



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

Dipartimento di
Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di laurea in
Ingegneria Informatica

Classe n. L-8

Sviluppo di un tool per l'applicazione di policy su query in linguaggio SPARQL e relativa riscrittura

Candidato: *Gianluca Maffioletti* Relatore: *Chiar.mo Prof. Stefano Paraboschi*

Matricola n. 1059062

Anno Accademico
2020/2021

Abstract

L'argomento di questa tesi si colloca nell'ambito del progetto MOSAICrOWN, promosso dall'Unione Europea e del cui consorzio fa parte l'Università degli Studi di Bergamo, il quale si pone la finalità di realizzare soluzioni per la condivisione e l'analisi di dati appartenenti a differenti proprietari, preservandone il controllo e la privacy.

L'oggetto è lo sviluppo di un sistema informatico in grado di acquisire query in linguaggio SPARQL e di applicare ad esse diverse tipologie di vincoli in relazione all'utente che le ha formulate, in modo da limitare l'accesso alle informazioni di cui non si ha autorizzazione.

L'obiettivo dell'elaborato è quindi quello di costruire una rappresentazione interna di una query fornita in ingresso, estrapolare e analizzare i suoi elementi, apportare eventuali modifiche imposte dalle policy di accesso, e restituire in uscita la query rettificata.

Per la realizzazione si è deciso di utilizzare il linguaggio Java, in quanto il paradigma ad oggetti risulta consono ad affrontare questo tipo di problema. Nel programma sono state congiunte classi sviluppate ad hoc con classi provenienti dal framework Apache Jena.

La tesi è articolata in cinque capitoli principali. Nel primo capitolo si contestualizza il problema affrontato e si introduce il tool da sviluppare. Nel secondo capitolo viene svolto un approfondimento teorico sulle due tecnologie alla base del progetto, ossia il modello di dati RDF e il linguaggio di interrogazione SPARQL. Nel terzo capitolo viene descritto dettagliatamente il sistema informatico, dai componenti fino al funzionamento. Nel quarto capitolo vengono presentati una serie di esempi di casi d'uso, per mostrare nel concreto il suo comportamento. Infine, nel quinto capitolo si espongono le conclusioni del lavoro di tesi.

Sommario

Capitolo 1: Introduzione al progetto	8
Capitolo 2: Tecnologie.....	9
2.1 RDF.....	9
2.1.1 Metadati	9
2.1.2 Semantic web.....	9
2.1.3 RDF strumento e componenti.....	10
2.1.4 RDF Data Model	10
2.1.5 Knowledge Graph.....	12
2.1.6 Namespace.....	12
2.1.7 RDF Schema.....	13
2.1.8 Confronto tra RDF data model e modello relazionale.....	14
2.2 SPARQL	16
2.2.1 SELECT query	16
2.2.1.1 Clausola SELECT.....	17
2.2.1.4 Clausola FROM	17
2.2.1.2 Clausola WHERE	17
2.2.1.3 Altre clausole	18
2.2.2 Formato Turtle.....	19
2.2.3 Oggetti Literal	20
2.2.4 Confronto tra SPARQL e SQL.....	21
Capitolo 3: Descrizione del tool sviluppato	23
3.1 Introduzione	23
3.2 Apache Jena	23
3.2.1 La classe <i>Query</i>	24
3.2.2 Elementi di <i>Query</i>	27
3.2.3 <i>ElementPathBlock</i>	28
3.2.4 <i>ElementFilter</i>	30
3.2.5 Costruzione di un oggetto <i>Query</i>	32
3.2.6 Esecuzione di un oggetto <i>Query</i>	35
3.3 Vincoli	36

3.3.1	Vincoli sui nodi	36
3.3.2	Vincoli sui predicati	37
3.3.3	Vincoli sugli attributi.....	39
3.3.4	Classi per rappresentare i vincoli	40
3.3.4.1	<i>NodeConstraint</i>	40
3.3.4.2	<i>PredicateConstraint</i>	41
3.3.4.3	<i>AttributeConstraint</i>	41
3.3.5	Contenitore di vincoli	42
3.4	<i>SPARQLQuery</i>	43
3.4.1	<i>VarTypes, UriTypes, PredicateTypes</i>	44
3.4.2	Costruttore	44
3.4.3	Metodi per ottenere gli elementi dell'oggetto <i>Query</i>	49
3.4.4	Metodi per modificare l'oggetto <i>Query</i>	51
3.4.4.1	Rimuovere variabili risultato	51
3.4.4.2	Rimuovere triple	52
3.4.4.3	Rimuovere filtri.....	53
3.4.4.4	Aggiungere filtri	53
3.5	<i>SPARQLParser</i>	58
3.5.1	Verifica e applicazione dei vincoli sui nodi	58
3.5.2	Verifica e applicazione dei vincoli sui predicati	59
3.5.3	Verifica e applicazione dei vincoli sugli attributi.....	62
3.6	Funzionamento.....	63
Capitolo 4: Casi d'uso		65
4.1	Descrizione del dataset	65
4.2	Esempi: vincoli sui nodi	66
4.3	Esempi: vincoli sui predicati.....	69
4.3	Esempi: vincoli sugli attributi.....	71
Capitolo 5: Conclusione		74

Indice delle figure

Figura 1: Struttura di una tripla RDF [7].....	11
Figura 2: Rappresentazione grafica di dati RDF [8]	11
Figura 3: Diagramma UML del Visitor Design Pattern [18]	27
Figura 4: Rappresentazione grafica del dataset [10]	65
Figura 5: Rappresentazione in formato Turtle semplificato del dataset [7]	66

Indice delle tabelle

Tabella 1: Risultati di un esperimento di confronto fra tempi di esecuzione di query su un database relazionale e su un database a grafo	15
Tabella 2: Esempi di letterali RDF con relativa tipologia.....	21
Tabella 3: Classi di espressioni	34
Tabella 4: Combinazioni di espressioni con l'espressione NOT IN	57

Indice dei listati

Listato 1: Dichiarazione di una classe RDF e di una sua istanza [10]	13
Listato 2: Dichiarazione di sottoclassi RDF [10]	13
Listato 3: Dichiarazione di proprietà RDF e impostazione di dominio e codominio [10]	14
Listato 4: Impostazione di label e comment di una proprietà RDF [10]	14
Listato 5: Struttura base di una query SPARQL di tipo SELECT [10].....	16
Listato 6: Esempio di clausola SELECT di una query SPARQL [10]	17
Listato 7: Esempio di clausola WHERE di una query SPARQL con triple e filtri [10]	18
Listato 8: Esempio di query SPARQL con filtro NOT EXISTS [10]	18
Listato 9: Esempio di query SPARQL con clausole ORDER BY e LIMIT [10].....	19
Listato 10: Esempio di contenuto di un documento in formato Turtle semplificato [7]	20
Listato 11: Esempio di oggetto della classe Query e dei suoi elementi	26
Listato 12: Codice Java per l'implementazione di un ElementWalker	28
Listato 13: Esempio di oggetti ElementPathBlock, PathBlock e TriplePath	29
Listato 14: Esempio di oggetto TriplePath con i suoi elementi.....	29
Listato 15: Esempio di Query con oggetti ElementFilter.....	30
Listato 16: Esempio di oggetti Expr e relativi elementi.....	32
Listato 17: Esempio creazione oggetto Query e ElementGroup	32
Listato 18: Esempio creazione oggetto ElementPathBlock e aggiunta oggetti Triple	33
Listato 19: Esempio creazione oggetti Expr e aggiunta ElementFilter	35
Listato 20: Esempio esecuzione oggetto Query	36
Listato 21: Esempio vincolo sui nodi in formato JSON.....	37

Listato 22: Esempio vincolo sui predicati in formato JSON	38
Listato 23: Esempio vincolo sugli attributi in formato JSON	40
Listato 24: Esempio costruttore Query, vettori di variabili e URI	45
Listato 25: Esempio costruttore Query, ricerca classi prima fase	46
Listato 26: Esempio costruttore Query, ricerca classi seconda fase.....	48
Listato 27: Esempio costruttore Query, RDF Schema proprietà :writer	48
Listato 28: Vincolo sui nodi generico relativo all'esempio 1	66
Listato 29: Vincolo sui nodi specifico relativo all'esempio 1	67
Listato 30: Esempio 1 vincoli sui nodi.....	67
Listato 31: Esempio 2 vincoli sui nodi.....	68
Listato 32: Vincolo sui nodi relativo all'esempio 2.....	68
Listato 33: Esempio 3 vincoli sui predicati	69
Listato 34: Vincolo sui predicati relativo all'esempio 3.....	69
Listato 35: Esempio 4 vincoli sui predicati	70
Listato 36: Vincolo sui predicati relativo all'esempio 4.....	70
Listato 37: Esempio 5 vincoli sugli attributi	71
Listato 38: Vincolo sugli attributi relativo all'esempio 5	72
Listato 39: Vincolo sugli attributi relativo all'esempio 6	72
Listato 40: Esempio 6 vincoli sugli attributi	73

Capitolo 1: Introduzione al progetto

Negli ultimi decenni lo sviluppo continuo del web sta portando innumerevoli vantaggi nell'ambito della distribuzione dei dati e della condivisione della conoscenza. Le nuove tecnologie consentono infatti di accedere in remoto ad informazioni collocate su differenti nodi di rete e rappresentate in diversi formati. Uno degli strumenti progettati a questo scopo è RDF, che fornisce un modello in grado di elaborare e gestire i dati contenuti nella rete.

Le informazioni costituiscono una risorsa tanto strategica quanto critica, che necessita di adeguati sistemi di sicurezza per impedire l'accesso ad utenti non autorizzati. Il progresso della condivisione comporta quindi in parallelo l'implementazione di nuove tecnologie volte a garantire privacy e confidenzialità.

In questo contesto, l'Unione Europea ha avviato il progetto MOSAICrOWN [1], che coinvolge le Università degli Studi di Bergamo e di Milano, il W3C e grandi società internazionali quali Dell, MasterCard e SAP. Il fulcro centrale del progetto è la realizzazione di strumenti per l'analisi collaborativa e tecniche di condivisione dati che garantiscono ai proprietari il controllo totale nel tempo, la divulgazione selettiva e meccanismi di sanitizzazione per preservare la riservatezza e la sensibilità delle informazioni.

Il gruppo di lavoro dell'Università di Bergamo si è occupato della realizzazione di un policy engine responsabile della valutazione e applicazione delle policy di MOSAICrOWN su query formulate in linguaggio SQL. Le regole di accesso sono state differenziate in base al soggetto che effettua l'interrogazione, il dataset target, la tipologia di operazione e la motivazione della richiesta.

Prendendo come riferimento il sistema citato, questa tesi si propone di elaborare una forma di vincoli di accesso a dati in formato RDF e sviluppare un tool in grado di valutarli e applicarli a query scritte in linguaggio SPARQL, il linguaggio di interrogazione per dati RDF.

Capitolo 2: Tecnologie

2.1 RDF

RDF, acronimo di Resource Description Framework, è lo strumento base proposto dal World Wide Web Consortium per la codifica, lo scambio e il riutilizzo di metadati strutturati e consente l'interoperabilità semantica tra applicazioni che condividono le informazioni sul web [2][3]. Per comprendere meglio questa tecnologia, occorre prima focalizzare l'attenzione sui concetti di metadati e semantic web.

2.1.1 Metadati

Nel web una risorsa è qualsiasi contenuto di informazioni, che sia un documento leggibile da un essere umano o un oggetto leggibile da una macchina, identificato univocamente mediante un URI (Uniform Resource Identifier).

Ogni risorsa è generalmente corredata di informazioni descrittive che prendono il nome di metadati, e che consistono in un insieme di asserzioni sui dati, ossia una lista di coppie attributo-valore.

I metadati relativi a un documento possono essere estratti dalla risorsa stessa (ad esempio se sono contenuti nella sezione Head di un documento HTML), da un'altra risorsa, oppure possono essere trasferiti assieme al documento (con richieste HTTP GET e POST).

Una caratteristica chiave dei metadati è il fatto che sono machine-understandable. Entrando nel contesto del semantic web, ciò consente ai software agent di poterli utilizzare per ottimizzare ricerche ed elaborazioni [4].

2.1.2 Semantic web

Con il termine semantic web si intende la transizione del World Wide Web ad un ambiente in cui i documenti pubblicati (pagine HTML, file, immagini, ...) sono associati ad informazioni e dati (metadati) che ne specificano il contesto semantico in

un formato adatto all'interrogazione, all'interpretazione e, più in generale, all'elaborazione automatica [5].

I vantaggi che ne scaturiscono sono molteplici, come la possibilità di ricerche più sofisticate, oppure l'evoluzione delle connessioni tra documenti dai semplici collegamenti ipertestuali a logiche più elaborate.

Questo processo di automatizzazione è stato favorito dalla presenza dei metadati, i quali, essendo machine-understandable, hanno consentito di superare le limitazioni dell'architettura originale del web nella quale le informazioni erano machine-readable.

Lo step successivo necessario è stato quello di introdurre delle convenzioni per la semantica, la sintassi e la struttura dei metadati. In questo scenario è intervenuto il W3C, "World Wide Web Consortium" [6], la cui finalità principale è lo sviluppo di tecnologie e protocolli comuni volti a favorire l'evoluzione del web e assicurarne l'interoperabilità.

2.1.3 RDF strumento e componenti

RDF è la tecnologia introdotta dal W3C per soddisfare le richieste di standardizzazione dei metadati e consentirne un efficace utilizzo.

RDF è costituito da due componenti:

- **RDF Model and Syntax:** descrive la struttura del modello dati RDF e una possibile sintassi;
- **RDF Schema:** espone la sintassi per definire schemi e vocabolari per i metadati.

2.1.4 RDF Data Model

Il modello dati RDF è basato su tre elementi principali: risorse, proprietà e valori.

Le risorse, come già detto precedentemente, sono qualunque oggetto identificabile univocamente mediante un URI, reperibile principalmente sul web, ma non solo.

Le proprietà sono delle relazioni che legano tra loro risorse e valori, e sono anch'esse identificate da URI. Un insieme di proprietà che fanno riferimento alla stessa risorsa viene detto "descrizione".

I valori, infine, sono letterali di tipo primitivo (stringhe, interi, date, ...) oppure URI di altre risorse.

Una risorsa, con una proprietà contraddistinta da un nome e il relativo valore, costituiscono un RDF statement, l'unità base per rappresentare informazioni in RDF. Uno statement è quindi una tripla del tipo Soggetto – Predicato – Oggetto, dove il soggetto è una risorsa, il predicato è una proprietà e l'oggetto è un valore letterale oppure un URI che punta ad un'altra risorsa.

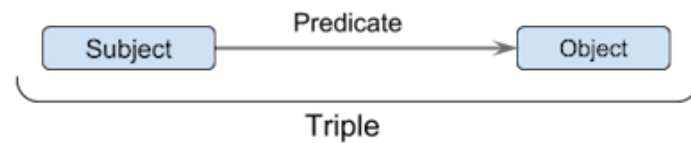


Figura 1: Struttura di una tripla RDF [7]

Graficamente, le relazioni tra risorsa, proprietà e valore vengono rappresentate mediante grafi etichettati orientati, in cui le risorse sono nodi e vengono identificate come ellissi, le proprietà come archi orientati e i valori come rettangoli. In Figura 2 un esempio di rappresentazione grafica di dati RDF.

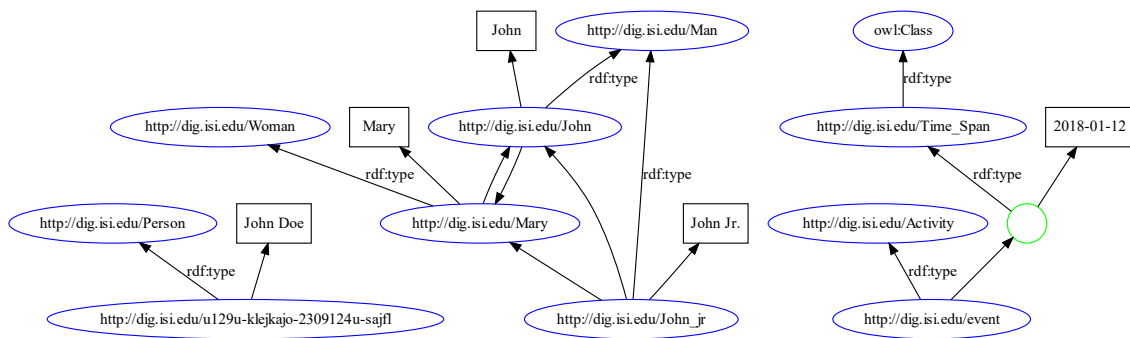


Figura 2: Rappresentazione grafica di dati RDF [8]

RDF mette a disposizione tre tipologie di container per consentire di fare riferimento ad un insieme di risorse piuttosto che ad una singola, in relazione al fatto che la proprietà può assumere valori multipli ordinati (Sequence) o non ordinati (Bag), oppure un singolo valore scelto tra un elenco di alternative (Alternative).

Un RDF statement può anche diventare l'oggetto di un altro statement tramite il processo chiamato reificazione (riduzione a oggetto), e il modello che ne risulta, chiamato *reified statement*, consente di costruire informazioni sempre più complesse, aumentando la capacità comunicativa del modello.

2.1.5 Knowledge Graph

La struttura a tripla degli RDF statement consente, impostando come valore dell'oggetto l'URI di un'altra risorsa, di creare una rete di relazioni, definendo un Knowledge Graph.

