Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Informatica in Informatica

RSLT: trasformazione di Open Linked Data in testi in linguaggio naturale tramite template dichiarativi

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Davide Quadrelli

Sessione II Anno Accademico 2015/2016

Indice

Indice del codice

Elenco delle figure

Introduzione

Questa dissertazione discute del problema della trasformazione di dati semantici, semplici da interpretare per una macchina, in un linguaggio naturale, di facile comprensione per l'uomo, discutendo delle soluzioni trovate in letteratura e, nel dettaglio, di RSLT, una libreria JavaScript che cerca di risolvere il problema appena introdotto. Verranno mostrati, inoltre, tutti i cambiamenti e tutte le modifiche introdotte nella versione 1.1 della libreria, la cui nuova funzionalità principale è il supporto a SPARQL 1.0.

Dalla nascita del Semantic Web, la rappresentazione di dati semantici in un formato facilmente comprensibili dagli esseri umani è sempre stato un problema ben noto. Indipendentemente da quale sia la trasformazione che è necessario eseguire, attualmente verrà gestita all'interno della logica dell'applicazione; dover descrivere ed effettuare queste operazioni all'interno del codice dell'applicativo, rende il riutilizzo di tali trasformazioni complicato. L'eventuale necessità di modificare l'output delle trasformazioni, inoltre, richiede modifiche all'interno del codice dell'applicazione, quindi di dover cambiare la logica dell'applicazione stessa.

Tale situazione è critica e richiede una soluzione che sia in grado di gestire trasformazioni di Open Linked Data in un linguaggio naturale, consentendo di semplificare la vita agli sviluppatori che necessitano l'utilizzo di queste informazioni. Il W3C ha definito RDF come standard per la descrizione di Open Linked Data e SPARQL come suo linguaggio di query; la nascita di questi standard ha permesso di cominciare a cercare soluzioni software in grado di eseguire queste trasformazioni, rimuovendo tale compito al codice delle applicazioni stesse. Sono nate differenti tipologie di soluzioni, basate sui diversi

standard che si sono utilizzati per risolvere il problema.
La prima tipologia di soluzioni si basa su linguaggi di trasformazioni già esistenti, modificando tali tecnologie, già conosciute ed utilizzate nel mondo, per aggiungere il supporto a RDF e SPARQL. Da questa idea sono nati XSPARQL [Bis12] e XSLT+SPARQL [BGH08]. XSPARQL è l'unione di XQuery, linguaggio di interrogazione per XML, e SPARQL, che consente di poter ottenere i dati tramite query. Questa soluzione, però, non ha una sintassi semplice, soprattutto per via della difficoltà di gestione dei dati (che hanno una struttura a grafo) all'interno di un meccanismo nato per informazioni con una struttura ad albero. XSLT+SPARQL, invece, nasce dall'unione di XSLT, linguaggio dichiarativo di trasformazione per XML, e SPARQL. Questo nuovo linguaggio consente la definizione di template per dati ottenuti tramite query; così facendo, consente un meccanismo di descrizione semplice su come mostrare i dati, ma continua ad avere problemi nella gestione dei dati (per lo stesso motivo di XSPARQL).
La seconda tipologia di soluzioni si basa su standard e linguaggi già esistenti, nati per la gestione di dati semantici, ed integrarli con nuove funzionalità per definire template dichiarativi e, quindi, consentire trasformazioni. Da questa idea sono nati STTL [CF15] e Fresnel [Pie06]. STTL si basa su SPARQL e aggiunge, a tale linguaggio, la possibilità di definire template. Questa soluzione permette quindi di definire, direttamente nella costruzione della query, come mostrare in output i dati ottenuti. STTL, inoltre, è completamente indipendente dal linguaggio di output, consentendo non solo output in linguaggio naturale, ma anche in altri linguaggi strutturati (JSON, XML, CSV, ecc). Fresnel, invece, si basa su un'ontologia per descrivere template in RDF stesso. Questa soluzione consente di descrivere, per ogni tipo di entità, quali sue proprietà mostrare e come farlo, permettendo la nascita di triplestore contenenti le descrizioni dei template.
La terza ed ultima tipologia di soluzioni presenti in letteratura si basa sulla creazione di nuovi linguaggi di trasformazioni, ispirando la loro sintassi a soluzioni già esistenti per consentire un apprendimento più semplice ed immediato. Da questa idea sono nati OWL-PL [BH09] e RSLT [PV15]. OWL-PL si basa su una sintassi in XHTML, nella quale è possibile definire i template voluti, utilizzando una sintassi basata su XSLT. RSLT, invece, si basa su una sintassi HTML, sempre ispirata ad XSLT. Questa soluzione consente di definire template all'interno della pagina stessa, e permette di definire visualizzazioni

complesse sfruttando più template, pensati per porzioni più semplici dell'output.
Queste soluzioni trovate sono molto differenti fra loro, ed offrono vantaggi e svantaggi rispetto agli altri, a seconda della tipologia di ambiente in cui si lavora. Nonostante ciò, però, XSPARQL e XSLT+SPARQL non sono ottimizzati perché basati su linguaggi nati per gestire dati con struttura ad albero, e non a grafo. XSPARQL, inoltre, avendo una definizione procedurale delle trasformazioni, ha una sintassi molto più complicata rispetto a tutte le altre soluzioni. STTL, OWL-PL e Fresnel hanno una definizione più semplice dei template, ma non consentono una loro integrazione in un'applicazione web che richiede l'interazione da parte di un utente. RSLT, invece, è l'unica soluzione presente in letteratura in grado di permettere la trasformazioni di questi dati nell'ambito di web application, permettendo l'interazione di un utente con i dati stessi. Offre, inoltre, dinamicità dell'output e la gestione dell'assenza di informazione.

RSLT, quindi, consente la trasformazione di dati semantici in formato RDF in un linguaggio naturale, facilmente comprensibile dall'uomo, tramite la definizione di template dinamici. Questa libreria nasce per essere utilizzata in ambito web, e perciò è stata scritta in JavaScript ed è basata su AngularJS. La definizione e l'utilizzo dei template che RSLT consente di definire permettono di costruire visualizzazione complesse di dati, a partire da piccoli template che si occupano di gestire la visualizzazione di porzioni ridotte del testo. Questa struttura consente una sintassi molto semplice e un forte riutilizzo del codice. Ogni template può essere associato ad un particolare tipo di entità e, quindi, può essere utilizzato per trasformare ogni singola istanza di quel tipo. I dati semantici utilizzati nelle trasformazioni vengono recuperati tramite particolari query SPARQL, costruite dalla libreria a partire dai grafi RDF definiti nei template. Nel caso in cui vengano, in una trasformazione, richieste informazioni che non si hanno a disposizione, RSLT consente di catturare quest'eventualità e gestirla, generando comunque dell'output. RSLT consente, inoltre, di utilizzare un qualsiasi linguaggio di output (all'interno di un qualche tag HTML), permettendogli di essere utilizzato anche per convertire dati RDF da un formato ad un altro.
Per semplificare il processo di creazione dei template, esiste una web application, chiamata RSLT Playground, che consente agli sviluppatori di testare i template che hanno

definito prima di integrarli all'interno della loro applicazione. Playground è rilasciato assieme alla libreria e consente di definire tutti i dati necessari alla simulazione di una vera esecuzione di RSLT; è possibile, inoltre, visualizzare tutti i dati ottenuti dalle query, per poter controllare che tutti le informazioni necessarie per le trasformazioni ci siano o meno. Playground semplifica anche l'apprendimento della sintassi di RSLT e da un valore aggiunto a questa libreria.

RSLT, essendo basato su AngularJS, sfrutta questa libreria per la creazione di ogni suo componente. Dalla versione 1.1, ogni meccanismo di RSLT è integrato o all'interno di direttive (come la gestione delle query e l'applicazione dei template) o di servizi (come la gestione delle triple, dei prefissi memorizzati e il parser della sintassi SPARQL) di AngularJS. Quest'ultima versione della libreria ha introdotto il supporto in RSLT allo standard SPARQL 1.0, prima assente a causa della mancanza del costrutto "BIND", utilizzato nelle query costruite da RSLT. Quest'aggiornamento ha anche risolto svariati bug legati alla gestione dei prefissi e al parser della definizione dei grafi; inoltre ha migliorato la stabilità generale del sistema, grazie al meccanismo di LIMIT delle query (che limita le possibilità di timeout da parte del triplestore con cui si comunica) e al supporto di un proxy che consente la risoluzione di problemi legati al CORS. La gestione del LIMIT, importante per impedire timeout nelle richieste, soprattutto con triplestore non molto efficienti, è stata inserita aggiungendo un valore di OFFSET e LIMIT nelle query inviate dall'applicazione al triplestore. Se, nel momento in cui si ottiene la risposta, ci sono ancora altri dati da ottenere, si aggiorna il valore di OFFSET e si ripete la richiesta, iterando fino all'ottenimento di tutti i dati.
La gestione di SPARQL 1.0 ha richiesto, data l'assenza del costrutto BIND, la generazione di una sintassi alternativa, compatibile con questo standard. Questo supporto è molto importante perché tantissimi triplestore del mondo utilizzano software che implementano ancora solo questa versione dello standard del W3C. Questa nuova sintassi richiede di definire, una volta per ogni variabile di cui si vogliono ottenere le informazioni, il grafo del template: questa situazione è ovviamente poco ottimizzata per il triplestore, che se non implementa un qualche meccanismo di caching, richiederà un aumento elevato del carico di lavoro. Per ottimizzare questa soluzione, e per problemi di integrazione di que-

sta nuova sintassi all'interno di RSLT, si è deciso di generare query differenti, in ognuna delle quali si definisce lo stesso grafo del template e si ottengono tutti i dati di una sola variabile. Con questa tecnica, il numero di query ottenuto è pari al numero di variabili a cui si è interessati: se il lavoro eseguito dal triplestore rimane elevato, queste query possono essere inviate in parallelo, ottimizzando i tempi di risposta delle richieste. Questa soluzione, quindi, va valutata, controllando quindi che i tempi complessivi richiesti da RSLT per la trasformazione dei dati, siano principalmente dovuti dai tempi di attesa alle risposte delle query e non alla computazione della libreria.

Per valutare, quindi, che eventuali tempi di attesa siano dovuti alle query, e cioè al limite della sintassi di SPARQL 1.0, e non alla computazione di RSLT, è stato necessario definire template e alcuni grafi da utilizzare per recuperare i dati ed eseguire alcune trasformazioni. Durante queste trasformazioni si sono presi i tempi richiesti dalla libreria per eseguire l'intera trasformazione e e i tempi richiesti dalle query per ottenere una risposta. I dati ottenuti da queste prove dimostrano che il 98% del tempo complessivo dell'esecuzione di RSLT è passato in attesa di ottenere le risposte delle query: ciò dimostra come l'implementazione della libreria sia efficiente e che non è possibile migliorare la situazione, in quanto il problema risiede in SPARQL stesso, che nella sua prima versione non supporta il costrutto BIND.
Per fornire un'ulteriore prova di come RSLT sia effettivamente in grado di avvicinarsi alla risoluzione della trasformazione di dati semantici in un linguaggio comprensibile anche dall'uomo, si è creato "LANCET Data Browser (for SPARQL 1.0)".
LANCET è un'applicazione web, interamente basata su RSLT, che consente di visualizzare dati RDF inerenti a fllm, attori, registi e produttori grazie all'output delle trasformazioni. Permette l'interazione da parte di un utente con i dati visualizzati, consentendo di eseguire ricerche e di navigare fra tutti i dati disponibili. Per dimostrare la dinamicità dei template, e come essi possano essere utilizzati, sono mostrati livelli diversi di dettaglio per ogni entità a seconda della schermata mostrata. Il corretto funzionamento di LANCET, infine, consente di dimostrare, con una prova tangibile, la reale capacità di

RSLT di essere utilizzato all'interno di un'applicazione web dinamica.

In conclusione, quindi, si è mostrato come sia possibile una soluzione al problema della trasformazione di dati semantici in un linguaggio naturale; anche se RSLT non è ancora pronto per adempiere completamente a questo scopo, si avvicina molto, consentendo già trasformazioni tramite template dichiarativi, gestendo l'assenza dei dati e fornendo un totale supporto allo standard SPARQL. Sono assolutamente necessarie, però, ulteriori integrazioni di nuove funzionalità, come il supporto a file RDF, l'aggiunta di alcune funzionalità nei template e la definizione delle trasformazioni in RDF, oltre che XML e JSON, anche in RDF stesso, tramite un'apposita ontologia.

Capitolo 1

Open Linked Data: RDF e SPARQL

Per comprendere da quali necessità è nato il bisogno di creare RSLT 1.1, bisogna prima discutere il contesto nel quale nasce tale progetto, ovvero quello del Semantic Web e degli Open Linked Data. Discuteremo di essi e degli standard che il W3C ha definito per gestire i dati semantici; inoltre, si mostreranno alcune delle più significative soluzioni, presenti in letteratura, per la trasformazione di dati semantici in formato RDF, valutandone aspetti positivi e negativi.

