

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“EXTENSIÓN DEL FRAMEWORK DE BASES DE DATOS
APACHE TINKERPOP PARA LA EJECUCIÓN DE
CONSULTAS SPARQL SOBRE EL MODELO DE PROPERTY
GRAPHS”

EDUARDO VICENTE RAMÍREZ FARÍAS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Carlos Buil Aranda
Profesor Correferente: Cecilia Reyes Covarrubias

Agosto - 2018

DEDICATORIA

A mis padres, por su cariño infinito.

AGRADECIMIENTOS

Muchas gracias al profesor Carlos Buil Aranda quien me acompañó y orientó durante todo este tiempo, a mis padres y hermano por siempre tener una palabra de aliento para enfrentar el gran desafío, a mis amigos de la universidad quienes hicieron del proceso más llevadero al poder vivir la experiencia juntos y a mis amigos de años con quienes pude contar siempre para el esparcimiento y el descanso tan necesarios durante la travesía.

RESUMEN

Resumen— Esta memoria aborda el problema de la incompatibilidad entre el lenguaje de consultas de bases de datos de grafos SPARQL y el framework de bases de datos de grafos Tinkerpop, que organiza sus datos basándose en el modelo *property graph*. Este modelo difiere de RDF, que se basa en *edge-labelled graph* y se consulta con SPARQL. Se busca poder consultar un grafo Tinkerpop, utilizando una consulta SPARQL. Para esto se diseña un mapeo de *property graph* a RDF y una traducción desde la consulta a *traversals* Gremlin. Además, se simplificó la sintaxis de las consultas utilizando SPARQL*, una extensión de SPARQL, y la complejidad de ejecución tipificando las variables a priori. Se logró implementar un software que funciona con operaciones de selección, filtrado, opcional y unión, además del calce de patrones básicos. El mapeo, la traducción, las simplificaciones y diseño del software pueden ser usados en el futuro como base para una capa de compatibilidad más amplia.

Palabras Clave— base de datos de grafos; RDF; SPARQL; Tinkerpop; grafo de propiedades; traducción

ABSTRACT

Abstract— This thesis is about the incompatibility problem between the graph database query language SPARQL and the graph database framework Tinkerpop, that organizes its data following the property graph model. This model differs from RDF, that follows the edge-labelled graph and it is queried using SPARQL. To do this, a mapping from property graph to RDF and a translation from the query to Gremlin traversals are made. Also, the query syntax is simplified using SPARQL*, a SPARQL extension, and the execution complexity is reduced by typing the variables a priori. It was possible to implement a software that works with operations like selection, filtering, optional and union, besides the basic pattern matching. The mapping, translations, simplifications and software design can be used in the future as base to a more wide compatibility layer.

Keywords— graph database; RDF; SPARQL; Tinkerpop; property graph; translate.

GLOSARIO

ACID: Atomicity, Consistency, Isolation, Durability. Pilares fundamentales de las BD SQL.

BASE: Basically available, Soft-state, Eventual consistency. Pilares fundamentales de las BD no SQL.

DB: Data base (o en español BD). Colección organizada de datos, almacenados computacionalmente.

EBNF: Extended Backus-Naur Form. Metalenguaje para expresar gramáticas de libre contexto que extiende del original BNF, al agregar simbología para simplificar sintaxis.

GDB: Graph Data Base (o BDG en español). BD que organiza sus datos como arcos y nodos.

IDE: Integrated Development Environment. Plataforma para desarrollar software que en general consiste en un editor de código, herramientas de automatización de configuración y construcción, depurador, integración con sistemas de control de versiones e inspección de código.

IRI: Internationalized Resource Identifier. Extensión de la URI que acepta caracteres Unicode.

NEN: Calce Nodo-Arco-Nodo: Contracción utilizada en la memoria para referirse al calce de dos arcos unidos por algún arco usando la sintaxis SPARQL*.

NPV: Calce Nodo-Propiedad-Valor: Contracción utilizada en la memoria para referirse al calce del valor de la propiedad de un nodo usando la sintaxis SPARQL*.

OLAP: Online Analytical Processing. Complementario a OLTP, concepto bajo el cual se orientan los sistemas de información diseñados para el procesamiento de consultas complejas, inferencia y análisis de grandes cantidades de datos.

OLTP: Online Transaction Processing. Concepto bajo el cual se orientan los sistemas de información diseñados principalmente para ingreso y egreso de datos.

PGB: Patrón de Grafo Básico. Parte de una BD de grafos en donde algunos valores son variables. Se utiliza para calzar estructuras dentro de una BDG.

PGC: Patrón de Grafo Complejo. PGB con operaciones sobre el grafo como unión, diferencia u opcional.

RDF: Resource Description Framework. Estándar de codificación de recursos web recomendado por la W3C, que organiza los datos como grafos.

SPARQL: SPARQL Protocol and RDF Query. Lenguaje de consulta abstracto para grafos RDF recomendado por la W3C.

SQL: Structured Query Language. Popular lenguaje de consultas de base de datos relacionales, que permite la definición, manipulación y creación de éstos.

Turtle: Terse RDF Triple Language. Especificación de RDF de sintaxis sencilla similar a SPARQL.

URI: Uniform Resource Identifier. Identificador ASCII para referenciar un recurso web de for-

ma unívoca.

URL: Uniform Resource Locator. Especificación de la URI, usada generalmente para referenciar páginas web.

W3C: Word Wide Web Consortium. Consorcio internacional que crea recomendaciones y estándares para el crecimiento y la estabilidad de la World Wide Web.

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	IV
GLOSARIO	V
ÍNDICE DE FIGURAS	X
ÍNDICE DE TABLAS	X
ÍNDICE DE CÓDIGOS	XI
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 Contexto	2
1.1.1 Bases de datos de grafos	2
1.1.2 Tinkerpop	5
1.1.3 SPARQL	6
1.2 Problema	6
1.3 Objetivos	8
1.3.1 Objetivo General	8
1.3.2 Objetivos Específicos	8
CAPÍTULO 2: MARCO CONCEPTUAL	9
2.1 Modelos de Bases de Datos de Grafos	9
2.1.1 Edge-labelled Graph	9
2.1.2 Property Graph	9
2.1.3 Patrones de Grafos Básicos	10
2.1.4 Patrones de Grafos Complejos	12
2.2 RDF	13
2.2.1 Conceptos básico	13
2.2.2 Serialización	14
2.2.3 Ontologías	15
2.3 SPARQL	16
2.3.1 Sintaxis	17
2.3.2 Patrones complejos	18
2.4 Tecnologías	21
2.4.1 JENA	21
2.4.2 Tinkerpop	23

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	29
3.1 Formas de calce	29
3.1.1 Calce de etiquetas	30
3.1.2 Calce de propiedades	31
3.1.3 Calce con Gremlin	33
3.1.4 Incompatibilidad entre modelo y <code>match</code>	35
3.2 Alternativa 1: SPARQL-Gremlin + SPARQL*	37
3.2.1 Desventajas	39
3.3 Alternativa 2: SPARQL-Gremlin + SPARQL puro + variables en predicados	40
3.3.1 Desventajas	41
3.4 Alternativa 3: Nuevo mapeo de Tinkerpop a RDF + SPARQL*	41
3.4.1 SPARQL* para simplificar consultas	44
3.5 Elección	46
3.6 Algoritmo	46
3.7 Input	47
3.8 Traducción SPARQL* a SPARQL	47
3.8.1 Justificación	47
3.8.2 Gramática y traducción	47
3.8.3 Algoritmo de traducción	49
3.9 Comprobación de sintaxis SPARQL	50
3.10 Asignar tipos a cada variable	50
3.10.1 Idea general	50
3.10.2 Consideraciones respecto al modelo	51
3.10.3 Algoritmo	52
3.11 Recorrer árbol y generar <code>match</code>	52
3.11.1 Operaciones soportadas	56
3.11.2 Traducción de operaciones SPARQL a Gremlin	56
3.11.3 Traducción de PGB	59
3.11.4 Traducción de Optional	63
3.11.5 Traducción de FILTER	65
3.12 Proyección	66
3.13 Output	66
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	67
4.1 Ambiente de desarrollo	67
4.2 Diseño del software	67
4.2.1 SPARQL2GREMLIN	68
4.2.2 <code>com.datastax.sparql.io</code>	68
4.2.3 <code>com.datastax.sparql.star</code>	69
4.2.4 <code>com.datastax.sparql.gremlin</code>	70
4.2.5 <code>com.datastax.sparql.ops</code>	73
4.2.6 <code>com.datastax.sparql.constants</code>	75
4.3 Código	76
4.4 Tests	76

CAPÍTULO 5: CONCLUSIONES	90
5.0.1 Mapeo	90
5.0.2 Gramática y SPARQL*	90
5.0.3 Traducción	91
5.0.4 Objetivos	91
5.1 Trabajo futuro	92
REFERENCIAS BIBLIOGRÁFICAS	94

ÍNDICE DE FIGURAS

1	Ejemplo de base de datos <i>edge-labelled graph</i>	4
2	Ejemplo de base de datos <i>property-graph</i>	4
3	Ejemplo de PGB sobre un <i>edge-labelled graph</i>	11
4	Ejemplo de PGB sobre un <i>property graph</i>	11
5	Ejemplo de grafo <i>RDF</i>	14
6	<i>Property graph Modern</i> incluido en Tinkerpop.	31
7	Grafo Tinkerpop 'The Crew' incluido en Gremlin.	34
8	Grafo Tinkerpop con metapropiedades	42
9	Representación en grafo <i>RDF</i> de 8 según mapeo	43
10	Paquete <code>com.datastax.sparql.io</code>	68
11	Paquete <code>com.datastax.sparql.star</code>	69
12	Paquete <code>com.datastax.sparql.gremlin</code>	71
13	Paquete <code>com.datastax.sparql.ops</code>	74
14	Paquete <code>com.datastax.sparql.constants</code>	75

ÍNDICE DE TABLAS

1	Calces del PGB de la Figura 3.	11
2	Calce del PGB de la Figura 4.	12
3	Resultado de consulta SPARQL del Código 5.	17
4	Resultado de consulta SPARQL del Código 7.	18
5	Calces de etiquetas usando (<code>person ?a ?b</code>) en grafo de Figura 6	31
6	Posible significado de un triple en un <i>property graph</i>	32
7	Calces de propiedades usando (<code>person ?a ?b</code>) en grafo de Figura 6.	32

8	Ejemplo de combinaciones de tipos de dos variables.	36
9	Ejemplo de calces con variables replicadas.	36
10	Posibles combinaciones entre variables, tipos conocidos y algoritmo de traducción a usar.	60
11	Algoritmo de traducción getVVVFromSOTypes	60
12	Algoritmo de traducción getVVVFromPType	61
13	Algoritmo de traducción getVUVFromPType	61
14	Algoritmo de traducción getVUUFromPType	62
15	Algoritmo de traducción getVVVFromO	62
16	Algoritmo de traducción getVVUFromSType	62
17	Algoritmo de traducción getVVUFromPType	63
18	Concatenación para triples Optional	64
19	Resultado de consulta con OPTIONAL	65
20	Traducción de expresiones FILTER	66

ÍNDICE DE CÓDIGOS

1	Ejemplo de sintaxis Gremlin	5
2	Ejemplo de sintaxis SPARQL	6
3	Grafo <i>RDF</i> expresado en <i>Turtle</i>	15
4	Ejemplo de RDFS	15
5	Consulta SPARQL simple	17
6	Datos RDF a ser consultados	17
7	Consulta SPARQL simple #2	18
8	Consulta SPARQL usando proyección con SELECT	18
9	Consulta SPARQL usando unión con UNION	19

10	Consulta SPARQL usando diferencia con MINUS	20
11	Grafo RDF para consulta con OPTIONAL	20
12	Consulta SPARQL usando opcionales con OPTIONAL	20
13	Consulta SPARQL usando filtrado con FILTER	21
14	Uso de JENA para instanciar el grafo de la Figura 5	21
15	Uso de JENA para consultar grafo instanciado en Código 14	22
16	Uso de Gremlin para instanciar grafo de la Figura 2	24
17	Uso de Gremlin para consultar grafo instanciado en 16	25
18	Consulta Gremlin #1	33
19	Consulta Gremlin usando match	33
20	Consulta SPARQL* # 1	38
21	Consulta SPARQL* # 1 traducida a Gremlin	38
22	Consulta SPARQL* # 2	38
23	Consulta SPARQL* # 2 traducida a Gremlin	39
24	Consulta SPARQL con variable en predicado	40
25	Consulta SPARQL con variable en el predicado # 2	41
26	Consulta SPARQL siguiendo nuevo modelo	44
27	Consulta SPARQL con OPTIONAL traducida a Gremlin	64
28	Consulta SPARQL con OPTIONAL	65

INTRODUCCIÓN

Las bases de datos son indispensables hoy en día. La cantidad de datos que los sistemas informáticos manejan es enorme y es necesario que la tecnología avance en función de facilitar su almacenamiento, su procesamiento, la forma de consultarlos, analizarlos y relacionarlos. Es por esto que existen numerosos sistemas de bases de datos, basados en modelos que prestan ventajas para cada necesidad. Desde los originales esquemas relacionales rígidos, rápidos para dominios sencillos y de uso masivo, hasta los modelos sin esquema, flexibles, escalables y distribuidos, muy en desarrollo actualmente.

En específico, existen las bases de datos de grafos, que organizan sus datos como un conjunto de nodos y arcos. Algunas implementaciones, como RDF, modelan el grafo como un conjunto de arcos etiquetados que relacionan datos en los nodos, y otros como Tinkerpop, extienden este concepto agregando propiedades en cada arco y nodo. Ambas tecnologías tienen formas de ser consultadas, la primera a través del lenguaje de consulta SPARQL y la segundo utilizando Gremlin. A pesar de que ambos trabajan sobre el mismo dominio, no son compatibles, es decir, no se puede utilizar SPARQL para consultar un grafo Tinkerpop ni Gremlin para consultar un grafo RDF.

El problema a tratar es justamente éste. A través de la presente memoria, se buscará una forma de poder consultar un grafo Tinkerpop utilizando SPARQL. Se comenzará definiendo el problema, centrándose en las diferencias entre los modelos subyacentes a ambas tecnologías, los cuales son *property graph* en Tinkerpop y *edge-labelled graph* en RDF, y las que hay entre ambos lenguajes de consulta. Luego se expondrá el marco teórico, desarrollando ambos modelos, las formas en que organizan los datos, cómo se consultan y cómo funcionan, al igual que las tecnologías mencionadas, sus características principales y ejemplos de funcionamiento.

Posteriormente se expondrán alternativas de solución, centrándose en la de mapear el modelo *property graph* a RDF, para así poder consultarlo utilizando SPARQL y posteriormente traducir la consulta a Gremlin. Se buscará disminuir la complejidad sintáctica de las consultas a través del uso de una versión extendida de SPARQL llamada SPARQL*, como también la complejidad de ejecución, diseñando un mapeo sin excesiva generalidad.

Se implementará el mapeo y algunas operaciones típicas sobre bases de datos de grafos. Se busca probar que el mapeo funcione, que las consultas se ejecuten y se obtenga lo deseado desde grafos Tinkerpop. Se expondrán los resultados obtenidos y propuestas para trabajos futuros que pretendan extender la solución.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

En esta sección se hablará en forma general del problema a tratar en la presente memoria. Se esbozarán los temas base necesarios para entender el problema y se expondrán los objetivos del trabajo.

1.1. Contexto

Antes de definir el problema a tratar, es necesario contextualizarlo y estudiar los conceptos que lo sustentan.

1.1.1. Bases de datos de grafos

En el área de las bases de datos (DB) existen dos grandes grupos: Las DB **SQL** y las **NoSQL**. Las primeras están basadas en tablas, llamadas relaciones, que consisten en datos organizados como filas y columnas. Este tipo de DB fueron las primeras desarrolladas y ocupaban por completo el mercado del almacenamiento y estructuración de datos hasta el año 2000, en donde las grandes empresas de tecnología comenzaron a invertir en investigación de nuevos enfoques, debido a las limitantes del modelo imperante [20]. Estas limitantes abarcan restricciones en escalabilidad, la pérdida de eficiencia en consultas de grandes cantidades de datos y la dificultad de administración y almacenamiento en DB de gran tamaño.

Las DB NoSQL surgen como solución a las limitantes del modelo SQL. Tienen como principio fundamental no ser relacionales. Muchas implementaciones están pensadas para que los datos existan a través de múltiples soportes hardware y así escalar horizontalmente de manera automática. Se caracterizan por no presentar un esquema rígido predefinido, es decir, la estructura de los datos que se insertan puede cambiar en cualquier momento y sin necesidad de intervención ni interrupciones.

Uno de los axiomas fundamentales de las DB SQL es ACID:

- **Atomicity:** Atomicidad, las actualizaciones de los datos son tareas completas que deben terminarse o abortarse completamente.
- **Consistency:** Consistencia, la base de datos debe estar siempre en un estado consistente una vez terminen las transacciones.
- **Isolation:** Aislamiento, no debe haber conflicto entre distintas transacciones.

- **Durability:** Durabilidad, los datos y las transacciones deben recuperarse aunque haya fallas en el soporte hardware.

Este axioma es reemplazado en las DB NoSQL por BASE, cuyos principios tratan de asegurar la disponibilidad del sistema en todo momento[21].

- **Basically available:** El sistema debe estar disponible todo el tiempo.
- **Soft-state:** El estado de un sistema puede cambiar incluso si no hay modificaciones por parte de clientes consultando, es decir, el sistema puede estar actualizándose mientras se consulta.
- **Eventual consistency:** Las lecturas pueden no reflejar el último estado de la base de datos mientras una transacción se distribuye entre los nodos del cluster, pero eventualmente lo harán (luego de la actualización completa).

Existen varios tipos de DB NoSQL[21][2]. Las que se usarán en el trabajo de título son las **Bases de Datos de Grafos** (GDB). En las GDB las entidades del dominio son representadas como un conjunto de nodos (o vértices) y sus relaciones entre sí a través de arcos (o aristas). Estos modelos permiten ejecutar consultas a la DB para reconocer diversos patrones estructurales o generar expresiones de navegación que entreguen conjuntos de datos relacionados de cierta forma [3].

Los dos tipos de modelos de datos de GDB que conciernen al presente trabajo de título son los **edge-labelled graphs** (grafos de arcos etiquetados) y los **property graphs** (grafos de propiedades). Los **edge-labelled graphs** cuentan con etiquetas en cada arco dirigido que expresa la relación entre el nodo de salida y el de entrada, dentro del dominio.

En la Figura 1 se muestra un ejemplo de una GDB **edge-labelled** de películas. La relación entre un actor y un personaje es `plays` y entre un personaje y una película es `movie`. Si bien la estructura de estos grafos es simple, es suficiente para describir relaciones complejas. Podría por ejemplo saberse la cantidad de películas en las que dos actores han actuado al mismo tiempo. Esta forma de grafos es la base para el estándar *Resource Description Framework* o **RDF** usado para codificar contenido web de forma que pueda ser leído por software. Se ahondará en esto en el Capítulo 3.

En el caso de los **property graphs**, se extiende la idea anterior agregando identificadores únicos a cada nodo y arco, permitiendo etiquetas en los nodos, y agregando varias propiedades a cada nodo y arco.

En la Figura 2 se ve que los datos de los personajes ahora están en las relaciones, con propiedades como `name` y `rol`. Los nodos tienen etiquetas que indican tipo de dato que representan, y tienen propiedades para más detalle.

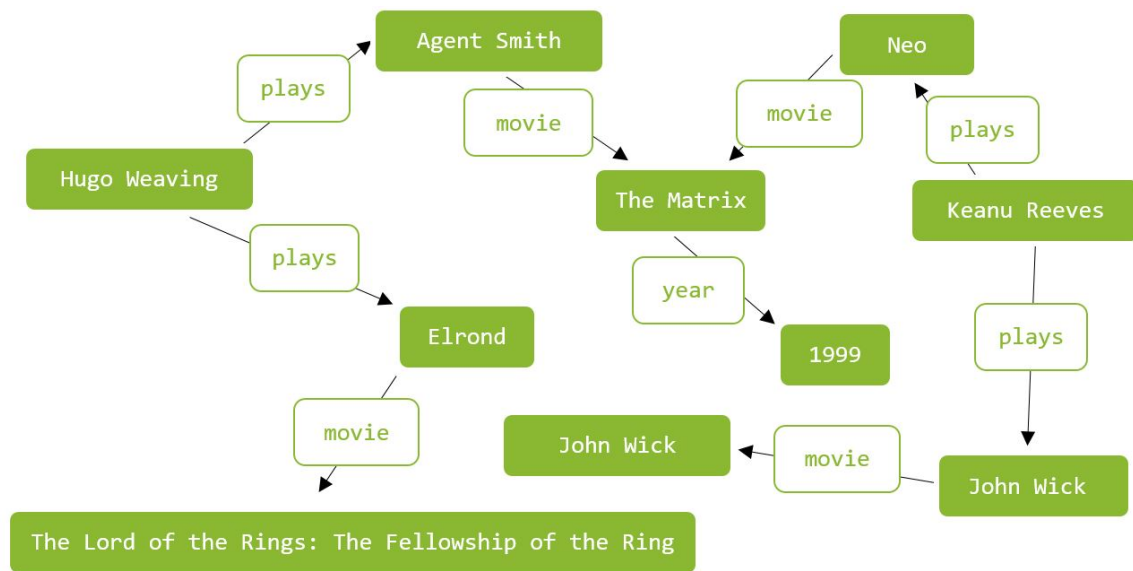


Figura 1: Ejemplo de base de datos *edge-labelled graph*.
Fuente: Elaboración propia.

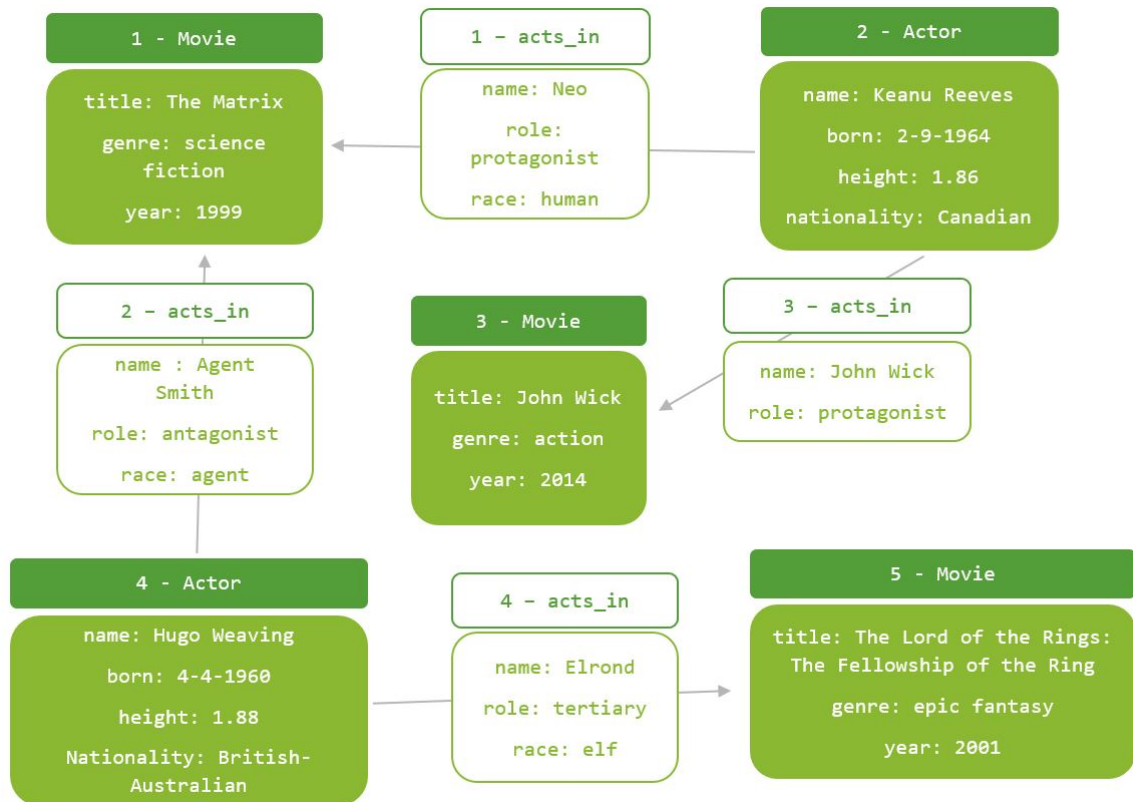


Figura 2: Ejemplo de base de datos *property-graph*.
Fuente: Elaboración propia.