La caratteristica fondamentale dei Knowledge Graph è quella di consentire l'interpretazione dei dati e la costruzione di nuovi fatti, ampliando la conoscenza dedotta dal dataset.

I dati RDF, quindi, possono essere rappresentati oltre che graficamente anche concettualmente come dei grafi orientati, in cui risorse e proprietà corrispondono a nodi e archi.

Si sottolinea, però, che non tutte le collezioni di triple RDF costituiscono dei Knowledge Graph: il requisito chiave è l'interconnessione dei dati e la creazione di nuova conoscenza che ne deriva.

2.1.6 Namespace

Per quanto riguarda invece la semantica, RDF sfrutta il meccanismo dei namespace (*XMLNs*) [9] per identificare univocamente le proprietà, lasciando così la libertà di definire la propria semantica alle singole comunità, rimuovendo il rischio di conflitti di nomenclatura.

Ogni namespace è individuato da un URI. Per questioni di leggibilità e di fruibilità, è opportuno definire un prefisso per ogni namespace utilizzato, ed evitare di riportare gli URI nel loro intero. Esempi comuni di prefissi sono:

- “rdf” per lo spazio dei nomi <http://www.w3.org/1999/02/22-rdf-syntax-ns#>, che fornisce le classi e proprietà standard di RDF;
- “rdfs” per <http://www.w3.org/2000/01/rdf-schema#>, che comprende classi e proprietà per la definizione di vocabolari RDF (paragrafo 2.1.7);

- “xsd” per <http://www.w3.org/2001/XMLSchema#>, che definisce i tipi di dati principali (paragrafo 2.2.3).

2.1.7 RDF Schema

RDF Schema (RDFS) è il componente di RDF che, attraverso un insieme di classi e proprietà, consente di strutturare vocabolari RDF, comprensibili sia ad un lettore umano che ad una macchina, la quale può sfruttarli per semplificare comprensione ed analisi dei dati cui fanno riferimento.

In RDF le risorse possono essere rappresentate come istanze di classi. Ogni risorsa è istanza della classe `rdfs:Resource`. Questa ha tre sottoclassi principali:

- `rdfs:Class`, che identifica una classe di risorse;
- `rdfs:Property`, che rappresenta le proprietà;
- `rdfs:Literal`, che rappresenta un letterale, una stringa di testo.

Si può affermare che una risorsa è istanza di una determinata classe tramite la proprietà `rdf:type`. In molti linguaggi e rappresentazioni, per questa proprietà viene riconosciuta comunemente l'abbreviazione “a”.

```
:Band a rdfs:Class .           # declaration of a class
:The_Beatles a :Band .         # declaring an instance of a class
```

Listato 1: Dichiarazione di una classe RDF e di una sua istanza [10]

È possibile dichiarare delle gerarchie fra classi definendo delle sottoclassi tramite la proprietà `rdfs:subClassOf`.

```
:Artist a rdfs:Class .
:Band rdfs:subClassOf :Artist .
:SoloArtist rdfs:subClassOf :Artist .
```

Listato 2: Dichiarazione di sottoclassi RDF [10]

Vi è inoltre la possibilità di stabilire dei vincoli sulle proprietà relativamente al dominio e al codominio, ossia le classi o il datatype rispettivamente di soggetto e oggetto di cui tale proprietà è predicato, tramite le proprietà `rdfs:domain` e `rdfs:range`.

```
:track a rdf:Property ;  
      rdfs:domain :Album ;  
      rdfs:range  :Song .  
  
:length a rdf:Property ;  
      rdfs:domain :Song ;  
      rdfs:range  xsd:integer .
```

Listato 3: Dichiarazione di proprietà RDF e impostazione di dominio e codominio [10]

Infine, vi sono due proprietà ritenute di buona pratica, benché non obbligatorie, utilizzate per fornire il nome e la descrizione di una risorsa che siano leggibili da un essere umano. Si tratta rispettivamente di `rdfs:label` e `rdfs:comment`.

```
:length rdfs:label "length (in seconds)" ;  
      rdfs:comment "The length of a song expressed in seconds".
```

Listato 4: Impostazione di label e comment di una proprietà RDF [10]

2.1.8 Confronto tra RDF data model e modello relazionale

L'analisi della struttura del modello dati RDF vista precedentemente permette di delineare alcune considerazioni e confronti con il modello di rappresentazione dati più diffuso, quello relazionale.

Soffermandoci ad un livello superficiale, astratto, si può dire che le due strutture sono abbastanza simili: nel modello ER abbiamo entità e relazioni, analogamente nel modello RDF abbiamo risorse e proprietà, ma quest'ultime sono oggetti identificati da un URI; un record rappresenta il nodo RDF, un nome del campo rappresenta una proprietà del nodo, e il campo del record rappresenta il valore.

Ovviamente però, analizzando più nel dettaglio, si evidenzia che le rappresentazioni fisiche differiscono.

Innanzitutto, il modello relazionale si fonda su schemi ben definiti, mentre il modello RDF, che si è visto assumere una rappresentazione a grafo, non prevede necessariamente dei dati strutturati. Risulta perciò adatto a rappresentare modelli di dati inconsistenti, flessibili, e responsivo a variazioni della struttura.

In RDF, quindi, le relazioni vengono stabilite a livello di record individuale, richiedendo un maggiore spazio di archiviazione, mentre in ER fanno riferimento a gruppi di record, intesi come tabelle. Questo comporta anche differenze sostanziali nei processi computazionali, in quanto su un grande quantitativo di record la configurazione a grafo richiede di esaminare ciascun record per determinarne la struttura.

Tuttavia, per determinati tipi di ricerche, basate maggiormente sulla connessione dei dati, la latenza di un database a grafo è minore. Ciò è dimostrato da un esperimento condotto da Jonas Partner e Aleksa Vukotic in cui sono stati confrontati i tempi di esecuzione di query con diversi livelli di profondità delle relazioni perimate su un database relazionale, MySQL, e su un database a grafo, Neo4j [11]. I risultati sono mostrati in Tabella 1, in cui i tempi di esecuzione sono espressi in secondi.

Profondità	Tempo di esecuzione – MySQL	Tempo di esecuzione – Neo4j
2	0,016	0,010
3	30,267	0,168
4	1.543,505	1,359
5	> 1 ora	2,132

Tabella 1: Risultati di un esperimento di confronto fra tempi di esecuzione di query su un database relazionale e su un database a grafo

Il punto centrale di questo confronto, però, è il ruolo del web. Infatti, il semantic web non è stato concepito per essere un nuovo modello dati, bensì risulta più appropriato come collegamento di dati provenienti da differenti modelli. Uno dei grandi vantaggi che esso mira a portare è proprio quello di poter processare un'enorme mole di informazioni distribuite in rete e provenienti da database eterogenei, consentendo scambi di dati e analisi collaborative.

2.2 SPARQL

SPARQL è un linguaggio di interrogazione per dati rappresentati in formato RDF e distribuiti sul web.

Essendo il modello di dati RDF fondato su triple della forma soggetto-predicato-oggetto, anche SPARQL utilizza questo pattern per estrapolare le informazioni. Di conseguenza, una query SPARQL consiste in un insieme di triple in cui ogni elemento può essere una variabile. Le soluzioni alle variabili vengono poi trovate confrontando le triple nella query con le triple all'interno del dataset.

SPARQL prevede quattro tipi di query:

- **SELECT**: utilizzata per estrarre valori in formato tabella, con una colonna per ogni variabile selezionata e una riga per ogni corrispondenza del modello;
- **ASK**: utilizzata per fornire un risultato booleano che indica se esiste almeno una corrispondenza tra il modello specificato nella query e le informazioni nel dataset;
- **CONSTRUCT**: utilizzata per estrarre valori in formato RDF, la cui struttura se non esplicitata corrisponde al modello specificato nella query;
- **DESCRIBE**: utilizzata per estrarre un grafo RDF di valori, non specificati dall'utente ma dal sistema, relativi a un determinato nodo o insieme di nodi.

Oltre al formato testuale (a tabella) e al grafo RDF, molti endpoint consentono di visualizzare i risultati in diversi altri formati, quali CSV, XML, JSON, e altri ancora.

2.2.1 SELECT query

Si analizza ora nel dettaglio la struttura della tipologia di query SELECT.

```
SELECT <variables>
WHERE {
    <graph pattern>
}
```

*Listato 5: Struttura base di una query SPARQL di tipo
SELECT [10]*

2.2.1.1 Clausola SELECT

La clausola SELECT precede un elenco di variabili o funzioni aggregate di variabili con i relativi alias che rappresentano il risultato della query, eventualmente accompagnate dalla keyword DISTINCT utilizzata per rimuovere i duplicati. Per selezionare tutte le variabili contenute nella query, invece di enumerarle, è possibile utilizzare il carattere “*”.

```
SELECT (count(?album) as ?count)
{
    ?album a :Album
}
```

Listato 6: Esempio di clausola SELECT di una query SPARQL [10]

2.2.1.4 Clausola FROM

Le query SELECT sono eseguite su un grafo predefinito, detto *default graph*.

In assenza di esso occorre specificare da dove caricare i dati. A questo scopo è possibile introdurre dopo la clausola SELECT la clausola FROM seguita dall’indirizzo a cui è possibile reperire le risorse web necessarie.

2.2.1.2 Clausola WHERE

La clausola WHERE è seguita da parentesi graffe al cui interno viene definito il modello da utilizzare per recuperare le informazioni richieste (il termine WHERE può anche essere omesso).

Il modello è composto principalmente da triple, i cui elementi possono essere variabili oppure URI. Come noto, gli URI non sono altro che stringhe rappresentanti il percorso da seguire per trovare la risorsa in questione. Una query con tanti URI potrebbe risultare complicata da scrivere e leggere. A tal proposito è possibile ricorrere all’utilizzo dei namespace, introducendo all’inizio della query, prima ancora della clausola SELECT, la clausola PREFIX seguita da un prefisso e il rispettivo percorso completo separati dal carattere “:”. Così facendo si può sostituire l’occorrenza di un URI con il relativo prefisso seguito dal nome della risorsa, semplificando scrittura e lettura della query per l’essere umano.

Oltre alle triple un elemento che assume un ruolo importante nella query è l'elemento FILTER. Questa espressione è utilizzata per filtrare i risultati restituiti, ponendo delle condizioni sulle variabili contenute nella query. Per scrivere tali vincoli SPARQL supporta i principali operatori di confronto, operatori logici, operatori matematici, e molte altre funzioni che lavorano su stringhe, date, elenchi di valori, elementi di RDF.

```
SELECT *
{
    ?album a :Album ;
           :artist ?artist ;
           :date ?date
    FILTER (year(?date) >= 1970)
}
```

Listato 7: Esempio di clausola WHERE di una query SPARQL con triple e filtri [10]

Ulteriori elementi che è possibile trovare all'interno della clausola WHERE sono:

- BIND, per assegnare l'output di una funzione a una variabile;
- UNION, per combinare le corrispondenze da due modelli diversi;
- MINUS, per rimuovere le corrispondenze di un modello da un altro;
- OPTIONAL, per indicare i modelli che possono esistere per alcuni nodi ma non per altri;
- NOT EXISTS, per trovare modelli che non esistono nel set di dati.

```
SELECT ?song {
    ?song a :Song .
    FILTER NOT EXISTS {
        ?song :length ?length .
    }
}
```

Listato 8: Esempio di query SPARQL con filtro NOT EXISTS [10]

2.2.1.3 Altre clausole

Le clausole che è possibile aggiungere successivamente alla clausola WHERE sono:

- GROUP BY, per raggruppare i risultati in base a una o più variabili;

- HAVING, per definire una condizione su un valore aggregato;
- ORDER BY, per ordinare i risultati in base ad una o più variabili nei due ordini ASC (default) e DESC;
- OFFSET, per saltare i primi N risultati;
- LIMIT, per limitare i risultati ad un numero N specificato.

```
SELECT *
{
    ?album a :Album ;
           :artist ?artist ;
           :date ?date
}
ORDER BY desc (?date)
LIMIT 2
```

Listato 9: Esempio di query SPARQL con clausole ORDER BY e LIMIT [10]

2.2.2 Formato Turtle

I documenti RDF possono essere rappresentati in diversi modi. Tra questi, si citano solamente N-Triples [12], JSON-LD [13] e RDF/XML [14].

Turtle, acronimo di “Terse RDF Triple Language” [15], è un formato ideato per esprimere dati RDF con una sintassi adatta a SPARQL.

Come ribadito più volte, le convenzioni RDF stabiliscono che le informazioni sono rappresentate per mezzo di triple, ciascuna delle quali consiste di un soggetto, un predicato e un oggetto. Ognuno di questi elementi è espresso come URI.

In un file Turtle (con estensione .ttl) ogni tripla è una sequenza di termini, separati da uno spazio, e terminata da un punto.

Spesso un soggetto può fare riferimento a più predicati. Turtle permette di evitare la ripetizione del soggetto terminando la tripla con un punto e virgola. In questo modo nella tripla successiva il soggetto è implicito, e si scrivono solamente il predicato e il valore.

Inoltre, quando ad una coppia soggetto-predicato corrispondono più oggetti, è possibile elencare questi valori separandoli con una virgola, senza necessità di ripetere la tripla.

Anche Turtle supporta il meccanismo dei namespace, per semplificare scrittura e lettura degli URI. All’inizio del dataset si possono elencare tutti i prefissi e i rispettivi percorsi preceduti dalla direttiva “@prefix” e terminati dal punto.

Inoltre, Turtle prevede la possibilità di scrivere commenti, utilizzando il simbolo “#”.

Per la sua struttura e le sue semplificazioni, Turtle è generalmente riconosciuto come il più leggibile e semplice da modificare manualmente. Un esempio è riportato nel Listato 10.

```
:The_Beatles      a :Band ;
                  :name "The Beatles" ;
                  :member :John_Lennon , :Paul_McCartney , :George_Harrison , :Ringo_Starr .
:John_Lennon      a :SoloArtist .
:Paul_McCartney   a :SoloArtist .
:Ringo_Starr      a :SoloArtist .
:George_Harrison  a :SoloArtist .
:Please_Please_Me a :Album ;
                  :name "Please Please Me" ;
                  :date "1963-03-22"^^xsd:date ;
                  :artist :The_Beatles ;
                  :track :Love_Me_Do .
:Love_Me_Do       a :Song ;
                  :name "Love Me Do" ;
                  :length 125 ;
                  :writer :John_Lennon , :Paul_McCartney .
```

Listato 10: Esempio di contenuto di un documento in formato Turtle semplificato [7]

2.2.3 Oggetti Literal

In una tripla RDF l’oggetto può essere un nodo identificato da un URI oppure un valore letterale.

La sintassi per i letterali prevede una stringa rappresentante il valore racchiusa tra apici singoli o doppi, con un tag lingua opzionale introdotto dal simbolo “@” e un tipo di dati opzionale in formato URI introdotto da “^^”.

I tipi di dati principali sono definiti da un namespace apposito, XML Schema Datatypes [16], comunemente abbreviato con il prefisso “xsd”, facente riferimento al seguente percorso <http://www.w3.org/2001/XMLSchema#>. Tra questi, si citano in Tabella 2 i più utilizzati, con un esempio dimostrativo.

Per comodità, i numeri possono essere scritti direttamente, senza apici e senza datatype esplicito, in quanto il sistema è in grado di riconoscere la tipologia analizzando la presenza o meno del punto (indicante la parte decimale) e dell’esponente.

Per i valori boolean TRUE e FALSE è prevista la stessa semplificazione.

Per quanto riguarda le stringhe, anche in questo caso è possibile omettere il datatype, mantenendo ovviamente gli apici. Inoltre, per facilitare la scrittura di valori contenenti apici oppure caratteri di nuova riga, SPARQL introduce un nuovo costrutto in cui i letterali sono racchiusi fra tre apici singoli o doppi.

Tipologia di dato	Esempio	Semplificazione
Integer	“5”^^xsd:integer	5
Decimal	“5.0”^^xsd:decimal	5.0
Double	“5.0E3”^^xsd:double	5.0E3
String	“sparql”^^xsd:string	“sparql”
Boolean	“true”^^xsd:boolean	true
Date	“1999-07-27”^^xsd:date	“1999-07-27”^^xsd:date

Tabella 2: Esempi di letterali RDF con relativa tipologia

2.2.4 Confronto tra SPARQL e SQL

SPARQL, proprio come SQL, fornisce le funzionalità di estrazione e manipolazione di dati strutturati.

Come è già stato evidenziato nel confronto tra modello RDF e relazionale, scrivendo che il modello RDF è adatto a riunire differenti rappresentazioni di dati, anche il linguaggio SPARQL consente di eseguire query su qualsiasi tipo di database, ad esempio quello relazionale, a patto che venga visualizzato come RDF tramite un software di mappatura.

La differenza sostanziale è che le query SPARQL non sono vincolate a lavorare all'interno di un database come in SQL, ma possono accedere a più datastore, facendo riferimento alla natura distribuita dei dati RDF nel web. Ciò è possibile perché SPARQL, oltre ad essere un linguaggio di interrogazione, è anche un protocollo di trasporto basato su HTTP. Infatti, l'acronimo SPARQL sta per “SPARQL Protocol and RDF Query Language”.

Entrando nei dettagli sintattici dei due linguaggi si evidenziano parecchie similitudini ma anche qualche differenza concettuale.