Il termine "Semantic Web" è stato coniato da Tim Barners Lee (uno dei maggiori esponenti e co-inventore del World Wide Web) per indicare un web di dati che possono essere facilmente processati dalle macchine [LHL01]: ciò descrive un ambiente che non contiene soltanto documenti e pagine web, ma uno in cui i suoi contenuti sono legati ad informazioni che specificano il loro contesto semantico. Tali informazioni sono ovviamente molto più semplici da indicizzare e ricercare per una macchina, ma sono molto complessi da comprendere per un essere umano.
Il W3C (World Wide Web Consortium) ha definito come standard per il salvataggio di queste informazioni il modello RDF (Resource Description Framework) [GB14] e, come linguaggio di interrogazione, SPARQL (SPARQL Protocol and RDF Query Language) [PS08]. RDF ha come unità base dell'informazione lo "statement", ovvero una tripla formata da soggetto, predicato ed oggetto; ogni elemento di uno statement è rappresentato da un IRI (Internationalized Resource Identifier), che identifica univocamente ogni risor-

sa. SPARQL, invece, si occupa di definire grafi (con una sintassi molto simile a quella di un database noSQL) per ottenere i dati corrispondenti ad essi da uno SPARQL point (ovvero un servizio che gestisce un triplestore RDF e sa ricevere, gestire e rispondere a richieste di query SPARQL).
Tali informazioni, essendo difficilmente comprensibili per un essere umano, devono essere in un qualche modo resi accessibili e comprensibili, perché per quanto il Semantic Web nasca apposta per semplificare l'indicizzazione e la ricerca dei dati per un computer, quei dati devono essere leggibili anche agli esseri umani.
La trasformazione di dati semantici in un formato comprensibile è un grosso problema: qualsiasi applicazione che ne faccia utilizzo, deve elaborare i dati che possiede all'interno del codice per decidere cosa (e come) mostrare all'utente. Questa elaborazione avviene tramite codice applicativo, che rende tremendamente complicato il riutilizzo di qualsiasi trasformazione applicata, portando a dover ogni volta riadattare il codice per una nuova applicazione; inoltre, la necessità di una semplice modifica sull'output, richiede modifiche al codice dell'applicativo, non sempre semplici, che rendono comunque l'output strettamente legato alla programmazione del applicazione che usa questi dati.
Questa situazione è decisamente complicata e il problema della trasformazione di dati semantici in un linguaggio comprensibile dall'uomo non possiede uno strumento semplice ed efficace per risolverla senza richiedere che, qualsiasi modifica all'output, necessiti modifiche alla programmazione dell'applicazione. Tale problematica del Semantic Web è ben nota nella letteratura scientifica, dove sono state proposte alcune soluzioni per la trasformazioni di dati RDF in un linguaggio naturale, facilmente comprensibile dall'uomo. Per progettare un sistema in grado di adempiere a tale scopo, sono possibili diverse metodologie che utilizzano standard e tecnologie differenti.

Di seguito vedremo, in specifico, le soluzioni più significative trovate in letteratura, spiegandone le funzionalità, le tecnologie utilizzate e, infine, confrontandole fra di loro.

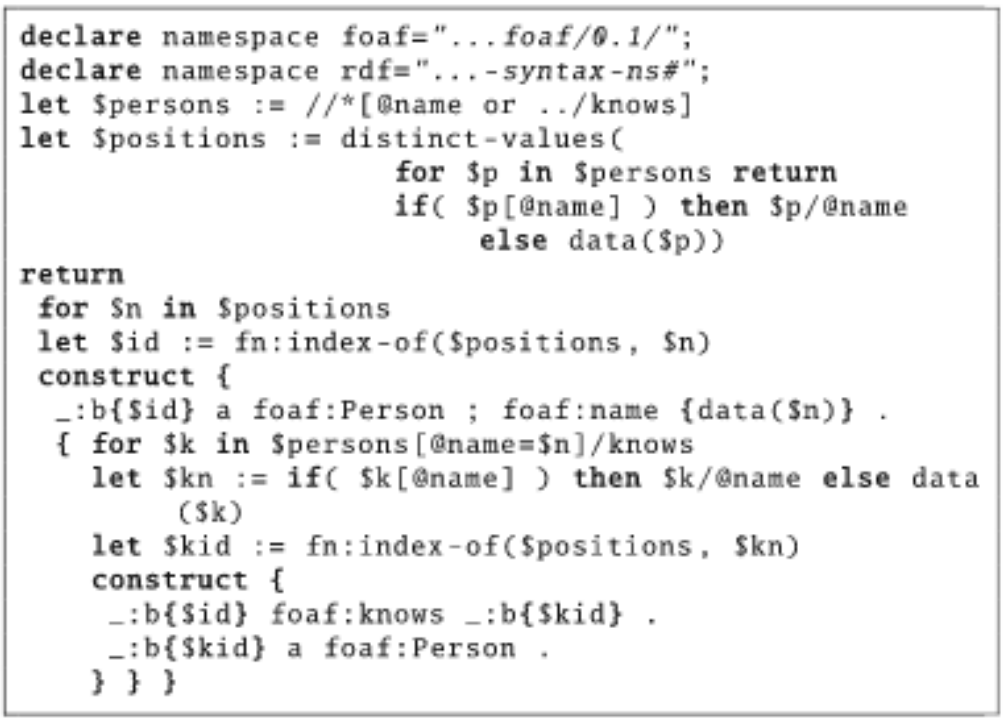
1.1 Adattamento di tecnologie per XML a RDF

La necessità di trasformare dati in RDF, standard recente per gli Open Linked Data, ha fatto riflettere sulla possibilità di utilizzare XML, e tutti i linguaggi di trasformazioni esistenti per esso, per risolvere questo nuovo problema. Riutilizzare vecchie soluzioni pensate per XML su RDF permetterebbe di avere, fin da subito, un bacino di utenza decisamente elevato, data l'elevata diffusione di XML, e di sfruttare sintassi già conosciute. Offrire una migliore comprensione degli statement RDF ad un essere umano risulta, quindi, possibile tramite strumenti già pensati per lo stesso scopo, cambiando soltanto il linguaggio di input.

Da questa idea, sono nati due progetti con lo scopo di integrare SPARQL in due differenti linguaggi di trasformazione per XML: XSLT [Kay07] e XQuery [Rob14]. Essi cercano di trasformare RDF integrando, all'interno dei linguaggi già esistenti, il supporto per query SPARQL, in grado di ottenere le informazioni richieste, e utilizzarle per le trasformazioni.

1.1.1 Trasformazioni tramite XQuery e SPARQL

XQuery è un linguaggio di interrogazione per XML, nato dopo XSLT, che permette di eseguire interrogazioni all'interno di un file XML e di generare dell'output contenente tali informazioni. XQuery è indipendente dal linguaggio di output, permettendo trasformazioni anche in un linguaggio naturale. Tale peculiarità rende XQuery adatto ad essere integrato con il supporto a query SPARQL, in modo da poter recuperare da un triplestore i dati RDF richiesti e utilizzarli per la generazione dell'output.
XSPARQL [Bis12] è un progetto che nasce per poter convertire dati RDF in XML e viceversa, utilizzando una versione modificata di XQuery che consente l'utilizzo di query SPARQL al suo interno. L'integrazione di uno standard all'interno dell'altro consente, a chi utilizza XSPARQL, di poter sfruttare assieme costrutti e sintassi dei due singoli linguaggi, aiutando un utente novizio ad imparare il nuovo linguaggio. XSPARQL è in grado di gestire sia alberi che grafi, e di poter convertire le due strutture tramite programmazione; per quanto sia un buon risultato, è necessario avere, per definire una trasformazione, capacità di programmazione. Programmare la trasformazione, come è



```
declare namespace foaf="...foaf/0.1/";
declare namespace rdf="...-syntax-ns#";
let $persons := //*[@name or ../knows]
let $positions := distinct-values(
                    for $p in $persons return
                    if( $p[@name] ) then $p/@name
                         else data($p))
return
 for $n in $positions
 let $id := fn:index-of($positions, $n)
 construct {
  _:b{$id} a foaf:Person ; foaf:name {data($n)} .
  { for $k in $persons[@name=$n]/knows
    let $kn := if( $k[@name] ) then $k/@name else data
         ($k)
    let $kid := fn:index-of($positions, $kn)
    construct {
     _:b{$id} foaf:knows _:b{$kid} .
     _:b{$kid} a foaf:Person .
    } } }
```

Figura 1.1: Esempio di definizione di trasformazione con XSPARQL
Immagine presa dall'articolo [Bis12].

possibile vedere nell'immagine 1.1, ha una difficoltà abbastanza elevata, ed è un requisito non necessariamente indispensabile nella trasformazione di dati. Questa difficoltà di programmazione delle trasformazioni, dovuta anche al passaggio da dati strutturati a grafo ad altri strutturati ad albero, e la disponibilità di conversione di RDF solo in XML, non permette a questo linguaggio di definire, in maniera semplice, trasformazioni di Open Linked Data in un testo in linguaggio naturale, ma pone comunque le basi per poter sviluppare linguaggi e progetti più funzionali e adatti a risolvere questo problema.

1.1.2 Integrazione di SPARQL in XSLT

XSLT (XML Style Language Transformation) è un linguaggio di trasformazione di stile per dati XML, nato per trasformare un file XML in un altro con una struttura differente. Essendo ogni documento XML un albero, XSLT nasce per gestire tale struttura: RDF, però, ha una struttura a grafo. Per tale motivo, si è cercato di integrare all'interno di XML delle query SPARQL, le cui variabili siano utilizzabili all'interno del documento, come descritto in [AL08], e, successivamente, di estendere XSLT con SPARQL, come in [BGH08], perché permette di gestire la formulazione di template dichiarativi per alberi e l'esecuzione di query su grafi contemporaneamente, utilizzando un linguaggio di trasformazione già ampiamente conosciuto e diffuso.
Con questa tecnica, si ottiene un linguaggio di trasformazione di dati RDF, in grado di trattare sia alberi che grafi, unendo le funzionalità delle query SPARQL con quelle di trasformazione di XSLT. Sono gestiti solo i costrutti "SELECT" e "ASK" di SPARQL, perché le risposte a tali query sono nel formato RDF/XML, che è perfettamente gestibile da XSLT. Tale soluzione è utile per chi conosce già sia SPARQL che XSLT, dato che non richiede la conoscenza di un nuovo linguaggio, e offre una buona libertà di azione per la definizione di template.
XSLT+SPARQL aggiunge alcune piccole funzioni a XSLT che consentono di memorizzare ed eseguire query SPARQL, utilizzandole poi all'interno dei template XSLT. Questa soluzione è completamente dichiarativa, permettendo la definizione di template senza dover scrivere codice applicativo.

```
<xsl:variable name="query">
  PREFIX skos: &lt;http://www.w3.org/2004/02/skos/core#&gt;
  PREFIX cat:  &lt;http://dbpedia.org/resource/Category:&gt;
  PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt;
  PREFIX foaf: &lt;http://xmlns.com/foaf/0.1/&gt;
  SELECT ?name ?person ?img
  WHERE { ?person skos:subject   cat:Spanish_actors .
          ?person rdfs:label     ?name .
          ?person foaf:depiction ?img .
          FILTER (lang(?name)="es") }
  ORDER BY ?name
</xsl:variable>

<xsl:template match="/">
  <html> <body>
      <h1>Spanish Actors</h1>
      <ul>
        <xsl:apply-templates select="sparql:sparqlEndpoint(
          $query, 'http://dbpedia.org/sparql')"/>
      </ul>
  </body> </html>
</xsl:template>

<xsl:template match="results:result">
  <li>
    <xsl:value-of select="results:binding[@name='name']"/>
    <img> <xsl:attribute name="src">
        <xsl:value-of select="results:binding[@name='img']"/>
    </xsl:attribute> </img>
  </li>
</xsl:template>
```

Figura 1.2: Esempio di definizione di trasformazione con XSLT+SPARQL
Immagine presa dall'articolo [BGH08].

Come è possibile vedere dalla figura 1.2, la definizione all'interno di XSLT+SPARQL delle query SPARQL e dei template da utilizzare, è completamente dichiarativa, e ciò rende più semplice l'utilizzo di questo linguaggio.
XSLT+SPARQL, però, non consente di definire template dinamici, ovvero permettere da un template l'esecuzione di altre query; inoltre, non è possibile ottenere un output che non sia in formato XML o un suo derivato (HTML, XHTML, ecc.), in quanto XSLT non gestisce altri tipi di file. Oltre a questa limitazione, non è possibile gestire l'assenza

di dati. Prendendo ad esempio i template in figura 1.2, se la query non dovesse trovare alcun attore spagnolo, si otterrebbe comunque l'intestazione della pagina "Spanish actors", senza nient'altro. Questa situazione non consente una corretta generazione di testi in linguaggio naturale: se fosse possibile prevedere l'assenza di informazioni, sarebbe stato possibile includere nell'output l'avvertimento, per il lettore, la mancanza delle informazioni cercate.

I progetti appena proposti, quindi, consentono la trasformazioni di dati RDF, ma non sono soluzioni dal semplice utilizzo: XSPARQL ha una sintassi procedurale molto complessa, mentre XSLT+SPARQL, per quanto permetta la definizione di query SPARQL all'interno di template statici e dichiarativi di XSLT, non consente la generazione di output in un formato differente da XML e non è in grado di gestire l'assenza di informazioni, nel caso la query non dovesse restituire un particolare dato.

1.2 Definizione di template in RDF e SPARQL

Oltre all'adattamento e alla modifica di tecnologia già esistenti per XML, come XSLT e XQuery, per aggiungere la gestione di dati RDF, come visto nel capitolo 1.1, è anche possibile cercare di utilizzare RDF stesso o SPARQL, pensati sin dalla loro progettazione per gestire dati con una struttura a grafo, per definire e utilizzare dei template dichiarativi per la trasformazione di Open Linked Data. Un template dichiarativo è una descrizione che non richiede l'utilizzo di codice di programmazione per la trasformazione dei dati, esattamente come XSLT per XML. Il vantaggio dell'utilizzo di template dichiarativi, rispetto alla trasformazione tramite codice applicativo, è che non sono richieste conoscenze di programmazione per poter definire la trasformazione da applicare, rendendone la definizione molto più semplice e chiara.
Di seguito, vedremo due soluzioni per la trasformazione di dati RDF basati su standard già esistenti: STTL [CF15], che consente la definizione di template all'interno di SPARQL e Fresnel [Pie06], che descrive i template direttamente in RDF, andando a definire un'ontologia apposita.