1.1.2. Tinkerpop

Hoy en día existen muchas tecnologías para el uso de GDB, cada una usando su propio esquema, similar o derivado de los dos expuestos anteriormente, dando prioridad a distintos aspectos como escalabilidad, rapidez, disponibilidad, etc. Algunos ejemplos son **ArangoDB**[14], **Sparksee**[22], **Neo4J**[19] y **Stardog**[23]. Las últimas dos tecnologías están basadas en **Apache Tinkerpop**. Este es un framework open-source de computación de grafos que provee herramientas para el almacenamiento y análisis de datos en forma de grafo y se basa en *property graphs*. Actualmente existen numerosas implementaciones sobre este sistema que proveen distintos enfoques y servicios, como alta disponibilidad, servicios de cloud computing, OLTP y OLAP distribuidos, análisis de *Big Data* por lotes en tiempo real, etc. [4].

El objetivo principal de TinkerPop es facilitar el desarrollo de aplicaciones basadas en grafos entregando una API y herramientas para cubrir las necesidades generales al trabajar con GDBs. Así, Tinkerpop provee una capa de abstracción sobre distintas bases de datos y procesadores de grafos, evitando la dependencia generada por los desarrolladores de tecnologías específicas. Todo esto permite que los desarrolladores prueben diferentes implementaciones con el mismo código para decidir el mejor entorno, pueden extender una implementación en base a sus necesidades específicas y genera un ambiente de seguridad al elegir una u otra tecnología ya que cualquier cambio futuro sería rápido y sin mayor impacto al software desarrollado.

El lenguaje utilizado para analizar y almacenar datos en TinkerPop es **Gremlin**. Es un lenguaje de recorrido, es decir, sirve para recorrer los grafos, calzar patrones, navegar por sus elementos y puede ser escrito de manera procedural o declarativa. Si bien la implementación principal¹ es en Java, existen también en Python, Ruby, Javascript, Go, C#, y otros. Puede ser usado directamente a través de la Consola Gremlin, o en el código del software desarrollado a través de la API [7].

```
1 g.V().has("name", "Neo").
2   out("knows").
3   out("knows").
4   values("name")
```

Código 1: Ejemplo de sintaxis Gremlin

En el Código 1 se toman todos los vértices del grafo *g* cuyo *name* sea *Neo*. De estos, se avanza a todos los conocidos (*knows*) de esos vértices, y luego a los conocidos de estos. De ese conjunto, se retornan los nombres.

¹Dado que la especificación del lenguaje es abstracta, esta puede ser concretada utilizando lenguajes de programación ya existentes que soporten composición de funciones.

1.1.3. SPARQL

Por otro lado está **SPARQL**, un lenguaje de consulta de bases de datos RDF basado en *edge-labelled graph*. Estas bases de datos se expresan como conjuntos de tripletas o **triples**, en donde cada una representa que el elemento inicial se relaciona con el tercer elemento a través del segundo elemento, llamados sujeto, objeto y predicado respectivamente[10]. En este caso los sujetos y objetos juegan el rol de vértices, y los predicados el de arcos. Este conjunto de triples genera un grafo, que puede ser consultado con este lenguaje, similar a como Gremlin consulta un grafo Tinkerpop.

En el Código 2 se muestra una consulta, que es básicamente un grafo al que le faltan valores concretos, que son reemplazados por variables. Se consulta por la variable `?name` y debe cumplirse dentro del grafo consultado que exista un sujeto (determinado por una variable cualquiera) que se relacione con 24 a través de `age` y que ese mismo sujeto se relacione con algún valor a través de `name`. Ese valor (o valores en caso de muchos casos que cumplan) es el que toma la variable consultada. Dicho de una forma más relacionada con el modelo, la consulta requiere que un vértice cualquiera se una a otro de valor 24 a través de un arco etiquetado como `age` y que además se una a otro cualquiera a través de un arco etiquetado como `name`.

```
1 SELECT ?name WHERE {  
2   ?someone age 24 .  
3   ?someone name ?name .  
4 }
```

Código 2: Ejemplo de sintaxis SPARQL

Ya en conocimiento de estas dos implementaciones de ambos modelos, salta a la vista la gran similitud que existe entre ellos. Sin embargo, no son compatibles y esto impide que el acceso a los datos guardados en ambos modelos este unificado, a pesar de ser modelos muy similares. No se puede consultar un grafo Tinkerpop con SPARQL ni se puede consultar un grafo RDF con Gremlin. Es este el problema que se desea abordar en el presente trabajo de título.

1.2. Problema

Cuando un usuario quiere interactuar con una GDB en un sistema basado en TinkerPop, debe hacerlo a través de Gremlin en alguna de sus variedades (implementaciones). Esto puede ser a través de una implementación de Gremlin en el lenguaje de programación que quiera usar o de algún compilador que transforme otro lenguaje de consulta a Gremlin [13]. En ambos casos está limitado por las tecnologías ya implementadas por otros desarrolladores.

Por otro lado, si se requiere consultar un grafo RDF debe hacerse a través de SPARQL o de

alguna API en algún lenguaje de programación que entregue las funciones necesarias para acceder a éste, por ejemplo **Apache JENA**[8].

Gremlin y SPARQL difieren principalmente en que el primero utiliza una secuencia de llamadas a funciones para generar navegación y calces sobre los elementos del grafo, mientras que el segundo define un lenguaje de consulta, similar a como lo hace SQL, el cual es parseado y ejecutado sobre el grafo. Esta diferencia paradigmática y estructural hace que TinkerPop sea incompatible con SPARQL. Sin embargo, el dominio sobre el cual trabajan es el mismo (bases de datos de grafos). Esto plantea la posibilidad de generar compatibilidad. El problema general que será estudiado en el trabajo de título es el de la incompatibilidad del lenguaje de consultas SPARQL sobre el framework Apache TinkerPop basado en *property graphs*. La idea será desarrollar una capa de software que tome como input un string que represente una consulta SPARQL y lo parsée de tal forma que pueda ser traducido a una consulta Gremlin, para que ésta acceda a los datos guardados en un grafo TinkerPop. Luego el resultado entregado por Gremlin pasará a esta capa, en donde se estructurará en algún formato usado por SPARQL para retornar.

Es necesario estudiar la forma en que se logrará la compatibilidad, dado que existen dos posibilidades: Traducir SPARQL a Gremlin, o mapear los modelos subyacentes. Ambos enfoques tienen ventajas y desventajas a considerar, relacionadas con la dificultad de implementación, lo amplio de este trabajo y la generalidad que entregarían.

Otro aspecto a considerar es la limitante de la sintaxis de SPARQL respecto al acceso a un *property graph*. SPARQL puede encargarse de 3 valores por *triple*, 2 en nodos y uno en arco. Nativamente no está preparado para analizar datos *dentro* de cada nodo o arco. Estos valores son las propiedades, no presentes en el modelo *edge-labeled*. La solución deberá considerar algún método para expresar a qué se refiere cada *triple*.

Existe un proyecto llamado SPARQL-Gremlin[11] que compila consultas SPARQL en recorridos Gremlin. Está basado en Apache Jena, que provee *parsing* del árbol sintáctico de las consultas SPARQL. Si bien Jena traduce ampliamente las operaciones disponibles en SPARQL, incluso en su versión más reciente 1.1, SPARQL-Gremlin solo implementa una pequeña parte de esas funciones, por lo que la compatibilidad entre ambos sistemas no alcanza un nivel aceptable. Tampoco maneja adecuadamente las propiedades de los arcos. Sin embargo, puede ser utilizado como base para el proyecto.

Se espera lograr un acercamiento a lo que podría convertirse, en el futuro con mayor profundización, en un *plugin* completo de compatibilidad entre SPARQL y sistemas TinkerPop, para que la integración entre sistemas se logre simplemente con la instalación de este.

1.3. Objetivos

1.3.1. Objetivo General

Extender el framework de bases de datos de grafo Apache TinkerPop para que acepte y ejecute óptimamente consultas realizadas con el lenguaje SPARQL, mediante un mapeo entre el modelo RDF y Property Graph.

1.3.2. Objetivos Específicos

1. Analizar los modelos RDF y Property Graph.
2. Diseñar un sistema de traducción y mapeo entre ambos modelos.
3. Implementar los componentes de software necesarios para la ejecución de las consultas, utilizando el mapeo diseñado.
4. Validar el funcionamiento del software, sometiéndolo a pruebas y comparaciones con resultados generados por el framework original.

CAPÍTULO 2

MARCO CONCEPTUAL

Se comenzará por revisar los dos modelos de bases de datos usados en el presente trabajo, las formas en que se pueden consultar y su relación con el modelo relacional. Luego se expondrán las principales características de RDF y SPARQL y su relación con uno de los modelos estudiados. Luego se pasará a estudiar los principales componentes de las tecnologías a usar para la solución del problema, estas son Gremlin y JENA, y establecer la relación del primero con el otro modelo estudiado.

2.1. Modelos de Bases de Datos de Grafos

Es necesario sentar las bases formales de los modelos antes de compararlos y analizarlos en profundidad. Se expresarán las estructuras formales de los modelos a través del uso de teoría de conjuntos y se hará un paralelo entre las consultas en estos modelos con las consultas al modelo relacional.

2.1.1. Edge-labelled Graph

Es el modelo base para la definición de BDGs. La idea es sencilla: cada nodo representa algún objeto del dominio en donde trabaja la BD, y cada arco entre estos representa su relación. Como se puede ver en la Figura 1, y como indica el nombre, cada arco está etiquetado para indicar qué relación existe entre ambos nodos. Formalmente, un *edge-labelled graph* está formado por:

1. Un conjunto de etiquetas L
2. Un conjunto de nodos V
3. Un conjunto de arcos $E \subseteq V \times L \times V$

2.1.2. Property Graph

Este modelo nace principalmente de la necesidad de agregar fácilmente tipos nuevos de meta-información a los arcos. Por ejemplo, indicar la fuente de donde proviene la información de que un actor actuó en una película. En el *edge-labelled graph* se tendría que hacer agregando un nuevo nodo con la fuente de esta información (por ejemplo, IMDb) y conectar

el actor con este nodo a través del arco `plays_source` o algo similar, lo cual es poco conveniente en búsquedas ya que genera una base de datos con muchos arcos. Otra forma de lograrlo es aplicando reificación, que se basa en representar los arcos como nodos [17]

Property graph extiende *edge-labelled graph* para solucionar este problema. Cada arco tiene un identificador único que sirve para acceder a datos propios del arco, llamados atributos o propiedades. Los nodos también tienen identificador y propiedades. Además ambas estructuras tienen asociada una etiqueta. Formalmente, un *property-graph* es una tupla $(V, E, \rho, \lambda, \sigma)$ donde:

1. V es un conjunto finito de nodos
2. E es un conjunto finito de arcos
3. $\rho : E \rightarrow (V \times V)$ es una función que asocia un arco y dos nodos, siendo el primer nodo el de salida y el segundo el de llegada.
4. $\lambda : (V \cup E) \rightarrow L$ es una función que asocia cada nodo o arco con una etiqueta del conjunto finito L .
5. $\sigma : (V \cup E) \times P \rightarrow V$ es una función que asocia cada propiedad del conjunto P , de cada nodo o arco, con un valor del conjunto V .

Cuando uno de estos modelos es consultado, puede hacerse por medio de la búsqueda de un patrón o de navegación. Hay dos formas de búsqueda de patrones, la primera es **patrones de grafo básicos** (*basic graph patterns*) que significa calzar en un grafo una estructura determinada. La otra es **patrones de grafo complejos** (*complex graph patterns*) que es lo mismo que la anterior pero incluyendo características relacionales en la consulta como proyecciones, uniones, opcionales y diferencias, generando resultados más específicos y filtrados. Mucha de la funcionalidad de SPARQL y Gremlin está enfocada en calzar patrones.

2.1.3. Patrones de Grafos Básicos

En palabras simples, un PGB (Patrón de Grafo Básico) es una base de datos de grafos (o una parte de esta) en donde no todos los nodos, arcos y propiedades están definidos, es decir, no toman valores sino que son variables. Cuando las variables del patrón pueden ser mapeadas a valores reales en una base de datos y los valores definidos también aparecen, todo esto manteniendo la estructura, se dice que el patrón calza. Cada forma de mapear las variables es un calce.

Por ejemplo, en el grafo de la Figura 1 se pueden buscar los actores que hayan interpretado a dos personajes con un patrón como el de la Figura 3. Las variables se denotan con x_1, x_2 y x_3 , donde la primera indica un nodo de salida, que en este caso tomará el valor de un actor,

y las otras dos son los nodos de llegada, que serán personajes. El resultado de este calce se puede apreciar en la Tabla 1.

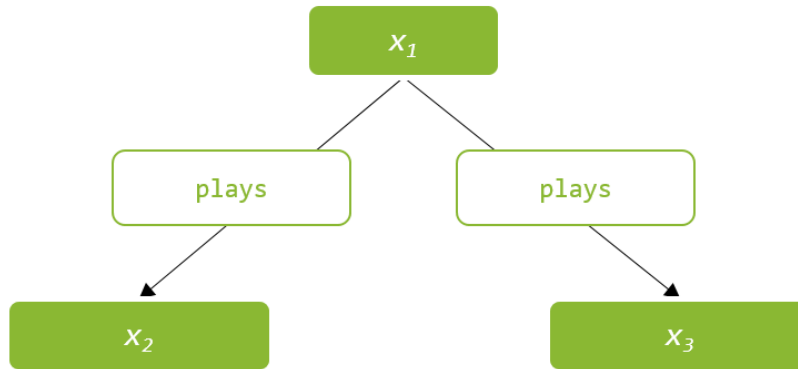


Figura 3: Ejemplo de PGB sobre un *edge-labelled graph*.
Fuente: Elaboración propia.

Tabla 1: Calces del PGB de la Figura 3.
Fuente: Elaboración propia.

x_1	x_2	x_3
Hugo Weaving	Elrond	Agent Smith
Keanu Reeves	Neo	John Wick

En el caso del *property graph* de la Figura 2, se pueden buscar los nombres y años de las películas en donde Keanu Reeves ha actuado como protagonista con un PGB como el de la Figura 4, obteniendo como resultado la Tabla 2. Notar que un patrón de *property-graph* puede incluir variables en valores, propiedades, identificadores y etiquetas.

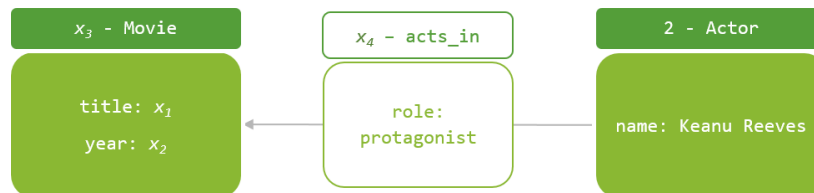


Figura 4: Ejemplo de PGB sobre un *property graph*.
Fuente: Elaboración propia.

Formalmente un calce en un *property graph* se define como:

Definición: Sea G un *property graph*, Q un PGB, C el conjunto de todas las constantes de G (ya sea identificadores, propiedades, valores o etiquetas) y V el conjunto de todas las

Tabla 2: Calce del PGB de la Figura 4.

Fuente: Elaboración propia.

x_1	x_2	x_3	x_4
The Matrix	1999	1	1
John Wick	2014	3	3

variables de Q . Un calce h de Q a G es un mapeo de $C \cup V$ a C tal que $h(c) = c \forall c \in C$ y $h(v) \in C \forall v \in V$.

El calce h no es más que un homomorfismo de Q a G , debido a que $h(Q)$ es un subgrafo de G . Si se restringe a que los valores a los que mapea cada variable sean distintos² en un calce, entonces lo que se tiene es un isomorfismo. La restricción de isomorfismo puede tomar varias semánticas. Por ejemplo, se pueden restringir todas las variables a ser distintas, solo los identificadores de los nodos o solo los identificadores de los arcos.

2.1.4. Patrones de Grafos Complejos

Los Patrones de Grafos Complejos (PGC) son simplemente PGB a los que se les agregan operaciones como las usadas en SQL:

- *Projection*: Seleccionar solo una parte de las variables del patrón para ser parte del output de la evaluación.
- *Join*: Unir los calces de dos patrones usando las variables que tienen en común cuando estas adquieren el mismo valor, es decir, los compatibles.
- *Union*: El resultado la unión de los calces de dos patrones es el conjunto de todos ellos.
- *Difference*: El resultado de la diferencia de los calces del patrón A con los del patrón B son todos los calces que aparezcan en A pero no en B .
- *Optional*: Igual a *Join* pero mantiene los calces incompatibles como parte del resultado.
- *Filter*: Filtra los calces obtenidos mediante la satisfacción de restricciones entre las variables con operadores lógicos, expresiones regulares y comparaciones entre valores.

Cada lenguaje de consulta a BDGs da soporte parcial o total a este tipo de operaciones. En secciones posteriores se profundizará en cómo lo hacen.

²En otras palabras, que h sea inyectivo

2.2. RDF

Una de las aplicaciones más directas del modelo *edge-labelled graph* es el *Resource Description Framework* o RDF. Este es un framework propuesto por la W3C para expresar hechos o relaciones ciertas entre objetos e información de Web. Se creó con el objetivo de satisfacer los siguientes postulados:

- Permitir a las aplicaciones intercambiar información entendible por computadores para facilitar la interoperación entre estos.
- Ser escalable para que su funcionamiento sea independiente de la cantidad de datos.
- Ser suficientemente flexible como para codificar cualquier información entendible por personas.
- Ser simple para poderse escribir, leer y consultar fácilmente.

Este modelo es uno de los varios componentes de la Web Semántica, debido a que es usado para guardar metadatos. Estos metadatos pueden ser usados para describir los contenidos de un recurso web, características especiales y su relación con otros, de manera explícita y disponible para los desarrolladores.

RDF sirve para establecer relaciones entre recursos, que pueden referirse a cualquier objeto ya sea un documento, valor, concepto, etc. Dos recursos están relacionados a través de otro recurso, de la misma forma en que se representan los *edge-labelled graph*.

2.2.1. Conceptos básico

Un recurso puede estar referenciado de tres formas:

- **IRI o URI:** Un *Internationalized Resource Identifier* o *Uniform Resource Identifier* es un string Unicode absoluto que sigue el formato especificado en [18]. Son identificadores que sirven para señalar de forma unívoca recursos en la web. Las IRIs son la generalización de las URIs y las URLs, que sirve para identificar páginas web.
- **Literal:** Strings Unicode cualquiera para referenciar cadenas de caracteres, números o fechas.
- **Nodo vacío:** Recurso sin información explícita, que se puede utilizar como agrupador de otros recursos.

Por ejemplo la URI:

`http://bestmovieever.com/movie/345`

podría estar asignada a la película The Matrix. Ésta se puede contraer definiendo prefijos. Por ejemplo si `mov` mapea a `http://bestmovieever.com/movie/` entonces queda:

`mov:345`

Usando esta sintaxis para identificar recursos, RDF define *triples* con el esquema:

Sujeto (URI/Nodo vacío) + Predicado (URI) + Objeto (URI/Literal/Nodo vacío)

que define relaciones entre un sujeto y un objeto a través de un predicado. Un conjunto de estas estructuras definen un grafo *edge-labelled* donde los nodos son recursos u objetos y los arcos son la relación, como muestra la Figura 5. Notar que el prefijo `char` puede servir, por ejemplo, para mapear a `http://bestmovieever.com/character/`. RDF usa `a` como mapeo a `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` que indica el tipo del sujeto.

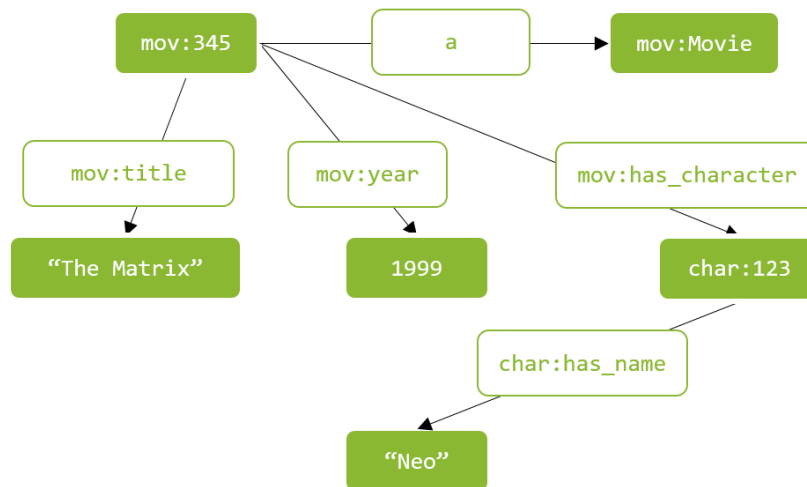


Figura 5: Ejemplo de grafo *RDF*.
Fuente: Elaboración propia.

2.2.2. Serialización

RDF es un modelo de datos, no un formato. Los datos (el grafo) a representar deben ser expresados en un formato web adecuado. Existen varios formatos³ de serialización como

³También llamados sintaxis concretas

Turtle, N-Triples, N-Quads, JSON-LD, N3 y RDF/XML. *Terse RDF Triple Language* o Turtle es el lenguaje recomendado por la W3C para expresar RDF. Tiene una sintaxis muy simple y es la base para la sintaxis de SPARQL. El Código 3 muestra la definición de la Figura 5 en Turtle.

```
1 @prefix mov: <http://bestmovieever.com/movie/>.
2 @prefix char: <http://bestmovieever.com/character/>.
3
4 mov:345 a mov:Movie;
5   mov:title "The Matrix"@eng ;
6   mov:year 1999 ;
7   mov:has_character char:123 ;
8 char:123 char:has_name "Neo" ;
```

Código 3: Grafo RDF expresado en *Turtle*

En la línea 1 y 2 se definen los dos prefijos, que serán usado en las declaraciones subsiguientes. En la 4 se define el triple que asigna un tipo al sujeto `mov:345`. Las líneas 5, 6 y 7 son triples anidados, útil cuando el sujeto es compartido entre varios triples, en este caso `mov:345`. En la línea 5 se usa `@eng` para especificar el lenguaje usado en el objeto. Finalmente en la línea 9 se muestra un triple sin anidación.

2.2.3. Ontologías

Similar a cómo se trabaja en Programación Orientada a Objetos, RDF expone la necesidad de expresar relaciones hereditarias entre entidades así como relaciones de pertenencia a una categoría. Estos dos conceptos son la **jerarquía** e **instanciación** respectivamente, expresadas por medio de **clases**. Siguiendo con las películas, `Movie` es una clase y la película específica `mov:345`, cuyo nombre es "The Matrix", es una instancia de esta (relación indicada con el término `a` en el Código 3). Por otro lado, si existiera por ejemplo la clase `Short` (cortometraje), esta sería una subclase de `Movie`.

En este contexto, un **vocabulario** se define como todos los términos usados en un dominio, como `Movie`, `Character`, `year`, `has_name`, `has_character` y una **ontología** como la estructura que pueden tomar estos términos, es decir, especificaciones de herencia (`Short` hereda de `Movie`), de que términos van en cada clase (`Movie` tiene `has_character`, `year`), etc. Para definir la ontología se puede usar RDF Schema o RDFS⁴, que utiliza sintaxis XML[15].

```
1 <?xml version = 1.0 encoding = UTF-8?>
2 <rdf:RDF
3   xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4   xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#" />
5
6   <rdfs:Class rdf:ID="Character" />
```

⁴OWL también es usado para definir ontologías

```
7   <rdfs:Class rdf:ID="Movie"/>
8   <rdfs:Class rdf:ID="Short">
9     <rdfs:subClassOf rdf:resource = "Movie"/>
10  </rdfs:Class>
11  <rdf:Property rdf:id="Name">
12    <rdf:domain rdf:resource="#Movie"/>
13    <rdf:range rdf:resource=
14      "http://www.w3.org/2000/01/rdf-schema#Literal"/>
15  </rdf:Property>
16  <rdf:Property rdf:id="Year">
17    <rdf:domain rdf:resource="#Movie"/>
18    <rdf:range rdf:resource=
19      "http://www.w3.org/2000/01/rdf-schema#Literal"/>
20  </rdf:Property>
21  <rdf:Property rdf:id="hasCharacter">
22    <rdf:domain rdf:resource="#Movie"/>
23    <rdf:range rdf:resource="#Character"/>
24  </rdf:Property>
25 </rdf:RDF>
```

Código 4: Ejemplo de RDFS

En el Código 4 se muestra la forma de definir parte de la ontología de películas. De la línea 6 a la 10 se definen las clases, donde *Short* hereda de *Movie*. Luego vienen definiciones de propiedades. Cada definición indica el identificador (*id*), la clase del sujeto desde quien enlazan (*domain*) y la clase del objeto al que pueden enlazar (*range*).