- Si nota che molti termini di clausole ed elementi sono i medesimi in SPARQL e SQL. Ad esempio, SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, così come anche la maggior parte di nomi di funzioni aggregate (SUM, COUNT, AVG, MIN, MAX).
- In SQL si parla di restrizione per selezionare le righe e proiezione per selezionare colonne specifiche di quelle righe. In SPARQL, nonostante la rappresentazione non sia tabellare, le stesse operazioni sono effettuate con le clausole SELECT e WHERE, combinando triple ed elementi FILTER.
- Gli operatori UNION e MINUS sono molto simili, ma in SQL prevedono che gli schemi dei due insiemi siano uniformi, mentre in SPARQL vi sono meno restrizioni, a seconda dello specifico endpoint.
- In SPARQL non esiste un concetto corrispondente al NULL di SQL, poiché non esiste un requisito RDF corrispondente al vincolo strutturale di SQL che ogni riga in un database debba essere conforme allo stesso schema.
- In SQL i vincoli di join, in genere, eliminano le righe se non sono presenti record con valori corrispondenti. I nodi di SPARQL non si legano se mancano attributi, perciò l'eliminazione del risultato è implicita. L'operatore LEFT OUTER JOIN, al contrario, esegue un join regolare ma non elimina le soluzioni se i vincoli di join non vengono soddisfatti. SPARQL utilizza l'elemento OPTIONAL per questo scopo.

Capitolo 3: Descrizione del tool sviluppato

3.1 Introduzione

Il sistema informatico oggetto della tesi combina le tecnologie presentate nel capitolo 2: le query che vengono trattate sono query SPARQL di tipologia SELECT, con una struttura base composta da triple ed elementi FILTER; il dataset con cui si interagisce contiene dati RDF, in formato Turtle semplificato.

In questo capitolo vengono descritti nei dettagli la composizione e il funzionamento dello strumento realizzato.

Nel paragrafo 3.2 si introduce il framework Apache Jena e le relative classi utilizzate, nel paragrafo 3.3 si espongono le tipologie di vincoli applicati alle query SPARQL e la loro rappresentazione in Java, nei paragrafi 3.4 e 3.5 verranno approfondite le due classi fondamentali per la gestione delle query e l'applicazione delle policy, ossia *SPARQLQuery* e *SPARQLParser*, e infine nel paragrafo 3.6 si riassume il funzionamento del sistema nel suo complesso.

3.2 Apache Jena

Apache Jena è un framework Java per lo sviluppo di applicazioni orientate al semantic web [17].

Esso fornisce API per leggere e scrivere grafi RDF, e un query engine, ARQ, che supporta il linguaggio SPARQL e la serializzazione dei grafi RDF da vari formati, tra cui Turtle.

Apache Jena mette a disposizione un set di librerie Java per la creazione, gestione, manipolazione ed esecuzione di query SPARQL, utili ai fini della realizzazione del progetto di questa tesi.

Nei prossimi paragrafi si descrivono le classi di Apache Jena utilizzate, citandone metodi e funzionalità.

3.2.1 La classe *Query*

La classe fondamentale su cui si basa l'intero progetto è la classe *Query*. Questa è in grado di rappresentare una query SPARQL, e fornisce una serie di metodi per ottenere o impostare i valori delle varie componenti di una query.

La classe è dotata di un costruttore privo di parametri per creare un oggetto *Query* vuoto. Per istanziare una query a partire da una stringa, invece, è possibile ricorrere alla classe *QueryFactory*, il cui metodo statico *create(String queryString)* restituisce un oggetto *Query* rappresentante la query testuale passata come parametro.

Come già visto, le query SPARQL possono essere di quattro tipologie: SELECT, ASK, CONSTRUCT, DESCRIBE. Per riconoscere la tipologia di query a partire da un oggetto *Query*, si possono utilizzare i metodi *isSelectType()*, *isAskType()*, *isConstructType()* e *isDescribeType()*, che restituiscono un valore booleano indicante se il tipo della query è il corrispondente. Viceversa, per impostare la tipologia della query si ricorre ai metodi *setQuerySelectType()*, *setQueryAskType()*, *setQueryConstructType()* e *setQueryDescribeType()*. Siccome il progetto di tesi si focalizza sulle query di tipologia SELECT, solamente i primi metodi dei due elenchi precedentemente citati sono utilizzati nel codice.

Per ogni clausola della query SELECT, la classe *Query* dispone di metodi per ottenere e impostare i relativi valori:

- Per la clausola SELECT, il metodo *getProject()* restituisce un oggetto della classe *VarExprList*, che consiste in una lista di coppie variabile-espressione rappresentanti le variabili risultato, ottenute con il metodo *getVars()*, e le rispettive espressioni nel caso siano formate da funzioni aggregate, ricavate con il metodo *getExpr(Var var)*.

Variabili ed espressioni recuperate da un oggetto *Query* sono istanze rispettivamente della classe *Var* e dell'interfaccia *Expr*. Della classe *Var* i due metodi più rilevanti sono il metodo statico *alloc(String varName)*, che permette di istanziare un oggetto *Var* passando come parametro il nome della variabile in forma testuale, e il metodo *getVarName()* che restituisce una stringa rappresentante il nome dell'oggetto che lo esegue. Dell'interfaccia *Expr* e delle

classi che la implementano si discuterà nel paragrafo 3.2.4, nel contesto degli elementi FILTER.

Per aggiungere invece variabili risultato, si ricorre al metodo *addResultVar()*, il cui parametro può essere una stringa o un oggetto *Var*, accompagnati eventualmente dall'espressione relativa se si tratta di una funzione aggregata (oggetto *Expr*).

Inoltre, è possibile verificare o impostare la rimozione dei duplicati per la query rispettivamente tramite i metodi *isDistinct()*, che restituisce un valore *boolean*, e *setDistinct(boolean b)*, specificando *true*;

- Per quanto riguarda la clausola WHERE, il suo contenuto può essere recuperato con il metodo *getQueryPattern()* che restituisce un oggetto *Element*, ma per estrapolarne i vari elementi la procedura è piuttosto articolata, e verrà trattata nel paragrafo 3.1.2;
- Per la clausola GROUP BY, il metodo *addGroupBy()* restituisce un oggetto *VarExprList*, già introdotto precedentemente, mentre per aggiungere un raggruppamento si utilizza il metodo *addGroupBy()*, passando come parametro la variabile in formato stringa o come oggetto *Var*, oppure un'oggetto *Expr* se si tratta di una funzione aggregata;
- Similmente, per la clausola HAVING, il metodo *getHavingExprs()* restituisce una lista di espressioni istanze di *Expr*, mentre il metodo *addHavingCondition(Expr expr)* ne aggiunge una alla query;
- Per la clausola ORDER BY, il metodo *getOrderBy()* fornisce una lista di oggetti della classe *SortCondition*, i quali sono composti da coppie variabile (*String* o *Var*) e direzione dell'ordinamento (*int*). Analogamente, per aggiungere un ordinamento alla query è possibile utilizzare il metodo *addOrderBy()* passando come parametro un oggetto *SortCondition* oppure direttamente la coppia variabile-ordinamento;
- Infine, per le clausole OFFSET e LIMIT, i metodi *getOffset()* e *getLimit()* restituiscono il rispettivo valore in formato *long*. Viceversa, per aggiungere i due valori si sfruttano i metodi *setOffset()* e *setLimit()* passando il valore *long* come parametro;

- Se nella query sono presenti dichiarazioni di prefissi, il metodo *getPrologue()* restituisce un oggetto della classe *Prologue*; richiamando poi il metodo *getPrefixMapping()* da quest'ultimo otteniamo un oggetto della classe *PrefixMapping*, al cui interno vi sono le coppie prefisso-percorso presenti nella query. Se si vuole invece aggiungere prefissi ad una query, occorre inizialmente creare un oggetto *PrefixMapping* e aggiungere le coppie prefisso-percorso tramite il metodo *setNsPrefix()*, passando le due stringhe. Successivamente si crea un oggetto *Prologue*, passando nel costruttore l'oggetto *PrefixMapping*. E infine si crea una nuova *Query* passando nel costruttore l'oggetto *Prologue*.

Nel Listato 11 viene presentato un esempio di oggetto Query con i valori ottenuti dai metodi precedentemente elencati.

```

SELECT ?album (COUNT(?song) AS ?countSong)
WHERE
  { ?album a      :Album ;
    :track ?song
  }
GROUP BY ?album
HAVING ( COUNT(?song) > 5 )
ORDER BY DESC(?album)
OFFSET 20
LIMIT 100

query.getPrologue().getPrefixMapping():
pm:={=http://stardog.com/tutorial/}

query.getProject():
[?album, ?countSong] // {?countSong=(AGG ?.0 COUNT(?song))}

query.getGroupBy():
[?album] // {}

query.getHavingExprs():
[(> (count ?song) 5)]

query.getOrderBy():
[(SortCondition DESC(?album))]

query.getLimit():
100

query.getOffset():
20

```

Listato 11: Esempio di oggetto della classe Query e dei suoi elementi

3.2.2 Elementi di *Query*

I vari elementi di una query SPARQL sono rappresentati da sottoclassi della classe astratta *Element*.

Come introdotto precedentemente, a partire da un oggetto *Query* si può ottenere il suo modello richiamando il metodo *getQueryPattern()*, che restituisce un riferimento della classe *Element*. La metodologia di analisi dell'oggetto segue il principio del Visitor Design Pattern, un modello risolutivo per separare un algoritmo dalla struttura di oggetti a cui è applicato. In Figura 3 viene mostrata una rappresentazione grafica sottoforma di diagramma UML del Visitor Design Pattern.

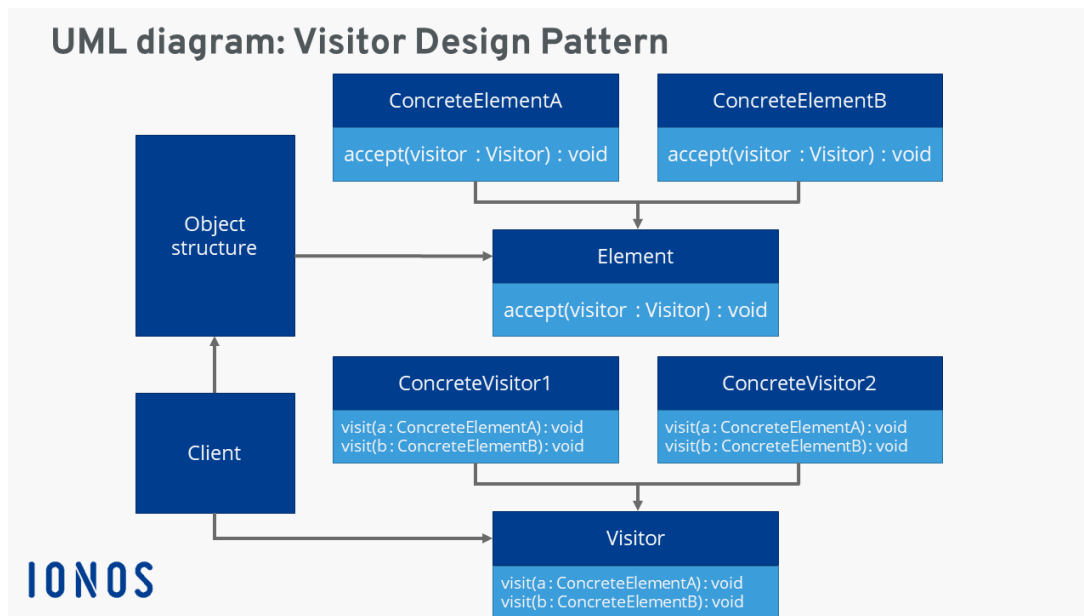


Figura 3: Diagramma UML del Visitor Design Pattern [18]

Apache Jena fornisce la classe *ElementWalker*, il cui metodo statico *walk(Element el, ElementVisitor visitor)* si occupa di percorrere l'albero interno dell'elemento passato come primo parametro e di applicare ad ogni elemento attraversato un visitatore, passato come secondo parametro. Il visitatore deve implementare l'interfaccia *ElementVisitor*, al cui interno sono predisposti dei metodi *visit()* per ciascun tipo di elemento. L'oggetto che sarà utilizzato è istanza della classe *ElementVisitorBase*, un visitatore vuoto. Quando viene eseguito il metodo *walk()* della classe *ElementWalker*, per il secondo parametro viene creato un nuovo oggetto *ElementVisitorBase* e viene

riscritto *inline* il metodo *visit()* con parametro la tipologia di elemento che si vuole analizzare ed elaborare (se presente).

Il Listato 12 mostra il codice Java per l'implementazione di un *ElementWalker*. All'interno dei metodi *visit()* verrà scritto il codice per lavorare sulle triple e sui filtri della query.

```
ElementWalker.walk(query.getQueryPattern(), new ElementVisitorBase() {  
  
    @Override  
    public void visit(ElementPathBlock el) {  
        // code for triples  
    }  
  
    @Override  
    public void visit(ElementFilter el) {  
        // code for filters  
    }  
  
});
```

Listato 12: Codice Java per l'implementazione di un *ElementWalker*

3.2.3 *ElementPathBlock*

L'elemento principale presente in tutte le query SPARQL sono le triple. La classe *ElementPathBlock*, sottoclasse di *Element*, rappresenta un contenitore di triple che compaiono in sequenza all'interno di una query.

Quando si riscrive il metodo *visit(ElementPathBlock)*, la classe *ElementWalker* analizza il contenuto dell'elemento passato come parametro e se trova e riconosce un oggetto della classe *ElementPathBlock* esegue il codice del metodo *visit()*. Ovviamente, le triple in una query possono costituire un blocco unico oppure più blocchi se si trovano intervallate da altri elementi, ad esempio un *FILTER*.

Il metodo *getPattern()* applicato ad un oggetto *ElementPathBlock* restituisce un oggetto *PathBlock*, che rappresenta la vera e propria collezione di triple. Eseguendo poi il metodo *getList()* su quest'ultimo, si ottiene una lista di oggetti *TriplePath*.

Un esempio che espone questi elementi è mostrato nel Listato 13, che riguarda il modello di triple della query del Listato 11.

```

ElementPathBlock:
?album a                <http://stardog.com/tutorial/Album> ;
    <http://stardog.com/tutorial/track> ?song

PathBlock:
[?album (<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>) http://stardog.com/tutorial/Album,
 ?album (<http://stardog.com/tutorial/track>) ?song]

TriplePath:
?album (<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>) http://stardog.com/tutorial/Album

TriplePath:
?album (<http://stardog.com/tutorial/track>) ?song

```

Listato 13: Esempio di oggetti *ElementPathBlock*, *PathBlock* e *TriplePath*

Un'istanza di *TriplePath* consiste a tutti gli effetti in una tripla. Infatti, è dotata dei metodi *getSubject()*, *getPredicate()* e *getObject()* che restituiscono i tre elementi della tripla come oggetti della classe *Node*. Un elemento della query può essere:

- **Variabile:** verificabile con il metodo booleano *isVariable()*. È possibile definire un oggetto *Var* a partire dall'oggetto *Node* richiamando il metodo statico *alloc(Node v)* della classe *Var*;
- **URI:** verificabile con il metodo booleano *isURI()*. Per gli URI non vi è una classe specifica, si mantengono le istanze di *Node*;
- **Letterale:** verificabile con il metodo booleano *isLiteral()*. Con il metodo *getLiteral()* si ottiene il relativo oggetto *LiberalLabel*.

```

TriplePath:
?artist (<http://stardog.com/tutorial/name>) "John Lennon"

Subject: ?artist
isVariable: true, isUri: false, isLiteral: false

Predicate: http://stardog.com/tutorial/name
isVariable: false, isUri: true, isLiteral: false

Object: "John Lennon"
isVariable: false, isUri: false, isLiteral: true

```

Listato 14: Esempio di oggetto *TriplePath* con i suoi elementi

3.2.4 *ElementFilter*

Un altro elemento fondamentale per le query SPARQL sono i FILTER, che permettono di applicare delle condizioni alle variabili. Questi elementi sono rappresentati tramite la classe *ElementFilter*, sottoclasse di *Element*.

Anche in questo caso, i filtri vengono ricavati dal query pattern tramite la classe *ElementWalker* e riscrivendo il metodo *visit()* dell'oggetto *ElementVisitorBase* nella forma *visit(ElementFilter el)*.

L'oggetto *ElementFilter* è composto da un'espressione, che nel concreto assume la forma di un riferimento dell'interfaccia *Expr*, ottenibile eseguendo il metodo *getExpr()*.