1.2.1 Definizione di template in SPARQL

Data la presenza di SPARQL come linguaggio standard per l'interrogazione di dati RDF, è stato proposta una sua potenziale estensione per poterlo utilizzare anche per definire template RDF [CF14]. Tale idea si sviluppa dal fatto che la generazione di template RDF tramite XSLT (come descritto nella soluzione al paragrafo 1.1.2), che è basato su una struttura ad albero, renda la scrittura e l'implementazione dei template più complessa;se si utilizzasse un linguaggio basato su una struttura a grafo, come SPARQL, la definizione dei template sarebbe decisamente semplificata.
Per questo motivo viene proposto STTL [CF15] (SPARQL Template Transformation Language), un linguaggio di regole di trasformazione generiche per RDF, basato su SPARQL. STTL offre due estensioni per SPARQL: uno per la definizione dei template e l'altro per la loro applicazione.

```
prefix rs: <http://www.w3.org/2001/sw/DataAccess/tests/result-set#>
template {
    "<td>"
    coalesce(
        st:call-template(st:display, ?val),
        " ")
    "</td>" ; separator = " "
}
where {
    ?x rs:solution ?in
    ?x rs:resultVariable ?var
    optional {
        ?in rs:binding [
        rs:variable ?var ; rs:value ?val ]
    }
}
order by ?var
```

Codice 1.1: Esempio di definizione di trasformazione con STTL

Come è possibile vedere dall'esempio 1.1, la prima estensione aggiunge il costrutto "TEMPLATE", che serve a definire il template dell'output e su quali dati andrà applicato. L'applicazione dei template definiti, invece, avviene tramite la funzione "call-template",

che permettono di eseguire la trasformazione da RDF al linguaggio di output definito.
La grande libertà che STTL offre è la possibilità di definire quali template utilizzare in base al nome del template, all'IRI per cui un template è stato definito e la variabile SPARQL per cui deve essere usato. Tale libertà di scelta permette di andare a definire una grande quantità di possibili trasformazioni, e la totale indipendenza dell'esecuzione dal linguaggio di output permette di generare documenti in linguaggio naturale, ma anche conversioni fra sintassi RDF differenti (ad esempio, da RDF/XML a Turtle), l'esportazione in un altro formato di dati (ad esempio, in JSON o CSV) e la conversione di statement in ontologie differenti [CFG15].
Tutti questi utilizzi sono davvero utili, e rendono STTL uno strumento altamente versatile e utile in tantissimi campi che necessitino della trasformazione di dati RDF. Tale soluzione non consente una totale portabilità dei template tra diversi utilizzi: ciò è dovuto dal fatto che, in STTL, i template e le query sono definite assieme, come è possibile vedere nell'esempio 1.1. Tale definizione impedisce l'esecuzione di template differenti senza specificare le query con cui recuperare i dati, creando un legame stretto tra la query che recupera i dati e il template che verrà applicato, impedendo un riuso dei template.
Questa difficoltà nasce dal fatto che, dato un template che stampa in un qualche linguaggio definito una descrizione di una persona, nel momento in cui provo ad utilizzare tale template su un nuovo triplestore, è possibile che la query con cui recupero i dati sia profondamente diversa: se la query è integrata nel template, allora è il template stesso che deve subire modifiche.

STTL, quindi, consente la trasformazione di dati RDF, indipendentemente dal linguaggio di output, tramite template dichiarativi basati su sintassi SPARQL arricchita. Questo nuovo linguaggio permette di definire template per la rappresentazione dei dati e query SPARQL insieme: questo rende la definizione degli stessi molto semplice rispetto a una soluzione basata su XML, ma non consente un buon riutilizzo dei template, perché per cambiare la query, è necessario modificare il template stesso.

1.2.2 Definizione di template tramite RDF

RDF, essendo un framework di descrizione di risorse, consente la definizione di ontologie per rappresentare un qualche dominio di interesse, in modo tale che la rappresentazione sia formale e condivisa. Tale caratteristica è applicabile anche a template dichiarativi: andando a definire una qualche ontologia che descriva il dominio della trasformazione di dati RDF, è possibile definire template in RDF.
Fresnel [Pie06] è un progetto che nasce proprio per definire un'ontologia per descrivere template in RDF, in grado di offrire un vocabolario per codificare le informazioni semantiche che permetta di specificare quali informazioni mostrare e come farlo. Per permettere un riutilizzo dei template maggiore possibile, Fresnel permette di definire separatamente la selezione del contenuto semantico a cui applicare un template dalla definizione della formattazione dei dati da mostrare. Fresnel, inoltre, consente la definizione di template, oltre che sui tipi di entità, come nella figura 1.3, anche su query SPARQL, in modo da poter definire trasformazioni grafiche anche su modelli di dati più complessi.

```
1 :PersonLens a fresnel:Lens ;
2     fresnel:instanceLensDomain
3       "SELECT ?mbox WHERE ( ?x foaf:name 'John Doe' )
4         ( ?x foaf:mbox ?mbox )"^^fresnel:sparqlSelector .
```

Codice 1.2: Esempio di definizione di template in Fresnel su query SPARQL.

L'utilizzo di questi template definiti direttamente in RDF permette di definire, per un'applicazione, quali dati mostrare e come, ma lascia la libertà a chi implementa un'applicazione, basata su Fresnel, di poter gestire lo stile con cui i dati vengono mostrati. Però, è difficile realizzare e generare con template così fortemente strutturati, sulla base delle singole informazioni da mostrare per un'entità, testi in linguaggio naturale. Fresnel consente di trasformare dati RDF, ma non consente di poter definire concatenazioni di trasformazioni di singole proprietà, in modo da costruire un testo comprensibile dall'uomo.
Ciò rende Fresnel ottimo per la creazione di browser di dati semantici, la cui visualizzazione viene definita dai template RDF e, l'output grafico, viene lasciato all'applicazione, ma la generazione di testi scritti risulta molto complicato: questo perché non esiste un legame tra come vengono mostrati i dati e l'effettiva visualizzazione nell'output. L'as-

```
:PersonLens a fresnel:Lens ;
 fresnel:classLensDomain foaf:Person ;
 fresnel:showProperties (
  foaf:name
  foaf:mbox
  [rdf:type fresnel:PropertyDescription;
   fresnel:alternateProperties (
    foaf:depiction foaf:img p3p:image )
   ] ) .

:nameFormat a fresnel:Format ;
    fresnel:propertyFormatDomain foaf:name ;
    fresnel:label "Name" .

:mboxFormat a fresnel:Format ;
    fresnel:propertyFormatDomain foaf:mbox ;
    fresnel:label "Mailbox" ;
    fresnel:value fresnel:externalLink ;
    fresnel:valueFormat [ fresnel:contentAfter "," ] .

:depictFormat a fresnel:Format ;
    fresnel:propertyFormatDomain foaf:depiction ;
    fresnel:label fresnel:none ;
    fresnel:value fresnel:image .
```

Figura 1.3: Esempio di definizione di un template con Fresnel.
Immagine presa dall'articolo [Pie06].

senza di questo legame rende Fresnel molto generico e applicabile

Fresnel definisce, quindi, un'ontologia per la costruzione di template per dati RDF in RDF stesso, in grado di definire come e quali attributi mostrare di ogni entità, lasciando la gestione della grafica all'applicazione che sfrutta questi template. Questa struttura, però, non consente la generazione di testi, arbitrariamente complessi, a causa della struttura stessa dei template.

1.3 Linguaggi di trasformazione ispirati a XSLT

Quando ancora non esisteva RDF, lo standard per la rappresentazione dei dati nel web più utilizzato era XML, e anche allora c'era la stessa necessità che c'è oggi per poter trasformare tali informazioni in un formato più comprensibile per l'uomo. Tecnologie come XSLT e XQuery sono nate proprio da questo bisogno e oggi sono molto diffuse e utilizzate. Inoltre, riutilizzare tecnologie già esistenti per il web, come HTML, CSS e JavaScript, permette di migliorare la diffusione di tecniche di trasformazione per dati RDF ex novo, ma basate su queste tecnologie. Ha senso, quindi, pensare a nuove tecnologie per RDF ispirandosi a quelle del passato, come XSLT per XML.

Con una sintassi ispirata a quella di XSLT, sono presenti in letteratura due soluzioni di trasformazioni di dati RDF: OWL-PL [BH09] e RSLT [PV15].

1.3.1 Integrazione template in XHTML

Come descritto sopra, il riutilizzo di vecchie tecnologie per consentire la traduzione di dati RDF in un formato leggibile da un essere umano, permette una rapida diffusione di tali meccanismi, in grado di risolvere questo grosso problema del Semantic Web. Data la grandissima diffusione di XML, HTML, CSS, JavaScript e JSON nel web, essi sono ottimi candidati per essere affiancati a nuovi meccanismi e tecnologie, consentendo infine una semplice traduzione di Open Linked Data. Nello specifico, XHTML, ovvero l'estensione di HTML con tag personalizzati, derivati dalla libertà di definizione di XML, consente

di definire nuovi tag che possono essere utilizzati per la trasformazione di dati RDF. OWL-PL [BH09] è un linguaggio per la trasformazione e la visualizzazione di dati RDF da pagine XHTML grazie alla definizione di template tramite tag personalizzati. Questo linguaggio consente di costruire pagine HTML che mostrano il risultato delle trasformazioni dei dati RDF definiti all'interno della pagina XHTML sorgente. La sintassi dei template, essendo basata su XSLT, consente tutte le funzionalità che il linguaggio permette su XSLT, però sfruttando la struttura a grafo dei dati su cui si basa.

```
1.  <p class="bold">Alumni:</p>
2.  <ul>
3.    <owlpl:for-each select="&R;hasMember" focus-on="y">
4.      <owlpl:if lhs="{&RDF;type}" op="eq" rhs="&R;Alumni">
5.        <li>
6.          <a owlpl:href="mailto:{&R;hasEmailAddress}">
7.            <owlpl:value-of select="&R;hasFirstName"/>
8.            <owlpl:value-of select="&R;hasLastName"/></a>
9.         (<owlpl:value-of select="&R;hasDegree"/>), 
10.         <owlpl:value-of select="&R;hasCurrentJob" default=""/>
11.       </li>
12.     </owlpl:if>
13.   </owlpl:for-each>
14. </ul>
```

Figura 1.4: Esempio di definizione di un template in OWL-PL

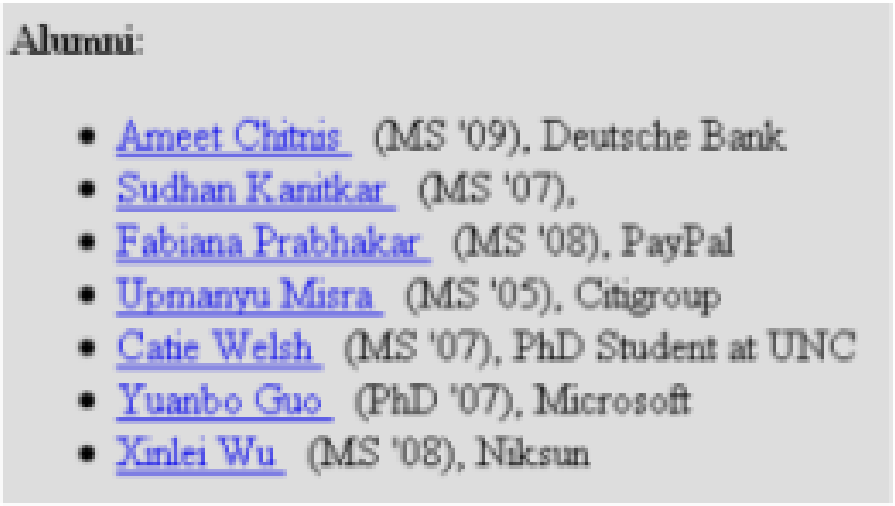


Figura 1.5: Esempio di risultato di una trasformazione in OWL-PL
Immagini prese dall'articolo [BH09].

Come è possibile vedere nell'immagine 1.4, la definizione delle trasformazioni dei dati RDF avviene direttamente all'interno dell'HTML della pagina: i tag personalizzati di OWL-PL permettono di definire la struttura di come mostrare le informazioni utili e, utilizzando appositi CSS e JavaScript, è possibile gestire l'estetica dell'output, come è mostrato nella figura 1.5.
OWL-PL, quindi, consente trasformazioni tramite semplice dichiarazioni di template dichiarativi, con una sintassi molto simile a XSLT: per quanto questa soluzione permetta una definizione semplice di template, la manipolazione di essi avviene esclusivamente durante il parsing del file, prima che la pagina venga restituita al client che ha formulato la richiesta. Una struttura simile non permette trasformazioni dinamiche o richieste successive di altri dati da parte di un utente, senza dover eseguire richieste dati al server che ha le pagine XHTML contenenti i template. Questa limitazione, per trasformazioni di dati condivisi sul web, non consente un utilizzo ottimale di OWL-PL all'interno web application, in quanto sia la struttura dell'output che i template stessi sono implementati solo sul server. Questa caratteristica è da tenere in considerazione prima di utilizzare

questo linguaggio all'interno di un qualche progetto che necessiti trasformazioni di dati RDF in un linguaggio naturale o, comunque, facilmente comprensibile dall'uomo grazie a tecnologie web (es. HTML e CSS).