2.3. SPARQL

Ya estudiado cómo representar metadatos y ontologías, es necesario un lenguaje que los manipule. Actualmente no existe un lenguaje de consulta de información RDF estandarizado, sino que muchos lenguajes de diversos paradigmas y capacidades, que en general no comparten sintaxis ni semántica [24]. La W3C recomienda a los desarrolladores que doten a sus lenguajes con las funcionalidades necesarias para cubrir los casos de usos expuestos en [24], que incluyen encontrar direcciones email, encontrar objetos multimedia desconocidos, monitorizar noticias, explorar vecindarios, encontrar documentos de entrada y salida para pruebas de casos de uso, descubrir recursos de aprendizaje, y un largo etc. Uno de estos lenguajes es SPARQL que es un lenguaje declarativo utilizado para consultar grafos RDF y cuya especificación la define la W3C. Este es uno de los lenguajes que serán estudiados en la presente memoria.

2.3.1. Sintaxis

Las consultas SPARQL son simplemente patrones de grafos básicos o complejos, como se revisó en las secciones 2.1.3 y 2.1.4. Estos deben ser calzados en un grafo RDF. Por ejemplo si la consulta:

```
1 SELECT ?title
2 WHERE
3 {
4   <http://bestmovieever.com/movie/345>
5     <http://bestmovieever.com/movie/title> ?title .
6 }
```

Código 5: Consulta SPARQL simple

se ejecuta sobre el grafo:

```
1 @prefix mov: <http://bestmovieever.com/movie/>.
2 @prefix char: <http://bestmovieever.com/character/>.
3 @prefix act: <http://bestmovieever.com/actor/>.
4
5 mov:112 a mov:Movie;
6   mov:title "The Lord of The Rings - The Fellowship of the Ring" ;
7   mov:year 2001 ;
8   mov:has_character char:3334 .
9 mov:345 a mov:Movie;
10  mov:title "The Matrix";
11  mov:year 1999 ;
12  mov:has_character char:123.
13 char:123 a mov:Character;
14   char:name "Neo" .
15 act:45 a act:Actor ;
16   act:name "Hugo Weaving" ;
17   act:plays char:123 ;
18   act:plays char:3334 .
19 char:3334 a mov:Character;
20   char:name "Elrond".
```

Código 6: Datos RDF a ser consultados

el resultado sería:

Tabla 3: Resultado de consulta SPARQL del Código 5.
Fuente: Elaboración Propia.

title
"The Matrix"

La consulta es similar a SQL, pero con sintaxis Turtle. Usa la variable ?title como campo para calzar los *triples* del grafo. Se puede traducir como "selecciona todos los ?title, en donde ?title sea el objeto de los *triples* que tengan sujeto

<http://bestmovieasever.com/movie/345> y predicado

<http://bestmovieasever.com/movie/title>". Otro ejemplo de consulta puede ser:

```
1 PREFIX char: <http://bestmovieasever.com/character/>
2 SELECT ?id ?name
3 WHERE
4 {
5     ?id a char:Character .
6     ?id char:name ?name .
7 }
```

Código 7: Consulta SPARQL simple #2

Que "selecciona todos los ?name e ?id, en donde ?name sea el objeto de los sujetos ?id a través de char:name y char:Character el objeto a través de a". En lenguaje natural, seleccionar los nombres e ids de todos los personajes. El resultado es:

Tabla 4: Resultado de consulta SPARQL del Código 7.

Fuente: Elaboración Propia.

id	name
char:123	Neo
char:3334	Elrond

Los ejemplos revisados hasta ahora corresponden a patrones básicos, es decir, solo calzar las variables dentro del grafo.

2.3.2. Patrones complejos

Se puede consultar un modelo *edge-labelled* incluyendo características relacionales a los patrones básicos. SPARQL da soporte a este tipo de operaciones a través del uso de varias *keywords* u operaciones en las consultas:

1. SELECT: Como ya se vio en los ejemplos anteriores, la característica de proyección la aporta esta *keyword*, ya que permite seleccionar un subconjunto de todas las variables usadas en una consulta:

```
1 PREFIX char: <http://bestmovieasever.com/character/>
2
3 SELECT ?name
4 WHERE {
```

```
5   ?movie a mov:Movie .
6   ?movie mov:has_character ?char .
7   ?movie mov:name ?name .
8   ?act act:plays ?char .
9   ?act act:name "Hugo Weaving"
10 }
```

Código 8: Consulta SPARQL usando proyección con SELECT

El Código 8 sirve para seleccionar el nombre de las películas en donde actuó Hugo Weaving.

2. Join: Esta operación es soportada implícitamente en cada calce de cada triple. En otras palabras, se hace un JOIN del conjunto de calces generados por el primer triple por si solo y el del segundo triple por si solo, luego se hace un join entre este conjunto de calces y los del tercer triple, y así hasta abarcar todos los triples. Es lo mismo que ocurre en SQL cuando se mantienen en el output final de un join solo las filas que compartían los valores de la columna sobre la cual se hizo el join.
3. Union: SPARQL soporta UNION con la misma keyword. Funciona de la misma forma que en SQL, incluyendo los calces de dos sub-patronos en el resultado final. Un ejemplo de su uso es el Código 9. En la variable ?name se guardaran los nombres de películas y actores. Sin el uso de UNION, esta misma consulta habría retornado un output vacío ya que el JOIN implicito sobre ?name no habría tenido valores que coincidieran.

```
1  PREFIX char: <http://bestmoviesever.com/character/>
2
3  SELECT ?name
4  WHERE
5  {
6    {
7      ?movie a mov:Movie .
8      ?movie mov:name ?name .
9    }
10   UNION
11   {
12     ?act act:plays ?char .
13     ?act act:name ?name .
14   }
15 }
```

Código 9: Consulta SPARQL usando unión con UNION

4. Difference: SPARQL provee de operación de diferencia a través de MINUS, que descarta los elementos del primer operando que aparecen en el segundo operando. En el Código 10 se muestra una consulta que selecciona los nombres de todos los personajes, menos de los que se llamen Elrond.

```
1 PREFIX char: <http://bestmovieseven.com/character/>
2
3 SELECT ?name
4 WHERE
5 {
6     ?char a mov:Character .
7     ?char char:name ?name .
8     MINUS
9     {
10         ?char char:name "Elrond" .
11     }
12 }
```

Código 10: Consulta SPARQL usando diferencia con MINUS

5. Optional: SPARQL puede incluir valores opcionales con OPTIONAL. Por ejemplo si se consulta el Grafo 11 con la consulta del Código 12, se obtienen los calces (Jorge, <jorge@example.com>) y (Ricardo,), en donde el segundo calce no tiene resultado para la variable email. Si no se hubiera usado OPTIONAL, el segundo calce habría sido descartado por no poder calzar la segunda variable.

```
1 @prefix foaf:    <http://xmlns.com/foaf/0.1/> .
2 @prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3
4 _:a rdf:type     foaf:Person .
5 _:a foaf:name    "Jorge" .
6 _:a foaf:email   <jorge@example.com> .
7
8 _:b rdf:type     foaf:Person .
9 _:b foaf:name    "Ricardo" .
```

Código 11: Grafo RDF para consulta con OPTIONAL

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name ?email
3 WHERE
4 {
5     ?x foaf:name ?name .
6     OPTIONAL { ?x foaf:email ?email }
7 }
```

Código 12: Consulta SPARQL usando opcionales con OPTIONAL

6. Filter: La operación de filtrado está presente en SPARQL a través de FILTER. En el Código 13 selecciona todos los personajes de Hugo Weaving que no aparezcan en una película de 1999.

```
1 PREFIX char: <http://bestmovieever.com/character/>
2
3 SELECT ?name
4 WHERE
5 {
6   ?char char:name ?name .
7   ?act act:name "Hugo Weaving" .
8   ?act act:plays ?char .
9   ?mov mov:has_char ?char .
10  ?mov mov:year ?year .
11  FILTER
12  (
13    year != 1999
14  )
15 }
```

Código 13: Consulta SPARQL usando filtrado con FILTER

2.4. Tecnologías

No basta solo con la especificación de RDF y SPARQL para lograr resultados reales. Es necesario que existan parsers sintácticos, compiladores y APIs para poder hacer uso real de estos. A continuación se exponen 2 tecnologías que sirven para usar estos lenguajes y manipular datos a través de código. También se mencionada la relación que tienen con la propuesta en la presente memoria.

2.4.1. JENA

Apache JENA es una API en Java para la manipulación de datos RDF. Tiene representaciones en clases de todos los elementos mencionados anteriormente del dominio de RDF, como nodos, variables, grafos, recursos, etc. Permite la creación de grafos, consultas, modificaciones, parseo y en general todas las tareas que se pueden realizar referente a bases de datos, RDF en este caso.

Por ejemplo, el grafo de la Figura 5 puede ser instanciado usando JENA como se muestra en el Código 14.

```
1 String mov = "http://bestmovieever.com/movie#";
2 String chara = "http://bestmovieever.com/character#";
3 Model graph = ModelFactory.createDefaultModel();
4 Resource movie = graph.createResource(mov + "345");
5 movie.addProperty(RDF.type, graph.createResource(mov + "Movie"));
```

```
6 movie.addProperty(graph.createProperty(mov + "title"), "The Matrix");
7 movie.addLiteral(graph.createProperty(mov + "year"), 1999);
8 Resource character = graph.createResource(chara + "123");
9 movie.addProperty(graph.createProperty(mov + "has_character"), character);
10 character.addProperty(graph.createProperty(chara + "has_name"), "Neo");
11 graph.setNsPrefix( "mov", mov );
12 graph.setNsPrefix( "char", chara );
13 graph.write(System.out, "TURTLE");
```

Código 14: Uso de JENA para instanciar el grafo de la Figura 5

En la jerga de Jena, un `Model` es un grafo RDF, un `Resource` es un nodo y una `Property` es un arco. Usando los métodos de construcción por defecto, se pueden instanciar los elementos que se necesiten y unirlos a través de propiedades. En la línea 5 se une un nodo de película con su tipo, utilizando la propiedad incluida en JENA RDF . type (<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> o su simplificación `a`). Al agregar una propiedad se puede especificar un nodo (línea 9) o un literal (línea 10). Las líneas 11 y 12 incluyen explícitamente los prefijos usados en las URIs. La línea 13 muestra en formato TURTLE el resultado:

```
<http://bestmovieever.com/character#123>
  <http://bestmovieever.com/character#has_name> "Neo" .
<http://bestmovieever.com/movie#345>
  a "http://bestmovieever.com/movie#Movie" ;
  <http://bestmovieever.com/movie#has_character>
    <http://bestmovieever.com/character#123> ;
  <http://bestmovieever.com/movie#title>
    "The Matrix" ;
  <http://bestmovieever.com/movie#year>
    "1999"^^<http://www.w3.org/2001/XMLSchema#long> .
```

La información en los grafos puede ser consultada con SPARQL. Esto es sencillo a través de la API de JENA, valiéndose de la clase `Query`. En el Código 15 puede verse un ejemplo de consulta al grafo anterior. La consulta definida en el `queryString` es pasada en la línea 8 para generar una consulta, que se ejecuta en la línea 9 sobre el grafo anterior. En las líneas posteriores se extraen sus resultados, se iteran y se imprimen. En este caso el resultado es simplemente `The Matrix`, `Neo`.

```
1 String queryString = "PREFIX mov: <http://bestmovieever.com/movie#>" +
2   "PREFIX char: <http://bestmovieever.com/character#>" +
3   "SELECT ?name ?char WHERE {" +
4   "?m a mov:Movie ;" +
5   "mov:title ?name ;" +
6   "mov:has_character ?c ." +
```

```
7         "?c char:has_name ?char . }" ;
8 Query query = QueryFactory.create(queryString) ;
9 QueryExecution qexec = QueryExecutionFactory.create(query, graph);
10 ResultSet results = qexec.execSelect() ;
11 for ( ; results.hasNext() ; ){
12     QuerySolution soln = results.nextSolution() ;
13     RDFNode movie_name = soln.get("name") ;
14     RDFNode chara_name = soln.get("char") ;
15     System.out.println(movie_name + ", " + chara_name);
16 }
```

Código 15: Uso de JENA para consultar grafo instanciado en Código 14

Los paquetes que más se usaran en el presente trabajo son los relacionados con la manipulación de la consulta SPARQL, es decir, los que permiten recorrer cada uno de sus componentes en operaciones de álgebra SPARQL o también en sus componentes gramaticales como triples, bloques, operadores, expresiones, etc. No interesará evaluarla sobre un grafo RDF, sino que iterar por sus componentes para representar cada operación usando Gremlin. La manipulación de SPARQL es posible con JENA gracias a su motor ARQ, cuya API provee interfaces que al implementarse pueden utilizarse en conjunto con varias clases para especificar código dependiendo del tipo de componente sobre el cual se esté iterando. Esta utilidad que presa el motor es fundamental para el proceso de traducción que se requiere. En la sección Propuesta de Solución se mostrará a través de pseudocódigo el recorrido de las consultas, y en la sección Validación de la Solución se profundizará en las formas en que el problema fue abordado a nivel de código.

2.4.2. Tinkerpop

Tinkerpop es un framework de computación de grafos. Provee todas las operaciones referentes a bases de datos, pero sobre el modelo de *property graph*. Este framework incluye Gremlin como el lenguaje para realizar todas las operaciones. Actualmente esta en su versión 3.3.3. Si bien la definición del lenguaje es abstracta, algunas de las implementaciones concretas son:

- **Gremlin-Java:** Implementación primaria y canónica de Gremlin. Principal compilador de código Gremlin a *bytecode* por su rapidez.
- **Gremlin-Groovy:** Lenguaje que por defecto utiliza la Consola Gremlin. Es un wrapper de Gremlin en Groovy que facilita la sintaxis al trabajar en la consola, como también por código.
- **Gremlin-Python:** Representa Gremlin dentro de Python y puede usarse dentro de cualquier máquina virtual Python como CPython y Jython.

- **Ogre:** Variante de Gremlin en Clojure, aumentando la expresividad.
- **SQL-Gremlin:** Compilador de SQL a *bytecode* Gremlin. Usado para herramientas de *Bussines Intelligence*.

A grandes rasgos, las operaciones que provee se pueden dividir en de **estructura** y de **procesamiento**. La estructura se refiere a la topología de los datos, es decir, los nodos, arcos, propiedades y valores en si. El procesamiento son las operaciones de análisis sobre la estructura.

Estructura: Compuesta por los nodos y los arcos, ambos con label y un conjunto de propiedades. Estas propiedades son una llave y un valor, donde la llave es siempre un string. Las propiedades de los vértices pueden a su vez tener un conjunto de propiedades, llamadas metapropiedades. Tinkerpop permite que las propiedades tengan varios valores, o dicho de otra forma, varias propiedades con la misma clave.

Un ejemplo de cómo instanciar el grafo de la Figura 2 con Gremlin en su versión Java aparece en el Código 16.

```
1 Graph graph = TinkerGraph.open();
2 Vertex matrix = graph.addVertex(T.label, "movie", "title", "The Matrix",
3   "genre", "science fiction", "year", 1999);
4 Vertex keanu = graph.addVertex(T.label, "actor", "name", "Keanu Reeves",
5   "born", "2-9-1964", "height", 1.68, "nationality", "Canadian");
6 Vertex john = graph.addVertex(T.label, "movie", "title", "John Wick",
7   "genre", "action", "year", 2014);
8 Vertex hugo = graph.addVertex(T.label, "actor", "name", "Hugo Weaving",
9   "born", "4-4-1960", "height", 1.88, "nationality", "British-Australian");
10 Vertex lotr = graph.addVertex(T.label, "movie", "title",
11   "The Lord of the Rings: The Fellowship of the Ring", "genre",
12   "epic fantasy", "year", 2001);
13 keanu.addEdge("acts_in", john, "name", "John Wick", "role", "protagonist");
14 keanu.addEdge("acts_in", matrix, "name", "Neo", "role", "protagonist",
15   "race", "human");
16 hugo.addEdge("acts_in", matrix, "name", "Agent Smith", "role", "antagonist",
17   "race", "agent");
18 hugo.addEdge("acts_in", lotr, "name", "Elrond", "role", "tertiary", "race",
19   "elf");
```

Código 16: Uso de Gremlin para instanciar grafo de la Figura 2

El ejemplo es bastante autoexplicativo. Se agregan los nodos o Vertex al grafo vacío instanciado en la línea 1 con la función `addVertex` que recibe un número indefinido de parámetros, en donde el *n*-ésimo es la llave de la propiedad y el *n*+1-ésimo es el valor de esta. Desde la línea 13 se empiezan a unir nodos usando arcos o Edge con `addEdge`, que recibe el label

como primer parámetro, el nodo de llegada como segundo y luego una cantidad indefinida para propiedades del mismo modo anterior.

Procesamiento: La forma en que se procesan los datos con Gremlin es a través de *traversals* sobre el grafo, que en palabras simple es la expresión de una caminata algorítmica sobre los elementos estructurales del grafo. La API de Tinkerpop provee una forma sistemáticamente natural de ir armando un *traversal* a través de la composición de funciones, que van indicando un paso dentro del grafo.

Por ejemplo, la consulta humana *¿Cuáles son los nombres de las películas en donde trabaja Hugo Weaving?* sobre el grafo de la Figura 2 expresada como pasos sería:

1. Tomar el o los nodos que sean actor
2. Considerar solo los que tengan nombre Hugo Weaving
3. Desde ahí, avanzar al siguiente nodo a través de un arco cuyo label sea `acts_in`
4. Obtener el valor de la propiedad `name` de esos nodos

Esto puede ser expresado en Gremlin. Todos los *traversals* parten de un `TraversalSource`, que pueden ser todos los vértices del grafo o todos los arcos. Cuando se inicia el proceso, se obtiene un `GraphTraversal`, cuyos métodos o pasos pueden ser encadenados. Existen varios tipos de pasos:

- *map*: Transforma el objeto *traverser* (el que recorre el *traversal*) en otro objeto.
- *flatMap*: Transforma el objeto *traverser* en un iterador de otros objetos.
- *filter*: Niega o permite el paso del *traverser* al siguiente paso.
- *sideEffect*: Hace continuar el *traverser* sin cambio, pero le inserta algún efecto comunicacional adicional.
- *branch*: Separa el *traverser* en varios y los envía a distintos lados del *traversal*.

Con Gremlin, la consulta anterior puede hacerse en Java como aparece en el Código 17, obteniendo como resultado el calce (The Lord of the Rings: The Fellowship of the Ring, The Matrix).

```
1 GraphTraversalSource s = graph.traversal();
2 System.out.println(s.V().has("name", "Hugo Weaving")
3   .out("acts_in").values("title").toSet());
```

Código 17: Uso de Gremlin para consultar grafo instanciado en 16

En este ejemplo, se obtienen un `GraphTraversal` con `V()` desde el `GraphTraversalSource` `s` y se comienza a avanzar paso a paso. `has` es un *filter* ya que deja algunos elementos en el *traversal*, según la propiedades y/o valor especificado, `out` es un `flatMap` que avanza al siguiente conjunto de vértices, conectados al inicial por algún arco del label especificado. El paso `values` es un *map* que entrega los valores de la propiedad especificada. Finalmente es necesario un paso que entregue los valores que se desea ya que todo paso retorna un *traversal* (lo que hace posible la composición). Para esto existen los pasos terminales, como `toSet`, que retorna un conjunto con los valores obtenidos.

A continuación se hace un resumen de los pasos más utilizados en el presente trabajo. Los ejemplos se simplificarán utilizando Gremlin en su versión Groovy, menos verbosa, sobre la consola Gremlin, que permite interacción por línea de comando con los grafos. Las líneas que comienzan con `>` son output.

- `and`: Se asegura que todos los *traversals* pasados como argumento entreguen al menos un resultado.

```
1 g.V().and(  
2   out("play"),  
3   values("age").is(lt(25))  
4 ).values("name")
```

Solo pasarán los vértices que tengan un arco con label `play` y que tengan edad menor a 25.

- `dedup`: Se filtran los objetos repetidos en el *traversal* actual. Los ejemplos muestran un output normal

```
1 g.V().values("some")  
2 > "val1"  
3 > "val1"
```

y uno de-duplicado

```
1 g.V().values("some").dedup()  
2 > "val1"
```

- `has`: Este paso tiene variantes que ayudan a filtrar objetos del *traversal*.
 - `has(key, value)`: Remueve los elementos que no cumplan con tener una propiedad `key` de valor `value`
 - `hasLabel`: Filtra según label
 - `hasId`: Filtra según id
 - `hasKeys`: Filtra según nombre de propiedades

- `hasValue`: Filtra según valores de las propiedades de los objetos
- `id()`: Entrega el id de los elementos.
- `is()`: Como se vio anteriormente, filtra los elementos según el predicado que se le entrega:

```
1 g.V().values("some")
2 > 1
3 > 2
4 > 3
5 > 4
6 > 5
7 g.V().values("some").is(gte(3))
8 > 3
9 > 4
10 > 5
11 g.V().values("some").is(lt(3))
12 > 1
13 > 2
14 g.V().values("some").is(inside(1,4))
15 > 2
16 > 3
```

- `or`: Deja pasar los objetos que logren avanzar por alguno de los *traversals* entregados como parámetro. El ejemplo entrega los nombres de todos quienes posean una moto o un auto.