Dall'oggetto *Expr* possiamo estrarre la funzione corrispondente con il metodo *getFunction()*, derivando un oggetto *ExprFunction*. Quest'ultimo è costituito da un simbolo testuale che ne descrive la tipologia di funzione, ricavabile dalla concatenazione di due metodi *getFunctionSymbol().getSymbol()*, e da una lista di argomenti, sempre riferimenti *Expr*, a cui si accede con il metodo *getArgs()*.

```
PREFIX :      <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?song a      :Song ;
          :length ?length ;
          :writer ?writer
    FILTER ( ?writer IN ( :John_Lennon, :Paul_McCartney ) )
    FILTER ( ( ?length > 100 ) && ( ?length != 200 ) )
  }

ElementFilter:
FILTER ( ?writer IN ( <http://stardog.com/tutorial/John\_Lennon>,
  <http://stardog.com/tutorial/Paul\_McCartney> ) )

ExprFunction:
(in ?writer <http://stardog.com/tutorial/John\_Lennon>
  <http://stardog.com/tutorial/Paul\_McCartney>)
Symbol: in

ElementFilter:
FILTER ( ( ?length > 100 ) && ( ?length != 200 ) )

ExprFunction:
(&& (> ?length 100) (!= ?length 200))
Symbol: and
```

Listato 15: Esempio di Query con oggetti ElementFilter

Un argomento può essere:

- **Variabile:** verificabile con il metodo booleano *isVariable()*. Con il metodo *getExprVar()* si ottiene un oggetto *ExprVar*, che implementa l'interfaccia *Expr*, da cui poi si genera l'oggetto della classe *Var* con il metodo *asVar()*;
- **Costante:** verificabile con il metodo booleano *isConstant()*. L'esecuzione del metodo *getConstant()* restituisce un oggetto della classe *NodeValue*. Questa classe è incaricata di rappresentare tutte le tipologie di dato, a partire da numeri *integer*, *decimal*, *double* fino a *string*, *date* e *boolean*. Per verificarne il tipo e generare un oggetto della corrispondente classe sono predisposti tutti gli appositi metodi, ad esempio *isInteger()* e *getInteger()*.
- **Funzione:** verificabile con il metodo booleano *isFunction()*. Un argomento può essere una funzione in caso di funzioni annidate, quali le operazioni logiche AND, OR, NOT. Discorso a parte per le funzioni aggregate, che non vengono gestite direttamente dall'interfaccia *Expr*, ma occorre effettuare un casting del riferimento *Expr* alla classe *ExprAggregator*. Da quest'ultima si può poi accedere con il metodo *getAggregator()* all'oggetto della classe *Aggregator* che, similmente alla classe *ExprFunction*, è dotato di una stringa che rappresenta il nome della funzione aggregata (COUNT, SUM, AVG, MAX, MIN) ottenibile con il metodo *getName()*, e un oggetto *ExprList*, ottenibile con il metodo *getExprList()*, che è di fatto un insieme di espressioni. La vera e propria lista di oggetti *Expr* viene restituita dal metodo *getList()* eseguito sull'istanza di *ExprList*.

In presenza di un oggetto *ExprFunction* o *Aggregator*, quindi, si procede ricorsivamente ad analizzare la lista delle sue espressioni fino a quando si ottengono valori variabile o costante.

Nel Listato 16 vengono mostrati gli elementi delle espressioni contenute nei filtri della query del Listato 15, specificandole il tipo.

```

ExprFunction:
(in ?writer <http://stardog.com/tutorial/John\_Lennon> <http://stardog.com/tutorial/Paul\_McCartney>)
Symbol: in

Variable: ?writer
Constant: <http://stardog.com/tutorial/John\_Lennon>
Constant: <http://stardog.com/tutorial/Paul\_McCartney>

ExprFunction:
(&& (> ?length 100) (!= ?length 200))
Symbol: and

Function: (> ?length 100)
Symbol: gt
Variable: ?length
Constant: 100

Function: (!= ?length 200)
Symbol: ne
Variable: ?length
Constant: 200

```

Listato 16: Esempio di oggetti Expr e relativi elementi

3.2.5 Costruzione di un oggetto *Query*

Finora si è visto come analizzare il contenuto della clausola WHERE di una query SPARQL generata a partire da una stringa fornita in ingresso. Per effettuare la procedura contraria, quindi costruire il modello di una query, bisogna produrre i singoli componenti per poi assemblarli in un unico elemento.

Il primo step è quello appunto di realizzare lo scheletro del modello, che consiste nell'istanziare un oggetto vuoto della classe *ElementGroup*.

```

Query query = new Query();
query.setQuerySelectType();
query.setQueryResultStar(true);

ElementGroup group = new ElementGroup();

```

Listato 17: Esempio creazione oggetto Query e ElementGroup

Poi si procede con le triple. Come visto prima, il contenitore delle triple è la classe *ElementPathBlock*, di cui viene creato un oggetto vuoto. Tramite il metodo *addTriple(Triple t)*, si va poi ad aggiungere una alla volta le triple necessarie.

Per creare una nuova tripla, si è scelto di non utilizzare la classe *TriplePath*, bensì la classe *Triple*, che concettualmente rappresenta lo stesso elemento ma che fornisce un costruttore di più semplice utilizzo, il quale richiede tre parametri *Node*, ossia uno per soggetto, predicato e oggetto:

- Se uno degli elementi è una variabile, è possibile passare come parametro l'oggetto della classe *Var* oppure crearlo con il metodo statico *alloc()* della classe *Var*, poiché sottoclasse di *Node*;
- Se un elemento è un URI, invece, si passa l'elemento *Node* corrispondente oppure lo si crea a partire da una stringa ricorrendo al metodo statico *asNode(String s)* della classe *NodeUtils*;
- Infine, se si tratta di un valore letterale, il parametro sarà un oggetto *NodeValue*. Si precisa che *NodeValue* è una superclasse generica. L'oggetto che viene istanziato è della classe *NodeValueInteger* se un intero, *NodeValueString* se una stringa e così via.

Completato l'inserimento di tutte le triple, si aggiunge al modello *ElementGroup* l'oggetto *ElementPathBlock* tramite il metodo *addElement(Element el)*.

```
ElementPathBlock block = new ElementPathBlock();

Triple triple = new Triple(
    Var.alloc("song"),
    NodeUtils.asNode("http://www.w3.org/1999/02/22-rdf-syntax-ns#type"),
    NodeUtils.asNode("http://stardog.com/tutorial/Song")
);
block.addTriple(triple);

triple = new Triple(
    Var.alloc("song"),
    NodeUtils.asNode("http://stardog.com/tutorial/length"),
    Var.alloc("length")
);
block.addTriple(triple);

group.addElement(block);
```

Listato 18: Esempio creazione oggetto ElementPathBlock e aggiunta oggetti Triple

Successivamente si passa agli elementi FILTER. In questo caso non si ha una classe contenitore di filtri, ma si procede alla creazione dei singoli oggetti e conseguente aggiunta al modello. Il costruttore della classe *ElementFilter* prevede come parametro

un oggetto che implementa l'interfaccia *Expr*. In Tabella 3 si fornisce l'elenco delle classi di espressioni che saranno utilizzate ai fini del progetto con la relativa funzione che realizzano e i parametri necessari per l'implementazione.

Classe	Funzione	Parametri
<i>E_Equals</i>	=	Un oggetto <i>ExprVar</i> che rappresenta la variabile a cui viene applicato il filtro e un oggetto <i>NodeValue</i> che rappresenta il valore di confronto della funzione.
<i>E_NotEquals</i>	≠	
<i>E_GreaterThan</i>	>	
<i>E_GreaterThanOrEqual</i>	≥	
<i>E_LessThan</i>	<	
<i>E_LessThanOrEqual</i>	≤	
<i>E_OneOf</i>	IN	Un oggetto <i>ExprVar</i> e un oggetto <i>ExprList</i> che rappresenta una lista di oggetti <i>Expr</i> che costituiscono i valori di confronto della funzione.
<i>E_NotOneOf</i>	NOT IN	
<i>E_LogicalAnd</i>	AND	Due oggetti <i>Expr</i> che rappresentano le funzioni a cui applicare l'operatore logico.
<i>E_LogicalOr</i>	OR	
<i>E_LogicalNot</i>	NOT	Un oggetto <i>Expr</i> che rappresenta la funzione a cui applicare l'operatore logico.

Tabella 3: Classi di espressioni

Vi è un'ulteriore tipologia di filtro in SPARQL che permette di imporre o vietare la presenza di nodi che corrispondono alla tripla specificata, il FILTER EXISTS e il FILTER NOT EXISTS. Ad esempio, se si vuole imporre che la variabile “artist” non sia istanza della classe “Band”, si utilizza il filtro seguente:

```
FILTER NOT EXISTS {?artist a :Band}
```

Per questi tipi di filtro, le classi di espressione corrispondenti sono *E_Exists* e *E_NotExists*, il cui unico parametro richiesto è un oggetto *Triple*.

Una volta creato l'oggetto *ElementFilter*, lo si aggiunge al modello *ElementGroup* tramite il metodo *addElementFilter(ElementFilter el)*.

```
Expr e1 = new E_GreaterThan(  
    new ExprVar( name: "length"),  
    new NodeValueInteger( i: 100)  
);  
Expr e2 = new E_NotEquals(  
    new ExprVar( name: "length"),  
    new NodeValueInteger( i: 200)  
);  
Expr e = new E_LogicalAnd(e1, e2);  
  
ElementFilter filter = new ElementFilter(e);  
  
group.addElementFilter(filter);
```

Listato 19: Esempio creazione oggetti Expr e aggiunta ElementFilter

Infine, si procede con l'assegnazione del modello alla query, utilizzando il metodo *setQueryPattern(Element el)* della classe *Query*, passando come parametro l'oggetto *ElementGroup* contenente tutti i sottoelementi.

3.2.6 Esecuzione di un oggetto Query

Successivamente alla costruzione della query da zero oppure a partire da una stringa fornita in ingresso, può essere necessario eseguirla.

Se il dataset da interrogare si trova su un file locale, il primo step è quello di creare un oggetto che lo rappresenti. Questo oggetto è istanza della classe *Model*, e viene costruito ricorrendo al metodo statico *loadModel(String file)* della classe *RDFDataMgr*, che ha come parametro la stringa relativa al percorso del file da leggere. Una volta ottenuto il modello si procede con l'esecuzione.

Prima di tutto si genera l'oggetto *QueryExecution* eseguendo il metodo statico *create(Query q, Model model)* della classe *QueryExecutionFactory* che ha come parametri la query da eseguire e il dataset che deve interrogare.

In seguito, si esegue la query con il metodo *execSelect()* dell'oggetto *QueryExecution*. Il risultato di questo metodo è un oggetto *ResultSet*, iteratore di oggetti *QuerySolution*. Gli iteratori impongono una scansione tramite ciclo while, in cui la condizione di

permanenza è che l'oggetto *ResultSet* abbia ulteriori soluzioni. La condizione viene scritta con il metodo booleano *hasNext()*, di cui dispone ogni iteratore. Infine, i risultati si leggono con il metodo *next()* che restituisce il successivo oggetto *QuerySolution*, e su questo il metodo *get(String v)* restituisce il valore della variabile specificato come parametro.

```
Model model = RDFDataMgr.LoadModel(uri: "fileString");

QueryExecution qe = QueryExecutionFactory.create(query, model);
ResultSet rs = qe.execSelect();

while (rs.hasNext()) {
    QuerySolution qs = rs.next();
    System.out.println(qs.get("variable"));
}
```

Listato 20: Esempio esecuzione oggetto Query

3.3 Vincoli

Lo scopo del progetto è quello di garantire la sicurezza e la riservatezza delle informazioni, limitando l'accesso ai dati in relazione alle autorizzazioni di cui dispone una determinata categoria di utenti. Per questo è necessario stabilire una serie di policy di accesso, che possano essere elaborate dal sistema e verificate sulle query in ingresso.

Considerando la natura e la struttura dei dati RDF e delle query SPARQL, si è deciso di definire tre tipologie di vincoli: vincoli sui nodi, vincoli sui predicati e vincoli sugli attributi.

Il sistema legge l'elenco dei vincoli in formato JSON da un file esterno.

Tutti gli oggetti JSON sono dotati di un campo *user*, che identifica l'utente o la categoria di utenti a cui fa riferimento il vincolo, e un campo *constraint* che equivale alla tipologia del vincolo e può assumere i valori “node”, “predicate” e “attribute”.

3.3.1 Vincoli sui nodi

La prima tipologia di vincolo si applica ai nodi, ossia permette di vietare l'accesso a una determinata classe di nodi oppure a degli specifici nodi appartenenti a questa.

Si ricorda che nei grafi RDF ogni elemento di informazione è considerato una risorsa o nodo, e che ciascuno di questi è istanza di una particolare classe, la quale può essere individuata tramite il predicato `rdf:type`.

Quando sono presenti più vincoli sulla stessa classe, se tra questi vi è un vincolo generico, ossia applicato all'intera classe e non a specifici nodi, si assume che questo ha la massima priorità, mentre gli altri perdono rilevanza in quanto già compresi nel vincolo generico.

I vincoli di questa tipologia dispongono di un campo *node-type*, che corrisponde all'URI della classe di nodi per cui viene limitato l'accesso, ed un campo facoltativo *nodes*, che rappresenta una lista di nodi su cui può venire ristretto il vincolo.

Il Listato 21 contiene un esempio di vincoli sui nodi.

```
{
  "user": "u1",
  "constraint": "node",
  "node-type": "http://stardog.com/tutorial/Band"
},

{
  "user": "u1",
  "constraint": "node",
  "node-type": "http://stardog.com/tutorial/SoloArtist",
  "nodes": [
    "http://stardog.com/tutorial/George_Harrison",
    "http://stardog.com/tutorial/John_Lennon",
    "http://stardog.com/tutorial/Paul_McCartney",
    "http://stardog.com/tutorial/Ringo_Starr"
  ]
},
```

Listato 21: Esempio vincolo sui nodi in formato JSON

3.3.2 Vincoli sui predicati

La seconda tipologia riguarda i predicati, vale a dire che se il predicato di una tripla, in cui soggetto e oggetto sono nodi e non letterali, corrisponde a quello specificato nel vincolo, viene limitato l'accesso agli elementi della tripla. In questo caso, il vincolo si differenzia in quattro varianti in base all'ampiezza della restrizione:

- Se vengono specificate solo le classi di soggetto e oggetto, allora la limitazione si applica a tutte le triple che hanno il predicato specificato e i cui soggetto e oggetto appartengono alle rispettive classi indicate.
- Se oltre alle classi viene segnalata anche una lista di nodi soggetto, allora la limitazione si applica a tutte le triple il cui soggetto compare nella lista;
- Analogamente, se viene segnalata una lista di nodi oggetto, allora la limitazione si applica a tutte le triple il cui oggetto compare nella lista;
- Infine, se vengono evidenziati sia soggetti che oggetti, il vincolo è relativo solo alle triple con soggetto predicato e oggetto corrispondenti a quelli indicati.

In presenza di più vincoli relativi allo stesso predicato e alle stesse classi di soggetto e oggetto, il vincolo generico, ossia quello che si applica sulle intere classi di soggetto e oggetto, ha la priorità più alta rispetto alle altre tre varianti.

I vincoli sui predicati possiedono i campi obbligatori *subject-type*, *predicate* e *object-type* che indicano rispettivamente gli URI della classe di nodi soggetto, del predicato, e della classe di nodi oggetto. Se il vincolo è applicato a degli specifici nodi soggetto o oggetto oppure ad una combinazione di questi allora i loro URI devono essere inseriti nei campi *subjects* e *objects*.

Il Listato 22 contiene un esempio di vincolo sui predicati.

```
{
  "user": "u4",
  "constraint": "predicate",
  "subject-type": "http://stardog.com/tutorial/Band",
  "subjects": ["http://stardog.com/tutorial/The_Beatles"],
  "predicate": "http://stardog.com/tutorial/member",
  "object-type": "http://stardog.com/tutorial/SoloArtist",
  "objects": [
    "http://stardog.com/tutorial/John_Lennon",
    "http://stardog.com/tutorial/Paul_McCartney"
  ]
},
```

Listato 22: Esempio vincolo sui predicati in formato JSON

3.3.3 Vincoli sugli attributi

La terza e ultima tipologia riguarda gli attributi. Anche in questo caso il vincolo si basa su una classe di soggetto e un predicato specificati, ma la differenza è che l'oggetto è un letterale e non più un nodo. Il vincolo può vietare l'accesso all'attributo per tutti i nodi appartenenti alla classe o per alcuni nodi specifici se ne è presente l'indicazione, oppure può imporre una limitazione sul dominio dei valori che può assumere in relazione alla funzione e ai valori precisati.

Le funzioni previste sono le seguenti:

- =
- ≠
- >
- ≥
- <
- ≤
- *BETWEEN*
- *IN*
- *NOT IN*

I vincoli sugli attributi sono costituiti dai campi obbligatori *subject-type* e *predicate*, che rappresentano rispettivamente gli URI della classe dei nodi soggetto e del predicato che servono per determinare l'attributo a cui fa riferimento il vincolo, e il campo *symbol* che rappresenta la funzione che restringe il dominio dei valori che l'attributo può assumere. Nel caso il simbolo sia "X", questo equivale a dire che l'utente non può accedere a quell'oggetto per i nodi soggetto della classe indicata, oppure solo per i nodi specificati nel campo facoltativo *subjects*. Altrimenti il simbolo può essere uno tra i seguenti: "=", "!=", ">", ">=", "<", "<=", "between", "in" e "notin". In tal caso vengono resi disponibili il campo *values*, che può contenere uno o più valori in base alla funzione, e la tipologia dell'attributo nel campo *object-type*, che può essere *integer*, *double*, *string* o *date*, di cui il sistema necessita per elaborare i valori.

Un esempio di vincolo sugli attributi è contenuto nel Listato 23.

```

{
  "user": "u6",
  "constraint": "attribute",
  "subject-type": "http://stardog.com/tutorial/Album",
  "predicate": "http://stardog.com/tutorial/date",
  "object-type": "date",
  "symbol": "between",
  "values": [
    "1990-01-01",
    "1999-12-31"
  ]
},

```

Listato 23: Esempio vincolo sugli attributi in formato JSON

3.3.4 Classi per rappresentare i vincoli

Nel codice sono state create delle classi in grado di acquisire e riprodurre i vincoli a partire dai relativi oggetti JSON. Per leggere il file JSON è stata utilizzata la libreria “json-simple” di Google Code [19].

Innanzitutto, è stata implementata la classe astratta *Constraint*, dotata del campo di tipo stringa *user*, che rappresenta il solo campo comune a tutti i tipi di vincoli e le rispettive varianti. La classe dispone del metodo *getUser()* che restituisce appunto il valore del campo *user*.