1.3.2 Integrazione template in HTML

Esattamente come OWL-PL, introdotto nel capitolo 1.3.1, anche RSLT [PV15] definisce un linguaggio di template dichiarativi fortemente ispirato a XSLT, per la trasformazione di dati RDF, utilizzando come linguaggio di output HTML.
Il linguaggio che RSLT (pronunciato come la parola in inglese "result") definisce, è pensato per esser il più simile possibile a XSLT, ma progettato per operare, in maniera efficiente, su grafi. RSLT non è un linguaggio, ma una libreria che, al suo interno, implementa il parser di un linguaggio di templating per dati RDF, andando ad utilizzare particolari query SPARQL per ottenere tutti i dati necessari all'applicazione dei template stessi.
I principi chiave alla base della libreria sono:

- Semplicità di integrazione della libreria all'interno di applicazioni web.
- Chiarezza della rappresentazione di statement RDF.
- Tutti gli IRI devono avere una rappresentazione semplice e comprensibile, ma devono rimanere memorizzati come sono per altri scopi (come link ipertestuali).
- Rappresentazioni complesse di dati devono essere costruite con la combinazione di più visualizzazioni semplici, che gestiscono poche risorse e proprietà.

```
<template match="?person -> foaf:Person">
    <p>Found <valueOf select="count(?person pro:holdsRoleInTime ?r)"></
    valueOf>
    papers authored by
        <valueOf select="?person foaf:givenName ?g"></valueOf>
        <valueOf select="?person foaf:familyName ?f"></valueOf>:
    </p>
    <ul>
```

```
8         <applyTemplates select='?person pro:holdsRoleInTime / pro:
      relatesToDocument ??work'>
9         </applyTemplates>
10     </ul>
11 </template>
```

Codice 1.3: Esempio di definizione di template su RSLT

In RSLT, l'associazione tramite un template e i dati su cui viene applicato, è definita direttamente dal tipo dell'entità RDF che si sta trasformando. Questa particolarità, ben visibile nell'esempio 1.3, permette la definizione di template non in base alle query SPARQL utilizzate, ma al tipo dei dati ottenuti in risposta. Ciò permette di riutilizzare lo stesso template per una particolare tipo di entità ogniqualvolta ce ne sia bisogno, indipendentemente da quale query ha ottenuto quella risorsa.
Dovendosi differenziare da XSLT nella gestione delle informazioni, RSLT richiede la definizione di un template principale: esso è il primo template ad essere eseguito dalla libreria per la trasformazione degli statement RDF e si occuperà di utilizzare gli altri template, ove necessario. Qui si definisce il grafo della "SELECT" SPARQL di cui si vogliono ottenere le informazioni, aggiungendo altri dati da poter richiedere, che possono servire per l'applicazione di altri template nell'esecuzione della trasformazione. Per il recupero dei dati, RSLT esegue particolari query SPARQL, che si occupano di richiedere tutti gli statement legati alle variabili del grafo definito nel template, permettendo quindi di andare a mostrare, nell'output, qualsiasi proprietà RDF si voglia, decidendone anche il formato. La particolarità di RSLT è il fatto di essere una libreria JavaScript per web application: essa infatti consente di poter applicare i template in maniera dinamica, di gestire l'assenza di dati (con un particolare tag) e di permettere l'integrazione completa di questi template all'interno di web application, consentendo l'interazione con i dati da parte di un utente.

RSLT, quindi, consente la generazione di template dichiarativi in maniera semplice e chiara, con una sintassi simile a quella di XSLT e, sfruttando l'integrazione della libreria all'interno di HTML, è possibile sfruttare tale proprietà per decidere, tramite JavaScript e CSS esterni, come questi dati vadano mostrati. Inoltre, è possibile generare sia output

strutturato (tabelle, elenchi puntati, ecc) che testo puro, ed è possibile generare output in qualsiasi linguaggio, che dovrà però esser gestito dalla pagina HTML stessa.

1.4 Confronto fra le soluzioni trovate

Tutte le soluzioni presentate nascono da approcci differenti al problema di generazione di template per dati RDF: ogni soluzione presente in letteratura è indipendente dal linguaggio di output (oppure ne utilizza uno, come l'HTML, che consente di inserire al suo interno qualsiasi altro linguaggio), però ognuna ha vantaggi e svantaggi rispetto alle altre.

XSPARQL (discusso nel capitolo 1.1.1) definisce le trasformazioni tramite un linguaggio procedurale, e non dichiarativo: questo approccio rende molto complicata la generazione di template. Inoltre, esattamente come le altre soluzioni che aggiungono il supporto a dati strutturati a grafo all'interno di un linguaggio nato per strutture ad albero, la gestione e il recupero dei dati è complicata e non ottimizzata. Per queste motivazioni, XSPARQL non è una soluzione adatta ad essere utilizzata su larga scala per il problema discusso in questa dissertazione.

STTL e XSLT+SPARQL (discussi rispettivamente nei capitoli 1.2.1 e 1.1.2) sono, come XSPARQL, proposte di modifiche a linguaggi esistenti per aggiungere il supporto di templating voluto: XSLT+SPARQL è affetta dalla stessa difficoltà di gestione dei grafi di XSPARQL, rendendola non efficace per una trasformazione rapida delle informazioni, mentre STTL, essendo un integrazione di SPARQL, non ha di queste problematiche. Quest'ultima soluzione, però, crea un forte legame fra la query che si occupa di recuperare i dati interessati da un triplestore e il template che si utilizzerà per mostrarli. Questa particolarità, ben visibile nell'esempio 1.1, non consente il riutilizzo dei template: per utilizzare lo stesso template, per un particolare tipo di entità, è necessario definirne due separati, in cui viene modificata la query che ottiene i dati. Considerando la natura dei dati semantici, di cui RDF è standard, ha senso voler applicare dei template ben definiti ad una particolare tipologia di risorse (ad esempio, un template per mostrare i dati ana-

grafici delle persone), e non alla query che recupera i dati dal triplestore. Una divisione fra la query e il template, consentirebbe un miglior riutilizzo dei template, esattamente come è possibile fare in Fresnel, OWL-PL e RSLT.

Fresnel (discusso nel capitolo 1.2.2), consente di definire regole di trasformazione per entità RDF utilizzando un'ontologia RDF appositamente pensata: questa peculiarità, assente in tutte le altre soluzioni, consente di memorizzare template all'interno di triplestore, per poterli utilizzare da qualsiasi applicazione, che si deve solo preoccupare di recuperare i template di cui ha bisogno. Tale funzionalità permette un elevatissimo riuso dei template in diversi ambiti, però la definizione di questa ontologia non consente di sfruttare questi template in maniera semplice per costruire testi in linguaggio naturale, facilmente comprensibili dall'uomo. Questo problema, facilmente visibile nella figura 1.3, è dovuto alla definizione forzata di come mostrare le singole proprietà di ogni entità, e non consentendo una definizione globale di trasformazione che utilizzi più proprietà assieme. Questa peculiarità rende Fresnel un ontologia perfetta per mostrare entità RDF, in una maniera comprensibile anche a un uomo, all'interno di una pagina web, ma non consente dinamicità nell'applicazione di questi template e la costruzione di porzioni di testo grammaticalmente corretto in un qualche linguaggio naturale.

Anche OWL-PL e RSLT (discussi rispettivamente nei capitoli 1.3.1 e 1.3.2) legano i propri template ad una particolare entità, ma questi due linguaggi di trasformazione per RDF, la cui sintassi si ispira a quella di XSLT, permettono di riutilizzare i template, indipendentemente da come i dati da trasformare siano stati recuperati. Nel dettaglio, OWL-PL consente di arricchire file XHTML con particolari tag per la trasformazione dei dati RDF, mentre RSLT arricchisce file HTML con un tag, all'interno del quale è possibile definire i template voluti. Queste soluzioni sono molto simili fra di loro, ma hanno due differenze fondamentali: la prima è che RSLT è una libreria JavaScript, pensata per essere utilizzata all'interno di applicazioni web, mentre OWL-PL, essendo gestito solo sul lato server, non consente una semplice integrazione dei template all'interno di applicazioni dinamiche che richiedono input; la seconda, invece, è che OWL-PL non supporta il recupero di dati tramite query SPARQL, ma solo da una sorgente dati RDF, mentre

RSLT supporta le query e il supporto a file RDF è già in via di sviluppo.

Per questo motivo, RSLT è attualmente il progetto, disponibile in letteratura, più completo per la trasformazioni di dati RDF in un linguaggio naturale. Tale affermazione non significa che questa libreria permetta di risolvere questo problema che affligge il Semantic Web fin dalla sua nascita, ma contiene al suo interno tutti i requisiti necessari per consentire la definizioni di trasformazioni semplici, tramite template dichiarativi, che permettano di rendere qualsiasi informazione RDF facilmente comprensibile da un essere umano. Inoltre, RSLT può essere integrato all'interno di una qualsiasi web application, consentendo anche la creazione di complicati tool per la visualizzazione di dati RDF.
Questa libreria, però, per poter avvicinarsi al raggiungimento della risoluzione del problema discusso, necessita di soluzioni a svariate problematiche, le cui due più importanti sono le seguenti:

- Il supporto a SPARQL è limitato alla versione 1.1 [Gro13], rendendo RSLT non compatibile con la totalità degli SPARQL point.
- Non esiste ancora un supporto a file RDF, locali o remoti.

La prima problematica descritta nasce dall'utilizzo, all'interno delle query generate da RSLT, di un costrutto presente solo nell'ultima versione di SPARQL. Qualsiasi SPARQL point nella versione 1.0 [PS08] non riesce a interpretare correttamente le query, impedendo l'ottenimento alla libreria dei dati necessari al suo funzionamento. L'aggiunta di tali funzionalità permetterebbero a RSLT, quindi, di raggiungere l'obiettivo che si pone di soddisfare: per dimostrare come questa libreria permetta di migliorare la comprensione di dati semantici, tramite template dichiarativi, ho deciso di integrare il supporto a SPARQL 1.0 in RSLT, dando un mio contributo verso la creazione di un sistema in grado di trasformare qualsiasi dato RDF in un formato facilmente comprensibile da un essere umano, utilizzando semplici template dichiarativi, riutilizzabili perché legati al significato semantico che l'informazione stessa rappresenta e non al come la si ottiene.
Tale contributo verrà mostrato e valutato nei prossimi capitoli, andando a spiegare come sia stata possibile l'integrazione del supporto sopra citato e di come ciò sia utile, controllando che non vada ad impattare, a livello computazionale, le prestazioni di RSLT,

fondamentali per l'interattività di un'applicazione basata su di esso.

Abbiamo visto, quindi, come il problema della trasformazione di dati semantici in un linguaggio naturale, consentirebbe di semplificare la vita agli sviluppatori di applicazioni che utilizzano pesantemente tali dati. Tutte le soluzioni mostrate, quindi, offrono meccanismi differenti per poter trasformare questi dati, utilizzando linguaggi o metodologie differenti. Considerando la trasformazione procedurale troppo complessa, sono rimaste come possibili soluzioni tutte quelle basate su template dichiarativi: l'unica, però, a permettere di definire trasformazioni in maniera semplice, indipendenti dalle query utilizzate per recuperare i dati, in grado di generare output in qualsiasi linguaggio (anche se incapsulandolo dentro a HTML) e utilizzabile all'interno di applicazioni web, è RSLT. Questa libreria, però, non è ancora in grado di risolvere, nella sua interezza, il problema esposto, e quindi necessita di modifiche e di funzionalità aggiuntive.

Capitolo 2

RSLT 1.1: panoramica

RSLT, come già detto nel capitolo 1.3.2, è una libreria, sviluppata per l'ambito web, per la trasformazione di dati semantici tramite template dichiarativi. Nasce per poter consentire di non dover eseguire trasformazioni di dati tramite codice applicativo, ma di utilizzare dei template completamente riutilizzabili, che consentono di migliorare sensibilmente l'approccio e la difficoltà di trasformare tali dati.

Adesso andremo ad introdurre le funzionalità offerte da RSLT, mostrando come questa libreria permetta di trasformare dati RDF in un altro linguaggio, in maniera semplice ed efficace. Analizzeremo, inoltre, Playground, una piattaforma di sviluppo basata su RSLT che consente di definire e testare l'esecuzione di template.

2.1 Funzionalità

RSLT consente l'utilizzo di template dichiarativi per eseguire trasformazioni RDF in un linguaggio naturale, consentendo la creazione di porzioni di testi semplici da comprendere rispetto ai dati stessi.