```
1 g.V().hasLabel("person").or(
2   out("has_car"),
3   out("has_motorcycle")
4 ).values("name")
```

- `values`: Entrega los valores de las propiedades de los elementos actualmente en el *traversal*. Estos elementos pueden ser nodos, arcos o propiedades de nodo. Se puede incluir un parámetro para limitar solo a los valores de la propiedad cuyo nombre es el parámetro, como `g.V().values("some_prop")`
- `out`: Como ya se ha visto, este paso simplemente avanza a los siguientes vértices desde los del actual *traversal*, cumpliendo con el label entregado o a todos, cuando no se entrega ninguno. Existen muchas variantes de este paso, cuya función es básicamente la de navegar sobre la estructura del grafo, algunos desde vértices y otros desde arcos:
 - `in`: Avanza a los nodos desde los cuales los arcos que llegan a los nodos actuales salen
 - `both`: Unión de los resultados de `in` y `out`

- `outE`: Avanza a los arcos que salen desde los vértices actuales
 - `inE`: Avanza a los arcos que llegan a los vértices actuales
 - `inV`: Avanza a los vértices a los que llegan los arcos actuales
 - `outV`: Avanza a los vértices de los que salen los arcos actuales
- `where`: Otra forma de filtro usado con predicados como los de `is`. Deja pasar los elementos que cumplan con el parámetro. Los dos ejemplos generan el mismo output, pero uno utiliza este paso

```
1 g.V().and(  
2   out('play'),  
3   values('age').is(lt(25))  
4 ).values('name')  
5 > john  
6 > stuart  
7 g.V().where(out('play')).where(values('age').is(lt(25))).values('name')  
8 > john  
9 > stuart
```

Existen una infinidad de pasos en Gremlin. Aquí solo se mencionó los que serán utilizados posteriormente. Algunos pasos más complejos que se utilizarán se explicarán en su respectiva sección. Para una lista exhaustiva con ejemplos, favor referirse a [9]

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

En esta sección se propondrán diversas soluciones a la problemática planteada. Cada una de esas soluciones presenta distintos desafíos, ventajas y desventajas y en base a estos se optará por una, que será implementada posteriormente. En esta sección se presentan algoritmos, diseños y traducciones sin importar el código en sí que une cada parte del programa final. Solo se especificará código cuando este sea Gremlin, pero tampoco se ahondará en su sintaxis específica en el lenguaje a utilizar posteriormente.

Lo importante en esta sección es que quede claro el flujo que seguirán los datos desde la obtención de la consulta hasta el output final, pasando por los pasos intermedios de descubrimiento de variables, traducción, armado de *traversals*, etc.

3.1. Formas de calce

Primero, es necesario establecer el curso de investigación hacia la mejor forma de traducción de un modelo al otro. En general las opciones son dos:

1. Interpretar la consulta SPARQL en bruto sobre el PG
2. Mapear el modelo PG a RDF.

Lo que se quiere es consultar una base de datos estructurada como PG, usando un lenguaje para consultar bases de datos RDF. Es decir, consultar una base de datos con más tipos de estructuras para almacenar datos con un lenguaje hecho para menos tipos.

En el caso 1, una consulta SPARQL es un *edge-labelled graph* al que le faltan partes, representadas con variables. Como los grafos PG tienen más componentes, una consulta SPARQL puede calzar muchas relaciones sujeto-predicado-objeto si cada uno de estos elementos puede ser un nodo, arco, propiedad, valor o metapropiedad, lo cual es de gran complejidad. En cuanto al uso de Gremlin, representa un gran reto hacer consultas si no se sabe a priori que son las variables, lo cual hace a esta opción un gran desafío. Esto sumado a que una variable no puede ligar dos tipos distintos de elementos.

Una traducción directa entre SPARQL y Gremlin puede sonar una buena opción pero es de difícil implementación. Las diferencias entre ambos lenguajes, como lo son su sintaxis, su paradigma y la forma de la consulta, hacen que no sea directo mapear cada operación de SPARQL a una de Gremlin. Si bien ambos lenguajes soportan las mismas operaciones sobre un grafo, estas no se encadenan de la misma forma en la consulta. SPARQL resuelve la consulta

recursivamente desde adentro hacia afuera, mientras que Gremlin la resuelve paso a paso encadenado.

En el caso 2, se busca entender el PG como si fuera un grafo RDF normal (solo vértices y arcos RDF). De esta forma el calce usando una consulta SPARQL sería más directo ya que cada elemento de la consulta SPARQL se referiría a un elemento de un grafo RDF, y por ende no se incurre en la excesiva complejidad mencionada anteriormente. Aun en este caso, es necesario finalmente generar una consulta Gremlin sobre el PG.

La dirección del presente trabajo es conseguir un mapeo de PG a RDF debido a que la otra opción, si bien es mas general, es muy amplia, muy compleja de implementar y de ejecutar y escapa del espectro de la memoria.

A continuación se profundiza en esto y se exponen ejemplos

3.1.1. Calce de etiquetas

Un triple RDF siempre se refiere a dos nodos y un arco que los une, a través de las URIs o valores literales, dado que sigue el modelo *edge-labelled graph*. En el caso del modelo *property graph*, cada elemento posee propiedades, por lo que el calce de un triple no es directo.

Suponiendo que un triple RDF sigue significando lo mismo en ambos modelos, es decir, calza las etiquetas, entonces no es suficiente para representar por completo un PG porque deja de lado las propiedades. Por ejemplo, El triple (`person` ?a ?b) podría ser calzado en el grafo de la Figura 6 de las formas que aparecen en la Tabla 5.

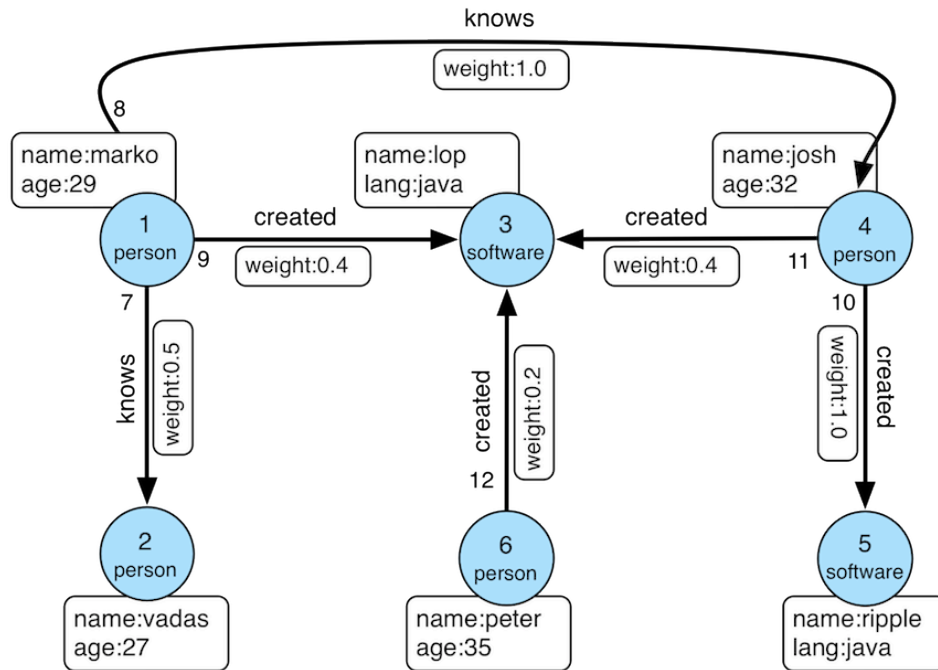


Figura 6: *Property graph Modern* incluido en Tinkerpop.

Fuente:

<http://tinkerpop.apache.org/docs/current/images/tinkerpop-modern.png>

Accedido: 18/10/2017.

Tabla 5: Calces de etiquetas usando (person ?a ?b) en grafo de Figura 6

Fuente: Elaboración Propia.

a	b
knows	person
created	software
knows	person
created	software
created	software
created	software

Con esta forma de calce nunca se podría acceder a las propiedades. Entonces es necesario reinterpretar el significado de los triples RDF para poder calzar toda la información presente dentro de un PG.

3.1.2. Calce de propiedades

Si se toma la label de los nodos y arcos como referencia, y se asume que el id es una propiedad como cualquier otra, un triple en este modelo puede significar lo mostrado en la Tabla 6.

Tabla 6: Posible significado de un triple en un *property graph*.

Fuente: Elaboración Propia.

Sujeto	Predicado	Objeto
label de nodo	key de propiedad	valor de propiedad
label de nodo	label de arco	label de nodo
label de arco	key de propiedad	valor de propiedad

Sabiendo esto, el mismo triple anterior podría calzar en el grafo como se muestra en la Tabla 7

Tabla 7: Calces de propiedades usando (*person* ?a ?b) en grafo de Figura 6.

Fuente: Elaboración Propia.

a	b
id	2
name	vadas
age	27
knows	person
id	1
name	marko
age	29
created	software
knows	person
id	4
name	josh
age	32
created	software
created	software
id	6
name	peter
age	35
created	software

De esta forma, se puede calzar más información del grafo. Si existiera un arco con label *person*, entonces habría más cantidad de calces, ya que se calzarían las propiedades de ese arco. A este modelo aún le faltan las metapropiedades que pueden tener las propiedades de nodo en Tinkerpop. Este tema será abordado en las secciones de Alternativas.

Aunque este modelo se ve prometedor, no es compatible con como se pretende hacer el calce en Gremlin. En las dos próximas secciones se revisará la funcionalidad principal de Gremlin que será usada para calzar patrones, y por que esta no es compatible con el actual modelo propuesto.

3.1.3. Calce con Gremlin

Gremlin cuenta con el paso `match` que provee una forma declarativa de expresar calces de patrones. Esta función recibe un número arbitrario de *traversals*, llamados en este caso patrones, que deben contener ligados a variables con el paso `as`, y retorna un mapeo de variables a valores de tal forma que las variables mantienen la consistencia a través de todos los patrones.

Dado que SPARQL es declarativo y sirve para calzar patrones, es lógico pensar que esta función es la indicada para la traducción. De hecho la relación entre SPARQL y la funcionalidad de `match` es mencionada explícitamente en la documentación oficial [5].

Para ilustrar el funcionamiento de `match` considerar el siguiente ejemplo: Se tiene el grafo de la Figura 7 que contiene todo lo que un PG puede contener, además de metapropiedades y multipropiedades de Tinkerpop. Se desea consultar todos los softwares que Stephen use y desarrolle. Usando Gremlin de la forma convencional, es decir, composición de funciones, podría ser como se muestra en el Código 18.

```
1 g.V().has('name', 'stephen').out('uses').as('software')
2   .in('develops').has('name', 'stephen').select('software').values('name')
```

Código 18: Consulta Gremlin #1

Esta consulta retorna `gremlin` y `tinkergraph`⁵. Ahora usando `match`, la consulta queda como la del Código 19.

```
1 g.V().match(
2   __.as('x').out('uses').as('y'),
3   __.as('x').out('develops').as('y'),
4   __.as('x').has('name', 'stephen')
5 ).select('y').values('name')
```

Código 19: Consulta Gremlin usando `match`

Retorna el mismo resultado. Incluso la sintaxis es similar a SPARQL, en donde se declaran variables que sirven para hacer *join* entre cada triple del patrón.

Cada patrón dentro de `match` debe cumplir con alguna de las siguientes estructuras:

- `__.as('x') ... .as('y')` Una variable al principio y al final. La variable al principio hace que el *traverser* parta en la ubicación etiquetada por esa variable (sean nodos, propiedades, arcos, etc.) y avance cada paso hasta el punto anterior a la variable del

⁵Una vez cada uno, debido a que se especificó `has('name', 'stephen')` luego de `in('develops')`. Si no se hubiera hecho esto, habría retornado 2 veces `tinkergraph` y 3 `gremlin`, pero solo se quería el software en donde estuviera involucrada esa persona.

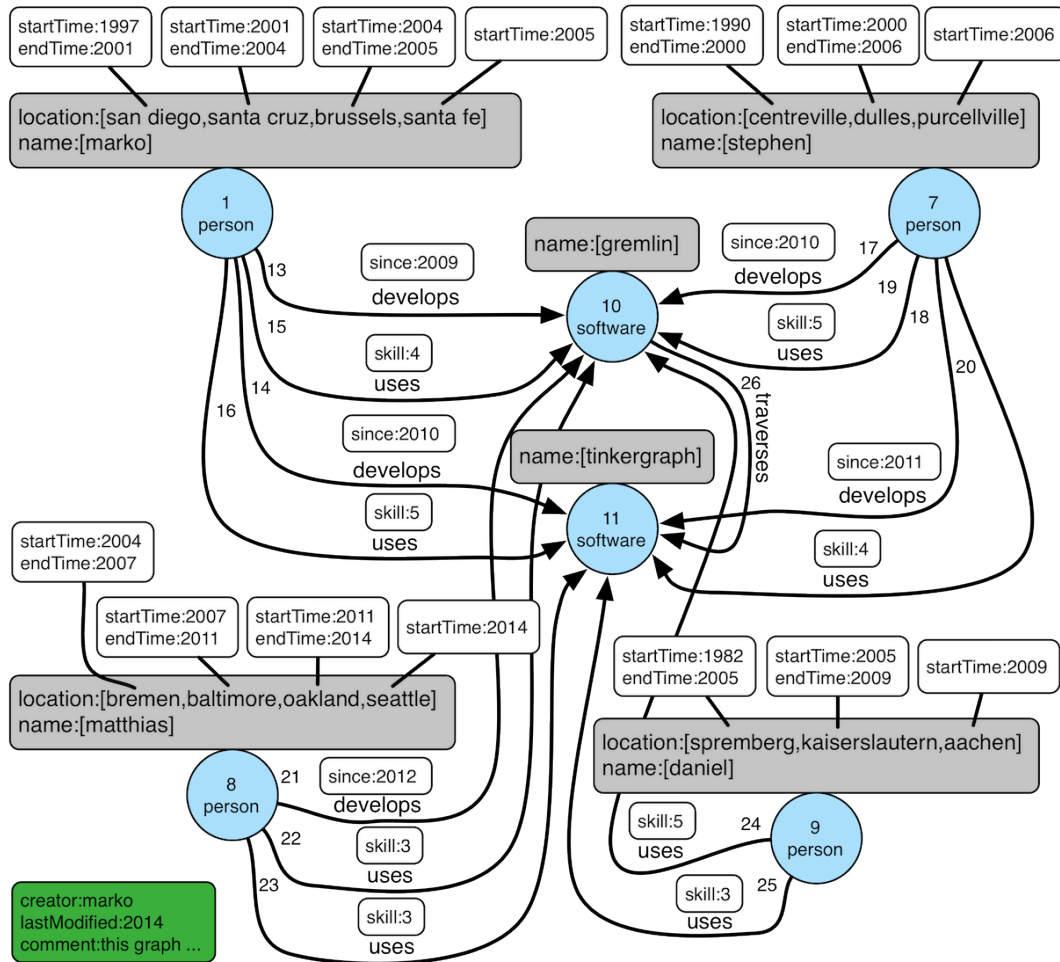


Figura 7: Grafo Tinkerpop 'The Crew' incluido en Gremlin.

Fuente: <http://tinkerpop.apache.org/docs/3.3.0/images/the-crew-graph.png>

Accedido: 12/10/2017.

final. Si esa variable final fue ligada en algún patrón anterior del `match`, entonces el *traverser* filtra los resultados recién obtenidos con los ya ligados, de manera que solo se mantengan los que coinciden. Si la variable no fue ligada anteriormente, entonces se etiqueta ese punto ligándolo a la variable del final, que puede ser usada después.

- `__ .as('x') ...` Una variable al principio. El *traverser* comienza de la posición etiquetada por la variable y recorre cada paso filtrando el contenido de la variable.
- Sin variables al principio ni al final. Esto se usa para utilizar operaciones como `or`, `and` y `where`, en donde los patrones ingresados como argumento se filtran según corresponda.

Si una variable se usa al principio de un patrón, esta debe haber sido ligada anteriormente, ya sea dentro o fuera de `match`. La única excepción es la variable inicial del `match` que se liga al *traversal* que viene antes de este.

Las variables ligadas dentro de `match` son únicas en todo el *traversal*, incluso fuera de `match`. Por esto es que pueden aplicarse selecciones, filtros y ordenamientos aún después de este.

3.1.4. Incompatibilidad entre modelo y `match`

Es muy difícil implementar una traducción a Gremlin utilizando `match` y calces tan genéricos como los mencionados anteriormente. Esto se debe en general al comportamiento de las variables dentro de la función. Una variable no puede ligar a varios tipos distintos de datos (nodos, arcos, propiedades, etc.).

Como la intención de las consultas SPARQL sobre el modelo es retornar cualquier elemento de un PG, sin limitarlo a etiquetas o valores como en el caso de RDF, es necesario considerar los nodos, aristas y metapropiedades como elementos que también pueden ser ligados a variables dentro de la consulta.

Con lo anterior, supongamos el triple `?x ?y 3`. En este caso, si se quisiera generalidad, `x` podría representar cualquier elemento e `y` podría ser la relación entre ese elemento y el valor 3. Por ejemplo, como se muestra en la Tabla 8, las combinaciones son bastantes. Si existe un calce en donde `?x` es un nodo y otro en donde es un arco, entonces dado el funcionamiento de `match`, esto no sería posible, no habría calce y ocurriría una excepción.

Tabla 8: Ejemplo de combinaciones de tipos de dos variables.

Fuente: Elaboración Propia.

x	y
nodo	propiedad
propiedad	valor
metapropiedad	valor
arco	propiedad
arco	id
nodo	id
arco	label
nodo	label

Una forma de solucionar esto sería generar varias variables ?x, con distintos nombres, cada una con la intención de calzar un determinado elemento dentro del grafo, a la vez que en vez de utilizar la variable ?y, se reemplazan todas las posibles relaciones entre las variables y el valor 3, como se ve en la Table 9. Así, todas las veces que ocurra calce en alguna variable, no interferirá con las otras. Luego de esto se tendría que unificar el resultado de todas estas en la variable original.

Tabla 9: Ejemplo de calces con variables replicadas.

Fuente: Elaboración Propia.

Variable	Predicado	Valor
?x_node	label	3
?x_node	id	3
?x_edge	label	3
?x_edge	id	3
?x_prop	value	3

Este método tiene la gran desventaja de que cada variable generada aumenta la complejidad en la ejecución. Además, si la variable interactúa con otra variable que también fue multiplicada, entonces habría que incluir en la consulta todas las permutaciones posibles entre estas⁶. Esto aumenta exponencialmente para más variables. Tampoco es posible la unificación de las variables en Gremlin.

La generalidad en la solución de la problemática de este proyecto trae muchos inconvenientes. Es por esto que se decidirá ir por una solución mas particular. Las alternativas mostradas a continuación toman como base el proyecto SPARQL-Gremlin [11], un nuevo estilo de consultas SPARQL y las ideas mencionadas anteriormente de generar un mapeo entre los dos modelos. Se estudiarán las ventajas y desventajas de las propuestas y se determinará la más adecuada para proseguir.

⁶En el caso de que en vez de un 3 fuera una variable, por ejemplo

3.2. Alternativa 1: SPARQL-Gremlin + SPARQL*

Se utiliza la misma sintaxis definida para SPARQL-Gremlin [11] para especificar arco, propiedad o valor de propiedad. Esta sintaxis no permite el acceso a propiedades de arcos, por lo que no representa bien un PG. Para esto se puede agregar la sintaxis de SPARQL* [16] que sugiere una nueva sintaxis a SPARQL que permita agregar metadatos a las relaciones (*statement-level metadata*), es decir, a los arcos.

La nueva forma de SPARQL trabaja sobre RDF*, que es una nueva forma de escribir grafos RDF con metadatos utilizando triples anidados para incluirlos y delimitando el triple principal con << >>. Por ejemplo en el siguiente código⁷

```
:n ns:name "Billy" .
:n ns:age 20 .
<< :n ns:fav_food "pizza" >> data:source <http://food.com/favs > ;
    data:creator :m .
```

se establecen 3 triples regulares y dos anidados. Los primeros dos establecen el nombre y la edad de la entidad representada con el nodo vacío :n. Luego se establece la comida favorita y como metadatos de esta relación se agregan la fuente y el creador de esta. El triple al que se le agregan los metadatos está delimitado y actúa como "sujeto" del triple formado por el, el predicado del metadato el objeto del metadato. Notar que también se puede utilizar la simplificación con ; para usar el mismo sujeto, que en este caso es el triple. No confundir los delimitadores de la URL con los propios de RDF*, estos son utilizados específicamente para URLs.

SPARQL* sigue el mismo principio. Una consulta SPARQL* es un grafo RDF* al que le faltan valores y en vez de estos tiene variables.⁸

Con esta sintaxis se podría incluir propiedades de los arcos dentro de las consultas. Por ejemplo para consultar el grafo de la Figura 7:

- Seleccionar todos los nombres de personas que tengan un nivel de habilidad 3 usando el software tinkergaph:

⁷Manteniendo la sintaxis Turtle utilizada a través de todo el documento

⁸Todo los detalles de esta sintaxis, fundamentos y aplicaciones pueden encontrarse en [16]

```
1 SELECT ?name WHERE {
2   ?x v:label "person" .
3   ?x v:name ?name .
4   ?y v:label "software" .
5   ?y v:name "tinkergraph" .
6   << ?x e:uses ?y >> v:skill 3 .
7 }
```

Código 20: Consulta SPARQL* # 1

SPARQL-Gremlin utiliza los prefijos *v* para referirse al valor del sufijo en el sujeto. Por ejemplo en la primera línea se estaría accediendo al valor del label de los nodos ligados a *x*, y estos deben valer *person*.

Por otro lado, utiliza el prefijo *e* para indicar el paso de un nodo a otro nodo a través de un arco de label igual al sufijo. De esta forma se indica que entre *x* a *y* hay un arco con label *uses*.

Para traducir a Gremlin es necesario crear variables. Esto se debe a que el arco debe ligarse para luego accederse a sus propiedades, especificadas como metadatos en la consulta. Por ejemplo la variable *x_uses_y* cumpliría este propósito. Una traducción es como la que aparece en el Código 21.

```
1 g.V().match(
2   __.as('x').values('name').as('name'),
3   __.as('x').outE('uses').as('x_uses_y'),
4   __.as('x_uses_y').inV().as('y'),
5   __.as('x').hasLabel('person'),
6   __.as('y').hasLabel('software'),
7   __.as('y').values('name').is('tinkergraph'),
8   __.as('x_uses_y').values('skill').is(3)
9 ).select('name')
```

Código 21: Consulta SPARQL* # 1 traducida a Gremlin

Otro ejemplo:

- Seleccionar todos los años de inicio de trabajo de todas las personas que hayan desarrollado algún software desde el 2010 en adelante, y que también lo usen con 4 o más de habilidad.

La consulta expresada en SPARQL* aparece en el Código 22. La traducción, siguiendo el esquema anterior, es como la del Código 23.

```
1 SELECT ?year WHERE {
2   ?x v:label "person" .
3   ?y v:label "software" .
4   ?x p:location ?location .
```

```

5    ?location v:startTime ?year .
6    << ?x e:develops ?y >> v:since ?since .
7    << ?x e:uses ?y >> v:skill ?skill .
8    FILTER (?since >= 2010 && ?skill >= 4)
9  }

```

Código 22: Consulta SPARQL* # 2

```

1
2  g.V().match(
3    __.as('x').properties('location').as('location'),
4    __.as('location').values('startTime').as('year'),
5    __.as('x').outE('develops').as('x_develops_y'),
6    __.as('x_develops_y').inV().as('y'),
7    __.as('x_develops_y').values('since').as('since'),
8    __.as('x').outE('uses').as('x_uses_y'),
9    __.as('x_uses_y').inV().as('y'),
10   __.as('x_uses_y').values('skill').as('skill'),
11   __.as('x').hasLabel('person'),
12   __.as('y').hasLabel('software'),
13   __.as('since').is(gte(2010)),
14   __.as('skill').is(gte(4))
15 ).where(
16   __.and(
17     __.as('since').is(gte(2010)),
18     __.as('skill').is(gte(4))
19   )
20 ).select('year')

```

Código 23: Consulta SPARQL* # 2 traducida a Gremlin

Estos ejemplos muestran que la traducción sería capaz de calzar todos los elementos de un PG que inicialmente se podían calzar en SPARQL-Gremlin, sumado a propiedades de cada arco. Notar que esta traducción también genera variables para ligar los arcos y luego acceder a sus propiedades.

De forma adicional, se muestra que la aplicación de `FILTER` es directa, incluyendo las operaciones booleanas dentro de un `where` luego de `match`. Se profundizará esto posteriormente.

3.2.1. Desventajas

- Una desventaja de esta forma de traducción es el hecho de que se está utilizando SPARQL-Gremlin sin ningún cambio. Este método tiene falencias que se mantienen en la traducción. La mayor falencia es que la única forma de referirse a un arco es a través

de su label. En caso de no saber la label o querer ejecutar un avance general por los elementos del grafo, no se puede hacer. Esto es una limitante inaceptable para lo que se desea lograr.

- Dado que no se especifica en ningún momento una variable para referirse a un arco, estos no puede seleccionarse. A pesar de que si existe una variable para referirse al arco, esta es generada y el usuario no sabe su identificador ⁹.

3.3. Alternativa 2: SPARQL-Gremlin + SPARQL puro + variables en predicados

La alternativa anterior no permite que en la consulta SPARQL existan variables en el predicado, ya que, en SPARQL-Gremlin, este siempre se utiliza para establecer el acceso desde el sujeto al objeto, ya sea valor, nodo o propiedad. La siguiente alternativa permite usar variables en el predicado que sirvan para ligar el o los arcos correspondientes y acceder posteriormente a sus propiedades. Esto permitiría mantener la sintaxis de SPARQL intacta. La consulta en el Código 20 quedaría como se muestra en el Código 24.