Poi sono state strutturate tre sottoclassi che estendono la classe *Constraint*, una per ogni tipologia di vincolo. In ciascuna di queste, il costruttore riceve come parametro un oggetto *JSONObject*, e per prima cosa richiama il costruttore della superclasse passandogli la stringa *user* ottenuta dall’oggetto JSON. Successivamente assegna i valori dei campi obbligatori alle rispettive variabili. E in seguito verifica la presenza dei campi facoltativi nell’oggetto JSON e in caso positivo aggiunge i valori presenti, altrimenti attribuisce il valore *null*.

3.3.4.1 *NodeConstraint*

La prima è la classe *NodeConstraint*, i cui campi sono la stringa *nodeType* e un *ArrayList* di stringhe *nodes*. La classe fornisce i metodi *getNodeType()* e *getNodes()*, che restituiscono i valori dei due campi tramutando le stringhe in oggetti *Node*, e il metodo *hasNodes()* che ritorna un valore booleano corrispondente al fatto che l’oggetto

NodeConstraint abbia o meno dei nodi specificati, ossia che il campo *nodes* sia istanziato oppure sia *null*.

3.3.4.2 PredicateConstraint

La seconda classe è *PredicateConstraint*, ed è dotata dei campi stringa *subjectType*, *predicate*, *objectType* e degli *ArrayList* di stringhe *subjects* e *objects*. Similmente alla classe precedente, i metodi implementati sono *getSubjectType()*, *getPredicate()*, *getObjectType()* che restituiscono la stringa corrispondente al campo richiesto, *getSubjects()* e *getObjects()* che invece ritornano una lista di oggetti *Node* ottenuta convertendo le stringhe contenute nelle liste dell'oggetto. Infine, sono presenti i metodi *hasSubjects()* e *hasObjects()* che indicano se i due campi sono istanziati oppure *null*.

3.3.4.3 AttributeConstraint

La terza classe, infine, si chiama *AttributeConstraint*, i cui campi sono le stringhe *subjectType*, *predicate*, *objectType* e *symbol*, e gli *ArrayList* di stringhe *subjects* e *values*. Anche questa classe stabilisce i consueti metodi che restituiscono i valori dei campi, ma a differenza delle altre due, per ritornare i valori del campo *values* sono presenti tre differenti metodi:

- *getValues()* restituisce la lista di stringhe così come è;
- *getNodeValues()* restituisce una lista di oggetti *Node*, ottenuti convertendo le stringhe;
- *getExprValues()* ritorna una lista di oggetti *Expr*, ottenuti ricorrendo ai metodi statici *makeInteger(String s)*, *makeDouble(Double d)*, *makeString(String s)* e *makeDate(String s)* della classe *NodeValue*, che sono nella forma dei letterali RDF, ossia con il datatype *XMLSchema*.

Inoltre, sono definiti tre metodi per il confronto dei valori:

- *compare(String s)*, che confronta il primo (e unico) valore del campo *values* (nel caso in cui la funzione del vincolo abbia bisogno di un solo valore, per esempio la funzione “=”) con il parametro passato e ritorna un valore negativo se il parametro è maggiore, positivo se il parametro è minore e nullo se sono uguali;

- *between(String s)*, che confronta il parametro passato con i due valori presenti nel campo *values* (nel caso in cui la funzione sia “between”) e ritorna il valore *true* se risulta maggiore o uguale del primo e minore o uguale del secondo, *false* altrimenti;
- infine, il metodo *contains(String s)*, che valuta se la stringa passata come parametro è contenuta nel campo *values* (nel caso l’operatore sia “in” o “notin”).

3.3.5 Contenitore di vincoli

Una volta stabilite le classi per rappresentare i vari tipi di vincoli, occorre implementare un’ulteriore classe per contenere la lista dei vincoli. Per questo è stata creata la classe *ConstraintsList*, i cui campi sono tre *ArrayList*, chiamati *nodeConstraints*, *predicateConstraints* e *attributeConstraints*, contenenti oggetti delle tre classi di vincoli.

Il costruttore della classe provvede solamente ad istanziare i nuovi oggetti. Successivamente viene richiamato il metodo *readConstraintsFile(String file)*, che ottiene come parametro il percorso del file da leggere, effettua il parsing della lista di oggetti JSON costruendo un oggetto *JSONArray*, e per ogni oggetto *JSONObject* all’interno della lista valuta il valore del campo *constraint*, che si ricorda può assumere i valori “node”, “predicate” e “attribute”, e aggiunge alla rispettiva lista un nuovo oggetto *Constraint* passando come parametro al costruttore l’oggetto *JSONObject*.

Gli altri metodi implementati sono:

- *getNodeConstraints(String user, Node type)*, che acquisisce una stringa che rappresenta l’utente e un oggetto *Node* corrispondente alla classe di nodi, scandisce la lista *nodeConstraints* e restituisce un *ArrayList* di oggetti *NodeConstraint* i cui campi *user* e *nodeType* corrispondono a quelli passati come parametro. Se durante la scansione viene individuato un vincolo generico, ossia senza lista di nodi, viene restituito interrompendo l’esecuzione del metodo, in quanto vincolo con la massima priorità;
- *getPredicateConstraints(String user, String subjectType, String predicate, String objectType)*, che acquisisce una stringa che rappresenta l’utente e tre stringhe corrispondenti alle classi di nodi soggetto e oggetto e al predicato, analizza gli

elementi di *predicateConstraints* e ritorna un *ArrayList* di oggetti *PredicateConstraint* i cui campi *user*, *subjectType*, *predicate* e *objectType* coincidono con quelli ricevuti in ingresso. Anche per questo tipo di vincoli, il vincolo generico, privo delle liste di nodi soggetto e oggetto, ha la massima priorità, e quindi se viene rilevato nella lista viene restituito, ignorando gli altri vincoli;

- *getAttributeConstraints(String user, String subjectType, String predicate)*, che acquisisce una stringa che rappresenta l'utente e due stringhe corrispondenti alla classe di nodi soggetto e al predicato, verifica la presenza nella lista *attributeConstraints* di vincoli i cui campi *user*, *subjectType* e *predicate* equivalgono ai valori dei parametri e restituisce un *ArrayList* di oggetti *AttributeConstraint*. In questo caso, i vincoli con simbolo "X" sono i più rilevanti perché vietano l'accesso all'attributo e non solo ad una porzione del dominio. Perciò se viene trovato un vincolo con simbolo "X" e senza lista di soggetti, allora questo ha la massima priorità e viene restituito. In assenza, hanno la priorità più alta i vincoli sempre con simbolo "X" ma che hanno una lista di nodi soggetto specificati. Altrimenti vengono ritornati tutti gli altri vincoli individuati.

3.4 *SPARQLQuery*

La classe fondamentale sviluppata per questo progetto è la classe *SPARQLQuery*, costruita per contenere una query SPARQL e fornire i metodi per modificarne gli elementi. I suoi campi sono:

- *user*, una stringa che indica l'utente che ha eseguito la query;
- *inputQuery*, un oggetto della classe *Query* che costituisce la query iniziale, fornita in ingresso al sistema;
- *outputQuery*, un oggetto della classe *Query* che costituisce la query finale, che viene modificata dall'eventuale applicazione dei vincoli;
- *varsTypes*, un *ArrayList* di oggetti *VarTypes*;
- *urisTypes*, un *ArrayList* di oggetti *UriTypes*;
- *predicateTypes*, un *ArrayList* di oggetti *PredicateTypes*.

3.4.1 *VarTypes, UriTypes, PredicateTypes*

Le classi *VarTypes*, *UriTypes* e *PredicateTypes* sono state sviluppate per contenere le variabili e gli URI presenti all'interno della query con i rispettivi possibili valori di classi e predicati.

La classe *VarTypes* rappresenta una variabile soggetto o oggetto di una tripla della query con un elenco di URI costituenti le possibili classi a cui può appartenere. I suoi campi sono un oggetto della classe *Var* denominato *variable* e un *ArrayList* di oggetti *Node* denominato *types*. I metodi implementati sono *getVar()* e *getTypes()* che semplicemente restituiscono i valori dei due campi.

La classe *UriTypes* ha la stessa funzione della classe precedente, a differenza che considera non più le variabili ma gli URI presenti nelle triple nel ruolo di soggetto o oggetto. I suoi campi sono un oggetto della classe *Node* denominato *uri* e un *ArrayList* di oggetti *Node* denominato *types*. Anche in questo caso i metodi *getUri()* e *getTypes()* ritornano i valori dei due campi.

La classe *PredicateTypes*, invece, serve per rappresentare i predicati che si trovano sotto forma di variabili e i possibili URI che possono assumere. I campi di cui è composta sono un oggetto *Var* nominato *predicate* e un *ArrayList* di oggetti *Node* chiamato *types*, e vengono restituiti rispettivamente dai metodi *getPredicate()* e *getTypes()*.

La finalità degli oggetti di queste classi è quella di poter accedere facilmente alle possibili classi di nodi e valori di predicati e costruire tutte le combinazioni di triple per poter valutare e applicare i vincoli.

3.4.2 Costruttore

Il costruttore della classe *SPARQLQuery* riceve in ingresso le stringhe *queryString* e *user* che equivalgono alla query da verificare e all'utente che l'ha formulata, e un oggetto *Model* che rappresenta il dataset su cui eseguire le query che servirà per recuperare gli URI delle classi e dei predicati.

Innanzitutto, il costruttore assegna la stringa *user* al campo *user* e crea due oggetti *Query* tramite il metodo statico *create(String queryString)* della classe *QueryFactory* a partire dalla stringa passata come parametro e li assegna ai campi *inputQuery* e

outputQuery. Questo perché quando viene istanziato l'oggetto *SPARQLQuery* le query iniziale e finale corrispondono.

Il compito successivo è quello di rilevare tutte le variabili e gli URI e trovarne le rispettive classi o predicati. Allora vengono creati tre oggetti *ArrayList*, uno per le variabili soggetto/oggetto (*vars*), uno per le variabili predicato (*predicates*) e uno per gli URI soggetto/oggetto (*uris*). Tramite un *ElementWalker* si analizza il contenuto della query e per ogni tripla vengono aggiunti alle rispettive liste le variabili e gli URI, evitando le ripetizioni.

Nel Listato 24 viene presentata una semplice query che richiede le canzoni dell'album "Let It Be", con relativa lunghezza e scrittori, e i vettori di variabili e URI. La lista dei predicati è vuota in quanto non ci sono variabili predicato.

```
PREFIX :      <http://stardog.com/tutorial/>

SELECT *
WHERE
  { :Let_It_Be :track ?song .
    ?song      :length ?length ;
              :writer ?writer
  }

vars: [?song, ?length, ?writer]
uris: [http://stardog.com/tutorial/Let\_It\_Be]
predicates: []
```

Listato 24: Esempio costruttore Query, vettori di variabili e URI

Poi si realizza un ciclo for che per ogni variabile contenuta nella lista *vars* crea un oggetto *VarTypes* passando come parametri al costruttore l'oggetto *Var* e un *ArrayList* di oggetti *Node* ottenuto richiamando il metodo *getVarTypes(Var v, Model model)*, e lo aggiunge al campo *varsTypes* dell'oggetto *SPARQLQuery*.

Il metodo *getVarTypes()* per prima cosa crea un oggetto *ElementPathBlock* vuoto, a cui aggiunge la tripla seguente:

```
?var rdf:type ?type
```

Dove "var" è la variabile di cui vogliamo trovare le classi e "type" è una variabile che conterrà i risultati del predicato *rdf:type*, ossia le classi della variabile soggetto.

Quindi esegue il metodo `getTypes(Node node, ElementPathBlock block, String resultString, Model model)`, a cui passa come parametri la variabile in questione tramutata in oggetto *Node* con il metodo `asNode()`, il blocco di triple appena costruito, una stringa che indica il nome della variabile oggetto della tripla precedentemente istanziata (ossia “type”) e l’oggetto *Model* ricevuto dal costruttore della classe.

Il metodo `getTypes()` è così costruito per dargli un carattere generale. Infatti, esso è utilizzato anche nelle fasi successive per trovare le classi degli URI e i valori delle variabili predicato. Questo metodo crea un nuovo oggetto *Query*, a cui assegna come unica variabile risultato quella indicata dalla stringa ricevuta come terzo parametro, e imposta la clausola `DISTINCT`. In seguito, effettua una scansione del campo `inputQuery` dell’oggetto e aggiunge al blocco di triple passato come secondo parametro tutte le triple che trova nella query. Stesso discorso per i filtri, che vengono aggiunti interamente alla nuova query. Infine, esegue la query sul dataset passato come quarto parametro e ritorna i risultati ottenuti aggiungendoli all’*ArrayList* di risultato. Si evidenzia che, durante la fase di scansione della query, se venissero individuate delle triple che hanno per soggetto la variabile che si sta trattando, passata come primo parametro, per predicato `rdf:type` e per oggetto un URI, allora si aggiungerebbero alla lista risultato gli URI del campo oggetto e conclusa la scansione si restituirebbe il risultato, in quanto si è determinato che la query specifica già al suo interno la classe o le classi della variabile.

```
var: ?song
types: [http://stardog.com/tutorial/Song]

var: ?length
types: []

var: ?writer
types: [http://stardog.com/tutorial/Band, http://stardog.com/tutorial/Producer,
http://stardog.com/tutorial/SoloArtist, http://stardog.com/tutorial/SongWriter]

uri: http://stardog.com/tutorial/Let\_It\_Be
types: [http://stardog.com/tutorial/Album]
```

Listato 25: Esempio costruttore Query, ricerca classi prima fase

Nel Listato 25 vengono mostrate le classi rilevate dalla prima fase per le variabili e URI della query di cui al Listato 24. La variabile “length” non ha classi poiché conterrà letterali.

Ritornando al metodo *getVarsTypes()*, dopo aver ottenuto la lista di URI con il metodo *getTypes()*, nominata *result*, viene effettuata un’ulteriore scansione del contenuto della query, e per ogni tripla in cui figura la variabile e il predicato è un URI, viene eseguito il metodo *getDomain(Node predicate, Model model)* se la variabile è il soggetto della tripla, oppure il metodo *getRange(Node predicate, Model)* se la variabile è l’oggetto della tripla.

Il metodo *getDomain(Node predicate, Model model)* crea una query in cui l’unica tripla è la seguente:

```
predicate rdfs:domain ?domain
```

dove “predicate” è l’oggetto *Node* passato come parametro. L’esecuzione di questa semplice query va ad interrogare il dataset, ed in particolare va a verificare se è specificato un RDF Schema per il predicato indicato, dal quale rilevare le classi di nodi che possono essere soggetti in una tripla con il predicato in questione.

Analogamente, il metodo *getRange(Node predicate, Model model)* crea una query con la tripla

```
predicate rdfs:range ?range
```

che individua invece le classi di nodi che possono essere oggetti in una tripla con il predicato in questione.

Entrambi i metodi, se il risultato delle query non è vuoto, aggiungono alla lista di classi anche le possibili sottoclassi, richiamando il metodo *getSubClassesOf(ArrayList<Node> classes, Model model)*, che per ogni elemento della lista passata come parametro, crea ed esegue una query la cui unica tripla è la seguente:

```
?subclass rdfs:subClassOf class
```

dove “class” è l’oggetto *Node* scandito dal ciclo for. La ricerca delle sottoclassi avviene ricorsivamente, in caso di gerarchie a più livelli.

```

var: ?song
types: [http://stardog.com/tutorial/Song]

var: ?length
types: []

var: ?writer
types: [http://stardog.com/tutorial/SongWriter]

uri: http://stardog.com/tutorial/Let\_It\_Be
types: [http://stardog.com/tutorial/Album]

```

Listato 26: Esempio costruttore Query, ricerca classi seconda fase

Tornando per l'ultima volta al metodo `getVarTypes()`, ottenuta una lista di classi e sottoclassi dai metodi `getDomain()` e/o `getRange()`, nominata *types*, si procede con la rimozione dalla lista *result* di tutti le classi che non compaiono anche nella lista *types*, se quest'ultima non è vuota, e infine si restituisce l'oggetto *ArrayList* contenente gli URI delle possibili classi della variabile. Questo procedimento con doppio passaggio per individuare le classi e poi confrontare i risultati si rivela necessario in quanto la prima fase determina una serie di classi meno precise, poiché restituisce tutte le classi a cui appartengono i nodi indicati ma con il rischio che una o più di queste non sia prevista come soggetto o oggetto di una particolare tripla contenuta nella query, mentre la seconda fase determina solo ed esclusivamente le classi ammesse per i predicati presenti, ma non si ha sicurezza che nel dataset sia sempre definito un RDF Schema completo. In questo modo, invece, si riesce a determinare in maniera più accurata l'elenco di classi, permettendo alle fasi successive di evitare iterazioni non necessarie ed eventuali malfunzionamenti.

A riprova di ciò, nell'esempio svolto riguardante la query del Listato 24, la prima fase ha individuato per la variabile "writer" quattro classi, mentre la seconda ne ha individuata una sola, mostrata nel Listato 26, analizzando l'RDF Schema del predicato "writer", riportata nel Listato 27.

```

:writer a rdf:Property ;
  rdfs:label "writer" ;
  rdfs:comment "A person or a group of people who participated in the creation of song." ;
  rdfs:domain :Song ;
  rdfs:range :Songwriter .