Questa libreria nasce per cercare di consentire, a sviluppatori che lavorano con il Semantic Web, di poter definire delle visualizzazioni per questi dati con una sintassi dichiarativa, direttamente all'interno di pagine HTML, che consenta la rimozione di trasformazioni

dall'interno della logica delle applicazioni. La semplicità della definizione dei template è indispensabile per poter costruire visualizzazioni complesse e articolate, formate dalla progettazione di singoli template più piccoli.

```
<template name="actor">
    <at select="??person movie:actor_name 'Ben Affleck'." preload="??movie
     movie:actor ?person."></at>
</template>

<template match="?person -> foaf:Person">
  <p>{{person['movie:actor_name']}} ha recitato in:</p>
    <ol>
        <at select='??movie movie:actor ?person'></at>
    </ol>
</template>

<template match="?movie -> movie:film">
    <li>{{movie['dcterms:title']}}</li>
</template>
```

Codice 2.1: Esempio di definizione di un template in RSLT

L'esempio appena mostrato definisce un template che genera in output, in un formato comprensibile, dati riferiti ad attori e film. Il template con "name" uguale a "actor" si occupa di definire un grafo RDF, che verrà elaborato ed utilizzato per la costruzione di una query SPARQL (utilizzando sintassi e meccanismi differenti in base a SPARQL 1.0 e 1.1) che si occupa di ottenere tutti i dati necessari alla libreria. Una volta ottenuti tali dati dal triplestore, RSLT controllerà il tipo di risorse ottenute in risposta, andando ad applicare i template corretti per esse.
L'output generato dalla computazione definita nell'esempio 2.1 è il seguente:

```
<p>Ben Affleck ha recitato in:</p>
<ol>
    <li>Forces of Nature</li>
    <li>Smokin' Aces</li>
    <li>State of Play</li>
    <li>Daredevil</li>
    <li>The Sum of All Fears</li>
```

```
8       <li>Chasing Amy</li>
9       <li>Dogma</li>
10      <li>The Town</li>
11      <li>Pearl Harbor</li>
12      <li>Armageddon</li>
13      ...
14  <ol>
```

Codice 2.2: Output dell'esempio 2.1 di template RSLT

Tale output, trattandosi di semplice HTML, consente di essere visualizzato all'interno della pagina web che contiene i template. La possibilità di utilizzare più template semplici per costruirne di più complessi, inoltre, permette una semplice definizione di visualizzazioni arbitrariamente complesse.

Figura 2.1: Esempio di visualizzazione dell'output generato dall'esempio 2.1

RSLT consente anche la gestione dell'assenza di dati, permettendo di definire dell'output anche in caso non si siano trovati i dati richiesti (o il triplestore non sia momentaneamente raggiungibile).
Per sfruttare tale funzionalità, basta inserire il tag "ifnone" all'interno degli "at". Basandoci sui template definiti nell'esempio 2.1, all'interno del tag "at" del template con

"name=actor", è sufficiente inserire il seguente codice HTML per ottenere la gestione richiesta.

```
<ifnone>
    <p>
    Nessun film di Ben Affleck trovato.
    </p>
</ifnone>
```

Codice 2.3: Esempio di gestione di assenza di dati in RSLT rispetto all'esempio 2.1

Nel caso non si dovessero ottenere dati corrispondenti al grafo definito nel template, allora si visualizzerà la rappresentazione definita all'interno del tag "ifnone": la semplicità nel definire tali comportamenti consente la creazione e, quindi, l'utilizzo di template dinamici e capaci di eseguire trasformazioni sensate in qualsiasi situazione.

RSLT consente anche la definizione di più template per lo stesso tipo di entità, consentendo di scegliere quale applicare a seconda della situazione; ciò permette di dare rappresentazioni diverse per lo stesso dato a seconda del contesto linguistico in cui le si vuole inserire. Il seguente esempio mostra come sia possibile sfruttare tale funzionalità di RSLT.

```
<template name="favouriteFilm">
    <at select="??person movie:actor_name 'Michael Caine'." preload="??
    movie movie:actor ?person.">
        <ifnone>
            <p>Non conosco nessun film di Michael Caine, quindi non ne ho
    uno preferito.</p>
        </ifnone>
    </at>
</template>

<template match="?person -> movie:actor">
  <p>
    {{person['movie:actor_name']}} &egrave; un famoso attore di cinema. Ha
     fatto
    <foreach select="??movie movie:actor ?person." collected> {{movie.
    length}} </foreach>
```

```
13      film.</p>
14    <p>Il mio preferito &egrave; <at select="??movie movie:actor ?person."
        mode="verbose"></at>
15    </p>
16    <p>Tutti gli altri film in cui ha recitato sono:
17      <ul><at select="??movie movie:actor ?person."></at></ul>
18  </template>
19
20  <template match="?movie -> movie:film" mode="verbose">
21    <span ng-if="movie['dcterms:title'] == titolo_preferito">{{movie['
        dcterms:title']}}, che &egrave; stato girato nel {{movie['dcterms:date
        '].substr(0,4)}} da
22      <at select="?movie movie:director ??director."></at>.
23    </span>
24  </template>
25
26  <template match="?movie -> movie:film">
27    <span ng-if="movie['dcterms:title']!=titolo_preferito"><li>{{movie['
        dcterms:title']}}</li></span>
28  </template>
29
30  <template match="?director -> movie:director">
31    <span>{{director['movie:director_name']}}</span><span ng-if="!last">, </
        span>
32  </template>
```

Codice 2.4: Esempio di utilizzo di più template per un tipo di risorse

L'output generato dall'esecuzione del template "favouriteFilm", ipotizzando che la variabile "titolo_preferita" assuma valore "Batman Begins", sarà il seguente risultato:

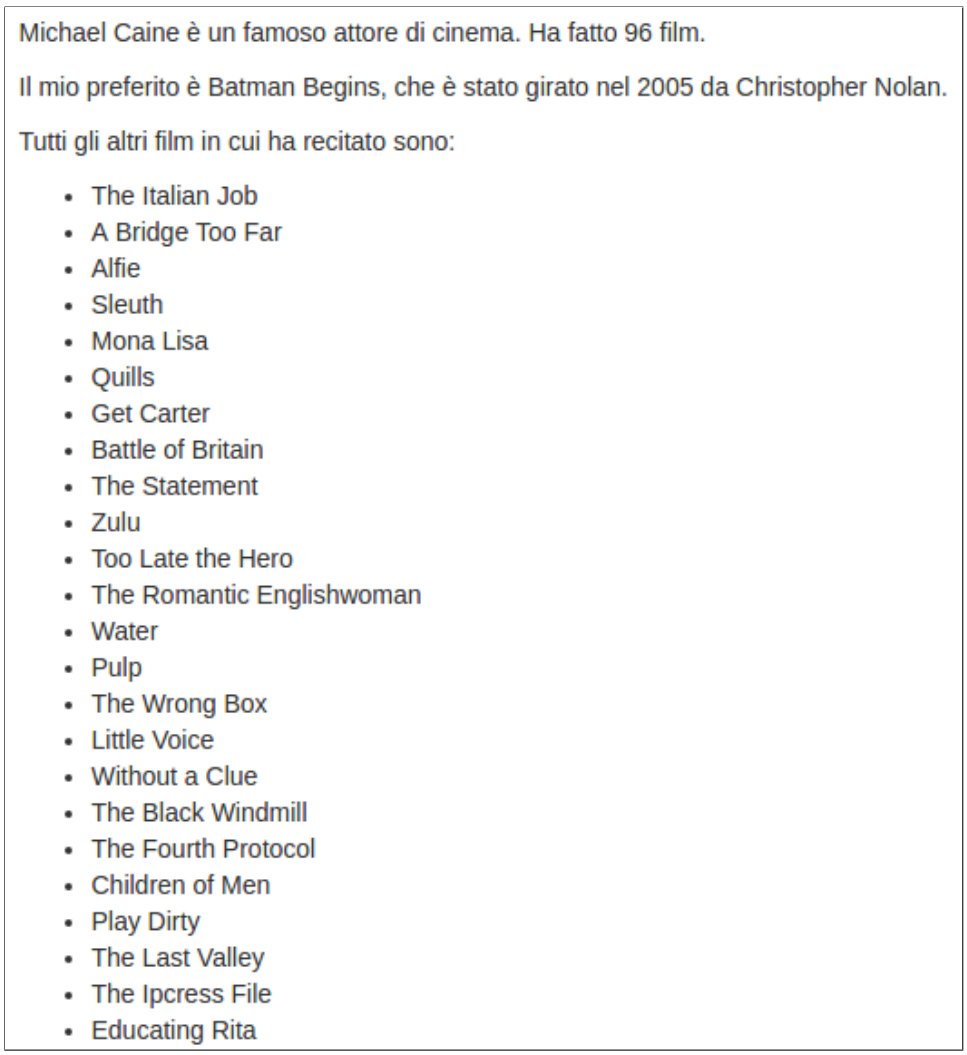


Figura 2.2: Esempio di utilizzo di più template per la stessa entità

Dall'output ottenuto, visibile in figura 2.2, è semplice notare come l'utilizzo di template differenti per la stessa tipologia di entità sia semplice e consenta di rendere più dinamico l'output ottenuto.
RSLT, inoltre, consente la definizione di più template, senza specificare il "mode", per una particolare entità, definendone la priorità; ciò permette di definire un ordine di preferenza nell'applicazione dei template in base alla loro priorità nel testo che si sta generando. Nel caso in cui ci siano più template con lo stesso valore di priorità, RSLT ne sceglierà uno random. Tale casualità può non essere utile (e quindi evitata) per trasfor-

mazioni in linguaggi strutturati (ad es. da RDF a JSON o XML), ma è stata pensata per la generazione di testi naturali, perché consente la generazione di testi diversi fra loro da esecuzioni identiche delle stesse trasformazioni, utilizzando gli stessi dati di partenza. Tale modularità nella generazione dell'output permette di non ottenere documenti con sempre la stessa struttura.

Abbiamo visto, quindi, come RSLT permetta la definizione di template e consenta, molto facilmente, di definire trasformazioni per dati RDF, offrendo funzionalità extra per consentire la dinamicità dell'output. Per poter imparare, però, a scrivere tali template, è necessaria una piattaforma dove scriverli e provarli; nel prossimo capitolo parleremo di RSLT Playground, applicazione web nata per lo sviluppo, il debug e il test di template per la libreria discussa in questa dissertazione.

2.2 RSLT Playground

Per imparare a creare template adatti alle proprie esigenze ed eseguire test su di essi, esiste un'applicazione web chiamata "RSLT Playground". Playground è rilasciato come tool di sviluppo per template di RSLT, ed aiuta gli sviluppatori ad eseguire prove e test per vedere quale sia l'output delle trasformazioni da loro definite.

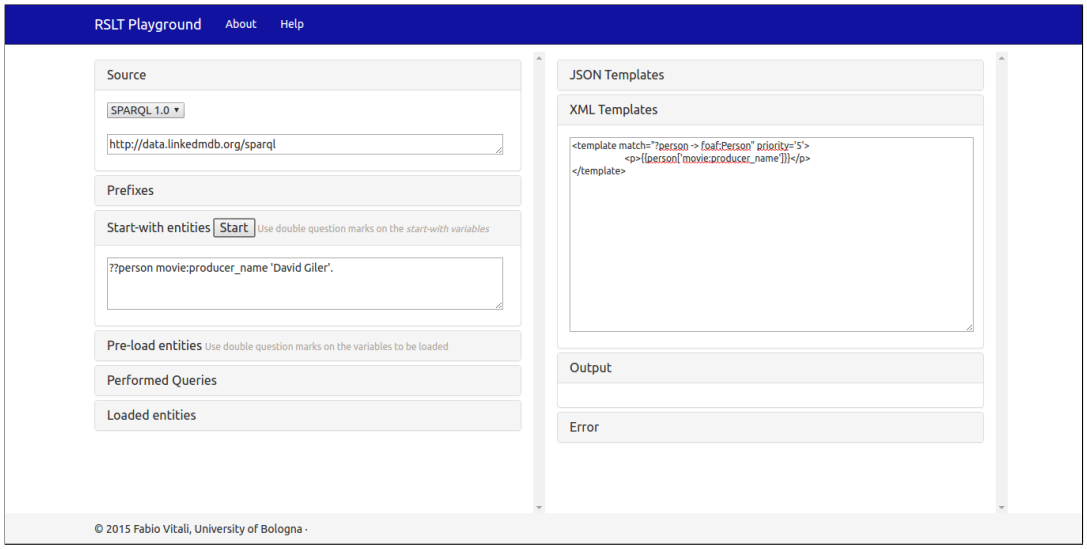


Figura 2.3: Interfaccia di Playground

Quest'applicazione permette di definire i dati da recuperare, mettendo nel riquadro "Start-with entities" ciò che viene definito nell'attributo "select" e in "Pre-load entities" il "preload" di un qualche template; definendo template XML e JSON, consente la trasformazione dei dati ottenuti, mostrando l'output nel riquadro apposito. Essendo uno strumento di sviluppo, Playground permette di visualizzare le query generate ed eseguite verso lo SPARQL point (nel riquadro "Performed Queries") e di navigare fra le entità ottenute in risposta (consentendo di visualizzare tutte le loro proprietà) nel riquadro "Loaded Entities").
Nell'aggiornamento di RSLT alla versione 1.1, anche Playground è stato aggiornato, aggiungendo la selezione della versione di SPARQL utilizzata e risolvendo alcuni bug dovuti

a vecchi errori nell'implementazione della libreria.

Per mostrare come Playground permetta di visualizzare il risultato dell'esecuzione di query e di applicare, ai dati ottenuti, i template definiti negli appositi campi, di seguito viene mostrata l'immagine dell'output dell'esecuzione dei template descritti nell'esempio 2.1, utilizzando come triplestore LinkedMDB (`http://data.linkedmdb.org/`), uno SPARQL point 1.0 contenente dati su film.