```
1 SELECT ?name WHERE {  
2   ?x ?e ?y .  
3   ?x v:label "person" .  
4   ?x v:name ?name .  
5   ?y v:label "software" .  
6   ?y v:name "tinkergraph" .  
7   ?e v:label "uses" .  
8   ?e v:skill 3 .  
9 }
```

Código 24: Consulta SPARQL con variable en predicado

La traducción es esencialmente la misma que la del Código 21, excepto que esta vez la variable que representa a los arcos tiene un nombre explícito (?e), que no es producto de la traducción. De este modo, cada vez que haya una variable en el predicado, esta se liga a arcos y las otras dos a nodos. Para el caso del Código 22 queda como la del Código 25. La traducción a Gremlin es también la misma.

⁹Los nombres de las variables en los códigos 21 y 23 son demostrativos

```
1 SELECT ?year WHERE {  
2   ?x ?e ?y .  
3   ?x ?e2 ?y .  
4   ?x v:label "person" .  
5   ?y v:label "software" .  
6   ?x p:location ?location .  
7   ?location v:startTime ?year .  
8   e? v:since ?dev_year .  
9   e? v:label "develops" .  
10  e2? v:skill ?skill .  
11  e2? v:label "uses" .  
12    FILTER (?dev_year >= 2010 && ?skill >= 4)  
13 }
```

Código 25: Consulta SPARQL con variable en el predicado # 2

3.3.1. Desventajas

- Esta sintaxis restringiría a que si existe una variable en el predicado de un triple, esta siempre representa un arco.
- Al necesitarse una nueva variable por cada arco, las consultas en general serán más largas, dificultando su escritura.

3.4. Alternativa 3: Nuevo mapeo de Tinkerpop a RDF + SPARQL*

Dadas las dificultades de utilizar SPARQL-Gremlin tal cual está, es necesario idear una nueva forma de interpretar un grafo Tinkerpop como grafo RDF, para así facilitar la consulta de este con SPARQL. Se requiere que este nuevo mapeo incluya todas las características de un grafo Tinkerpop, para poder ejecutar calces entre todos sus componentes como sugiere la definición formal. También se desea que sea un mapeo bidireccional, es decir, que teniendo la definición de los datos en RDF o en Tinkerpop¹⁰, ambos signifiquen lo mismo y se pueda mapear en ambas direcciones.

Se sugiere el siguiente esquema, basado en SPARQL-Gremlin pero con varios agregados y cambios:

1. Los nodos, arcos y propiedades de nodos Tinkerpop son nodos RDF vacíos.
2. Si un nodo `_ :n` tiene un *id* `i`, se cumple el triple `_ :n n:id i`

¹⁰Se entrará en detalles de formatos en la sección de Validación de la Solución

3. Si un nodo `_:n` tiene un *label* `l`, se cumple el triple `_:n n:label l`
4. Si un nodo `_:n` tiene una propiedad `_:p` llamada *property*, se cumple el triple `_:n np:property _:p`
5. Si una propiedad `_:p`, que le pertenece a un nodo, tiene un valor de `v`, se cumple el triple `_:p np:value v`
6. Si una propiedad `_:p`, que le pertenece a un nodo, tiene una metapropiedad llamada *mprop* que vale `v`, se cumple el triple `_:p meta:mprop v`
7. Si dos nodos `_:n1` y `_:n2` están conectados por un arco `_:edge`, se cumplen los triples `_:n1 e:out _:edge` y `_:edge e:in _:n2` simultáneamente.
8. Si un arco `_:edge` tiene un *id* `i`, se cumple el triple `_:edge e:id i`
9. Si un arco `_:edge` tiene un *label* `l`, se cumple el triple `_:edge e:label l`.
10. Si un arco `_:edge` tiene una propiedad llamada *property* que vale `v`, se cumple el triple `_:edge ep:property v`

En la Figura 8 se muestra un grafo Tinkerpop y en la Figura 9 como se representaría en RDF con el mapeo anterior. Este grafo cumple todo lo listado anteriormente. Se utilizan los prefijos `n`, `e`, `np`, `ep` y `meta` para facilitar la escritura, lectura y posterior traducción de la consulta SPARQL a Gremlin. Es irrelevante saber que URI representan. Si bien el usuario obtendría el mismo resultado utilizando los prefijos o las URIs, solo haría que las consultas quedaran mucho más largas.

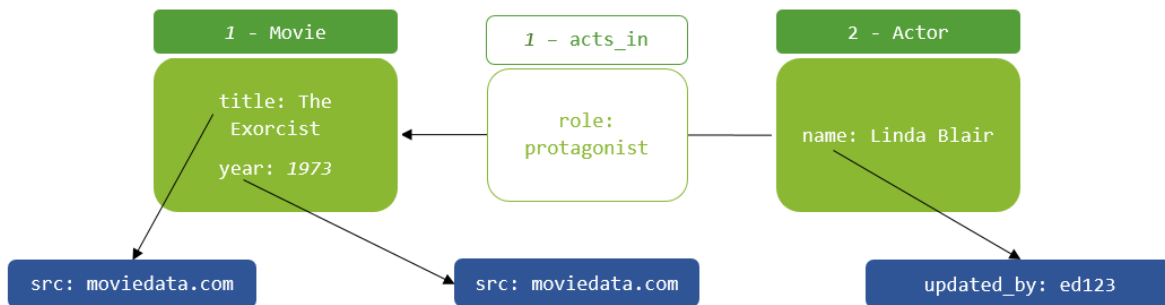


Figura 8: Grafo Tinkerpop con metapropiedades
Fuente: Elaboración propia.

En Tinkerpop, solo las propiedades de nodos pueden tener meta-propiedades, las de los arcos no. Es por esto que se optó por representar solo las propiedades de nodo Tinkerpop como nodo RDF vacío, y las de los arcos como el arco desde el nodo RDF vacío que representa el arco Tinkerpop al valor de esta.

En el Código 26 se muestra la consulta del Código 22 según este modelo.

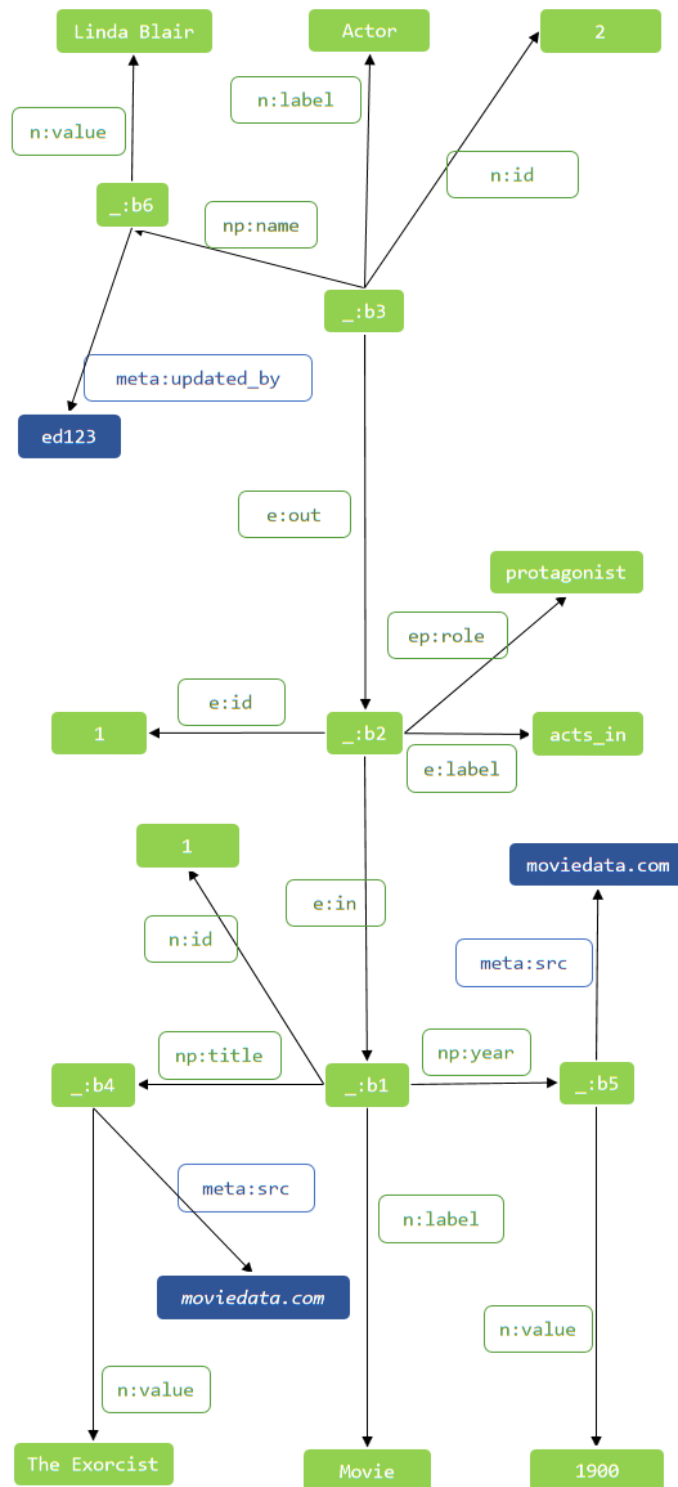


Figura 9: Representación en grafo RDF de 8 según mapeo
Fuente: Elaboración propia.

```
1 SELECT ?year WHERE{
2   ?x n:label "person" .
3   ?y n:label "software" .
4   ?x np:location ?location .
5   ?location meta:startTime ?year .
6   ?x e:out ?e1 .
7   ?x e:out ?e2 .
8   ?e1 e:in ?y .
9   ?e2 e:in ?y .
10  ?e1 e:label "develops" .
11  ?e1 ep:since ?since .
12  ?e2 e:label "uses" .
13  ?e2 ep:skill ?skill .
14  FILTER (?since >= 2010 && ?skill >= 4)
15 }
```

Código 26: Consulta SPARQL siguiendo nuevo modelo

3.4.1. SPARQL* para simplificar consultas

La evidente excesiva verbosidad de la consulta puede ser reducida si se use la sintaxis de SPARQL*. Se propone utilizar esta sintaxis para evitar escribir variables que ligan a nodos vacíos que no se pretende seleccionar. Cada una de estas contracciones deberá ser traducida primero a SPARQL según este modelo antes de pasar a Gremlin.

- **Uso 1: Calce rápido Nodo-Propiedad-Valor (NPV):** Se puede expresar el calce del valor de una propiedad de un nodo de la forma

<< ?node np:property "value" >> .

Para traducir esto a SPARQL se crea una variable auxiliar y se generan dos triples:

?node np:property ?aux .
?aux n:value "value".

De la misma forma que en la Alternativa 1, se puede incluir triples anidados que hagan uso de la propiedad (variable) implícita. En este caso interesaría calzar meta-propiedades

<< ?node_variable np:property "value" >> ;
meta:meta_property1 "meta_value1";
meta:meta_property2 "meta_value2".

Para la traducción con triples anidados, cada uno toma como sujeto a la variable creada:

```
?node_variable np:property ?aux .  
    ?aux n:value "value".  
?aux meta:meta_property1 "meta_value1".  
?aux meta:meta_property2 "meta_value2".
```

- **Uso 2: Calce rápido Nodo-Arco-Nodo (NEN):** Se puede expresar el calce de dos nodos unidos por un arco de la forma

```
<< ?node1 e:to ?node2 >> .
```

lo que creando una variable auxiliar en SPARQL significa

```
?node1 e:out ?aux .  
    ?aux e:in ?node2.
```

Notar que se agregó la nueva URI `e:to` al vocabulario para representar el avance de un nodo a otro. A diferencia de la Alternativa 1, no es necesario especificar un label de arco para calzarlo. Igual que en el uso anterior, se puede incluir triples anidados que hagan uso de la variable implícita. En este caso interesaría calzar `id`, `label` o propiedades del arco.

```
<< ?node1 e:to ?node2 >> ;  
    e:id 1 ;  
    e:label "label";  
    ep:prop "value".
```

lo que se traduce igual que el caso anterior:

```
?node1 e:out ?aux .  
    ?aux e:in ?node2 .  
    ?aux e:id 1 .  
    ?aux e:label "label".  
    ?aux ep:prop "value".
```

Es necesario señalar que la sintaxis original de SPARQL* no permite el uso de un triple delimitado con `<< >>` que no tenga además metadatos incluidos a continuación. Es por esto que los calces rápidos NPV y NEN no cumple 100 % con la sintaxis sugerida por el autor, más si con la idea de incluir meta-datos referentes al *statement* (el *triple*). Por ejemplo el triple `<< ?a np:property ?c >> .` no cumple con la sintaxis original porque esta solo permite la existencia de `<< >>` cuando este se utiliza como sujeto u objeto de un triple superior (por

ejemplo `<<?a np:property ?b>> x:y value .`), pero si cumple con la sintaxis propuesta en el presente trabajo.

Recordar que la idea de usar SPARQL* es simplificar la sintaxis de consultas. Utilizándola tal cual lo propone el autor las consultas no siempre podrían simplificarse ya que, aun cuando no se quiera acceder a metadatos de un triple, ya sea propiedades de arco o propiedades de arco, se tendría que incluir algo al lado del triple delimitado para cumplir con la sintaxis, por ejemplo variables que luego no se utilizarán, como en el triple `<<?a np:prop ?b >> ?x ?y .`

3.5. Elección

Luego de este análisis es claro que la opción más adecuada para el diseño y desarrollo de la solución es la Alternativa 3. Esto se debe a que tiene todo lo que se pretendía. Por un lado, es capaz de calzar todos los elementos de un grafo Tinkerpop utilizando SPARQL y tiene una sintaxis precisa, entendible y no tan extensa provista por las ideas de SPARQL*. Además, dado que se basa en SPARQL-Gremlin es claro que la utilización de `match` es bastante directa.

A continuación se explicará el diseño y flujo de la solución sin entrar en detalles de implementación más que cuando sea necesario incluir pseudocódigo, el cual se combinará un poco con las funciones que provee Gremlin para que sea más entendible, y para que el posterior paso a analizar el código en la sección de validación sea más directo.

3.6. Algoritmo

El siguiente paso es determinar el algoritmo para convertir un string de consulta SPARQL* en un calce de variables a valores de un grafo Tinkerpop. A grandes rasgos, las operaciones son las siguientes:

1. Obtener el input usuario, que especifica un grafo Tinkerpop y una consulta SPARQL*
2. Traducir la consulta SPARQL* a SPARQL mediante la expansión de cada triple entre `<< >>` y sus triples anidados
3. Comprobar que la consulta SPARQL obtenida cumpla con la sintaxis SPARQL
4. Asignar tipos a cada variable
5. Recorrer el árbol de operaciones dentro de WHERE y generar un conjunto de *traversals* a calzar con `match`
6. Agregar al *traversal* generado todas las operaciones de proyección especificadas en la consulta.

7. Ejecutar el *traversal* y obtener el output con los calces generados.

3.7. Input

El programa debe recibir una consulta en SPARQL* y un grafo Tinkerpop sobre el cual ejecutarse. El formato en que se entregan ambos datos se especificará en la sección 4.2.2

3.8. Traducción SPARQL* a SPARQL

Una vez obtenido el string de consulta y determinado sobre que grafo se va a ejecutar, corresponde traducir a SPARQL la consulta. Si esta no cuenta con los delimitadores << >>, se asume que no es SPARQL* y este paso se salta.

3.8.1. Justificación

Como se va a utilizar JENA para el parsing de la consulta SPARQL, el procedimiento general sería hacer que la gramática SPARQL* también sea entendida por JENA, pudiendo armar el árbol de operaciones de la misma forma que lo hace con consultas puras. Para esto es necesario extender JENA, ya que no provee una forma directa a través de su API de agregar nuevas gramáticas además de las ya existentes 1.0 y 1.1, especificadas en [12] y [10] respectivamente¹¹. Luego de analizar el funcionamiento de JENA ARQ, especialmente los paquetes `org.apache.jena.query` y `org.apache.jena.sparql.lang`, se pudo determinar que la extensión no es directa y requiere de un estudio amplio del funcionamiento de todas las clases subyacente. Dada la complejidad de esto, se propone una solución mas ad-hoc y se sugiere esto para trabajo futuro.

3.8.2. Gramática y traducción

A continuación se muestra la gramática SPARQL* permitida por el programa, que es la formalización de lo expuesto en la sección 3.4.1. Se utiliza para esto la notación EBNF, también usada para expresar la gramática de SPARQL [6]. Los símbolos no terminales utilizados aquí no coinciden con las variables utilizadas en el código, pero generan la misma gramática.

¹¹JENA también soporta la sintaxis propia ARQ (que agrega funcionalidades extra a SPARQL) y el uso directo del álgebra SPARQL

```
SPARQL_STAR ::= NPV | NEN
NPV ::= ONE_NPV | MANY_NPV
NEN ::= ONE_NEN | MANY_NEN
NPV_MAIN_TRIPLE ::= '<<' VAR N_PROP (VALUE | VAR) '>>'
NESTED_NPV ::= META (VALUE | VAR)
ONE_NPV ::= NPV_MAIN_TRIPLE '.'
MANY_NPV ::= NPV_MAIN_TRIPLE ';' ( NESTED_NPV ';' )* NESTED_NPV '.'
NEN_MAIN_TRIPLE ::= '<<' VAR 'e:to' VAR '>>'
NESTED_NEN ::= ( 'e:id' | 'e:label' | E_PROP ) (VALUE | VAR)
ONE_NEN ::= NEN_MAIN_TRIPLE '.'
MANY_NEN ::= NEN_MAIN_TRIPLE ';' ( NESTED_NEN ';' )* NESTED_NEN '.'
```

El significado de los no terminales sin definición es:

- VAR es cualquier variable SPARQL
- N_PROP es cualquier string np:property donde property es el nombre de alguna propiedad de nodo
- VALUE es cualquier valor numérico o string entre comillas simple
- META es cualquier string meta:property donde property es el nombre de alguna metapropiedad
- E_PROP es cualquier string ep:property donde property es el nombre de alguna propiedad de arco

El punto de entrada es SPARQL_STAR, que puede situarse dentro de una consulta SPARQL en cualquier lugar en donde pueda haber varios triples, dado que el resultado de la traducción es también un string de triples correlativos.

Una vez que se comprueba que hay un string que cumple con SPARQL_STAR dentro de la consulta, se puede obviar que es cada elemento y se traduce directamente, según sea NPV o NEN. Si el string

<< A B C >> ; M1 V1 ; M2 V2 ; ... ; MN VN .

es un NPV, la traducción es

A B X . X n:value C . X M1 V1 . X M2 V2 X MN VN .

y si es un NEN, queda

A e:out X . X e:in C . X M1 V1 . X M2 V2 X MN VN .

X es la variable aleatoria única generada. Lógicamente, si no hay triples anidados, no se considera esa parte de la traducción.

No se enfatizó en el uso de espacios blancos entre elementos de la gramática por razones de orden y claridad. Estos detalles son abordados directamente en el código.

3.8.3. Algoritmo de traducción

El Algoritmo 1 muestra a grandes rasgos la traducción de la consulta completa.

Algoritmo 1: Traducción de consulta SPARQL* a SPARQL

Input: $q \leftarrow$ consulta SPARQL*

Output: q traducida a SPARQL

```

1  $npv \leftarrow$  expresión regular para calzar Nodo Propiedad Valor y sus anidados;
2  $nen \leftarrow$  expresión regular para calzar Nodo Arco Nodo y sus anidados;
3  $calces \leftarrow []$ ;
4 Buscar calces con  $npv$  en  $q$  y agregar a  $calces$ ;
5 Buscar calces con  $nen$  en  $q$  y agregar a  $calces$ ;
6  $substrings \leftarrow []$ ;
7 foreach  $calce \in calces$  do
8    $i \leftarrow$  índice inicial de  $calce$  en  $q$ ;
9    $f \leftarrow$  índice final de  $calce$  en  $q$ ;
10  Agregar a  $substrings$  la tupla  $(calce, i, f)$ ;
11 foreach  $tupla \in substrings$  do
12    $main \leftarrow$  triple entre  $\langle \langle \rangle \rangle$  de  $tupla.calce$ ;
13    $nested \leftarrow$  triples anidados en  $tupla.calce$ ;
14    $var \leftarrow$  nombre de variable aleatoria único;
15    $main\_transalte \leftarrow$  traducción de  $main$  utilizando  $var$ ;
16    $nested\_transalte \leftarrow$  traducción de cada triple en  $nested$  utilizando  $var$ ;
17    $translate \leftarrow$  concatenación de  $main\_transalte$  y todos los strings en
     $nested\_transalte$ ;
18   Insertar  $translate$  en  $q$  entre  $tupla.i$  y  $tupla.f$ ;
19 return  $q$ ;
```

En la línea 14 se generan los nombres aleatorios de las variables usadas para expandir las expresiones. Notar que no se crea un nuevo string de consulta sino que las expansiones se insertan dentro del mismo. El string final no contendrá $\langle \langle \rangle \rangle$ y será SPARQL puro.

3.9. Comprobación de sintaxis SPARQL

Una vez se obtiene la consulta, debe comprobarse que cumple con la sintaxis SPARQL. Esto es sencillo utilizando Jena, ya que provee un método para transformar un string de consulta en álgebra SPARQL que posteriormente puede recorrerse. Si este paso falla, se reporta el posible problema encontrado por Jena y el programa se detiene. Si no, guarda el álgebra generado y continua.

3.10. Asignar tipos a cada variable

Este paso surge de la necesidad de disminuir la complejidad de la ejecución del traversal Gremlin final, evitar errores en tiempo de ejecución y disminuir también la dificultad de escribir una traducción a Gremlin que sea altamente general.

3.10.1. Idea general

Las variables en el modelo propuesto se pueden referir a nodos, arcos, propiedades, meta-propiedades, valores y a la relación entre estos elementos. Esto hace que triples como `?x ?y 1` puedan significar muchas cosas:

- Un nodo tiene un id 1
- Un nodo tiene un label 1
- Un arco tiene una propiedad que vale 1
- Un arco tiene un id 1
- Un arco tiene un label 1
- Una propiedad de nodo tiene un valor 1
- Una propiedad tiene una metapropiedad que vale 1

Calzar todas esas opciones, utilizando las mismas variables, sin que falle, es muy difícil por como funciona Gremlin por debajo. Por ejemplo, no todos los pasos que provienen de una propiedad pueden ser aplicados a un arco. Si una variable estuviera ligada a arcos y se avanza por un paso que solo es de propiedades, generaría error. En general, sin saber a priori que es cada variable no se pueden lograr calces. La naturaleza del lenguaje (componer pasos) permite al usuario saber que es lo que espera encontrar en cada paso y por ende esta característica no es un problema cuando el lenguaje se utiliza normalmente.

Para lograr ese nivel de generalidad en Gremlin, habría que lanzar muchos *traversals* distintos en donde cada uno asumiera que la variable es de un determinado tipo. Luego ejecutarlos todos e ignorar los que fallen. A eso sumarle la complejidad de todas las posibles combinaciones de tipos si el número de variables aumenta. Esta solución tiene una complejidad de ejecución muy alta y esta muy lejos del alcance de este proyecto.

El objetivo de este paso en el algoritmo es reducir lo más posible la generalidad del significado de cada variable, para lograr en la medida que sea posible una traducción uno a uno de un triple a un *traversal*. Esto se logra identificando el rol de cada variable analizando su función en el triple y las URIs/variables que la acompañan.

Como ejemplo práctico, imaginar el patrón `?x n:id 1 . ?x ?y 'person'`. Se puede determinar claramente que `?x` liga nodos ya que actúa como sujeto en un triple donde el predicado es `n:id`, que solo se usa entre nodos y literales. Luego de descubrir esto, se puede saber que `?y` es `n:id` o `n:label`, ya que solo estos predicados están entre nodos y valores. Este análisis permitió disminuir el alcance de las variables, y así facilitar la traducción y ejecución posterior.

El hecho de descubrir el tipo de una variable puede significar que otra variable puede ser descubierta posteriormente, por lo que es un proceso repetitivo hasta que no se puedan encontrar más tipos.

Este ejemplo también sirve para ver la utilidad de usar predicados específicos, como `n:id`. Si para especificar ids se hubiera optado por algo general para uso en nodos y arcos como por ejemplo `preds:id`, en la consulta anterior no se habría podido determinar que `?x` era un nodo, ya que este predicado podría usarse en nodos y arcos. Esta característica de diseño ayuda considerablemente en la traducción.

3.10.2. Consideraciones respecto al modelo

Existen varias consideraciones a tomar en cuenta para la búsqueda de los tipos de cada variable, que surgen de la forma en que se definió el modelo:

- Los sujetos de todos los triples son siempre variables, que ligan a nodos, arcos o propiedades de nodo, ya que todas las relaciones parten con un nodo vacío.
- Los objetos son literales que representan un valor o son variables que ligan a nodos, arcos, propiedades o literales.
- Los predicados son variables o URIs que representan una relación.
- Si el predicado es una URI, entonces siempre se puede saber el tipo la variable del sujeto, y de objeto si lo fuera. Esto se debe a que cada uno es único y solo puede utilizarse entre un determinado par de tipos de sujeto y objeto.

- Los únicos tipos de predicados que pueden ser descubiertos son `e:out`, `e:in` y `np:some_property` porque solo estos se pueden determinar inequívocamente dado el par de tipos de sujeto y objeto, que son nodo-arco, arco-nodo y nodo-propiedad, respectivamente.
- Si se sabe el tipo del sujeto y el predicado, y estos son `nodo-np:some_property`, `nodo-e:out` o `arco-e:in`, entonces se puede determinar el tipo del objeto, siendo propiedad, arco o nodo respectivamente.
- El proceso inverso a anterior, es decir, saber solo el tipo del objeto, también funciona para esos casos.

3.10.3. Algoritmo

El Algoritmo 2 es el encargado de iterar sobre todos los triples dentro de la consulta, asignando tipos a las variables, y repitiendo el proceso cada vez que una nueva variable se descubra.

El caso de la línea 11 es cuando se tiene variable en el sujeto y predicado. En este caso no se puede obtener un tipo para estos, dado que el literal del objeto no entrega información. Para los demás, se intenta asignar tipos con tres funciones¹²

La función `AsignarVV` en el Algoritmo 3 intenta buscar los tipos cuando el triple tiene 3 variables, consultado lo que ya se sabe respecto a esas variables. Hay casos, como en la línea 4 y 17 que falta información para determinar el tipo. Como se mencionó en la sección 3.10.2, hay casos en donde siempre se puede determinar el tipo (líneas 10 y 13) y otros casos que depende si es posible, según la estructura del modelo, determinar inequívocamente el tipo.

La función `AsignarUV` en el Algoritmo 4 es para cuando hay variables en el sujeto y objeto. Dado que se *p* es una URI, los tipos de las otras dos variables siempre se pueden determinar.

Finalmente la función `AsignarUU` en el Algoritmo 5 es para cuando solo esta presente la variable del sujeto. En este caso también se puede determinar siempre.

Al finalizar el algoritmo se obtiene un mapeo entre nombres de variables y tipos, que será utilizado posteriormente para la traducción. Los detalles del descubrimiento de tipos se abordaran en el código.

3.11. Recorrer árbol y generar `match`

Ya con la consulta SPARQL traducida y los tipos de las variables obtenidos, se pasa al proceso central de algoritmo, que es la traducción de la consulta SPARQL a un traversal Gremlin.

¹²Las siglas significan VV = variable-variable, UV = uri-variable, UU = uri-uri. Estas se refieren al predicado y objeto, ya que se sabe que el sujeto es siempre una variable.

Algoritmo 2: Obtención de tipos de cada variable

Input: $q \leftarrow$ consulta SPARQL

Output: Mapeo de nombres de variables a tipos

