no menor que se podría agregar a la interfaz de RDFExplorer es la capacidad de señalar la relación existente entre el recurso que se tenga seleccionado y las alternativas de este, de forma tal que el usuario pueda tener una idea de la razón de porqué su consulta no tiene resultados. Por el momento, la única forma que tiene el usuario de conocer esta relación es manualmente.

ANEXOS

Anexo A. Reglas de RDFS *entailment*

número	Si contiene:	Agregar:
1	uuu aaa lll . Con lll literal	_:nnn rdf:type rdfs:Literal . donde _:nnn identifica un blank none
2	aaa rdfs:domain xxx . uuu aaa yyy .	uuu rdf:type xxx .
3	aaa rdfs:range xxx . uuu aaa vvv .	vvv rdf:type xxx .
4a	uuu aaa xxx .	uuu rdf:type rdfs:Resource .
4b	uuu aaa vvv .	vvv rdf:type rdfs:Resource .
5	uuu rdfs:subPropertyOf vvv . vvv rdfs:subPropertyOf xxx .	uuu rdfs:subPropertyOf xxx .
6	uuu rdf:type rdf:Property .	uuu rdfs:subPropertyOf uuu .
7	aaa rdfs:subPropertyOf bbb . uuu aaa yyy .	uuu bbb yyy .
8	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf rdfs:Resource .
9	uuu rdfs:subClassOf xxx . vvv rdf:type uuu .	vvv rdf:type xxx .
10	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf uuu .
11	uuu rdfs:subClassOf vvv . vvv rdfs:subClassOf xxx .	uuu rdfs:subClassOf xxx .
12	uuu rdf:type rdfs:ContainerMembershipProperty .	uuu rdfs:subPropertyOf rdfs:member .
13	uuu rdf:type rdfs:Datatype .	uuu rdfs:subClassOf rdfs:Literal .

REFERENCIAS BIBLIOGRÁFICAS

[Andreşel *et al.*, 2018] Andreşel, M., Ibáñez-García, Y., Ortiz, M., y Šimkus, M. (2018). Relaxing and restraining queries for obda.

[Angles *et al.*, 2017] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., y Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50:1–40.

[Berners-Lee *et al.*, 2001] Berners-Lee, T., Hendler, J., y Lassila, O. (2001). The semantic web. *Scientific american*, 284(5):34–43.

[Calvanese *et al.*, 2007] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., y Rosati, R. (2007). Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated Reasoning*, 39:385–429.

[Dellal *et al.*, 2019] Dellal, I., Jean, S., Hadjali, A., Chardin, B., y Baron, M. (2019). Query answering over uncertain rdf knowledge bases: explain and obviate unsuccessful query results. *Knowledge and Information Systems*, 61.

[Elbassuoni *et al.*, 2011] Elbassuoni, S., Ramanath, M., y Weikum, G. (2011). Query relaxation for entity-relationship search. volumen 6643, pp. 62–76.

[Fokou *et al.*, 2015] Fokou, G., Jean, S., Hadjali, A., y Baron, M. (2015). Cooperative techniques for sparql query relaxation in rdf databases.

[Hogan *et al.*, 2012] Hogan, A., Mellotte, M., Powell, G., y Stampouli, D. (2012). Towards fuzzy query-relaxation for rdf. volumen 7295, pp. 687–702.

[Huang *et al.*, 2008] Huang, H., Liu, C., y Zhou, X. (2008). Computing relaxed answers on rdf databases. volumen 5175, pp. 163–175.

[Hurtado *et al.*, 2006] Hurtado, C., Poulovassilis, A., y Wood, P. (2006). A relaxed approach to rdf querying. pp. 314–328.

[Pérez *et al.*, 2009] Pérez, J., Arenas, M., y Gutierrez, C. (2009). Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45.

[Saleem *et al.*, 2015] Saleem, M., Ali, M. I., Hogan, A., Mehmood, Q., y Ngonga Ngomo, A.-C. (2015). Lsq: The linked sparql queries dataset.

[Vargas *et al.*, 2019] Vargas, H., Buil-Aranda, C., Hogan, A., y López, C. (2019). *RDF Explorer: A Visual SPARQL Query Builder*, pp. 647–663.

[Vrandečić y Krötzsch, 2014] Vrandečić, D. y Krötzsch, M. (2014). Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57:78–85.

[w3c, 2008] w3c (2008). Sparql protocol for rdf. https://www.w3.org/TR/rdf-sparql-protocol/. Accessed: 2020-04-25.

[w3c, 2013] w3c (2013). W3c semantic web activity. https://www.w3.org/2001/sw/. Accessed: 2020-04-25.

[w3c, 2014a] w3c (2014a). Rdf 1.1 concepts and abstract syntax. https://www.w3.org/TR/rdf11-concepts/. Accessed: 2020-04-25.

[w3c, 2014b] w3c (2014b). Resource description framework (rdf). https://www.w3.org/RDF/. Accessed: 2020-04-25.