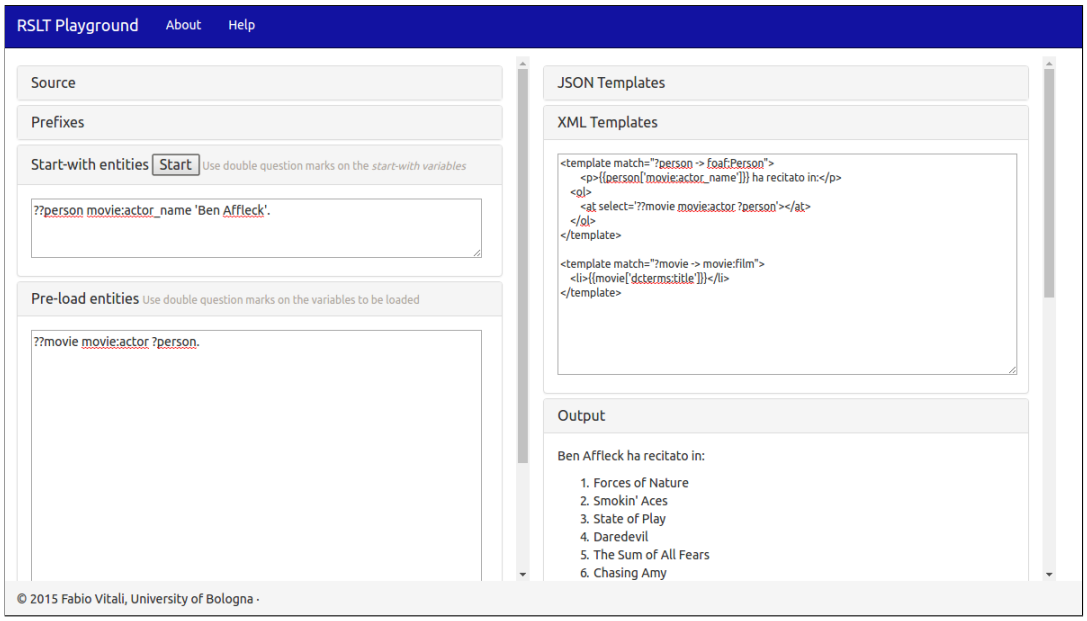


Figura 2.4: Esempio di esecuzione di trasformazione dati su Playground

Playground, quindi, permette di imparare ad utilizzare i template di RSLT e vedere quale risultato si ottiene in output, prima di andare ad utilizzarli all'interno dell'applicazione che si vuole sviluppare.
Utilissima funzionalità, per conoscere tutte i dati ottenuti dalle query eseguite, è la possibilità di visualizzare tutte le entità e le loro proprietà nel riquadro "Loaded Entities". Tutte le risorse RDF sono suddivise per tipo, ed i dati disponibili sono visualizzabili cliccando sull'entità stessa. Conoscere tutte le informazioni di cui si dispone permette di scoprire come migliorare o modificare i template che si stanno creando, dando allo sviluppatore un grande aiuto per migliorare i risultati ottenuti.

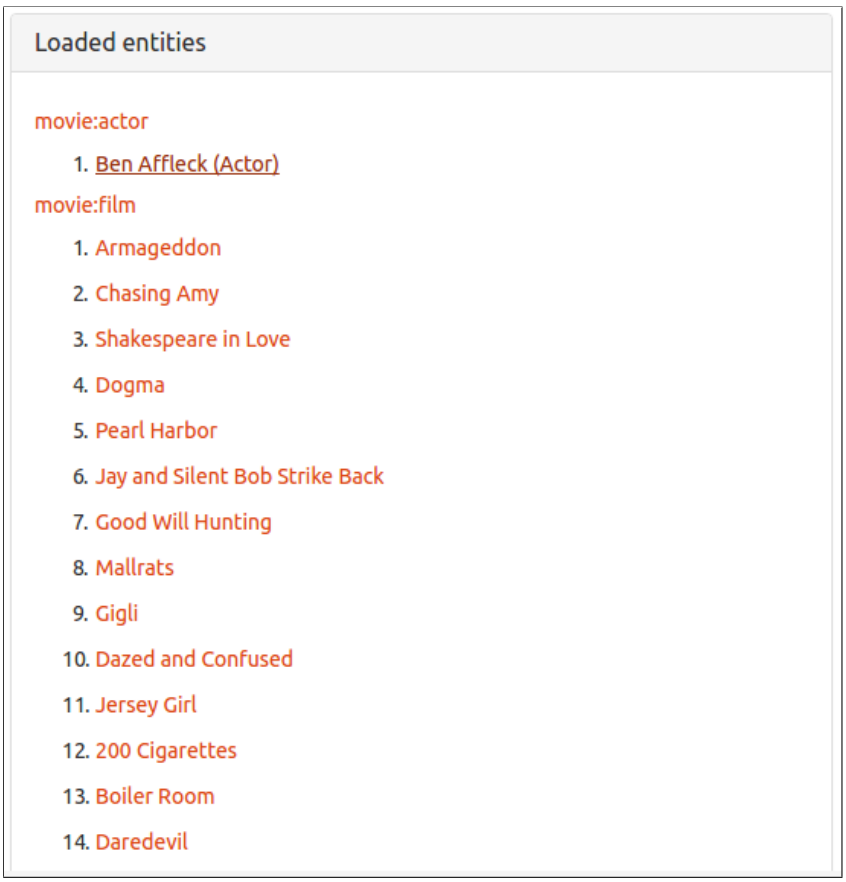


Figura 2.5: Schermata di Playground delle entità caricate

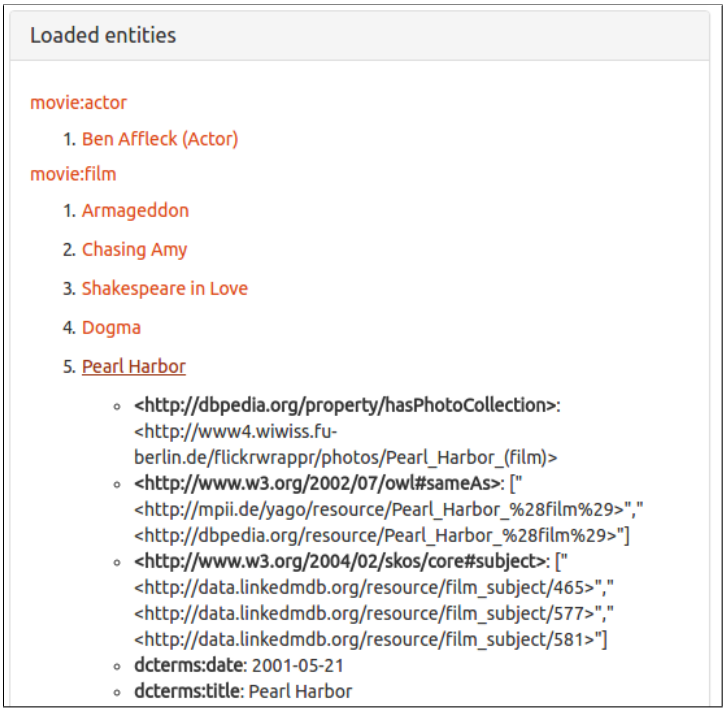


Figura 2.6: Schermata di Playground dei dati delle entità caricate

Playground è, quindi, una piattaforma che consente di generare e testare template per RSLT prima di utilizzarli all'interno di una qualche applicazione. Permette, inoltre, di poter visualizzare tutte le proprietà conosciute per ogni entità, consentendo allo sviluppatore di vedere tutte le informazioni che può sfruttare per la costruzione dei template, aiutandolo a scegliere quali utilizzare e scoprire se, per ottenerne altre, è necessario modificare il grafo SPARQL definito.

RSLT e Playground, quindi, consentono di poter testare la reale capacità di trasformazione di questa libreria, prima di andarla ad utilizzare all'interno di una qualche applicazione. Abbiamo visto come sia possibile generare questi template e come Playground possa aiutare lo sviluppatore in ciò. Gli strumenti offerti da Playground sono di grandissima utilità, e rendono RSLT ancora più interessante e semplice da imparare ad utilizzare.

Capitolo 3

RSLT 1.1: dettagli implementativi

RSLT 1.1, come già accennato nel capitolo precedente, ha introdotto diverse modifiche strutturali nella libreria, oltre ad aver aggiunto funzionalità e risolto problemi causati da errori precedenti. In questa versione viene introdotta, come funzionalità principale, il supporto a SPARQL 1.0.
Vedremo, nel dettaglio, tutte le modifiche apportate da tale aggiornamento, ovvero la nuova struttura di RSLT, le modifiche e le correzioni inserite e, soprattutto, l'integrazione del supporto a SPARQL 1.0. Inoltre, sarà illustrato come sia possibile integrare RSLT all'interno di un'applicazione web e la sintassi dei template, per consentire la creazione di qualsiasi template RSLT sappia riconoscere e gestire.

3.1 Struttura del codice

La nuova struttura del codice di RSLT è stata realizzata per permettere alla libreria di utilizzare tutte le funzionalità offerte da AngularJS. Si parla di "angolarizzazione" del codice, necessaria per motivi di pulizia dello scope globale del JavaScript (prima era tutto memorizzato in variabili globali), che utilizza dei nuovi servizi che rendono necessarie anche modifiche alle direttive, da sempre utilizzate, per poter accedere correttamente ai dati. I servizi presenti in RSLT sono i seguenti:

- Template Dictionary
  Servizio che implementa il dizionario che memorizza tutti i template definiti, inclusi

alcuni template di default, presenti nel codice stesso, che vengono applicati ad una particolare entità solo nel caso in cui non sia stato definito come trasformarla.

- Prefix Dictionary
  Servizio che implementa il dizionario che memorizza tutti i prefissi definiti all'interno di RSLT oltre a "RDF" e "RDFS", che sono sempre necessari alla libreria e, quindi, sempre presenti.

- Triple Store
  Servizio in cui sono memorizzati tutti gli statement RDF ottenuti dalle query, trasformandoli in un dizionario di oggetti contenenti tutti i dati. Offre anche una funzione per memorizzare la versione di SPARQL utilizzata, che viene richiamata in fase di inizializzazione della libreria, e tutte le funzioni necessarie alla ricerca dei dati richiesti da un template. Se i dati richiesti non dovessero essere trovati, si occupa anche di richiedere al servizio che si occupa di generare le query, i dati necessari.

- Query Manager
  Servizio che si occupa di definire funzioni ausiliare all'interno delle classi String e Object di JavaScript, utilizzate all'interno delle direttive.

- Query Generator
  Servizio che si occupa di gestire correttamente la generazione e l'invio delle query allo SPARQL point, gestendone le risposte e differenziando il proprio comportamento in base alla versione del linguaggio di query utilizzata. Tale servizio si è reso necessario dall'integrazione del supporto a SPARQL 1.0: ogni metodo che implementa viene utilizzato dal gestore delle query e, quindi, tutto il codice è strutturato all'interno di switch, che definiscono il comportamento di ogni funzione in base alla versione di SPARQL.

- Select Parser
  Servizio che implementa un parser, generato tramite una specifica libreria JavaScript, PegJS (`http://pegjs.org/`), che si occupa di validare ed estrapolare tutti

i dati necessari alla ricerca delle informazioni necessarie per l'utilizzo del template stesso.

- Template Parser
  Servizio che implementa un parser, generato anch'esso come il Select Parser grazie a PegJS, che si occupa di validare ed estrapolare la definizione di un template, permettendo di categorizzarlo e memorizzarlo nel Template Dictionary.

Il resto della struttura di RSLT, ovvero la gestione di ogni "tag" che è possibile utilizzare nei template e il gestore delle query e delle loro risposte, è implementato tramite le direttive di AngularJS. Per ogni possibile tag utilizzabile nei template, esiste una direttiva che implementa il codice ad esso legato. L'unica direttiva che necessiti spiegazioni ulteriori è "dataMgrFactory", che si occupa di definire "mgr", ovvero il modulo che gestisce le query. In esso vengono definite tutte le funzioni per la creazione delle query e la corretta gestione della loro risposte.

La struttura e i meccanismi implementati all'interno di questo modulo verranno spiegate nelle sezioni successive, in quanto sono il cuore della gestione delle query ed è la componente di RSLT che ha subito più modifiche nell'aggiornamento alla versione 1.1 della libreria, ma è importante aver compreso come i servizi di Angular implementino, ognuno, funzionalità differenti. Inoltre, gli altri componenti della libreria, ovvero quelle basate sulle direttive di AngularJS, si occupano di gestire l'esecuzione dei template e la gestione del recupero dei dati.

3.2 Dalla versione 1.0 alla 1.1

Il passaggio di RSLT dalla versione 1.0 alla 1.1 ha ristrutturato la maggior parte della struttura del codice, utilizzando i servizi di AngularJS e risolvendo diversi bug che impedivano il corretto funzionamento della trasformazione dei dati. Altre funzionalità di minore importanza, come l'aggiunta del LIMIT nelle query, che sono state aggiunte in questa versione di RSLT, hanno permesso alla libreria di raggiungere una migliore

stabilità, oltre all'integrazione del supporto a SPARQL 1.0.

Di notevole importanza è l'introduzione del valore di LIMIT all'interno delle query (sia per SPARQL 1.0 che 1.1), in modo da cercare di impedire che le risposte delle query vadano in timeout. Tale valore è stato scelto e fissato a 2000 triple, ma è possibile ridefinirlo in qualsiasi momento. Tale possibilità è lasciata agli sviluppatori per permettere a RSLT di adattarsi alle prestazioni di ogni triplestore, consentendo valori di LIMIT elevati in caso di SPARQL point molto efficienti e valori bassi per server con poche risorse.
Per risolvere alcuni bug, si è cambiata la sintassi nei template da utilizzare per il recupero delle informazioni dallo scope di AngularJS, ottenendo la sintassi mostrata nell'esempio 2.1, ovvero:

```
{{ variabile_della_query['prefisso:proprieta'] }}
```

Questa modifica ha permesso di risolvere alcuni problemi riguardo l'utilizzo degli IRI con, al loro interno, caratteri speciali che davano problemi e, inoltre, per rendere la sintassi per accedere alle proprietà delle entità più simile a quella di SPARQL.
Infine, sono stati risolti due bug legati all'assenza dei prefissi RDF e RDFS, che vengono utilizzati all'interno del codice stesso di RSLT e che, quindi, ora sono sempre memorizzati all'interno della libreria e non è necessario che siano definiti dall'utente.