```
1  $triples \leftarrow$  obtener todos los triples en  $q$ , en cualquiera de sus componentes;
2 do
3    $continuar \leftarrow$  false;
4   foreach  $triple \in triples$  do
5      $s \leftarrow$  sujeto de  $triple$ ;
6      $p \leftarrow$  predicado de  $triple$ ;
7      $o \leftarrow$  objeto de  $triple$ ;
8     if  $p$  es variable then
9       if  $o$  es variable then
10         $continuar \leftarrow$   $continuar$  OR AsignarVV( $s, p, o$ );
11      else
12        /* No es posible */
13      else
14        if  $o$  es variable then
15           $continuar \leftarrow$   $continuar$  OR AsignarUV( $s, p, o$ ) ;
16        else
17           $continuar \leftarrow$   $continuar$  OR AsignarUU( $s, p$ ) ;
18 while  $continuar$ ;
19 return;
```

Algoritmo 3: AsignarVV

Output: Sí se asignó o no un tipo a una o más variables

```
1 Funcion AsignarVV(s, p, o)
2   if No conozco s then
3     if No conozco p then
4       if No conozco o then                                     */
5         /* No se puede asignar nada
6       else
7         Asignar tipos a s y p con o, si se puede;
8         if Se pudo then return true;
9     else
10      if No conozco o then
11        Asignar tipos a s y o con p;
12        return true;
13      else
14        Asignar tipo a s con p;
15        return true;
16  else
17    if No conozco p then
18      if No conozco o then                                     */
19        /* No se puede asignar nada
20      else
21        Asignar tipo a p con s y o, si se puede;
22        if Se pudo then return true;
23    else
24      if No conozco o then
25        Asignar tipo a o con s y p, si se puede;
26        if Se pudo then return true;
27      else
28        /* Conozco todo                                     */
29    return false;
```

Algoritmo 4: AsignarUV

Output: Sí se asignó o no un tipo a una o más variables

```
1 Funcion AsignarUV(s, p, o)
2   if No conozco s then
3     if No conozco o then
4       Asignar tipos a s y o con p;
5       return true;
6     else
7       Asignar tipo a s con p;
8       return true;
9   else
10    if No conozco o then
11      Asignar tipo a o con p;
12      return true;
13    else
14      /* Conozco todo                               */
15  return false;
```

Algoritmo 5: AsignarUU

Output: Sí se asignó o no un tipo a una o más variables

```
1 Funcion AsignarUU(s, p)
2   if No conozco s then
3     Asignar tipo a s con p;
4     return true;
5   return false;
```

Valiéndose de las herramientas que provee JENA, se puede obtener un árbol de álgebra SPARQL a partir de una consulta. Este árbol puede recorrerse en postorden y pueden ejecutarse operaciones en cada nodo, lo cual es muy conveniente para la traducción.

3.11.1. Operaciones soportadas

Para esta versión del trabajo, se soportan las operaciones SELECT, WHERE, UNION, OPTIONAL y FILTER. La gramática de consulta permitida es la siguiente:

```
QUERY ::= REGULAR_QUERY | UNION_QUERY
REGULAR_QUERY ::= 'SELECT' VARS 'WHERE' WHERE_BODY
WHERE_BODY ::= '{' BGP (FILTER | OPTIONAL)* '}'
FILTER ::= 'FILTER {' CONDITIONS '}'
OPTIONAL ::= 'OPTIONAL {' BGP_NO_PRED_VAR '}'
UNION_QUERY ::= 'SELECT * WHERE {' WHERE_BODY 'UNION' WHERE_BODY '}'
```

VARS es cualquier número de variables en formato SPARQL (?x) o un asterisco. BGP es la traducción a SPARQL de cualquier PGB en SPARQL* como los que cumple con las reglas de la sección 3.8.2. CONDITIONS es cualquier expresión lógica con operadores =, !=, >=, <=, > y < sobre variables y literales. BGP_NO_PRED_VAR es cualquier PGB sin variable en los predicados. Notar que UNION solo se soporta con selección de todas las variables.

3.11.2. Traducción de operaciones SPARQL a Gremlin

El algoritmo general se muestra en 6. Este es el punto de entrada del proceso de traducción, en donde principalmente ocurre lo siguiente (líneas indicadas entre paréntesis):

1. Se comienza el *traversal* obteniendo todos los nodos del grafo (2)
2. El proceso se separa en caso de que exista o no UNION en la consulta. La solución presentada en este trabajo de título solo contempla el uso de UNION en el nivel superior de la consulta y solo una vez
3. La función `getTraversals(q)` (6, 7, 10) toma una consulta SPARQL que contenga triples, operaciones WHERE y/o operaciones OPTIONAL, y retorna una lista de *traversals* Gremlin que son recibidos por la función `match`. El proceso de traducción ocurre allí dentro
4. La función `union` (8) de Gremlin es la encargada de unir los resultados de las dos partes de la consulta UNION SPARQL. Si no está presente, simplemente se concatena la función `match`. Esta operación sigue el concepto de union de SQL. En este caso el resultado

de la consulta de la izquierda y se une al de la derecha. Aún cuando una variable aparezca en ambas sub-consultas y sus resultados independientes no son compatibles, se entregan todos los resultados. En otras palabras, no se hace join de los resultados de ambas sub-consultas, solo se unen

5. Se retorna el *traversal* armado

Algoritmo 6: Convertir SPARQL A Traversal Gremlin

Input: $q \leftarrow$ consulta SPARQL, $t \leftarrow$ mapeo de variables a tipos, $g \leftarrow$ grafo a consultar

Output: Traversal Gremlin obtenido de la traducción

```

1  $hay\_union \leftarrow$  Si hay o no presente una operación UNION en  $q$ ;
2  $traversal \leftarrow g.V()$ ;
3 if  $hay\_union$  then
4    $left \leftarrow$  Obtener la sub-consulta de la izquierda de UNION en  $q$ ;
5    $right \leftarrow$  Obtener la sub-consulta de la derecha de UNION en  $q$ ;
6    $ltraversals \leftarrow$  getTraversals( $left$ );
7    $rtraversals \leftarrow$  getTraversals( $right$ );
8    $traversal \leftarrow traversal.union(\_\_.match(ltraversals), \_\_.match(rtraversals))$ 
9 else
10   $traversals \leftarrow$  getTraversals( $q$ );
11   $traversal \leftarrow traversal.match(traversals)$ 
12 return  $traversal$ ;

```

El Algoritmo 7 recorre el árbol de forma recursiva y comienza el proceso de traducción local dependiendo de que sub-consulta se está visitando. Estas sub-consultas son expresadas en álgebra SPARQL de 3 formas:

1. **BGP:** Es igual a nuestra definición de patrón de grafo básico de la sección 2.1.3 . Define un conjunto de triples con variables. Por ejemplo la traducción de SPARQL a álgebra de la consulta

```

SELECT ?s ?o
  WHERE { ?s ?p ?o }

```

es

```

(base <http://example/base/>
  (project (?s ?o)
    (bgp (triple ?s ?p ?o))
  )
)

```

En donde `<http://example/base/>` es la URI base de la consulta, `project` es la selección y `bgp` el PGB en cuestión, con un triple dentro. Este árbol y sus operadores son los que utiliza JENA para recorrer desde las hojas recursivamente la consulta, que en este paso ya no existe como tal, sino que se usa la estructura algebraica.

2. **OPTIONAL**: Operación que sigue el concepto de *left-join* de SQL. En este caso el resultado de la sub-consulta de la derecha es adicionado a la de la izquierda, aun cuando no se obtengan resultados de la consulta de la derecha. Por ejemplo la consulta

```
{ ?s :p1 ?v1
  OPTIONAL {?s :p2 ?v2 }
}
```

se traduce en álgebra como

```
leftjoin(
  bgp(triple(?s :p1 ?v1)),
  bgp(triple(?s :p2 ?v2)),
  true
)
```

Aquí `leftjoin` es el **OPTIONAL**.

3. **FILTER**: Operación de filtrado que sigue el mismo concepto que el **WHERE** de SQL. Especifica condiciones binarias que deben cumplir los resultados de la consulta inmediatamente anterior. Si no cumplen las condiciones, son descartados. En álgebra SPARQL la consulta

```
{ ?s :p1 ?v1 } FILTER ( ?v1 < 3 )
```

queda

```
filter( ?v1 < 3 , bgp( triple(?s :p1 ?v1)))
```

El álgebra SPARQL es un tema amplio y con mucha fundamentación matemática. No es necesario indagar en este tema más allá de lo mencionado.

El algoritmo 7 `getTraversals` toma la consulta q y desanida todas las sub-consultas que tenga, creando una lista. Se itera sobre esta lista y se traduce cada sub-consulta dependiendo de cual se trata.

Es importante el caso de la línea 5 ya que, como se mencionó anteriormente, el recorrido es en post-orden, por lo que la detección del PGB es previa a la detección de la operación

algebraica *left-join* o *where*. Si el PGB no pertenece a estas estructuras, entonces se procesa como un conjunto de triples a calzar dentro del grafo. En la línea 9 está el caso *OPTIONAL* que será profundizado luego, y en la 11 el caso *FILTER*, en donde se utiliza la función de Gremlin *where* para filtrar los resultados que no cumplen con las condiciones. Cada condición expresada en SPARQL es traducida a un traversal Gremlin y agregada a una lista, para luego esta ser pasada a *where*¹³. Las traducciones específicas se listan en las siguientes secciones.

Algoritmo 7: getTraversals

Input: $q \leftarrow$ consulta

Output: Lista de traversals traducidos

```

1 Funcion getTraversals( $q$ )
2    $qs \leftarrow$  lista de sub-consultas en  $q$ ;
3    $traversals \leftarrow$  lista vacia;
4   foreach  $q \in qs$  do
5     if  $q$  es un PGB que no pertenece a un OPTIONAL o FILTER then
6       foreach  $triple \in q$  do
7          $traversal \leftarrow$  traducción de  $triple$  según algoritmo indicado en Tabla 10;
8         Agregar  $traversal$  a  $traversals$ ;
9     else if  $q$  es un OPTIONAL then
10      Agregar getOptional( $q$ ) a  $traversals$ ;
11    else
12      foreach  $expresion \in q$  do
13         $traversal \leftarrow$  Traducción de  $expresion$  según Tabla 20 ;
14         $traversal \leftarrow \_\_.where(traversal)$ ;
15        Agregar  $traversal$  a  $traversals$ ;
16  return  $traversals$ ;

```

3.11.3. Traducción de PGB

Para traducir un PGB, se debe traducir individualmente cada uno de sus triples a traversals, dependiendo de sus estructuras y de los tipos de cada uno. En la Tabla 10 se muestran las combinaciones posibles entre variables y tipos, y el algoritmo aplicado. Los casos marcados con * no fueron desarrollados dado su excesiva generalidad.

En la Tabla 10 se indican los métodos a utilizar en caso de que se sepan o no los tipos de las variables. Recordar que el sujeto siempre es una variable y que si el predicado es una URI, se puede saber siempre el tipo de las demás variables del triple. Los algoritmos de traducción se comportan según lo indicado en las Tablas 11, 12, 13, 14, 16 y 17.

¹³Gremlin automáticamente incluye las condiciones de *where* dentro del *match* concatenado inmediatamente antes. Este comportamiento es muy conveniente para el modelo propuesto.

Tabla 10: Posibles combinaciones entre variables, tipos conocidos y algoritmo de traducción a usar.

Fuente: Elaboración propia.

Predicado	Objeto	Tipo Sujeto	Tipo Predicado	Tipo Objeto	Algoritmo
VAR	VAR	NO	NO	NO	*
VAR	VAR	NO	NO	SI	*
VAR	VAR	NO	SI	NO	getVVVFromPType
VAR	VAR	NO	SI	SI	getVVVFromPType
VAR	VAR	SI	NO	NO	*
VAR	VAR	SI	NO	SI	getVVVFromSOTypes
VAR	VAR	SI	SI	NO	getVVVFromPType
VAR	VAR	SI	SI	SI	getVVVFromPType
VAR	URI	NO	NO	-	getVVUFromO
VAR	URI	NO	SI	-	getVVUFromPType
VAR	URI	SI	NO	-	getVVUFromSType
VAR	URI	SI	SI	-	getVVUFromPType
URI	VAR	SI	-	SI	getVUVFromPType
URI	URI	SI	-	-	getVUUFromPType

En la Tabla 11 se muestran los casos en donde no se sabe que es el predicado, pero si lo demás. En algunos casos se usa `union` porque existe más de un arco posible entre los nodos correspondientes. Por ejemplo, en el modelo, el arco RDF entre un nodo Tinkerpop y un valor Tinkerpop puede ser `n:id` o `n:label`. Es por eso que se requiere unir los resultados.

Tabla 11: Algoritmo de traducción `getVVVFromSOTypes`

Fuente: Elaboración propia.

Sujeto	Objeto	Traducción
Nodo	Arco	<code>__.as(s).outE().as(o)</code>
Nodo	Propiedad	<code>__.as(s).properties().as(o)</code>
Nodo	Valor	<code>__.as(s).union(__.label(), __.id()).as(o)</code>
Arco	Nodo	<code>__.as(s).inV().as(o)</code>
Arco	Valor	<code>__.as(s).union(__.properties().value(), __.label(), __.id()).as(o)</code>
Propiedad	Valor	<code>__.as(s).union(__.value(), __.properties().value()).as(o)</code>

En todos los casos, `s` y `o` representan las variables del sujeto y el objeto respectivamente.

En el caso de la Tabla 12, solo se sabe el tipo del predicado. En casi todos los casos, saber esto es sinónimo de saber exactamente el valor, ya que el arco RDF solo puede tomar valores fijos según el modelo. En el caso de las propiedades y metapropiedades, el valor del arco RDF entrega mayor información ya que no es fijo. En el caso de `np`, se toman todas las propiedades. En los dos otros casos, debido a que el modelo no soporta enlazar variables a

propiedades de arco o metapropiedades, se toman todos los valores.

Tabla 12: Algoritmo de traducción getVVVFromPType
Fuente: Elaboración propia.

Tipo Predicado	Traducción
n:value	__.as(s).value().as(o)
meta:?	__.as(s).values().as(o)
n:id	__.as(s).id().as(o)
e:id	__.as(s).label().as(o)
n:label	__.as(s).label().as(o)
e:label	__.as(s).label().as(o)
np:?	__.as(s).properties().as(o)
ep:?	__.as(s).values().as(o)
e:in	__.as(s).inV().as(o)
e:out	__.as(s).outE().as(o)

En la Tabla 13, el predicado es una URI, por lo que ahora se puede extraer el valor de la propiedad (o la propiedad misma, en el caso de np) en cuestión y ligarse a la variable del objeto.

Tabla 13: Algoritmo de traducción getVUVFromPType
Fuente: Elaboración propia

Tipo Predicado	Traducción
n:value	__.as(s).value().as(o)
meta:property	__.as(s).values("property").as(o)
n:id	__.as(s).id().as(o)
e:id	__.as(s).id().as(o)
n:label	__.as(s).label().as(o)
e:label	__.as(s).label().as(o)
np:property	__.as(s).properties("property").as(o)
ep:property	__.as(s).values("property").as(o)
e:in	__.as(s).inV().as(o)
e:out	__.as(s).outE().as(o)

En la Tabla 14 también se conoce el valor del predicado, pero esta vez el objeto también es una URI, por lo que no todos los casos son posibles. Por ejemplo es imposible que, si el objeto es una URI, el predicado sea e:in, e:out o np: ? ya que estos solo se usan entre nodos RDF vacíos.

Tabla 14: Algoritmo de traducción getVUUFromPType

Fuente: Elaboración propia

Tipo Predicado	Traducción
n:value	__.as(s).hasValue(o)
meta:property	__.as(s).values("property").is(o)
n:id	__.as(s).hasId(o)
e:id	__.as(s).hasId(o)
n:label	__.as(s).hasLabel(o)
e:label	__.as(s).hasLabel(o)
ep:property	__.as(s).values("property").is(o)

En la Tabla 15 se muestra el caso en que solo se tiene el objeto. Se usa `or` dado que no se sabe que elemento esta actualmente ligado al sujeto.

Tabla 15: Algoritmo de traducción getVVVFromO

Fuente: Elaboración propia

Traducción
__.as(s).or(__.hasId(o), __.hasLabel(o), __.properties().value().is(o), __.hasValue(o))

En la Tabla 16 solo se sabe el tipo del sujeto, pero a diferencia del caso de la Tabla 11, el objeto es una URI, por lo que este paso debe ser de filtrado (no se busca ligar), y se debe usar `or` en vez de `union`.

Tabla 16: Algoritmo de traducción getVVUFromSType

Fuente: Elaboración propia

Tipo Sujeto	Traducción
Nodo	__.as(s).or(__.hasLabel(o), __.hasId(o))
Arco	__.as(s).or(__.hasLabel(o), __.hasId(o), __.properties().value().is(o))
Propiedad	__.as(s).or(__.hasValue(o), __.properties().value().is(o))

Finalmente en la Tabla 17 se usa el tipo del predicado para filtrar los objetos ligados al sujeto según el valor del objeto. Nuevamente no es posible que el predicado tenga los tipos que van entre dos nodos vacíos (`e:in`, `e:out` y `np:property`).

Tabla 17: Algoritmo de traducción getVVUFromPType

Fuente: Elaboración propia

Tipo Sujeto	Traducción
n:value	__.as(s).hasValue(o)
meta:property	__.as(s).values().is(o)
n:id	__.as(s).hasId(o)
e:id	__.as(s).hasId(o)
n:label	__.as(s).hasLabel(o)
e:label	__.as(s).hasLabel(o)
ep:property	__.as(s).values().is(o)

Con todo esto ya queda lista la traducción de un BGP en la consulta.

3.11.4. Traducción de Optional

El Algoritmo 7 hace uso en la línea 10 de `getOptional`, especificado en el Algoritmo 8. La forma en que se simuló el comportamiento del OPTIONAL de SPARQL en Gremlin es la siguiente:

1. El PGB dentro de OPTIONAL no liga variables, con excepción de la variable del objeto del último triple
2. Si se encuentra un calce en la variable del final, se liga la variable final a lo encontrado, si no, se liga a un String vacío
3. Las variables usadas dentro del PGB solo sirven para avanzar en el grafo hasta el valor opcional deseado
4. Los predicados no pueden ser variables

En la Tabla 18 se ve el paso que se concatena por cada predicado de la consulta. Esta Tabla muestra las traducciones de cada triple en la línea 5 y 6 del algoritmo. Finalmente en la línea 8 se genera el *traversal* que se retorna. Notar que usa `choose`, que toma el *traversal* del primer argumento y si retorna algo, se propaga el segundo argumento, sino el tercero. En este caso se propaga el mismo *traversal* y luego el resultado se liga a la variable del objeto. La función `constant` se utiliza para emitir un valor constante y agregarlo al flujo de datos. En este caso se emite un string vacío.

Algoritmo 8: getOptional

Input: $q \leftarrow$ consulta OPTIONAL

Output: Traversal resultante de traducción de OPTIONAL

```

1 Funcion getOptional( $q$ )
2    $s \leftarrow$  sujeto del primer triple;
3    $traversal \leftarrow$  traversal vacío ;
4   foreach  $triple \in q$  do
5     | Concatenar a  $traversal$  la traducción de  $triple$  según lo indicado en Tabla 18;
6    $o \leftarrow$  objeto de último triple;
7    $result \leftarrow \_\_.as(s).choose(traversal, traversal, \_\_.constant()).as(o);$ 
8   return  $result$ ;

```

Tabla 18: Concatenación para triples Optional
Fuente: Elaboración propia

Predicado	Traducción
n:value	.value()
np:property	.properties("property")
meta:property	.properties("property")
n:id	.id()
n:label	.label()
e:in	.inV()
e:out	.outE()
e:id	.id()
e:label	.label()

La consulta de ejemplo del Código 28 muestra una consulta SPARQL con OPTIONAL. En este caso se pretende ligar a ?y los años desde que las personas ligadas a ?x desarrollan un software, de existir.

```

1 g.V().match(
2   __.as('x').label('person').as('name'),
3   __.as('x').property('name').as('p1'),
4   __.as('p1').value().as('name'),
5   __.as('x').choose(__.outE().properties('since'),
6     __.outE().properties('since'),
7     __.constant(''))
8   ).as('y'),
9 ).select('name', 'software_name')

```

Código 27: Consulta SPARQL con OPTIONAL traducida a Gremlin

La traducción a Gremlin está en el Código 27. El resultado sobre el grafo que hemos estado

utilizando como ejemplo (Figura 7) aparece en la Tabla 19. En el caso de `daniel`, como no ha desarrollado ningún software, no aparece resultado. Esto se logra con el `OPTIONAL`, debido a que si no hubiera sido opcional, `daniel` tampoco habría aparecido en la respuesta, porque habría sido filtrado por no cumplir con un patrón.