```

Listato 27: Esempio costruttore Query, RDF Schema proprietà :writer

La procedura per determinare le possibili classi degli URI è sostanzialmente la stessa. Per ogni URI contenuto nella lista individuata si aggiunge al campo *urisTypes* un nuovo oggetto *UriTypes* creato passando al costruttore l'oggetto *Node* e un *ArrayList* di oggetti *Node* ottenuto eseguendo il metodo *getUriTypes(Node uri, Model model)*, in cui i passaggi sono gli stessi del metodo *getVarTypes()*, a differenza che la tripla inserita inizialmente è

```
uri rdf:type class
```

Dove “uri” è il nodo di cui si stanno cercando le classi.

Infine, per ogni variabile predicato nella lista individuata viene creato un oggetto *PredicateTypes* generato passando al costruttore l'oggetto *Var* corrispondente al predicato e un *ArrayList* di oggetti *Node* ricavato dal metodo *getPredicateTypes(Var predicate, Model model)*, e successivamente lo aggiunge al campo *predicatesTypes*.

Anche in questo caso, il metodo *getPredicateTypes()* sfrutta il metodo *getTypes()* già descritto, passandogli però un oggetto *ElementPathBlock* vuoto, poiché nei precedenti casi il risultato da trovare erano le classi di un nodo variabile, mentre ora occorre trovare gli URI della stessa variabile predicato, e quindi non è necessario aggiungere alcuna tripla alla query iniziale. Perciò il metodo *getTypes()* di fatto esegue la query originale, e restituisce i valori distinti che il predicato variabile può assumere nella query e restituisce la lista al metodo *getPredicateTypes()*.

Con questa procedura, i campi dell'oggetto *SPARQLQuery* sono completati e l'oggetto stesso consente al sistema di poter risalire rapidamente alle variabili e URI presenti all'interno della query, alle classi di soggetti e oggetti e ai possibili valori delle variabili predicato, senza l'onere di dover ripetutamente analizzare il contenuto dell'oggetto *Query*.

3.4.3 Metodi per ottenere gli elementi dell'oggetto *Query*

Successivamente al costruttore e ai metodi ausiliari ad esso, sono stati implementati una serie di metodi che restituiscono i vari elementi della query e che risultano utili per semplificare determinati passaggi nei metodi successivi:

- I metodi *getUser()*, *getInputQuery()* e *getOutputQuery()* sono i più semplici, poiché restituiscono gli oggetti *String* e *Query* contenuti nei campi *user*, *inputQuery* e *outputQuery* dell'oggetto *SPARQLQuery*;
- Poi sono presenti i metodi *getNodeVars()* e *getNodeUris()* che restituiscono tutte le variabili e tutti gli URI corrispondenti ai nodi soggetto o oggetto contenuti nella query scansionando le liste *varsTypes* e *urisTypes* e ricavando i risultati dagli oggetti *VarTypes* e *UriTypes* con i metodi *getVar()* e *getUri()*.
Associati a questi due, ci sono i metodi *getNodeTypesByVar(Var v)* e *getNodeTypesByUri(Node uri)* che ritornano l'elenco delle possibili classi dell'oggetto specificato ricercando gli oggetti *VarTypes* e *UriTypes* i cui campi *variable* e *uri* corrispondono a quelli passati come parametro e ricavando la lista di oggetti *Node* con il metodo *getTypes()*;
- Allo stesso modo, il metodo *getPredicateTypesByVar(Var predicate)* scansiona la lista *predicatesTypes* cercando l'oggetto *PredicateTypes* il cui campo *predicate* equivale alla variabile passata come parametro e restituisce il relativo elenco di oggetti *Node* che rappresentano i possibili URI che la variabile predicato può assumere;
- Per quanto riguarda le triple, è disponibile il metodo *getTriples()* che restituisce l'elenco di tutte le triple presenti nella query analizzando il contenuto dell'oggetto *outputQuery* con un *ElementWalker* e aggiungendo alla lista risultato tutti gli oggetti *TriplePath* contenuti nell'elemento *ElementPathBlock*. Inoltre, sono presenti i metodi *getTriplesByVar(Var v)* e *getTriplesByUri(Node uri)* che leggono tutte le triple fornite dal metodo precedente e restituiscono quelle in cui il soggetto o l'oggetto equivale al parametro specificato;
- Anche per i filtri è disponibile il metodo *getFilters()* che restituisce l'elenco di tutti i filtri presenti nell'oggetto *outputQuery*;
- Infine, sempre riguardante i filtri, vi è il metodo *getVarsByFilter(ElementFilter filter)* che restituisce l'elenco delle variabili contenute nel filtro passato come parametro. Questo metodo estrae l'oggetto *ExprFunction* dall'oggetto *ElementFilter* e lo passa al metodo *getVarsByFunc(ExprFunction expr)*, il quale analizza ciascuno degli argomenti dell'espressione: se l'argomento è una costante lo ignora; se l'argomento è una variabile lo aggiunge all'*ArrayList*

result; se l'argomento è una funzione richiama ricorsivamente il metodo *getVarsByFunc()*; infine, se l'argomento è una funzione aggregata richiama il metodo *getVarsByAgg(ExprAggregator agg)* passandogli come parametro l'espressione dell'argomento dopo aver effettuato il casting verso la classe *ExprAggregator*. Il metodo *getVarByAgg()* analizza a sua volta gli argomenti dell'espressione, e come prima: se l'argomento è una costante lo ignora; se l'argomento è una variabile lo aggiunge all'*ArrayList result*; infine, se l'argomento è una funzione richiama il metodo *getVarsByFunc()*. Il caso in cui l'argomento sia una funzione aggregata non è previsto in quanto la query non ammette funzioni aggregate composte con altre funzioni aggregate. Tutti e tre i metodi restituiscono un *ArrayList* di oggetti *Var*. Riassumendo, a partire dal metodo *getVarsByFilter()* il sistema inizia una ricerca “ad albero”, analizzando gli argomenti di tutte le funzioni composte fino a raggiungere i componenti “foglia”.

3.4.4 Metodi per modificare l'oggetto *Query*

L'applicazione di vincoli ad una query può comportare l'aggiunta di un filtro oppure la rimozione di variabili o triple. Nei prossimi sottoparagrafi si descrivono i metodi implementati per soddisfare queste funzioni.

3.4.4.1 Rimuovere variabili risultato

Il primo metodo è *removeResultVar(Var var)*, il cui fine è quello di rimuovere la variabile specificata dalla clausola *SELECT* della query. La classe *Query* non fornisce dei metodi per rimuovere una variabile dalle variabili risultato o per impostarle da zero; perciò, l'unica via percorribile è quella di costruire una nuova query aggiungendo tutte le variabili risultato meno quella da rimuovere, più tutti gli altri elementi presenti nella query.

Quindi il metodo inizia con il creare un oggetto *Query* vuoto, imposta il tipo *SELECT* della query e aggiunge la mappa dei prefissi presa dalla query *outputQuery*.

Successivamente, scansiona la lista delle variabili risultato della query *outputQuery*, ottenuta con il metodo *getProject()* che restituisce un oggetto *VarExprList*, e per ognuna

di queste valuta se la variabile equivale a quella da rimuovere, in caso di variabile semplice, oppure, in caso di funzione, se la lista delle variabili contenute nell'espressione, ottenute con i metodi *getVarsByFunc()* o *getVarsByAgg()* a seconda che la funzione sia aggregata o meno, contiene la variabile da rimuovere. In questi casi, il metodo prosegue il ciclo trascurando la variabile o la funzione, mentre se le due condizioni non sono soddisfatte aggiunge la variabile o l'espressione alla clausola SELECT della query con il metodo *addResultVar()*.

Poi il metodo si occupa di copiare nella nuova query il resto degli elementi, ossia le triple, i filtri e le altre eventuali clausole (DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET, ORDER BY), e infine assegna al campo *outputQuery* il riferimento al nuovo oggetto *Query*.

3.4.4.2 Rimuovere triple

Il secondo metodo è *removeTriple(TriplePath triple)*, che si occupa di rimuovere la tripla specificata dalla query. Anche in questo caso non esiste un metodo per rimuovere direttamente una tripla dall'oggetto *Query*. Perciò viene costruito un nuovo oggetto *ElementPathBlock* vuoto, a cui si aggiungono tutte le triple presenti nella query, ottenute con il metodo *getTriples()*, ad eccezione di quella passata come parametro. Poi si crea un contenitore *ElementGroup* a cui si aggiunge l'elemento *ElementPathBlock* costruito e tutti i filtri presenti nella query ottenuti tramite il metodo *getFilters()*. Infine si imposta il pattern dell'oggetto *outputQuery* con il nuovo elemento, passato come parametro al metodo *setQueryPattern(Element element)*.

Quando si rimuove una tripla, questa azione può comportare degli effetti a cascata. Perciò il metodo, prima di esaurirsi, richiama altri tre metodi ausiliari:

- Uno di questi è *removeResultVarsNotInTriples()*, il cui scopo è rimuovere dalle variabili risultato le variabili che non compaiono più nelle triple della query. Concretamente, il metodo crea una lista di variabili risultato analizzando l'oggetto *VarExprList* ottenuto dal campo *outputQuery*, e per ognuno degli oggetti *Var* contenuti nella lista esegue il metodo *getTriplesByVar(Var v)* che ritorna l'elenco delle triple in cui compare la variabile in oggetto. Se l'elenco è

vuoto, viene rimossa la variabile risultato richiamando il metodo *removeResultVar(Var v)*.

- Un altro metodo è *removeVarsUrisPredicatesNotInQuery()*. Esso si occupa di mantenere aggiornate le liste *varsTypes*, *urisTypes* e *predicatesTypes* rimuovendo le variabili e gli URI che non compaiono più nelle triple della query. Per fare questo crea tre *ArrayList*, uno per le variabili soggetto/oggetto, uno per gli URI soggetto/oggetto e uno per le variabili predicato, a cui aggiunge tutti gli elementi corrispondenti trovati nelle triple attualmente presenti nella query. Poi confronta queste liste con i tre campi dell'oggetto *SPARQLQuery*: se la variabile dell'oggetto *VarTypes* o *PredicateTypes* oppure l'URI dell'oggetto *UriTypes* non compare nella rispettiva lista, allora lo rimuove.
- L'aggiornamento delle liste di variabili e URI è utile soprattutto per il metodo *removeFiltersWithVarsNotInTriples()*, che ha l'obiettivo di rimuovere i filtri che contengono variabili che non esistono più nelle triple della query. Il metodo esegue un ciclo for sulla lista di oggetti *ElementFilter* ottenuta dal metodo *getFilters()*. Per ciascuno di essi, confronta le variabili contenute all'interno del filtro, identificate con il metodo *getVarsByFilter(ElementFilter filter)*, con le variabili delle liste *varsTypes* e *predicatesTypes*, e se almeno una variabile del filtro non è presente nelle variabili delle triple della query, allora il filtro viene aggiunto alla lista dei filtri da rimuovere. Terminato il ciclo for, tutti i filtri individuati vengono rimossi.

3.4.4.3 Rimuovere filtri

Il metodo per rimuovere i filtri è *removeFilter(ElementFilter filter)*, che analogamente al metodo *removeTriple()*, crea un contenitore *ElementGroup* in cui inserisce tutte le triple e i filtri della query escluso il filtro passato per parametro, e lo imposta come nuovo pattern dell'oggetto *outputQuery*.

3.4.4.4 Aggiungere filtri

Viceversa, per aggiungere un filtro, il metodo implementato è *addFilter(ElementFilter filter)*, che deve valutare se il filtro è già presente oppure se sono presenti altri filtri che contengono la stessa variabile, e in tal caso cercare di combinarli per ottenere un unico

filtro. Il metodo quindi per prima cosa costruisce l'oggetto vuoto della classe *ElementGroup* e il blocco di triple *ElementPathBlock* copiando tutte quelle contenute nella query. Poi esegue un ciclo for sulla lista dei filtri ottenuta con il metodo *getFilters()*, e confronta ciascuno di questi con quello passato come parametro:

- Se i due filtri sono uguali, allora viene impostata a *true* la variabile booleana *filterAlreadyExists*;
- Se il filtro della query contiene la stessa variabile del filtro da aggiungere, allora viene inserito in una lista di filtri da combinare;
- Altrimenti, se il filtro contiene variabili differenti oppure si tratta di un filtro NOT EXISTS, viene aggiunto al contenitore *ElementGroup*.

Ulteriori condizioni per poter combinare più filtri sono:

- Ciascuno di essi, quindi sia quello da aggiungere che quelli già presenti, deve contenere una sola variabile, anche ripetuta in caso di funzioni composte, in quanto con più variabili si rischierebbe di mettere insieme condizioni appartenenti a variabili differenti;
- La variabile del filtro da aggiungere non deve comparire in triple il cui predicato è variabile, poiché la variabile potrebbe appartenere a classi o tipi diversi e quindi si rischierebbe di valutare non correttamente la combinazione dei valori delle funzioni.



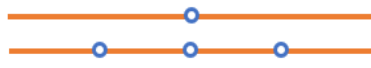

Al termine del ciclo for:




- Se la lista dei filtri con la stessa variabile è vuota, oppure se il filtro da aggiungere è di tipo NOT EXISTS e non è già presente, il filtro viene aggiunto al contenitore;
- Se la lista contiene un solo elemento ma questo equivale a quello da aggiungere, allora non si effettua alcuna modifica;
- Altrimenti, i filtri nella lista vengono rimossi dalla query (concretamente non vengono aggiunti al corpo della query), il filtro da aggiungere viene inserito nella lista, e viene creato un nuovo oggetto *ElementFilter* ottenuto combinando la lista di filtri tramite il metodo *combineFilters(ArrayList<ElementFilter> filters)*.

Il metodo *combineFilters()* riceve una lista di almeno due oggetti *ElementFilter*. Crea un oggetto *Expr* denominato *expr* che conterrà l'espressione utilizzata per costruire l'oggetto *ElementFilter* da restituire. Inizialmente alla variabile *expr* viene assegnata l'espressione del primo filtro contenuto nella lista ottenuta con il metodo *getExpr()*. Successivamente si esegue un ciclo *for* che scandisce la lista dei filtri a partire dal secondo, e si confronta l'oggetto *expr* con l'espressione del filtro dell'iterazione corrente. Innanzitutto, si verifica che l'oggetto *expr* non contenga il valore *null*. In caso contrario, il metodo termina restituendo il valore *null* poiché una combinazione di filtri ha determinato una condizione impossibile. Inoltre, se almeno uno dei simboli di funzione delle due espressioni corrisponde alle funzioni logiche AND, OR, NOT, allora si effettua un AND logico tra le due espressioni, si assegna l'espressione risultante alla variabile *expr* e si procede con la successiva iterazione. Questo perché il metodo è in grado di combinare solamente due vincoli semplici, ossia costituiti da una variabile, un operatore e un valore o lista di valori.

Se non si verificano le due situazioni precedenti, si procede con il confrontare le espressioni in base ai relativi simboli di funzione e ad assegnare alla variabile *expr* il risultato della combinazione dei due filtri.

A titolo di esempio, si riportano in Tabella 4 le possibili combinazioni di espressioni assumendo che la seconda espressione abbia simbolo di funzione *NOT IN*.

=	Se la costante della prima espressione non è contenuta nella lista delle costanti della seconda espressione allora l'espressione combinata è la prima espressione, altrimenti si restituisce il valore <i>null</i> .	
		
≠	Se la costante della prima espressione è contenuta nella lista delle costanti della seconda espressione allora l'espressione combinata è la seconda espressione, altrimenti l'espressione combinata è la seconda espressione aggiungendo alla lista delle costanti la costante della prima espressione.	
		

>	<p>Se nessuna delle costanti della seconda espressione è maggiore della costante della prima espressione, allora l'espressione combinata è la prima espressione, altrimenti l'espressione combinata è la congiunzione logica della prima espressione con una nuova espressione con simbolo \neq se una sola costante è maggiore oppure con simbolo <i>NOT IN</i> se più costanti sono maggiori.</p>
	
\geq	<p>Se nessuna delle costanti della seconda espressione è maggiore o uguale della costante della prima espressione allora l'espressione combinata è la prima espressione, altrimenti l'espressione combinata è la congiunzione logica della prima espressione con una nuova espressione con simbolo \neq se una sola costante è maggiore o uguale oppure con simbolo <i>NOT IN</i> se più costanti sono maggiori o uguali.</p>
	
<	<p>Se nessuna delle costanti della seconda espressione è minore della costante della prima espressione, allora l'espressione combinata è la prima espressione, altrimenti l'espressione combinata è la congiunzione logica della prima espressione con una nuova espressione con simbolo \neq se una sola costante è minore oppure con simbolo <i>NOT IN</i> se più costanti sono minori.</p>
	
\leq	<p>Se nessuna delle costanti della seconda espressione è minore o uguale della costante della prima espressione, allora l'espressione combinata è la prima espressione, altrimenti l'espressione combinata è la congiunzione logica della prima espressione con una nuova espressione con simbolo \neq se una sola costante è minore o uguale oppure con simbolo <i>NOT IN</i> se più costanti sono minori o uguali.</p>








			
<i>IN</i>	<p>Se una sola costante della prima espressione non è contenuta nella lista delle costanti della seconda espressione allora l'espressione combinata è una nuova espressione con simbolo = e la costante individuata, se invece più costanti non sono contenute nella lista allora l'espressione combinata è una nuova espressione con simbolo <i>IN</i> e la lista di costanti individuate, altrimenti si restituisce il valore <i>null</i>.</p>		
			
<i>NOT IN</i>	<p>L'espressione combinata è una nuova espressione con simbolo <i>NOT IN</i> e con lista di costanti l'unione delle liste delle due espressioni, tolte le ripetizioni.</p>		
			

Tabella 4: Combinazioni di espressioni con l'espressione *NOT IN*

I valori costanti ottenuti dalle espressioni sono istanze della classe *NodeValue*, che si ricorda essere una superclasse generica. Per confrontare due oggetti *NodeValue* è stato implementato il metodo *compareNodeValues(NodeValue x, NodeValue y)*, che verifica la tipologia di valore (*Integer*, *Double*, *String* o *Date*) e restituisce un intero positivo se il primo parametro è maggiore del secondo, negativo se minore e nullo se sono uguali.