Abbiamo visto, quindi, quali miglioramenti siano stati fatti in RSLT 1.1, e come questi cambiamenti influenzano la stabilità e le funzionalità offerte da RSLT.
Nelle prossime sezioni verranno descritte, nel dettaglio, tutte le funzioni e i miglioramenti aggiunti in tale versione di RSLT, ovvero la gestione del LIMIT nelle query e il supporto a SPARQL 1.0.

3.2.1 Gestione timeout query

Tutte le query che RSLT esegue sono ricevute da uno SPARQL point, che deve elaborare la richiesta inviata per poter rispondere. Dovendo gestire quantità di dati completamente arbitrarie, RSLT, essendo un tool generico, deve essere in grado di gestire ed evitare, se possibile, eventuali timeout del server con cui comunica. Possedendo SPARQL stesso un costrutto per limitare il numero di informazioni della risposta (LIMIT) e uno per definire

un offset dal quale cominciare a restituire i dati trovati (OFFSET), si sono modificate le sintassi delle query per gestire entrambi i costrutti all'interno di esse, scrivendo in RSLT un meccanismo di iterazione di chiamate e di gestione delle risposta parziali per la stessa query, con offset differenti.
Tale meccanismo invia, come sempre, una richiesta al triplestore, attendendo la sua risposta: quando arriva, controlla quanti dati si sono ricevuti in risposta. Se il numero di dati è inferiore al valore di LIMIT, allora si sono ottenuti tutti i dati disponibili; se, invece, il numero di statement ottenuti è uguale al valore di LIMIT, è probabile ci siano altri statement da recuperare, per cui si procede a memorizzare i risultati parziali fin'ora ottenuti in un array (identificato, all'interno di un dizionario, da un codice univoco che viene inviato in ogni query, ma che non varia tra un'iterazione e l'altra della stessa), si incrementa il valore di OFFSET e si esegue la nuova richiesta.
Nel caso in cui il numero di statement ottenuto sia uguale al valore di LIMIT, non si ha la certezza che ci siano altri dati, ma solamente una grande probabilità perché, raramente, si potrebbe avere come valore di LIMIT il numero esatto di triple che, per questa query, il server deve restituire. Eseguire un'ulteriore richiesta, in tal caso, è totalmente inutile, e non restituirà dati, ma non è possibile saperlo a priori. Per questa motivazione, quando si riceve un numero di triple pari al valore di LIMIT, si esegue sempre la richiesta di altri possibili dati.
Quando l'ultima iterazione della query restituisce meno triple del valore di LIMIT, allora si sono ottenuti tutti i dati: si recupera tutto l'array di dati concatenati da quelli ottenuti dalle varie richieste e si richiama il gestore della risposta.

La gestione del LIMIT, quindi, avviene tramite un meccanismo abbastanza semplice, che consente di iterare le query con valori di OFFSET e LIMIT differenti, diminuendo le possibilità di timeout delle query. Questa soluzione è ottimale, tranne che nella remota possibilità in cui il valore di LIMIT vada a coincidere col numero esatto di triple che il server ha: in tal caso, si eseguirà una query inutile, che non restituirà nuovi dati.

3.2.2 Gestione SPARQL point 1.0

L'introduzione del supporto a SPARQL 1.0 ha richiesto la gestione di due flussi di esecuzione differenti (uno per ogni versione di SPARQL), oltre all'ideazione e all'implementazione di una nuova sintassi per le query, con stessa capacità espressiva di quella per SPARQL 1.1, ma senza utilizzare il costrutto BIND, disponibile solo nell'ultima versione del linguaggio. Per spiegare da quale problema nasca la necessità di due sintassi differenti, è necessario spiegare come venga usato il BIND nelle query SPARQL 1.1. Per fare ciò, è utile utilizzare un esempio di query inviata dalla libreria.

Per l'esempio, utilizzo il seguente grafo, definito in un qualche tag "at" di un template.

```
1 ??person movie:actor_name "Daniel Radcliffe".
2 ??movie movie:actor ?person;
3     movie:director ??director.
```

Codice 3.1: Esempio di grafo RDF definito in RSLT

La query, a partire da questo grafo, realmente eseguita da RSLT è la seguente:

```
SELECT DISTINCT ?s ?p ?o ?startWith WHERE {
    # graph
    ?person movie:actor_name "Daniel Radcliffe".
    ?movie movie:actor ?person;
        movie:director ?director.

    # unions
    {
        bind(true as ?startWith)
        bind(?person as ?s)   ?s ?p ?o.
    } UNION {
        bind(?movie as ?s)   ?s ?p ?o.
    } UNION {
        bind(?director as ?s)   ?s ?p ?o.
    }
}
```

Codice 3.2: Esempio di query eseguita da RSLT per SPARQL 1.1

Come si può vedere dall'esempio, il grafo del template viene arricchito e trasformato con i BIND in modo che, oltre a restituire tutte le risorse volute, si ottengano tutti i predicati e i valori definiti per esse. Nello specifico, ogni variabile (a parte lo startWith, che serve solo internamente a RSLT), con il BIND, viene considerata come soggetto "?s" della tripla "?s ?p ?o": così facendo si ricevono tutte le informazioni riguardanti tutte le entità di quella variabile e, tramite le UNION, si costruisce il grafo contenente tutte le informazioni dei singoli BIND.
Lo startWith, nel dettaglio, serve solo ed esclusivamente a separare le variabili definite all'interno dell'attributo "select" del template da quelle definite in "preload" e rappresenta, quindi, le entità da utilizzare per iniziare ad applicare template.

Per integrare il supporto a SPARQL 1.0, è prima necessaria una sintassi equivalente che non faccia uso di BIND. La sintassi trovata, dopo svariate prove e tentativi, è la seguente:

```
SELECT DISTINCT ?s ?p ?o WHERE {
  {
    #graph for ?person#
    ?s movie:actor_name "Daniel Radcliffe".
    ?movie movie:actor ?s;
      movie:director ?director.
    #bindings#
    ?s ?p ?o.
  }
  UNION
  {
    #graph for ?movie#
    ?person movie:actor_name "Daniel Radcliffe".
    ?s movie:actor ?person;
      movie:director ?director.
    #bindings#
    ?s ?p ?o.
  }
  UNION
  {
    #graph for ?director#
    ?person movie:actor_name "Daniel Radcliffe".
    ?movie movie:actor ?person;
      movie:director ?s.
    #bindings#
    ?s ?p ?o.
  }
}
```

Codice 3.3: Esempio di query per SPARQL 1.0

Questa sintassi, ad ogni iterazione, in ogni sotto-grafo, prende una delle variabili definite in RSLT col doppio punto interrogativo, la trasforma in "?s", in modo da recuperarne con "?s ?p ?o" tutte le informazioni e, grazie alle UNION, restituisce tutti gli statement legati ad ogni variabile. Questa sintassi, però, non permette la gestione dello startWith, perché non è possibile risalire, dalla risposta, quale statement sia legato a quale variabile

del grafo originale.

Per poter risolvere tale problema, ho pensato di modificare la singola query, spezzandola in un numero maggiore di richieste singole, una per ogni variabile dichiarata con il doppio punto interrogativo nel grafo di RSLT. Utilizzando sempre il grafo descritto nell'esempio 3.1 si ottengono le seguenti query:

- 1° query

```
SELECT DISTINCT ?person ?p ?o WHERE   {
  #graph for ?person#
  ?person movie:actor_name "Daniel Radcliffe".
    ?movie movie:actor ?person;
            movie:director ?director.
  #bindings#
  ?person ?p ?o.
}
```

Codice 3.4: Esempio della 1° query spezzata per SPARQL 1.0

- 2° query

```
SELECT DISTINCT ?movie ?p ?o WHERE  {
  #graph for ?movie#
  ?person movie:actor_name "Daniel Radcliffe".
    ?movie movie:actor ?person;
        movie:director ?director.
  #bindings#
  ?movie ?p ?o.
}
```

Codice 3.5: Esempio della 2° query spezzata per SPARQL 1.0

- 3° query

```
SELECT DISTINCT ?director ?p ?o WHERE   {
  #graph for ?director#
  ?person movie:actor_name "Daniel Radcliffe".
    ?movie movie:actor ?person;
        movie:director ?director.
  #bindings#
  ?director ?p ?o.
}
```

Codice 3.6: Esempio della 3° query spezzata per SPARQL 1.0

In queste query non viene sostituito il nome delle variabili con "?s", lasciando il nome della variabile come definito dal template. Tale modifica permette, memorizzando all'interno di RSLT quali variabili facciano parte dello startWith, di riconoscere quali statement siano riferite a variabili dello startWith (ovvero le risposte a query con, come soggetto, variabili della "select" e non del "preload") o meno. Le query ottenute, essendo completamente indipendenti fra di loro, possono essere eseguite in parallelo, andando a permettere un miglioramento dei tempi richiesti per ottenere tutti i dati.

L'implementazione di tale sintassi, con la parallelizzazione delle query, ha richiesto l'aggiunta del servizio Query Generator, descritto nel capitolo 3.1, e la modifica del gestore delle query. Ogni query che viene generata da un grafo, viene inviata contemporaneamente (per SPARQL 1.1 si continua ad avere sempre e comunque una sola query), e si attende la risposta di tutte le query. Quando tutte le query avranno risposto, e anche tutte le eventuali iterazioni con valori di OFFSET differenti, verrà gestita la risposta ottenuta. La gestione del LIMIT rimane invariata, mentre la condizione di esecuzione della gestione dei dati ottenuti in risposta deve essere cambiata: tale controllo, ora, lo esegue Query Generator che, quando il numero di query da cui si sono ottenuti tutti i dati è pari al numero di richieste inviate, permette l'esecuzione della gestione della risposta.

Per poter sincronizzare l'esecuzione della gestione dei dati alla conclusione di tutte le richieste asincrone, che sono completamente slegate fra di loro, si è utilizzato un deferred. Un deferred è un particolare meccanismo, implementato in AngularJS, che consente

di eseguire codice in base a condizioni generate e controllate dall'esecuzione di codice asincrono. Per questa motivazione, è lo strumento perfetto per poter attendere che ogni risposta sia ricevuta prima di gestire tutti i dati ottenuti.

Abbiamo visto come sia implementato RSLT, come le funzionalità della versione 1.1 siano state introdotte, analizzando la struttura e i meccanismi interni della libreria. Per poter verificare che esso riesca ad adempiere ai propri scopi, è necessario mostrare come l'esecuzione dei template sia estremamente efficiente e che eventuali rallentamenti siano condizionati solo dai tempi di attesa di risposta dalle query. Inoltre, è necessario mostrare un'applicazione vera e propria che utilizzi la libreria, mostrando come sia davvero possibile integrare RSLT ed utilizzarlo in un'applicazione web interattiva.

3.3 Integrazione di RSLT in applicazioni web

RSLT, essendo una libreria basata su AngularJS, può essere integrata all'interno di una qualsiasi applicazione web. Per integrare correttamente RSLT all'interno di un qualsiasi altro tool, è necessario prima soddisfare alcuni requisiti, che sono elencati qui di seguito.

- Includere le dipendenze.
  RSLT, oltre ad AngularJS, ha come dipendenze jQuery(`https://jquery.com/`) e RDF-Ext(`https://github.com/rdf-ext/rdf-ext`): è necessario includerle nel progetto che si sta sviluppando. Si noti come RDF-Ext sia una libreria modulare: l'unico modulo utilizzato in RSLT è il parser XML, e quindi è necessario avere solamente questo modulo all'interno della versione di RDF-Ext che si include. Per semplicità, RSLT viene rilasciato con una versione di tale libreria, già correttamente configurata, nei file sorgente.

- Definire l'applicazione di AngularJS.
  Perché i servizi di AngularJS che RSLT utilizza vengano correttamente associati all'applicazione che si sta sviluppando, è assolutamente necessario chiamare tale applicazione "myQuery", includendo "rslt" come dipendenza, perché ogni servizio di RSLT è associato a quel nome. Il seguente esempio, spiega come procedere.

```
//...
angular.module('myQuery',['rslt']);
//...
```

Codice 3.7: Esempio della definizione del modulo di AngularJS per RSLT

- Specificare la versione di SPARQL utilizzata.
  Definire che tipo di sorgente dati si andrà ad utilizzare permette a RSLT di sapere quale sintassi per le query utilizzare (tra quella per SPARQL 1.1 e quella per SPARQL 1.0). Tale scelta va salvata, nella definizione del controller, in "$scope.$root.selectedSource". I valori accettati sono descritti dall'array "sourceTypes", memorizzato nel servizio "QueryGenerator". Per avere accesso a tale servizio, basta richiederlo nella definizione del controller. Il seguente esempio mostra come si possa procedere.

```
//...
angular.module('myQuery').controller('controllerName',['$scope', '
    QueryGenerator', function($scope,QueryGenerator) {
    //...
    console.log(QueryGenerator.sourceTypes)
    //['SPARQL 1.1','SPARQL 1.0']
  //ad esempio, dichiaro di utilizzare la versione 1.0
  $scope.$root.selectedSource=QueryGenerator.sourceTypes[1];
  //...
}]);
//...
```

Codice 3.8: Esempio di definizione della versione di SPARQL da utilizzare all'interno di RSLT

Con questi 3 semplici passi, RSLT è già pronto per essere utilizzato. Per dare maggiore libertà di sviluppo e di utilizzo della libreria agli sviluppatori, sono possibili le seguenti azioni all'interno di RSLT.