```

1 SELECT ?name ?since WHERE {
2   ?x n:label "person" .
3   << ?x np:name ?name >> .
4   OPTIONAL {
5     ?x e:out ?e .
6     ?e ep:since ?y .
7   }
8 }
```

Código 28: Consulta SPARQL con `OPTIONAL`

Tabla 19: Resultado de consulta con `OPTIONAL`
Fuente: Elaboración propia

name	y
marko	2009
marko	2010
stephen	2010
stephen	2011
mathias	2012
daniel	

3.11.5. Traducción de `FILTER`

Finalmente, si existe un `FILTER` en la consulta, el Algoritmo 7 traduce cada expresión de este según la Tabla 20. Esta traducción deja de manifiesto que ambos lenguajes, a pesar de sus diferencias, comparten las mismas operaciones, debido a que ambos trabajan sobre dominios iguales. Esto se ve reflejado en que la traducción de uno al otro es prácticamente directa en ese caso, utilizando los pasos lógicos y booleanos provistas por Gremlin.

Las últimas dos traducciones usan `translate`, que no es más que una forma sencilla de mostrar que ambas expresiones en los parámetros del `and` y `or` de SPARQL se traduce nuevamente según la misma tabla. No es una función como tal.

Cada uno de estos filtros es agregado al *traversal* a través de la función `where`.

Tabla 20: Traducción de expresiones FILTER

Fuente: Elaboración propia

Expresión	Traducción
?v = N	__.as(?v).is(eq(N))
?v != N	__.as(?v).is(neq(N))
?v < N	__.as(?v).is(lt(N))
?v <= N	__.as(?v).is(lte(N))
?v > N	__.as(?v).is(gt(N))
?v >= N	__.as(?v).is(gte(N))
E1 and E2	__.and(translate(E1), translate(E2))
E1 or E2	__.or(translate(E1), translate(E2))

3.12. Proyección

La operación de proyección es ejecutada por la función `select` utilizada en todas las traducciones anteriores.

3.13. Output

Finalmente, se ejecuta el traversal encadenado a través de toda la traducción. Si no hay problemas con la ejecución, se mostraran los resultados. Cada linea muestra los calces posibles de las variables seleccionadas.

Es necesario mencionar que el mapeo ideado permite la selección de variables que se refieren a elementos del PG, como el nodos o la propiedad en si, además de sus valores. La forma en que se mostrarán estos resultados será de la misma forma que los muestra Gremlin por defecto. Un resultado se muestra de la forma `x[y]`, donde `x` es `v`, `e`, `vp` o `p`, dependiendo de si es nodo, arco, propiedad de vértice o propiedad de arco (o metapropiedad) respectivamente. Entre corchetes aparece el id del elemento en caso de ser nodo o arco, o un string de la forma `'property' -> 'value'` indicando el nombre y valor de la propiedad, en caso de serlo.

Un ejemplo de output donde se calzaron las variables `x` y `d`, y esta última era opcional, podría ser:

```
{x=v[7] , d=2012}
{x=v[8] , d=2013}
{x=v[9] , d=}
```

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

Para demostrar que el diseño de la solución es factible y funciona, se implementará un software siguiéndolo.

En esta sección se menciona brevemente el ambiente de desarrollo, el diseño de software, como este se relaciona con el algoritmo y finalmente se muestran los test realizados.

4.1. Ambiente de desarrollo

Como el software fue desarrollado basándose en el proyecto Gremlin-SPARQL, se mantuvo la configuración del proyecto y se comenzó desde allí. La versión de Java del proyecto es 1.8. Se utilizó el IntelliJ IDEA como IDE para desarrollar. Se escogió por las siguientes razones:

- Integración nativa con proyectos Java basados en Maven, como el proyecto base que se utilizó
- Autocompletado de código rápido y preciso
- Completo análisis y refactorización de código
- Fácil definición de tareas para ejecutar comandos con parámetros
- Integración completa con toda la plataforma Java
- Amplia comunidad
- El autor lo ha usado en varios proyectos anteriormente

4.2. Diseño del software

Dado que la organización del código base no era suficiente para soportar las nuevas funcionalidades y modelo, se decidió cambiar completamente.

En general, cada uno de los pasos mencionados en la Propuesta está encapsulado dentro de clases o paquetes específicos, encargadas de solo una parte delimitada del proceso.

Muchos de los casos condicionales de los algoritmos de la Propuesta fueron abarcados a través de herencia, estableciendo comportamientos específicos en cada clase dependiendo

del caso, o a través de mapeos. Es por esto que la cantidad de condiciones del código es menor que en los algoritmos.

Se mostrará el diagrama de clases de cada paquete del software.

4.2.1. SPARQL2GREMLIN

Punto de entrada del programa.

4.2.2. com.datastax.sparql.io

Incluye las clases encargadas de la entrada y la salida del software. Su diagrama aparece en la Figura 10.

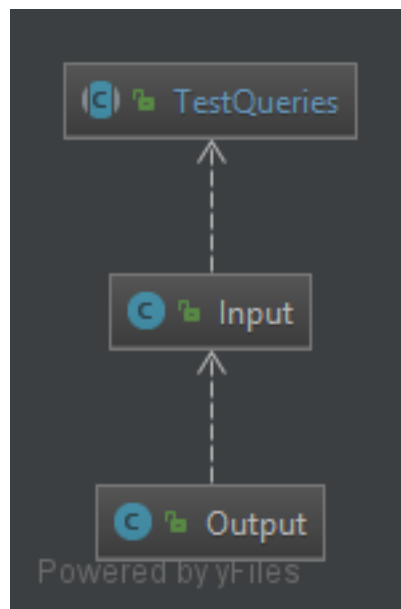


Figura 10: Paquete `com.datastax.sparql.io`
Fuente: Elaboración propia.

Input:

Selecciona el flujo del programa según los argumentos. Con `file` se especifica la ruta del archivo con la consulta SPARQL, que también puede ser pasada por línea de comandos. Con `graph` se especifica uno de los tres grafos que se incluyen en el framework (`classic`, `modern 6` y `crew 7`) o un archivo en formato `.kryo`¹⁴ que especifique el contenido del grafo, que es

¹⁴Para mayor información referirse a <https://github.com/EsotericSoftware/kryo>

parseado e instanciado a través de la API de Gremlin. Si se pasa test, se que el archivo con la consulta SPARQL contenga también el separador === y luego la respuesta esperada por el programa.

Si se utiliza la opción `verbose`, se muestra en el output final la consulta, la traducción si corresponde, el traversal Gremlin, y el resultado. Si no, solo el resultado es mostrado.

Si no se pasan parámetros, se usa el string en `TestQueries.test` como consulta y el grafo `crew`, y el modo `verbose`.

Output:

Ejecuta el *traversal* generado a través de la traducción. Imprime el resultado. Maneja los casos cuando se trata de una test (archivo con consulta y resultado) o una prueba de desarrollo.

TestQueries:

Clase con consultas de prueba. Están pensadas para consultar el grafo `crew`.

4.2.3. `com.datastax.sparql.star`

Incluye las clases encargadas de la traducción de la consulta SPARQL* a SPARQL. Su diagrama aparece en la Figura 11.

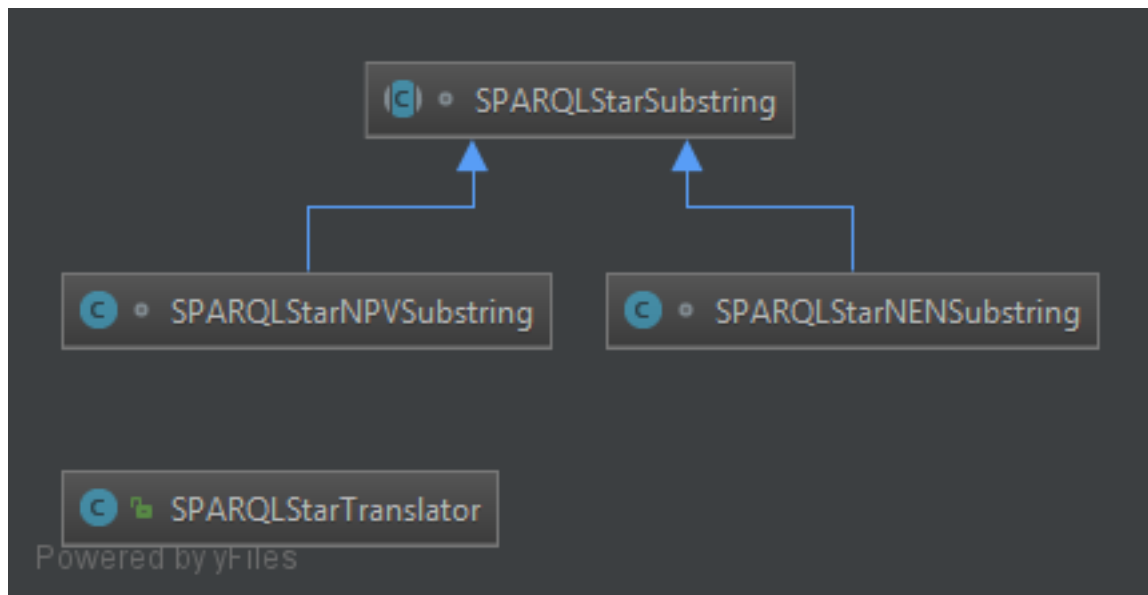


Figura 11: Paquete `com.datastax.sparql.star`
Fuente: Elaboración propia.

SPARQLStarTranslator:

Punto de entrada para la traducción. Mantiene expresiones regulares compiladas con `java.util.regex.Pattern` para asegurarse de que la sintaxis de la consulta cumple con lo establecido en 3.8.2 Gramática y Traducción.

En `translate` se obtienen todas las traducciones de cada triple SPARQL* y se reemplazan, sobre la consulta, por los strings de los cuales fueron obtenidas. Se retorna la consulta SPARQL.

SPARQLStarSubstring:

Representación de una composición de triples con sintaxis SPARQL*, determinados por el string en si y el punto de partida y final en la consulta original. Se instancia a partir del calce de las expresiones regulares que determinan esta composiciones, utilizando un `java.util.regex.Matcher`. Traduce según lo indicado en 3.8.3, especificando el comportamiento con dos subclases.

`replace` utiliza los índices para ir reemplazando los strings traducidos.

`getSPARQLTriples` hace la diferencia entre los triples delimitados con `<< >>` con triples anidados y los que no. Primero genera una variable nueva aleatoria con `com.datastax.sparql.gremlin.Randomizer`. Utilizando un `StringBuilder` va armando la traducción del string. Con `buildMainTriples` arma la traducción del triple entre `<< >>`, método que es sobrescrito por las subclases. Si además cuenta con triples anidados, los itera y los va traduciendo, extrayendo el predicado y objeto.

SPARQLStarNPVSubstring:

Subclase de `SPARQLStarSubstring`, que especifica la traducción del triple principal cuando es de tipo NPV.

SPARQLStarNENSubstring:

Subclase de `SPARQLStarSubstring`, que especifica la traducción del triple principal cuando es de tipo NEN.

4.2.4. com.datastax.sparql.gremlin

Contiene las clases encargadas de la construcción del traversal Gremlin a partir de la consulta. Su diagrama aparece en la Figure 12.

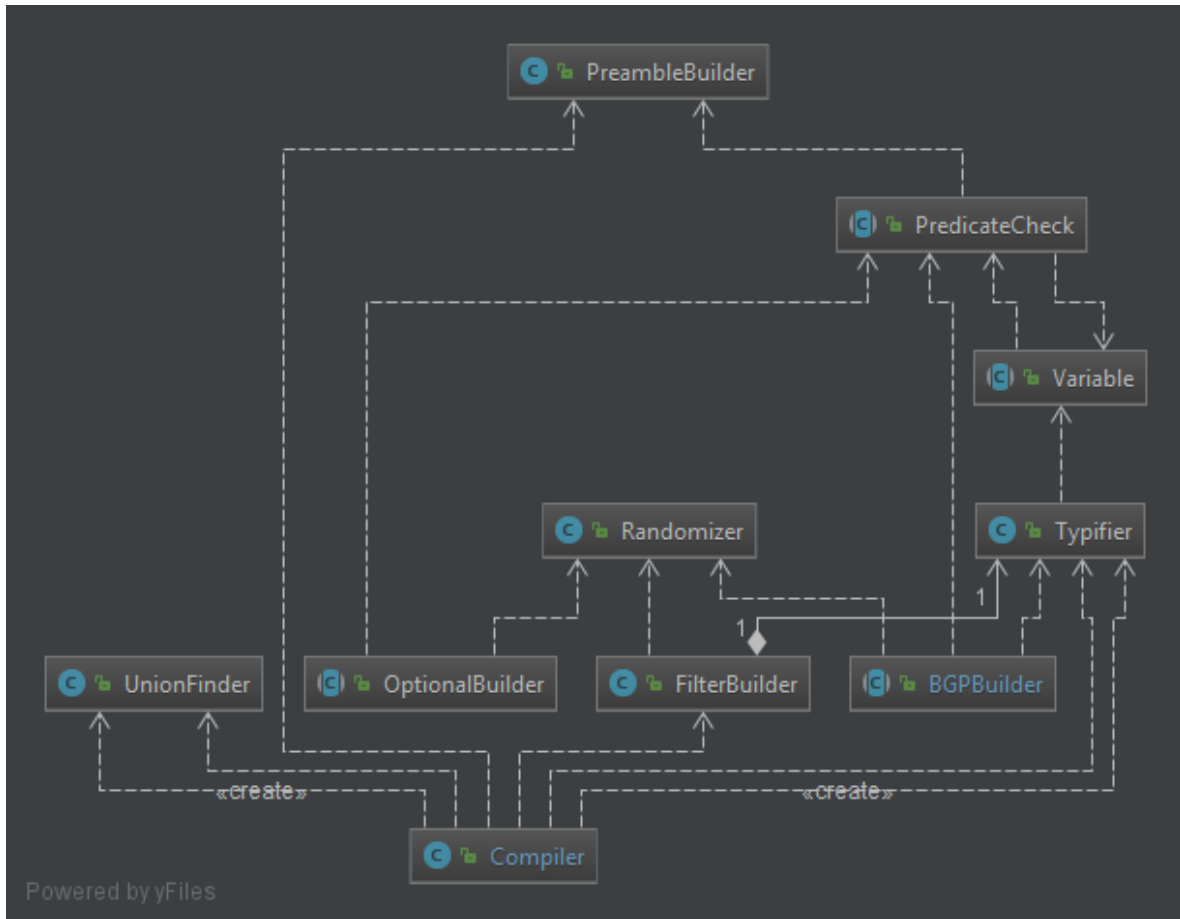


Figura 12: Paquete `com.datastax.sparql.gremlin`

Fuente: Elaboración propia.

Compiler:

Punto de entrada. Inicia el encadenamiento del traversal con `g.traversal().V()` y comprueba la sintaxis de la consulta SPARQL (3.9 Comprobación de sintaxis SPARQL) valiéndose de `org.apache.jena.query.QueryFactory.create`, que lanza excepción si no es una consulta válida.

`convertToGremlinTraversal` realiza el proceso de traducción según lo indicado en 3.11.2. Primero consigue los tipos con `Typifier`. Luego revisa si la consulta cuenta con UNION valiéndose de la estructura algebraica de la consulta, obtenida con `Algebra.compile`, que retorna un `org.apache.jena.sparql.algebra.Op`. Estas estructuras pueden recorrerse recursivamente con una clase que herede de `org.apache.jena.sparql.algebra.OpVisitorBase`, en donde se pueden sobrescribir los métodos `visit` con diferentes parámetros, cada uno representado una operación algebraica, para especificar un comportamiento en caso de visitar aquella operación durante el recorrido.

Se hace uso de la clase `OpsFinder`, subclase de `OpVisitorBase`, para implementar la lógica de la función `getTraversals` en 7

UnionFinder:

Subclase de `OpVisitorBase`, usada por `Compiler`, que se encarga de identificar la existencia de `UNION` en la consulta, por lo que solo sobrescribe `visit(OpUnion)`.

Typifier:

Clase encargada de obtener los tipos de las variables. Se instancia y guarda todos los triples de la consulta, para luego recorrerlos, según lo indicado en 3.10.3.

Los algoritmos `AsignarVV`, `AsignarUV` y `AsignarUU` son tienen sus equivalentes en esta clase, que se valen de los mapeos dentro de `Variable` para obtener los tipos, dados los tipos conocidos.

La clase en si es un hash (hereda de `HashMap<String, Variable.Type>`), que se encarga de mapear nombres de variables a tipos.

Variable:

Contiene las definiciones de cada tipo de variable posible (`enum Type`) y mapeos para saber tipos de variables dados otros tipos de variables conocidos. Por ejemplo el mapa `PtoSO` permite saber el tipo del sujeto y objeto dado el del predicado.

PredicateCheck:

Se encarga de retornar el tipo de la URI del predicado.

Randomizer:

Encargada del componente aleatorio del software. Genera nombres aleatorios de variables de la forma `?rN` donde `N` es un entero. Se encarga de que no se repitan nunca los nombres generados. Estos nombres son usados cuando se crean las variables que emergen al expandir las contracciones `NPV` y `NEN`.

`dup` sirve para poner un sufijo a las variables repetidas al usar `UNION`. Cuando las variables se usan a ambos lados de la operación `UNION`, en `Gremlin` se hace `JOIN` de los resultados, por lo que se pierden los que no sean iguales. Este no es el comportamiento deseado con esta operación, en donde se busca mantener todos los resultados. Es por esto que a todas las variables del segundo operando de `UNION` se le agregan estos sufijos. Con esto se calzan normalmente las nuevas variables con el paso `union` y al final se les quitan estos sufijos.

Este proceso se gatilla al encontrarse visitando el segundo operando de `UNION` en donde se llama `Randomizer.setDuplicatedVariable`, desde la clase `OpsFinder`.

PreambleBuilder:

Clase encargada de precargar el preámbulo de las consultas SPARQL. De esta forma, los prefijos mapeados a URIs no tienen que ser incluidos en la consulta por el usuario, ya que siempre existen. Este solo debe usar los prefijos conocidos.

BGPBuilder:

Se encarga de transformar un triple a su equivalente *traversal* en Gremlin. Encapsula todo lo indicado en 3.11.3, traduciendo según tipos conocidos y variables dentro del triple.

OptionalBuilder:

Se encarga de transformar el PGB de argumento de OPTIONAL al *traversal* equivalente en Gremlin, según lo indicado en 3.11.4.

FilterBuilder:

Se encarga de transformar las expresiones de filtrado de argumento en FILTER a sus *traversals* equivalentes según 3.11.5. Se implementa de tal forma que actúa según la clase de la expresión que se está visitando, generando el *traversal* correspondiente según la Tabla 20.

Los transform de las operaciones `and` y `or` hacen uso nuevamente de las demás funciones, para traducir las expresiones en sus parámetros.

4.2.5. com.datastax.sparql.ops

Este paquete contiene las clases encargadas de la visita de cada operación SPARQL y especificar la construcción de los *traversals* específicos. Implementan el Algoritmo 7. Su diagrama aparece en la Figura 13.

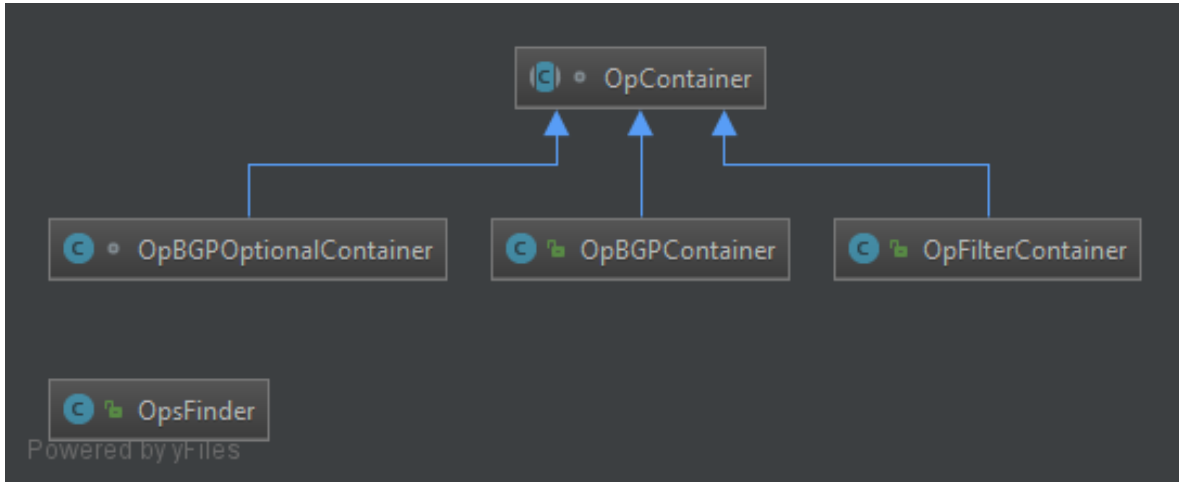


Figura 13: Paquete `com.datastax.sparql.ops`
Fuente: Elaboración propia.

OpContainer:

Clase abstracta que guarda un `Op`. Sus subclases implementan `getTraversals` según sea el tipo específico en la operación.

OpBGPContainer:

Subclase de `OpContainer`. Implementa `getTraversals` para el caso de `OpBGP` (subclase de `Op`). Hace uso de `BGPBuilder`.

OpBGPOptionalContainer:

Subclase de `OpContainer`. Implementa `getTraversals` para el caso de visitar un `OpBGP` que sea parámetro de una operación `OPTIONAL`. Hace uso de `OptionalBuilder`.

OpFilterContainer:

Subclase de `OpContainer`. Implementa `getTraversals` para el caso de `OpFilter`. Hace uso de `FilterBuilder`.

OpsFinder:

Clase encargada de las visitas a las operaciones de la consulta. Cuenta con una lista de `OpContainer` para guardarlos y luego ejecutar el `getTraversals` de cada uno.

Implementa 3 sobrecargas de `visit`, para cada operación. Notar que cuando es `OPTIONAL` el guardado no es directo. Si el `PGB` que recibe `OPTIONAL` como parámetro ya fue procesado anteriormente como `OpBGPContainer`, este se reemplazará por el correspondiente `OpBGPOptionalContainer`.

Una vez visitada toda la estructura y con los `Container` listos, se llama `getTraversalsArray` desde `Compiler`, para realizar la transformación y retornar un arreglo con todos los *traversals*.

4.2.6. `com.datastax.sparql.constants`

Contiene interfaces con constantes. Su diagrama se muestra en la Figura 14.

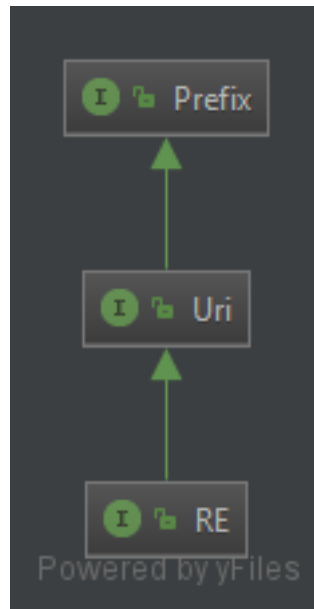


Figura 14: Paquete `com.datastax.sparql.constants`
Fuente: Elaboración propia.

Prefix:

Contiene las URIs que actúan como prefijos, es decir, sin el valor final. También tiene los prefijos que se usan para acortar las URIs.

Uri:

Contiene las URIs completas usadas en los predicados.

Re:

Contiene las expresiones regulares necesarias para calzar consultas SPARQL* según la gramática en 3.8.2. Se usa la sintaxis usada para calzar con `java.util.regex` y se usa `s`, `p` y `o` como agrupadores de sujetos, predicado y objeto de triples respectivamente.

4.3. Código

Todo el código implementado se encuentra en el repositorio público de Github <https://github.com/db-di-utfsm/sparql-gremlin>

4.4. Tests

Para comprobar que el software funciona correctamente, se escribieron pruebas para cada funcionalidad implementada. A continuación se muestran las consultas de prueba realizadas. Se muestra en lenguaje natural, en SPARQL* y en Gremlin. La consulta en Gremlin no es la traducción, sino que es un ejemplo de consulta regular que se ejecutaría para obtener el resultado deseado. Todas son ejecutadas sobre el grafo `the_crew_7` y los resultados obtenidos se muestran en el formato del software. Todas las consultas Gremlin generan el mismo resultado que las SPARQL, por lo que todos los test pasan y comprueban que la traducción es válida. Algunas consultas en lenguaje natural son similares pero se expondrán formas distintas de obtener los resultados en SPARQL para demostrar las distintas funcionalidades implementadas.

1. Obtener los nodos de las personas que sepan usar el software `tinkergraph` con nivel de habilidad 3

- Gremlin

```
g.V().as('x').outE().has('skill',3).  
  inV().has('name','tinkergraph').select('x')
```

- SPARQL