Il metodo *combineFilters()*, quindi, può restituire al metodo *addFilter()* un oggetto *ElementFilter*, e in tal caso questo viene aggiunto alla query, oppure può restituire un valore *null*, poiché la combinazione dei filtri potrebbe aver portato ad una condizione impossibile, e perciò, oltre ad aver già rimosso tutti i filtri contenenti la variabile e non avere aggiunto il nuovo filtro, occorre anche rimuovere tutte le triple in cui figura la variabile in questione, in quanto l'utente non può più accedervi.

3.5 *SPARQLParser*

Dopo aver descritto la classe *SPARQLQuery*, incaricata di rappresentare una query nel sistema sviluppato, occorre definire un'altra classe fondamentale, la classe *SPARQLParser*, il cui compito è acquisire una query SPARQL, recuperare i vincoli attribuiti all'utente che ha formulato la query e applicarli, restituendo una query modificata e autorizzata dalle policy.

I campi della classe sono due: un oggetto *Model* denominato *model*, che contiene il dataset su cui eseguire le query, e un oggetto *ConstraintsList* chiamato *constraintsList*, che contiene l'elenco dei vincoli disponibili per il dataset in questione.

Il costruttore, perciò, riceve come parametri in ingresso le stringhe *datasetFile* e *constraintsFile*, che rappresentano rispettivamente il percorso del file in formato Turtle contenente il dataset e il percorso del file in formato JSON contenente la lista dei vincoli. Al campo *model* viene assegnato un nuovo oggetto *Model*, costruito con il metodo *loadModel(String file)* della classe *RDFDataMgr*, mentre al campo *constraintsList* viene assegnato un nuovo oggetto vuoto della classe *ConstraintsList*, e successivamente si esegue il metodo *readConstraintsFile(String file)* che importa nell'oggetto tutti i vincoli, separandoli in tre liste, una per ogni tipologia di vincolo.

Dopo essere stato istanziato, l'oggetto *SPARQLParser* attende che venga eseguito il metodo *parseQuery(String queryString, String user)*, il quale crea un oggetto *SPARQLQuery* nominato *sparqlQuery* passando al costruttore le stringhe rappresentanti query e utente e il dataset contenuto nel campo *model*. In seguito, procede con la verifica dei vincoli.

3.5.1 Verifica e applicazione dei vincoli sui nodi

Il metodo *parseQuery()* realizza un ciclo for che scandisce l'elenco delle variabili soggetto o oggetto presenti nell'oggetto *SPARQLQuery* e per ciascuna di esse esegue il metodo *checkVarNodeConstraints(Var v, SPARQLQuery sparqlQuery)*. Questo metodo crea un *ArrayList* di oggetti *ElementFilter* vuoto, che conterrà i filtri da aggiungere alla query al termine del metodo, e valuta per ogni classe del nodo variabile la presenza di vincoli *NodeConstraints*, ricavandoli tramite il metodo *getNodeConstraints()* applicato

al campo *constraintsList* del parser. Se la lista di vincoli è vuota si procede con la successiva iterazione. Se invece contiene degli elementi, allora si distinguono due casi:

- Se il primo vincolo della lista non ha nodi specificati, ossia ha validità sull'intera classe, allora è sicuramente l'unico elemento della lista, e viene aggiunto all'elenco dei filtri un nuovo filtro NOT EXISTS

```
FILTER NOT EXISTS {?var a class}
```

dove “var” è la variabile passata come parametro e “class” è la classe del nodo variabile della corrente iterazione, il quale implica che la query non può processare i nodi di quella classe, e incrementa il contatore *contNotExists*;

- Altrimenti, la lista può contenere uno o più vincoli con nodi specificati, e in tal caso si aggiunge all'elenco dei filtri un nuovo filtro NOT IN con variabile quella passata come parametro e lista di valori l'insieme degli URI contenuti nel campo *nodes* dei vincoli.

Al termine delle iterazioni, se la lista delle classi del nodo variabile non è vuota e la sua dimensione corrisponde al contatore *contNotExists*, ciò comporta che la variabile non ha classi a cui l'utente può accedere, e quindi occorre rimuovere tutte le triple nella query che la contengono e rimuovere la variabile dalle variabili risultato se presente. In caso contrario, si aggiungono alla query tutti i filtri presenti nella lista costituita.

Il metodo *parseQuery()* poi ripete la stessa procedura per tutti gli URI soggetto o oggetto contenuti nella query, richiamando il metodo *checkUriNodeConstraints(Node uri, SPARQLQuery sparqlQuery)*, con l'unica differenza che se i vincoli hanno una lista di nodi specificati, allora se l'URI è contenuto in esse vengono rimosse tutte le triple contenenti l'URI, mentre in precedenza veniva creato un filtro NOT IN.

3.5.2 Verifica e applicazione dei vincoli sui predicati

Dopo i vincoli sui nodi, il metodo *parseQuery()* verifica i vincoli sui predicati realizzando un ciclo for che scandisce le triple della query e per ciascuna di esse richiama il metodo *checkPredicateConstraints(TriplePath triple, SPARQLQuery sparqlQuery)*. Per prima cosa il metodo crea un *ArrayList* di oggetti *ElementFilter*, il cui scopo è contenere i filtri da aggiungere alla query al termine del metodo, e istanzia tre *ArrayList* di oggetti *Node*:

- *subjectTypes*, che contiene le classi del nodo soggetto, ottenute con il metodo *getNodeTypesByVar()* se il soggetto è una variabile o *getNodeTypesByUri()* se il soggetto è un URI;
- *predicateTypes*, che contiene i possibili valori del predicato ricavati con il metodo *getPredicateTypesByVar()* se il predicato è variabile, altrimenti l'URI contenuto nella tripla;
- *objectTypes*, che contiene le classi del nodo oggetto, ottenute con il metodo *getNodeTypesByVar()* se l'oggetto è una variabile o *getNodeTypesByUri()* se l'oggetto è un URI.

Poi esegue tre cicli for annidati, uno per ciascuna delle tre liste precedenti, che scansionano tutte le possibili combinazioni di predicato, classe di soggetto e classe di oggetto, in questo ordine.

All'inizio dei cicli per le classi di soggetto e di oggetto, viene effettuata la verifica che la classe di soggetto e la classe di oggetto in questione possano essere rispettivamente nel dominio e nel range del predicato dell'iterazione corrente, riscontrando che i suddetti siano contenuti negli *ArrayList* restituiti dai metodi *getDomain()* e *getRange()*. Se non sono presenti, allora si procede con la successiva iterazione.

Successivamente si ottiene la lista dei vincoli richiamando il metodo *getPredicateConstraints()*. Se questa è vuota, si procede con la successiva iterazione. Se invece contiene degli elementi, allora si distinguono due casi:

- Se il primo vincolo non ha né lista di nodi soggetto né lista di nodi oggetto, ciò significa che il vincolo ha validità su tutti i nodi delle classi di soggetto e di oggetto e che è anche l'unico elemento della lista; in tal caso: se il predicato della tripla è una variabile allora viene aggiunto all'elenco dei filtri un filtro NOT EXISTS la cui tripla è composta dal soggetto della tripla (variabile o URI), dal predicato dell'iterazione corrente e dall'oggetto della tripla, e si incrementa il contatore *contNotExists*; se invece il predicato è un URI si rimuove la tripla e si conclude il metodo;
- Altrimenti, si realizza un ciclo for sulla lista dei vincoli, che può contenere uno o più elementi. Per ogni vincolo che possiede o la lista di nodi soggetto o la lista di

nodi oggetto specificati, questi vengono raccolti in due *ArrayList*, *subjects* e *objects*, e:

- se il soggetto o l'oggetto è una variabile, allora viene aggiunto all'elenco dei filtri un nuovo filtro NOT IN con la variabile del soggetto o dell'oggetto e la lista di URI individuata;
- viceversa, se il soggetto o l'oggetto è un URI, se questo è presente nella rispettiva lista di URI vietati, allora si rimuove la tripla e si conclude il metodo.

Per ogni vincolo, invece, che possiede sia il campo *subjects* che il campo *objects*, si distinguono differenti casi:

- Se soggetto e oggetto sono variabili, allora si aggiunge all'elenco dei filtri un nuovo filtro AND negato (NOT AND) tra l'espressione NOT IN con la variabile del soggetto e la lista degli URI soggetto del vincolo e l'espressione NOT IN con la variabile dell'oggetto e la lista degli URI oggetto del vincolo. In questo modo, la query non può accedere alle triple per cui valgono entrambe le condizioni allo stesso tempo;
- Se il soggetto è una variabile e l'oggetto è un URI, allora se l'URI dell'oggetto è contenuto nella lista degli URI oggetto vietati si aggiunge all'elenco dei filtri un filtro NOT IN con la variabile del soggetto e la lista di URI soggetto del vincolo;
- Se il soggetto è un URI e l'oggetto è una variabile, allora se l'URI del soggetto è contenuto nella lista degli URI soggetto vietati si aggiunge all'elenco dei filtri un filtro NOT IN con la variabile dell'oggetto e la lista di URI oggetto del vincolo;
- Infine, se sia soggetto che oggetto sono URI ed entrambi sono contenuti nelle rispettive liste di URI vietati, allora se il predicato è una variabile si aggiunge all'elenco dei filtri un filtro NOT EXISTS con soggetto e oggetto della tripla e il predicato dell'iterazione corrente e si incrementa il contatore *contNotExists*, altrimenti si rimuove la tripla e si conclude il metodo.

Al termine delle iterazioni, se il contatore *contNotExists* non è nullo ed equivale alla dimensione dell'*ArrayList predicateTypes*, allora si rimuove la tripla, altrimenti si aggiungono alla query tutti i filtri contenuti nell'elenco costruito.

3.5.3 Verifica e applicazione dei vincoli sugli attributi

Per ultimo, il metodo *parseQuery()* verifica i vincoli sugli attributi eseguendo un ciclo for sull'elenco delle triple contenute nella query e richiamando per ciascuna il metodo *checkAttributeConstraints(TriplePath triple, SPARQLQuery sparqlQuery)*. Anche in questo caso vengono recuperate le classi del soggetto e i valori del predicato e vengono eseguiti due cicli for annidati, il primo per i predicati e il secondo per le classi del nodo soggetto. L'esecuzione dell'iterazione prosegue se la classe di soggetto risulta nel dominio del predicato dell'iterazione corrente e se la lista di vincoli ottenuta con il metodo *getAttributeConstraints()* non risulta vuota, altrimenti si passa all'iterazione successiva.

Ottenuta la lista di vincoli, si procede ad analizzarli singolarmente in base al simbolo della funzione e si crea un *ArrayList* di oggetti *ElementFilter* che conterrà i filtri da aggiungere alla query. Per i vincoli con simbolo "X", ossia quelli che impediscono l'accesso all'attributo:

- Se il vincolo si applica ad una lista di nodi soggetto specifici:
 - Se il soggetto della tripla è una variabile si aggiunge all'elenco dei filtri un filtro NOT IN con la variabile soggetto e la lista di nodi del vincolo;
 - Se il soggetto è un URI ed è contenuto nella lista di nodi del vincolo e il predicato della tripla è una variabile, allora si aggiunge all'elenco dei filtri un filtro NOT EXISTS con soggetto e oggetto della tripla e il predicato dell'iterazione corrente e si incrementa il contatore *contNotExists*;
 - Se il soggetto è un URI ed è contenuto nella lista di nodi del vincolo e il predicato della tripla è un URI, allora si rimuove la tripla e si conclude il metodo;
- Se invece il vincolo si applica all'intera classe di nodi soggetto:

- Se il predicato è variabile si aggiunge all'elenco dei filtri un filtro NOT EXISTS con soggetto e oggetto della tripla e il predicato dell'iterazione corrente e si incrementa il contatore *contNotExists*;
- Se invece il predicato è un URI si rimuove la tripla e si conclude il metodo.

Per tutti gli altri simboli di funzione, si distinguono i seguenti casi:

- Se l'oggetto della tripla è una variabile allora si aggiunge all'elenco un nuovo filtro corrispondente al simbolo di funzione con la variabile dell'oggetto e i valori del vincolo;
- Se l'oggetto è un letterale e rientra nei valori vietati dal vincolo:
 - Se il predicato è variabile si aggiunge all'elenco dei filtri un filtro NOT EXISTS con soggetto e oggetto della tripla e il predicato dell'iterazione corrente e si incrementa il contatore *contNotExists*;
 - Se invece il predicato è un URI si rimuove la tripla e si conclude il metodo.

Terminati i cicli sulle combinazioni predicato-classe di nodo soggetto, se il contatore *contNotExists* non è nullo ed equivale al prodotto tra le dimensioni del vettore di predicati e del vettore di classi del nodo soggetto, allora si rimuove la tripla, altrimenti si aggiungono alla query tutti i filtri contenuti nell'elenco costruito.

3.6 Funzionamento

Dopo aver descritto tutte le classi, con i relativi campi e metodi, utilizzate nel progetto, si presenta una sintetica descrizione del funzionamento nel suo intero.

Il sistema innanzitutto richiede l'inserimento delle stringhe corrispondenti ai percorsi dei file del dataset e dei vincoli. Con questi costruisce l'oggetto *SPARQLParser*.

Poi il sistema acquisisce una query testuale e l'utente che l'ha eseguita. Richiamando il metodo *parseQuery()* dell'oggetto *SPARQLParser*, viene costruito un oggetto *SPARQLQuery*, che contiene all'interno un oggetto *Query* iniziale, un oggetto *Query* finale che verrà modificato con i vincoli, e delle liste che indicano le classi delle

variabili, le classi degli URI e i valori delle variabili predicato contenuti nella query, costruite interrogando il dataset, ed in particolare l’RDF Schema.

Poi il parser procede con la verifica dei vincoli. Innanzitutto, si valutano i vincoli sui nodi, che vietano l’accesso a determinate classi di nodi o a specifici nodi. Poi si passa ai vincoli sui predicati, che limitano l’accesso ai nodi soggetto o oggetto di una tripla che contiene un determinato predicato. Si termina con i vincoli sugli attributi, che vietano l’accesso a particolari attributi individuati da coppie classe di soggetto – predicato specificate, oppure che ne limitano il dominio di valori che possono assumere.

L’applicazione dei vincoli alla query può comportare la rimozione di triple e di variabili risultato, oppure l’introduzione di filtri, che sotto determinate condizioni, possono essere combinati per dare un filtro unico. Ciascuno di questi passaggi viene specificato all’utente.

Infine, il sistema restituisce una query finale, mostrando le eventuali differenze con la query iniziale.

Capitolo 4: Casi d'uso

4.1 Descrizione del dataset

Per effettuare la fase di testing del sistema è stato utilizzato un dataset a tema musicale [10], in cui sono rappresentati artisti, band, album, produttori, canzoni e scrittori. In Figura 4 un estratto del dataset in rappresentazione grafica mentre in Figura 5 lo stesso in formato Turtle semplificato.

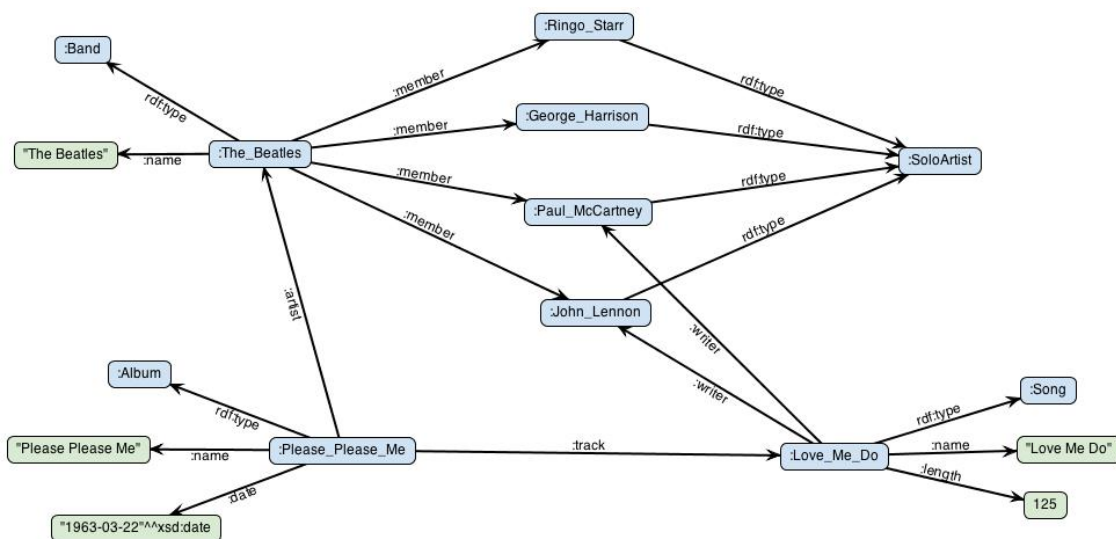


Figura 4: Rappresentazione grafica del dataset [10]

I nodi principali sono i *SoloArtist*, dotati di un nome e di una descrizione, e le *Band*, anch'esse costituite da nome e descrizione, e collegate ai propri membri, istanze della classe *SoloArtist*, tramite il predicato *member*. Entrambi sono sottoclassi di *Artist*.

Poi ci sono gli *Album*, caratterizzati da nome, descrizione e data di pubblicazione. Ciascun *Album* è in relazione con il proprio artista tramite il predicato *artist*, il cui range è la classe *Artist*, superclasse di *SoloArtist* e *Band*. Inoltre, ogni *Album* è prodotto da uno o più *Producer*, identificati con il predicato *producer*. Infine, gli *Album* prevedono un elenco di canzoni, della classe *Song*, composte da un nome, una descrizione, una lunghezza in secondi e uno o più *SongWriter*, individuati con il predicato *writer*.

```

:the_beatles      a :Band ;
                  :name "The Beatles" ;
                  :member :John_Lennon , :Paul_McCartney , :George_Harrison , :Ringo_Starr .

:John_Lennon     a :SoloArtist .
:Paul_McCartney  a :SoloArtist .
:Ringo_Starr     a :SoloArtist .
:George_Harrison a :SoloArtist .
:Please_Please_Me a :Album ;
                  :name "Please Please Me" ;
                  :date "1963-03-22"^^xsd:date ;
                  :artist :the_beatles ;
                  :track :Love_Me_Do .

:Love_Me_Do      a :Song ;
                  :name "Love Me Do" ;
                  :length 125 ;
                  :writer :John_Lennon , :Paul_McCartney .

```

Figura 5: Rappresentazione in formato Turtle semplificato del dataset [7]

Un nodo può essere istanza di più classi: in particolare, un *SoloArtist* e una *Band* possono essere allo stesso tempo *Producer* e *Songwriter*, mentre non è possibile che un nodo appartenga sia alla classe *SoloArtist* che *Band*.

Il dataset è dotato di un RDF Schema completo che definisce classi, sottoclassi e proprietà con i relativi domini e codomini.

Nei prossimi paragrafi vengono presentati una serie di esempi di casi d'uso, che mostrano l'effettivo funzionamento del sistema realizzato.

4.2 Esempi: vincoli sui nodi

La query del primo esempio, Listato 30, richiede l'accesso agli *Album* e ai relativi *Artist*. L'utente *u1* però non è autorizzato ad accedere alle *Band* e ad alcuni *SoloArtist*, come indicato nei Listati 28 e 29. Il sistema aggiunge quindi un filtro NOT EXISTS in cui impone che la variabile non possa essere istanza della classe *Band* e un filtro NOT IN che vieta l'accesso ai *SoloArtist* specificati. Viene evidenziato poi che non sono presenti vincoli sui predicati e sugli attributi, ed infine si restituisce la query modificata.

```

{
  "user": "u1",
  "constraint": "node",
  "node-type": "http://stardog.com/tutorial/Band"
},

```

Listato 28: Vincolo sui nodi generico relativo all'esempio 1

```
{
  "user": "u1",
  "constraint": "node",
  "node-type": "http://stardog.com/tutorial/SoloArtist",
  "nodes": [
    "http://stardog.com/tutorial/George_Harrison",
    "http://stardog.com/tutorial/John_Lennon",
    "http://stardog.com/tutorial/Paul_McCartney",
    "http://stardog.com/tutorial/Ringo_Starr"
  ]
},
```

Listato 29: Vincolo sui nodi specifico relativo all'esempio 1

```
--- USER: u1 ---
--- QUERY: ---
PREFIX : <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?album a :Album ;
    :artist ?artist
  }

--- CHECK NODE CONSTRAINTS ---
Add filter: FILTER ( ?artist NOT IN (<http://stardog.com/tutorial/George_Harrison>,
<http://stardog.com/tutorial/John_Lennon>, <http://stardog.com/tutorial/Paul_McCartney>,
<http://stardog.com/tutorial/Ringo_Starr>) )
Add filter: FILTER NOT EXISTS { ?artist a <http://stardog.com/tutorial/Band> }

--- CHECK PREDICATE CONSTRAINTS ---
No predicate constraints applied

--- CHECK ATTRIBUTE CONSTRAINTS ---
No attribute constraints applied

--- OUTPUT QUERY: ---
PREFIX : <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?album a :Album ;
    :artist ?artist
    FILTER ( ?artist NOT IN (:George_Harrison, :John_Lennon, :Paul_McCartney,
:Ringo_Starr) )
    FILTER NOT EXISTS { ?artist a :Band }
  }
```

Listato 30: Esempio 1 vincoli sui nodi

Il secondo esempio, Listato 31, espone ancora l'applicazione di un vincolo sui nodi, in questo caso applicato ad un oggetto URI e non variabile. La query richiede l'accesso alle canzoni scritte dal nodo *John_Lennon*, che oltre ad essere *SoloArtist* è anche istanza di *SongWriter*. All'utente *u2* corrisponde un vincolo su una lista di nodi della classe in questione, Listato 32. Il sistema, quindi, confronta l'oggetto con la lista di URI e, trovando corrispondenza, rimuove la tripla contenente l'URI.

```

--- USER: u2 ---
--- QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?song a      :Song ;
        :writer :John_Lennon
  }

--- CHECK NODE CONSTRAINTS ---
Remove triple: ?song (<http://stardog.com/tutorial/writer>) http://stardog.com/tutorial/John_Lennon

--- CHECK PREDICATE CONSTRAINTS ---
No predicate constraints applied

--- CHECK ATTRIBUTE CONSTRAINTS ---
No attribute constraints applied

--- OUTPUT QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?song a :Song }

```

Listato 31: Esempio 2 vincoli sui nodi

```

{
  "user": "u2",
  "constraint": "node",
  "node-type": "http://stardog.com/tutorial/Songwriter",
  "nodes": [
    "http://stardog.com/tutorial/George_Harrison",
    "http://stardog.com/tutorial/John_Lennon",
    "http://stardog.com/tutorial/Paul_McCartney",
    "http://stardog.com/tutorial/Ringo_Starr"
  ]
},

```

Listato 32: Vincolo sui nodi relativo all'esempio 2

4.3 Esempi: vincoli sui predicati

Il successivo esempio, Listato 33, riguarda invece i vincoli sui predicati. La query richiede l'accesso alle *Band* e ai relativi componenti. All'utente *u4* è vietato l'accesso ai membri *John_Lennon* e *Paul_McCartney* della *Band The_Beatles*, come evidenziato nel Listato 34. Ciò implica che l'utente può accedere agli altri membri della *Band*. Quindi il sistema aggiunge un filtro AND negato, che impedisce il verificarsi allo stesso tempo delle due condizioni indicate.

```
--- USER: u4 ---
--- QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?band :member ?member }

--- CHECK NODE CONSTRAINTS ---
No node constraints applied

--- CHECK PREDICATE CONSTRAINTS ---
Add filter: FILTER ( ! ( ( ?band = <http://stardog.com/tutorial/The_Beatles> ) && ( ?member IN
  (<http://stardog.com/tutorial/John_Lennon>, <http://stardog.com/tutorial/Paul_McCartney>) ) ) )

--- CHECK ATTRIBUTE CONSTRAINTS ---
No attribute constraints applied

--- OUTPUT QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?band :member ?member
    FILTER ( ! ( ( ?band = :The_Beatles ) && ( ?member IN (:John_Lennon, :Paul_McCartney) ) ) )
  }
```

Listato 33: Esempio 3 vincoli sui predicati

```
{
  "user": "u4",
  "constraint": "predicate",
  "subject-type": "http://stardog.com/tutorial/Band",
  "subjects": ["http://stardog.com/tutorial/The_Beatles"],
  "predicate": "http://stardog.com/tutorial/member",
  "object-type": "http://stardog.com/tutorial/SoloArtist",
  "objects": [
    "http://stardog.com/tutorial/John_Lennon",
    "http://stardog.com/tutorial/Paul_McCartney"
  ]
},
```

Listato 34: Vincolo sui predicati relativo all'esempio 3

Un ulteriore esempio per i vincoli sui predicati è presentato nel Listato 35. In questo caso la query richiede i membri della *Band The_Beatles*, per cui il soggetto è un URI specificato. Considerando il vincolo contenuto nel Listato 36, il sistema verifica che l'utente *u3* non può accedere al suddetto nodo, e perciò rimuove la tripla e la variabile dalla clausola SELECT. Si nota che la query iniziale non conteneva ulteriori triple e variabili, quindi il risultato finale è una query vuota.

```

--- USER: u3 ---
--- QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT *
WHERE
  { :The_Beatles :member ?member }

--- CHECK NODE CONSTRAINTS ---
No node constraints applied

--- CHECK PREDICATE CONSTRAINTS ---
Remove triple: http://stardog.com/tutorial/The_Beatles (<http://stardog.com/tutorial/member>) ?member
Remove result var: ?member

--- CHECK ATTRIBUTE CONSTRAINTS ---
No attribute constraints applied

--- OUTPUT QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT
WHERE
  { # Empty BGP

  }

```

Listato 35: Esempio 4 vincoli sui predicati

```

{
  "user": "u3",
  "constraint": "predicate",
  "subject-type": "http://stardog.com/tutorial/Band",
  "subjects": [
    "http://stardog.com/tutorial/Queen_(band)",
    "http://stardog.com/tutorial/The_Beatles",
    "http://stardog.com/tutorial/The_Rolling_Stones"
  ],
  "predicate": "http://stardog.com/tutorial/member",
  "object-type": "http://stardog.com/tutorial/SoloArtist"
},

```

Listato 36: Vincolo sui predicati relativo all'esempio 4

4.3 Esempi: vincoli sugli attributi

Per quanto riguarda invece i vincoli sugli attributi, il primo esempio è contenuto nel Listato 37. La query richiede le canzoni con la relativa lunghezza, applicando a tale variabile un filtro maggiore o uguale. Il vincolo corrispondente all'utente *u5*, Listato 38, implica che l'attributo non debba assumere determinati valori. Il sistema in questo caso è in grado di combinare il filtro esistente con il filtro aggiunto dal vincolo.

```
--- USER: u5 ---
--- QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?song a      :Song ;
    :length ?length
    FILTER ( ?length >= 300 )
  }

--- CHECK NODE CONSTRAINTS ---
No node constraints applied

--- CHECK PREDICATE CONSTRAINTS ---
No predicate constraints applied

--- CHECK ATTRIBUTE CONSTRAINTS ---
Remove filter: FILTER ( ?length >= 300 )
Add combined filter: FILTER ( ( ?length > 300 ) && ( ?length != 350 ) )

--- OUTPUT QUERY: ---
PREFIX :    <http://stardog.com/tutorial/>

SELECT *
WHERE
  { ?song a      :Song ;
    :length ?length
    FILTER ( ( ?length > 300 ) && ( ?length != 350 ) )
  }
```

Listato 37: Esempio 5 vincoli sugli attributi

```

{
  "user": "u5",
  "constraint": "attribute",
  "subject-type": "http://stardog.com/tutorial/Song",
  "predicate": "http://stardog.com/tutorial/length",
  "object-type": "integer",
  "symbol": "notin",
  "values": [
    "200",
    "250",
    "300",
    "350"
  ]
},

```

Listato 38: Vincolo sugli attributi relativo all'esempio 5

L'ultimo esempio, Listato 40, concerne ancora i vincoli sugli attributi, ma cambia la tipologia di dato, da *Integer* a *Date*. La query richiede gli *Album* pubblicati dal 1985 in poi, ma all'utente *u6* è concesso accedere agli *Album* pubblicati nel decennio 1990-1999, Listato 39. Il vincolo BETWEEN, non esistendo nel linguaggio SPARQL, viene applicato come un filtro maggiore e uguale e un filtro minore e uguale. Perciò il sistema aggiunge due vincoli, effettuando due volte la combinazione dei filtri.

```

{
  "user": "u6",
  "constraint": "attribute",
  "subject-type": "http://stardog.com/tutorial/Album",
  "predicate": "http://stardog.com/tutorial/date",
  "object-type": "date",
  "symbol": "between",
  "values": [
    "1990-01-01",
    "1999-12-31"
  ]
},

```

Listato 39: Vincolo sugli attributi relativo all'esempio 6


```

--- USER: u6 ---
--- QUERY: ---
PREFIX : <http://stardog.com/tutorial/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT *
WHERE
  { ?album a :Album ;
    :date ?date
    FILTER ( ?date >= "1985-01-01"^^xsd:date )
  }

--- CHECK NODE CONSTRAINTS ---
No node constraints applied

--- CHECK PREDICATE CONSTRAINTS ---
No predicate constraints applied

--- CHECK ATTRIBUTE CONSTRAINTS ---
Remove filter: FILTER ( ?date >= "1985-01-01"^^<http://www.w3.org/2001/XMLSchema#date> )
Add combined filter: FILTER ( ?date >= "1990-01-01"^^<http://www.w3.org/2001/XMLSchema#date> )
Remove filter: FILTER ( ?date >= "1990-01-01"^^<http://www.w3.org/2001/XMLSchema#date> )
Add combined filter: FILTER ( ( ?date >= "1990-01-01"^^<http://www.w3.org/2001/XMLSchema#date> )
  && ( ?date <= "1999-12-31"^^<http://www.w3.org/2001/XMLSchema#date> ) )

--- OUTPUT QUERY: ---
PREFIX : <http://stardog.com/tutorial/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT *
WHERE
  { ?album a :Album ;
    :date ?date
    FILTER ( ( ?date >= "1990-01-01"^^xsd:date ) && ( ?date <= "1999-12-31"^^xsd:date ) )
  }

```

Listato 40: Esempio 6 vincoli sugli attributi

Capitolo 5: Conclusione

L'obiettivo di questo progetto di tesi era lo sviluppo di un sistema informatico volto ad applicare policy di accesso a dati RDF su query formulate in linguaggio SPARQL.

A questo scopo, sono state implementate tre tipologie di vincoli che consentono di diversificare l'accesso alle informazioni e garantire un buon livello di sicurezza.

Il tool prodotto è in grado di acquisire query SPARQL con una struttura base, analizzarle ed apportare le modifiche necessarie imposte dalle policy, restituendo in output delle query autorizzate.

Possibili evoluzioni di questo strumento riguardano soprattutto il grado di complessità delle componenti: il sistema potrà estendere la sua capacità di analisi a query più articolate, prevedendo ulteriori clausole e sottoquery, e valutare differenti forme di vincoli, che prevedano condizioni combinate.

Lavorare a questo progetto di tesi è stato molto coinvolgente e interessante in quanto ho avuto la possibilità di interfacciarmi con tecnologie che nel corso dei prossimi anni continueranno a svilupparsi e assumeranno grande rilevanza nel mondo informatico, proponendo ulteriori spunti di ricerca.

Bibliografia

- [1] «MOSAICrOWN – Multi-Owner data Sharing for Analytics and Integration respecting Confidentiality and OWNeR control», <https://mosaicrown.eu/>, consultato il giorno 27-07-2021
- [2] «Resource Description Framework», https://it.wikipedia.org/wiki/Resource_Description_Framework, consultato il giorno 26-08-2021
- [3] RDF Working Group, «Resource Description Framework (RDF)», <https://www.w3.org/RDF/>, consultato il giorno 13-09-2021
- [4] Oreste Signore, «RDF per la rappresentazione della conoscenza», <http://www.w3c.it/papers/RDF.pdf>, consultato il giorno 27-08-2021
- [5] «Web semantico», https://it.wikipedia.org/wiki/Web_semantico, consultato il giorno 26-08-2021
- [6] «W3C», <https://www.w3.org/>, consultato il giorno 13-09-2021
- [7] «RDF Graph Data Model», <https://www.stardog.com/tutorials/data-model>, consultato il giorno 28-07-2021
- [8] W3C, «RDF Data Visualization», <https://www.w3.org/2018/09/rdf-data-viz/>, consultato il giorno 11-09-2021
- [9] W3C, «Namespaces in XML 1.0 (Third Edition)», <https://www.w3.org/TR/xml-names/>, consultato il giorno 13-09-2021
- [10] «Learn SPARQL», <https://www.stardog.com/tutorials/sparql/>, consultato il giorno 28-07-2021
- [11] «How much faster is a graph database, really?», <https://neo4j.com/news/how-much-faster-is-a-graph-database-really/>, consultato il giorno 10-09-2021

- [12] W3C, «RDF 1.1 N-Triples. A line-based syntax for an RDF graph», <https://www.w3.org/TR/n-triples/>, consultato il giorno 13-09-2021
- [13] «JSON for Linking Data», <https://json-ld.org/>, consultato il giorno 13-09-2021
- [14] W3C, «RDF 1.1 XML Syntax», <https://www.w3.org/TR/rdf-syntax-grammar/>, consultato il giorno 13-09-2021
- [15] W3C, «RDF 1.1 Turtle. Terse RDF Triple Language», <https://www.w3.org/TR/turtle/>, consultato il giorno 10-09-2021
- [16] W3C, «W3C XML Schema Definition Language (XSD) 1.1. Part 2: Datatypes», <https://www.w3.org/TR/xmlschema11-2/>, consultato il giorno 07-07-2021
- [17] «Apache Jena», <https://jena.apache.org/>, consultato il giorno 31/07/2021
- [18] «Cosa si cela dietro al Visitor design pattern?», <https://jena.apache.org/>, consultato il giorno 12-09-2021
- [19] «JSON.simple», <https://code.google.com/archive/p/json-simple/>, consultato il giorno 05-08-2021