```
SELECT ?x WHERE {  
  ?x e:out ?e .  
  ?e e:label 'uses' .  
  ?e ep:skill 3 .  
  ?e e:in ?software .  
  ?software np:name ?n .  
  ?n n:value 'tinkergraph' .  
}
```

- Resultado

```
v[8]  
v[9]
```

2. Obtener el nodo cuyo nombre sea `gremlin`

- Gremlin

```
g.V().has('name','gremlin')
```

- SPARQL

```
SELECT ?x WHERE {  
  ?x np:name ?p .  
  ?p ?what 'gremlin' .  
}
```

- Resultado

```
v[10]
```

3. Obtener el año de inicio de trabajo de todos quienes hayan trabajado en santa cruz

- Gremlin

```
g.V().properties().hasValue('location','santa cruz')  
  .properties('startTime').value()
```

- SPARQL

```
SELECT ?s WHERE {  
  ?x np:location ?p .  
  ?p ?what 'santa cruz' .  
  ?p meta:startTime ?s .  
}
```

- Resultado

```
2001
```

4. Obtener los nodos cuyo nombre sea marko

- Gremlin

```
g.V().has('name','marko')
```

- SPARQL

```
SELECT ?x WHERE {  
  ?x np:name ?p .  
  ?p n:value 'marko' .  
}
```

- Resultado

```
v[1]
```

5. Obtener los nodos cuyo nombre sea marko (version SPARQL*)

- Gremlin

```
g.V().has('name','marko')
```

■ SPARQL

```
SELECT ?x WHERE {  
  <<?x np:name 'marko' >>. }
```

■ Resultado

```
v[1]
```

6. Consulta sin resultados para validar sintaxis

■ SPARQL

```
SELECT ?p WHERE {  
  << ?x np:name 'a' >>.  
  << ?x np:name 'b' >>;  
  meta:p1 'c' ;  
  meta:p2 'd'.  
  ?x ?p ?t .  
  << ?a np:p3 ?b >> ;  
  meta:mp4 'e' .  
  << ?x e:to ?y >> ;  
  ep:p6 ?p .  
}
```

■ Resultado

```
-
```

7. Obtener el año de inicio de trabajo de daniel en aachen

■ Gremlin

```
g.V().has('name','daniel').properties('location').  
  hasValue('aachen').values('startTime')
```

■ SPARQL

```
SELECT ?startTime WHERE {  
  << ?x np:name 'daniel' >> .  
  <<?x np:location 'aachen'>> ;  
  meta:startTime ?startTime .  
}
```

■ Resultado

```
2009
```

8. Obtener las labels de todos los nodos

- Gremlin

```
g.V().label()
```

- SPARQL

```
SELECT ?label WHERE {  
  ?x n:label ?label .  
}
```

- Resultado

```
person  
person  
person  
person  
software  
software
```

9. Obtener los nodos cuyo label sea person

- Gremlin

```
g.V().hasLabel('person')
```

- SPARQL

```
SELECT ?x WHERE {  
  ?x n:label 'person' .  
}
```

- Resultado

```
v[1]  
v[7]  
v[8]  
v[9]
```

10. Obtener el nodo de id 1

- Gremlin

```
g.V(1)
```

- SPARQL

```
SELECT ?x WHERE {  
  ?x n:id 1 .  
}
```

- Resultado

```
v[1]
```

11. Obtener los nodos y sus ids

- Gremlin

```
g.V().as('x').id().as('id').select('id','x')
```

- SPARQL

```
SELECT ?id ?x WHERE {  
  ?x n:id ?id .  
}
```

- Resultado

```
{id=1, x=v[1]}  
{id=7, x=v[7]}  
{id=8, x=v[8]}  
{id=9, x=v[9]}  
{id=10, x=v[10]}  
{id=11, x=v[11]}
```

12. Obtener el nodo del cual sale el arco de id 17

- Gremlin

```
g.V().outE().hasId(17).outV()
```

- SPARQL

```
SELECT ?x WHERE {  
  ?x e:out ?e .  
  ?e e:in ?y .  
  ?e e:id 17 .  
}
```

- Resultado

```
v[7]
```

13. Obtener los nodos a los que llegue un arco

- Gremlin

```
g.V().out()
```

- SPARQL

```
SELECT ?y WHERE {  
  <<?x e:to ?y >>.
```

- Resultado

```
v[11]
v[11]
v[10]
v[10]
v[10]
v[10]
v[11]
v[11]
v[10]
v[10]
v[11]
v[10]
v[11]
v[11]
```

14. Obtener los nodos softwares que sean desarrollados por alguna persona

■ Gremlin

```
g.V().out('develops')
```

■ SPARQL

```
SELECT ?y WHERE {
  <<?x e:to ?y >> ;
  e:label 'develops' .
}
```

■ Resultado

```
v[10]
v[11]
v[10]
v[11]
v[10]
```

15. Obtener los nodos persona que desarrollen un software desde el 2011

■ Gremlin

```
g.V().outE('develops').has('since',2011).outV()
```

■ SPARQL

```
SELECT ?x WHERE {
  <<?x e:to ?y >> ;
  ep:since 2011 .}
```

■ Resultado

v[7]

16. Obtener los ids de todos los arcos

- Gremlin

```
g.V().outE().id()
```

- SPARQL

```
SELECT ?c WHERE {  
  ?x e:out ?e .  
  ?e e:in ?y .  
  ?e e:id ?c .}
```

- Resultado

```
16  
15  
13  
14  
19  
20  
17  
18  
22  
23  
21  
24  
25  
26
```

17. Obtener los labels de todos los arcos

- Gremlin

```
g.V().outE().label()
```

- SPARQL

```
SELECT ?c WHERE {  
  ?x e:out ?e .  
  ?e e:in ?y .  
  ?e e:id ?c .}
```

- Resultado

```
uses  
uses  
develops
```

```
develops
uses
uses
develops
develops
uses
uses
develops
uses
uses
traverses
```

18. Obtener los valores de todas las habilidades

■ Gremlin

```
g.V().outE().values('skill')
```

■ SPARQL

```
SELECT ?s WHERE {
  << ?x e:to ?y>> ;
  ep:skill ?s .
}
```

■ Resultado

```
5
4
5
4
3
3
5
3
```

19. Obtener los nodos de los cuales salga un arco que llegue a un nodo de label o id 11

■ Gremlin

```
g.V().as('x').out().
or(hasLabel('11'),hasId(11)).select('x')
```

■ SPARQL

```
SELECT ?x WHERE {
  << ?x e:to ?y>> .
  ?y ?p 11 .
}
```

- Resultado

```
v[1]
v[1]
v[7]
v[7]
v[8]
v[9]
v[10]
```

20. Obtener los nodos de label o id 11 a los que llegue un arco que venga de un nodo de label person.

- Gremlin

```
g.V().hasLabel('person').
  out().or(hasLabel('11'),hasId(11))
```

- SPARQL

```
SELECT ?y WHERE {
  ?x n:label 'person' .
  << ?x e:to ?y>> .
  ?y ?p 11 .
}
```

- Resultado

```
v[11]
v[11]
v[11]
v[11]
v[11]
v[11]
```

21. Obtener los nodos de los cuales salga un arco con alguna propiedad que valga 4

- Gremlin

```
g.V().as('x').outE().values().is(4).select('x')
```

- SPARQL

```
SELECT ?x WHERE {
  ?x n:label 'person' .
  << ?x e:to ?y>> ;
  ?p 4 .
}
```

- Resultado

```
v[1]  
v[7]
```

22. Obtener los nodos cuya ubicación sea san diego, y que esta propiedad tenga una metapropiedad que valga 2001

- Gremlin

```
g.V().as('x').has('location','san diego')  
.properties().values().is(2001).select('x')
```

- SPARQL

```
SELECT ?x WHERE {  
  << ?x np:location 'san diego'>>;  
  ?p 2001 .  
}
```

- Resultado

```
v[1]
```

23. Obtener los nodos que tengan una label o id 16

- Gremlin

```
g.V().or(hasLabel('16'),hasId(16))
```

- SPARQL

```
SELECT ?v WHERE {  
  ?v ?p 16 .  
}
```

- Resultado

```
-
```

24. Obtener los pares de nodos que estén unidos por un arco y tengan el mismo label. Mostrar ambos nodos y el label.

- Gremlin

```
g.V().as('x').out().as('y').label().as('v').  
match(__.as('x').label().as('v')).select('x','y','v')
```

- SPARQL

```
select ?x ?y ?v where{  
  ?x n:label ?v .  
  << ?x e:to ?y >> .  
  ?y ?z ?v .  
}
```

■ Resultado

```
{x=v[10], y=v[11], v=software}
```

25. Obtener la propiedad `name` (deduplicada) de todos los nodos `persona` que manejen un `software` con nivel de habilidad mayor a 3.

■ Gremlin

```
g.V().match(
  __.as('x').label().as('v'),
  __.as('x').outE().values('skill').as('v2'),
  __.as('x').properties('name').as('n')
).where(__.as('v').is('person'))
  .where(__.as('v2').is(gt(3)))
  .dedup('n').select('n')
```

■ SPARQL

```
SELECT DISTINCT ?n WHERE{
  ?x n:label ?v .
  << ?x e:to ?y >>;
  ep:skill ?v2.
  ?x np:name ?n.
  FILTER (?v = 'person' && ?v2 > 3)
}
```

■ Resultado

```
vp[name->marko]
vp[name->stephen]
vp[name->daniel]
```

26. Obtener los nodos (deduplicados) que tengan habilidad 3 en el uso de un `software` y opcionalmente el año en donde comenzaron a desarrollar un `software`

■ Gremlin

```
g.V().match(
  __.as('x').outE().values('skill').as('s'),
  __.as('s').is(3),
  __.as('x').choose(
    __.outE().values('since'),
    __.outE().values('since'),
    __.constant('')).as('d')
).dedup('x','d').select('x','d')
```

■ SPARQL

```
SELECT DISTINCT ?x ?d WHERE {  
  << ?x e:to ?y>> ;  
  ep:skill 3 .  
  OPTIONAL {  
    ?x e:out ?e .  
    ?e ep:since ?d.  
  }  
}
```

■ Resultado

```
{x=v[8], d=2012}  
{x=v[9], d=}
```

27. Obtener el nombre de todas las personas y opcionalmente el año donde comenzaron a desarrollar un software

■ Gremlin

```
g.V().match(  
  __.as('x').hasLabel('person'),  
  __.as('x').values('name').as('name'),  
  __.as('x').choose(  
    __.outE().values('since'),  
    __.outE().values('since'),  
    __.constant('')).as('d')  
  ).select('name','d')
```

■ SPARQL

```
SELECT ?name ?y WHERE {  
  ?x n:label 'person' .  
  << ?x np:name ?name >> .  
  OPTIONAL {  
    ?x e:out ?e .  
    ?e ep:since ?y .  
  }  
}
```

■ Resultado

```
{name=marko, y=2009}  
{name=marko, y=2010}  
{name=stephen, y=2010}  
{name=stephen, y=2011}  
{name=matthias, y=2012}  
{name=daniel, y=}
```

28. Obtener los nodos persona que usen un software con habilidad mayor a 3. Además las personas que desarrollen un software desde después del 2009. Obtener también los nodos software en cuestión, como también las habilidades y años encontrados.

■ Gremlin

```
g.V().union(
  __.match(
    __.as('x').outE().as('e1'),
    __.as('e1').inV().as('y'),
    __.as('e1').values('skill').as('s'),
    __.as('s').where(is(gt(3)))
  ),
  __.match(
    __.as('x2').outE().as('e2'),
    __.as('e2').inV().as('f'),
    __.as('e2').values('since').as('t'),
    __.as('t').where(is(gt(2009)))
  )
)
```

■ SPARQL

```
SELECT * WHERE {
  {
    << ?x e:to ?y>> ;
    ep:skill ?s .
    FILTER (?s > 3)
  }
  UNION
  {
    <<?x e:to ?f>>;
    ep:since ?t.
    FILTER (?t > 2009)
  }
}
```

■ Resultado

```
{s=5, x=v[1], y=v[11], r80257=e[16][1-uses->11]},
{s=4, x=v[1], y=v[10], r80257=e[15][1-uses->10]},
{x=v[1], f=v[11], t=2010, r52033=e[14][1-develops->11]},
{s=5, x=v[7], y=v[10], r80257=e[19][7-uses->10]},
{s=4, x=v[7], y=v[11], r80257=e[20][7-uses->11]},
{x=v[7], f=v[10], t=2010, r52033=e[17][7-develops->10]},
{x=v[7], f=v[11], t=2011, r52033=e[18][7-develops->11]},
{x=v[8], f=v[10], t=2012, r52033=e[21][8-develops->10]},
```

{s=5, x=v[9], y=v[10], r80257=e[24][9-uses->10]}

29. Obtener todas las habilidades mayores a 4 y opcionalmente los años de inicio de desarrollo.

- Gremlin

```
g.V().match(
  __.as('x').outE().values('skill').as('s'),
  __.as('s').is(gt(4)),
  __.as('x').choose(
    __.outE().values('since'),
    __.outE().values('since'),
    __.constant('')).as('d')
).select('s','d')
```

- SPARQL

```
SELECT ?s ?d WHERE {
  << ?x e:to ?y>> ;
  ep:skill ?s .
  FILTER (?s > 4) .
  OPTIONAL {
    ?x e:out ?e .?e ep:since ?d.}
}
```

- Resultado

```
{s=5, d=2009}
{s=5, d=2010}
{s=5, d=2010}
{s=5, d=2011}
{s=5, d=}
```

CAPÍTULO 5

CONCLUSIONES

El desarrollo del software basado en el modelo propuesto fue exitoso. Este último demostró representar muy bien las potencialidades de un PG y logró ser suficientemente simple para lograr su implementación sin tener que recurrir a cambiar código interno del framework usado, sino que valerse de las herramientas que este provee públicamente. A continuación se consideran algunos puntos que es necesario concluir luego de la memoria.

5.0.1. Mapeo

Representar un PG como RDF a través del mapeo ideado resultó como se esperaba. Si bien no se mostró una demostración formal que cualquier grafo Tinkerpop puede representarse con el mapeo, con los ejemplos dados, el desarrollo del programa y los test pasados queda bastante claro que esto es posible. En conclusión, el mapeo permite ver el PG como si fuera un *edge-labelled graph*, que cuenta con una estructura calzable con patrones básicos a través de SPARQL. Si bien es cierto que quien desee consultar un grafo Tinkerpop a través de este software requiere tener conocimiento previo del mapeo, este es suficientemente simple y auto-explicativo como para un fácil aprendizaje. Esto sumado a un conocimiento previo de bases de datos, lenguajes de consulta de bases de datos y familiarización con SPARQL, RDF o similar, permitiría hacer consultas rápidamente. Todo esto provee a los desarrolladores familiarizados con el dominio la posibilidad de usar una nueva tecnología valiéndose de sus conocimientos. La reutilización de conocimientos es un aspecto muy valioso en la industria, y este trabajo es un pequeño intento por darle valor a esto.

5.0.2. Gramática y SPARQL*

Quedó de manifiesto que el uso de SPARQL* como simplificación de la sintaxis fue efectivo. Las consultas mostradas en varios ejemplos y sobre todo en los tests pudieron ser escritas sin explicitar variables de arcos o propiedades de nodo que no fueran a ser consultadas. Además el uso de la URI `_:to` hacia más natural el calce de tres elementos del grafo tan solo refiriéndose a dos, dando la opción de incluir restricciones y calces para el arco.

El desarrollo fue ampliamente facilitado por la utilización de expresiones regulares para la generación de una gramática que permitía identificar y traducir posteriormente las partes de la consulta que fueran SPARQL*. Si bien es una solución *ad-hoc*, es más genérica que utilizar condicionales sobre el string de consulta, pero más particular que aumentar las capacidades de JENA en cuanto al parseo de consultas. Esto último escapaba del ámbito de la memoria.

Sumado a lo anterior, las URIs reducidas usadas como predicado en el mapeo permitieron

que las consultas fueran más cortas.

En general, como SPARQL está enfocado al calce de patrones y Gremlin a la navegación sobre un grafo, las consultas que buscan calzar patrones quedan más claras utilizando SPARQL. En los ejemplos se ve que muchas consultas en su versión Gremlin eran más complicadas de leer que en su versión SPARQL*. Más aún, algunas de Gremlin recurrían a los mismos pasos que utiliza el traductor, como `match`, que es naturalmente la función para calzar patrones.

Finalmente, como ambas sintaxis son permitidas (con y sin `<< >>`), el usuario tiene mayor libertad y no esta obligado a usar algo si no lo necesita.

5.0.3. Traducción

La implementación de la solución fue en gran parte facilitada por que en ambos lenguajes existen operaciones similares, dado que ambos trabajan sobre grafos y tienen en general los mismos objetivos.

Un aspecto que dificultó la traducción fue el funcionamiento del paso `match` y de cómo funciona el ligado de variables dentro del traversal Gremlin. Como las variables son únicas a través de todo el *traversal*, sea cual sea el paso actual que se resuelve, ocurrían problemas al utilizar la misma variable en ambos cuerpos del `union`, ya que en vez de sumar los resultados, se hacía un join y se perdían los que no calzaran con el segundo patrón. Es por esto que tuvo que recurrirse al renombramiento de las variables del segundo cuerpo del `union` (con valores aleatorios) y posterior regreso al nombre original, para evitar colisiones de nombre dentro del *traversal*.

5.0.4. Objetivos

Los objetivos de la memoria fueron logrados. Del análisis de RDF y Tinkerpop, y sus modelos subyacentes, se logró llegar a tres opciones de traducción, cuyas cualidades fueron sopesadas y se llegó a la conclusión de que el mapeo y el uso de SPARQL* eran lo indicado, por ser la solución más general y que permitía una sintaxis más limpia y reducida.

En base a lo anterior, se diseñó un algoritmo que permitía llevar la consulta SPARQL* a un conjunto de calces de datos en un grafo Tinkerpop. Con esto, se escribió un software en Java que extiende las capacidades del framework, haciendo uso de sus funciones mediante la traducción. Finalmente se escribieron tests probando lo implementado, obteniendo resultados satisfactorios.

Se espera que quienes lean este documento lo vean de varias formas. Por un lado como un esfuerzo de demostrar que se puede lograr compatibilidad entre sistemas que trabajan sobre dominios similares ideando una capa intermedia que haga de puente entre ambos.

La creación de esta capa se facilita mientras más similares sean los dominios sobre los que ambos sistemas trabajan.

Respecto a lo anterior, durante el desarrollo del trabajo de título, Amazon lanzó Amazon Neptune, una base de datos de cloud y distribuida, que soporta los modelos RDF y Property Graph, y puede ser consultada con SPARQL y Gremlin. Sorprende ver que las ideas planteadas en el presente trabajo tengan una implementación real y a la escala del gigante tecnológico mencionado. Esto deja de manifiesto que efectivamente es una buena idea compatibilizar ambas tecnologías dado que trabajan sobre dominios iguales y de formas similares, aprovechando los conocimientos de los desarrolladores al máximo. Para mayor información sobre el servicio favor referirse a [1].

También se busca generar crítica acerca de la generalidad y particularidad de las soluciones. Durante el desarrollo del trabajo se mantuvo una constante disputa entre las soluciones generales y las particulares respecto a la problemática planteada. Por un lado lo general es bueno porque sirve para más casos, pero casi siempre la implementación es más costosa, ya que requiere ver el problema de forma global, con mayor cantidad de casos borde y de escenarios específicos. Sin embargo para efectos de demostrar una idea, o de hacer pruebas de concepto, las soluciones particulares pueden ayudar a darse cuenta de problemas que puedan surgir posteriormente en la búsqueda de soluciones generales, o de simplemente crear algo pequeño para demostrar parcialmente que un postulado es factible. En la memoria se optó a veces por generalidad, como por ejemplo el mapeo de un modelo al otro y a veces por particularidad, como la implementación de la unión solo con selección de todas las variables, o la implementación de solo unas cuantas operaciones de patrones complejos.

También se busca hacer de esta memoria una introducción al mundo de las bases de datos de grafos. Conocer nuevos modelos y tecnologías de bases de datos es necesario a medida que las organizaciones y el mundo de los datos crece. Las formas originales de modelar el mundo real no son suficientes cuando los datos cambian constantemente, y es por esto que nuevos modelos como los estudiados en la memoria han aparecido en los últimos años.

Finalmente, se invita al lector a extender el presente trabajo. A continuación se sugieren temas que podrían desarrollarse.

5.1. Trabajo futuro

- Incluir vocabulario RDF real en vez de URIs inventadas, para acceder al PG
- Implementar las 3 traducciones que faltan
- Extender JENA para que entienda SPARQL*
- Implementar UNION con selección de variables
- Implementar las 3 combinaciones de tipos conocidos/desconocidos faltantes

- Mejorar la forma en que se generan las variables aleatorias
- No incluir variables autogeneradas en los resultados de UNION sin proyección
- Demostración formal de que cualquier grafo Tinkerpop puede ser representado con el modelo propuesto.
- Encontrar una forma de iniciar consulta desde los arcos en vez de los nodos e implementar la solución de tal forma que se pueda elegir entre ambas.

REFERENCIAS BIBLIOGRÁFICAS

- [1] Amazon Web Services. Amazon Neptune. Accedido: 1/6/2018.
- [2] Renzo Angles. A comparison of current graph database models. *Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012*, (April 2012):171–177, 2012.
- [3] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc. *Foundations of Modern Graph Query Languages*. V, 2016.
- [4] Apache Software Foundation. Apache TinkerPop3 Implementations. <http://tinkerpop.apache.org/docs/current/reference/#implementations>. Accedido: 25/06/2017.
- [5] Apache Software Foundation. Documentación de match. Accedido: 12/10/2017.
- [6] Apache Software Foundation. Gramática de SPARQL. Accedido: 2/12/2017.
- [7] Apache Software Foundation. Gremlin main feautres. <http://tinkerpop.apache.org/gremlin.html>. Accedido: 25/06/2017.
- [8] Apache Software Foundation. Mirror oficial de Apache Jena - Paquete org.apache.jena. Accedido: 30/11/2017.
- [9] Apache Software Foundation. Pasos en Gremlin. Accedido: 1/4/2018.
- [10] Apache Software Foundation. SPARQL 1.1 Query Language. <https://www.w3.org/TR/sparql11-query/>. Accedido: 30/11/2017.
- [11] Apache Software Foundation. SPARQL-Gremlin Repository. <https://github.com/dkuppitz/sparql-gremlin>. Accedido: 25/06/2017.
- [12] Apache Software Foundation. SPARQL Query Language for RDF. <https://www.w3.org/TR/rdf-sparql-query/>. Accedido: 25/06/2017.
- [13] Apache Software Foundation. TinkerPop-Enabled Providers. <http://tinkerpop.apache.org/providers.html#query-language-providers>. Accedido: 25/06/2017.
- [14] ArangoDB. ArangoDB Documentation. <https://docs.arangodb.com/3.1/Manual/index.html>. Accedido: 25/06/2017.
- [15] G. Hiranmay C. Santanu, M. Anupama. Multimedia Ontology. pages 7–22, 2016.
- [16] Olaf Hartig. Foundations of rdf* and sparql*: (an alternative approach to statement-level metadata in rdf). *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web 2017*, 1912, 2017.

- [17] D Hernández, A Hogan, and M Krötzsch. Reifying rdf: What works well with wikidata? 1457:32–47, 01 2015.
- [18] M. Duerst, M. Suigard. IRI RFC. Accedido: 04/10/2017.
- [19] Neo Technology. Neo4j features. <https://neo4j.com/product/>. Accedido: 25/06/2017.
- [20] Supriya S Pore and Swalaya B Pawar. Comparative Study of SQL & NoSQL Databases. *Ijarcet*, 4(5):1747–1753, 2015.
- [21] Vatika Sharma and Meenu Dave. SQL and NoSQL Databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8):2277–128, 2012.
- [22] Sparsity Technologies. Sparksee Documentation. <http://www.sparsity-technologies.com/>. Accedido: 25/06/2017.
- [23] Stardog. Stardog Documentation. <http://www.stardog.com/docs/>. Accedido: 25/06/2017.
- [24] World Wide Web Consortium. RDF Data Access Use Cases and Requirements. <https://www.w3.org/TR/rdf-dawg-uc/>. Accedido: 25/06/2017.