- Cambio del valore di LIMIT.
  Per cambiare il valore di limit di deafult (ovvero di 2000 triple), è sufficiente richiamare la funzione "setQueryLimit" del servizio QueryGenerator, passando come

parametro il valore desiderato. Il seguente esempio mostra come cambiare il valore di LIMIT con un valore di 50000 triple.

```
//...
angular.module('myQuery').controller('controllerName',['$scope', '
    QueryGenerator', function($scope,QueryGenerator) {
     //...
  QueryGenerator.setQueryLimit(50000);
  //...
}]);
//...
```

Codice 3.9: Esempio di personalizzazione del valore di LIMIT in RSLT

- Cambio del proxy per l'esecuzione delle query.
  Per gestire l'esecuzione di query ad un triplestore raggiungibile ad un dominio diverso da quello dell'applicazione, è necessario utilizzare un qualche meccanismo di proxy per impedire che il browser blocchi la richiesta (vedasi `https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS`). RSLT viene rilasciato con un semplice script server-side, scritto in PHP, che si occupa di risolvere questo problema: è possibile, però, cambiarlo ed utilizzarne un altro, e tale scelta viene lasciata allo sviluppatore. Bisogna considerare che all'URL del proxy verrà concatenato quello della query: esso deve, quindi, essere in grado di gestire tale URL. Il seguente esempio mostra come sia possibile personalizzare il proxy utilizzato.

```
//...
angular.module('myQuery').controller('controllerName',['$scope', '
    QueryGenerator', function($scope,QueryGenerator) {
     //...
  console.log(QueryGenerator.proxy)
  // "../../proxy.php?q="
  QueryGenerator.proxy = "path/to/proxy?query_url=";
  //se non si vuole utilizzare alcun proxy, basta scrivere
    QueryGenerator.proxy = "";
     //...
}]);
```

```
10 //...
```

Codice 3.10: Esempio di utilizzo di un proxy personalizzato in RSLT

- Accesso diretto ai dati ottenuti dalle query.
  Per permettere agli sviluppatori di accedere ai dati ottenuti dalle query SPARQL, è possibile definire come dipendenza del controller il servizio "Triplestore", che contiene al suo interno tutti i dati ottenuti. Per accedere ad una risorsa SPARQL memorizzata in questo servizio, si utilizza l'URI della risorsa. Ogni risorsa è un oggetto JavaScript, le cui proprietà sono i predicati RDF, descritti tramite prefisso, se definito, altrimenti tramite URI esteso. Il seguente esempio mostra come sia possibile accedere ad un attributo di una entità.

```
1 //...
2 angular.module('myQuery').controller('controllerName',['$scope', '
    QueryGenerator', 'Triplestore', function($scope,QueryGenerator,
    Triplestore) {
3     //...
4     var entity = Triplestore['<http://data.linkedmdb.org/resource/
    actor/29392>'];
5   console.log(entity['movie:actor_name']);
6   //"Babe Ruth"
7   //...
8 }]);
9 //...
```

Codice 3.11: Esempio di accesso ad una risorsa nel Triplestore locale di RSLT

- Esecuzione dinamica dei template.
  RSLT permette di poter eseguire dei template anche dal codice JavaScript, e non solo tramite tag "calltemplate". Per utilizzare tale funzionalità, è necessario dichiarare "rslt" come dipendenza del controller e andare a definire lo scope di AngularJS per settare tale funzione. La funzione "calltemplate" richiede tre parametri:
  - Il nome del template da eseguire.
  - Il selettore CSS che identifica dove verrà mostrato l'output generato.

– Un valore booleano che indica se rimpiazzare o appendere in coda l'output generato all'interno del contenitore. Il contenuto viene sostituito con valore della variabile "true".

L'esempio seguente mostra come sia possibile richiedere e utilizzare tale funzione.

```
//...
angular.module('myQuery').controller('controllerName',['$scope','
    QueryGenerator','Triplestore','rslt',function($scope,
    QueryGenerator,Triplestore,rslt) {
    //...
    var calltemplate = rslt.calltemplate.bind($scope);
    calltemplate('template_name','#output_container_selector',true);
    //true definisce che il contenuto dell'elemento definito da #
    output_container_selector vada rimosso
    //Utilizzando false, l'output viene appeso all'attuale contenuto.

  //...
}]);
//...
```

Codice 3.12: Esempio di esecuzione di un template di RSLT da JavaScript

Con le operazioni appena mostrate, quindi, è possibile notare come sia semplice integrare RSLT all'interno di una web application; inoltre, le funzionalità aggiuntive che la libreria offre, consentono agli sviluppatori di avere più libertà possibile nella progettazione e nella creazione delle loro applicazioni.

3.4 Sintassi template

La sintassi dei template, come spiegato nel capitolo 1.3.2, cerca di essere il più simile possibile a XSLT, utilizzando comandi e proprietà del linguaggio suddetto. La sintassi è definita da alcuni tag HTML, da inserire all'interno della web application che utilizza RSLT. I tag, e le loro proprietà, sono descritti di seguito:

- rslt: è il tag principale di RSLT, all'interno del quale si possono definire i template. Esso deve avere l'attributo "triplestore", che va a definire l'URL dello SPARQL point da interrogare.
- prefix: permette di definire un prefisso SPARQL attraverso l'attributo "ns" (namespace).
- calltemplate: permette di richiedere l'esecuzione di un template, specificandone il nome con l'attributo "name". Un template senza "name" non è applicabile tramite calltemplate. Questo tag richiede anche la definizione dell'attributo "renderTo", che indica il selettore CSS che definisce dove sarà mostrato l'output.
- template: Permette la definizione di un nuovo template per dati RDF. Per ogni template, è possibile definire le seguenti proprietà:
  - name: definisce il nome del template, usato per la direttiva "calltemplate". Il primo template che viene eseguito all'avvio di RSLT si deve chiamare "start".
  - match: definisce per quali tipo di risorse il template va applicato, specificando il nome della variabile con cui si accederà ai dati richiesti.
  - mode: definisce la modalità di utilizzo del template. Se nel momento della selezione dei dati, per i quali verrà applicato un template, si definisce il "mode", si utilizzerà il template con quel particolare valore di mode.
  - priority: definisce la priorità del template, nel caso ci siano più template con lo stesso "mode" per lo stesso tipo di entità. A parità di priorità, RSLT sceglierà casualmente quale template applicare.
- at: detto anche "apply template", permette di selezionare alcuni dati legati a quello attuale tramite sintassi SPARQL (per farlo, bisogna definire la proprietà

"select"). Qui si può definire il "mode" del template che si dovrà applicare ai dati ottenuti. All'interno di questo tag è possibile, anche, definire il tag "ifnone", che definisce cosa bisogna mostrare nel caso in cui non si sia trovato nemmeno una entità corrispondente alla "select".

- foreach: permette di definire, all'interno di questo tag, un template per ogni dato che si ottiene dall'attributo "select" del tag stesso. Il foreach supporta l'attributo "collected", che invece che eseguire il template all'interno del tag una volta per ogni elemento, lo fa solo una volta, salvando nella variabile definita nella select un array con tutti i dati corrispondenti trovati.

Oltre a questi costrutti e le loro proprietà, si possono utilizzare, all'interno dei template tutte le funzionalità di AngularJS, mescolando il linguaggio di output con i comandi offerti dalla libreria. I template si scrivono in due linguaggi, XML e JSON, ed hanno la stessa capacità espressiva.

In conclusione, abbiamo illustrato la struttura di RSLT, mostrando da quali componenti è formato e le responsabilità che ognuno di essi ha. Tale struttura è stata ottenuta nell'aggiornamento di RSLT alla versione 1.1, che ha risolto bug legati ai prefissi, introdotto il supporto al LIMIT e integrato una sintassi per poter gestire anche SPARQL point 1.0. Oltre alla struttura e alle funzionalità che RSLT, si è discusso di come sia possibile integrare, con pochi e semplici passi, tale libreria in una qualsiasi applicazione web; inoltre, è possibile anche poter personalizzare alcuni parametri della libreria (come il valore di LIMIT o il proxy utilizzato per le query) ed avere direttamente accesso al dizionario dei dati ottenuti dalle query.

Capitolo 4

Valutazione prestazioni query

L'integrazione del supporto di SPARQL 1.0, introdotto in RSLT 1.1, ha permesso di raggiungere il supporto a qualsiasi SPARQL point nell'applicazione, permettendo di avvicinare RSLT all'obiettivo di poter trasformare qualsiasi dato RDF in un qualsiasi linguaggio naturale, permettendo una facile comprensione di tali dati ad un essere umano; perché tale risultato sia accettabile, però, è necessario che RSLT sia veloce e reattivo per permettere all'utente di interagire con un'applicazione vera e propria. Per questi motivi, è necessario valutare le prestazioni che RSLT riesce ad ottenere, e valutare se siano risultati accettabili o meno.

In questo capitolo, quindi, valuteremo, con dati oggettivi a supporto, la velocità di RSLT, scoprendo se i risultati che si ottengono permettano di RSLT di essere utilizzata come libreria all'interno di applicazioni web, che necessitano di reattività per l'interazione da parte di un utente. Vedremo, inoltre, LANCET, un semplice e minimale data browser di dati semantici in formato RDF.LANCET è la prima applicazione basata su RSLT in versione 1.1, ed è una prova reale di come possa tale libreria essere perfettamente utilizzata in un'applicazione di questo tipo.

4.1 Valutazione tempi di RSLT

Il supporto a SPARQL 1.0 ha reso necessaria la definizione di una nuova sintassi per le query, che genera più query separate, nelle quali viene definito lo stesso grafo più volte: a meno di meccanismi particolari di caching, implementati all'interno degli SPARQL point, non si ha un grande livello di ottimizzazione. Il basso livello di ottimizzazione delle query è dovuto dal fatto che, ogni singola richiesta, ridefinisce lo stesso grafo, ma essendo tutte richieste differenti (anche solo per uno statement), il server deve ricercare gli stessi dati ogni volta. Nonostante questo problema di efficienza, l'integrazione di questa sintassi per SPARQL 1.0 è comunque necessaria per poter permettere a RSLT di funzionare con moltissimi SPARQL point. Ottenere un livello di prestazioni abbastanza elevato da permettere la reattività richiesta da una web application, è fondamentale per la libreria.
Per valutare, quindi, l'efficienza della soluzione utilizzata, si sono presi i tempi di esecuzione di due template differenti, con diverso numero di triple: il quantitativo di dati differente serve per poter controllare come, cambiando il carico di lavoro della libreria, cambino i tempi totali necessari a completare la trasformazione dei dati, separando il tempo passato ad aspettare le risposte delle query da quello di computazione della libreria.

I template utilizzati per ottenere i dati sono i seguenti:

- 1° template

```
<template name="actor">
  <at select="??person rdf:type movie:actor" preload="?person
    movie:actor_name ?name. ??movie movie:actor ?person.">
  </at>
</template>

<template match="?person -> movie:actor">
  <h3>{{person['movie:actor_name']}} had acted in:</h3>
      <ol>
        <at select='??movie movie:actor ?person'></at>
    </ol>
```

```
</template>

<template match="?movie -> movie:film">
  <h3>{{ movie['dcterms:title'] }}</h3>
  <p>The movie was released on {{movie['dcterms:date']}}.
  </p>
</template>
```

Codice 4.1: 1° template per valutazione efficienza RSLT

- 2° template

```
<template name="director">
    <at select="??person rdf:type movie:director" preload="?person
    movie:director_name ?name. ??movie movie:director ?person."></at>
</template>

<template match="?person -> movie:director">
    <p>{{person['movie:director_name']}} ha girato:
        <ol>
            <at select='??movie movie:director ?person'></at>
        </ol>
    </p>
</template>

<template match="?movie -> movie:film">
    <li>{{movie['dcterms:title']}} :
        <ol>
            <li ng-repeat="(key,val) in movie">{{key}} : {{val}}</li>
        </ol>
    </li>
</template>
```

Codice 4.2: 2° template per valutazione efficienza RSLT

Per andare ad applicare un quantitativo di dati differente di triple ad ogni template, si è sfruttato il costrutto "FILTER" di costrutto SPARQL, che permette di definire regex da applicare a particolari variabili. I filtri utilizzati sono i seguenti:

- 1° filtro

```
FILTER regex(?name,"^Dav")
```

- 2°filtro

```
FILTER regex(?name,"James")
```

- 3°filtro

```
FILTER regex(?name,"^O")
```

Definiti i template e i filtri da utilizzare, si sono raccolti i tempi richiesti dall'intera trasformazione, utilizzando come valore di LIMIT 10.000 e come SPARQL point LinkedMDB, separando il tempo passato in attesa di risposte dalle query dal tempo di esecuzione del codice, tenendo anche in considerazione il numero di statement RDF restituiti dalle query.

I risultati ottenuti sono i seguenti:

Template	Filtro	Tempi (s)			N° Triple
		Totale	Query	Computazione	
1	1	19,16	18,10	1,06	17.800
	2	42,04	41,7	0,34	8.995
	3	23,50	22,11	1,39	13.936
2	1	144,10	143,82	0,28	5.392
	2	22,62	21,48	1,14	4.594
	3	6,17	5,84	0,33	776

Dai dati raccolti è semplice notare come la maggior parte del tempo di esecuzione dei template risieda nell'attesa della risposta delle query SPARQL, mentre il salvataggio dei dati e la generazione dell'output occupino una parte del tempo veramente minima. I seguenti grafici mostrano, appunto, la relazione fra i tempi di esecuzione delle query e della computazione di RSLT.

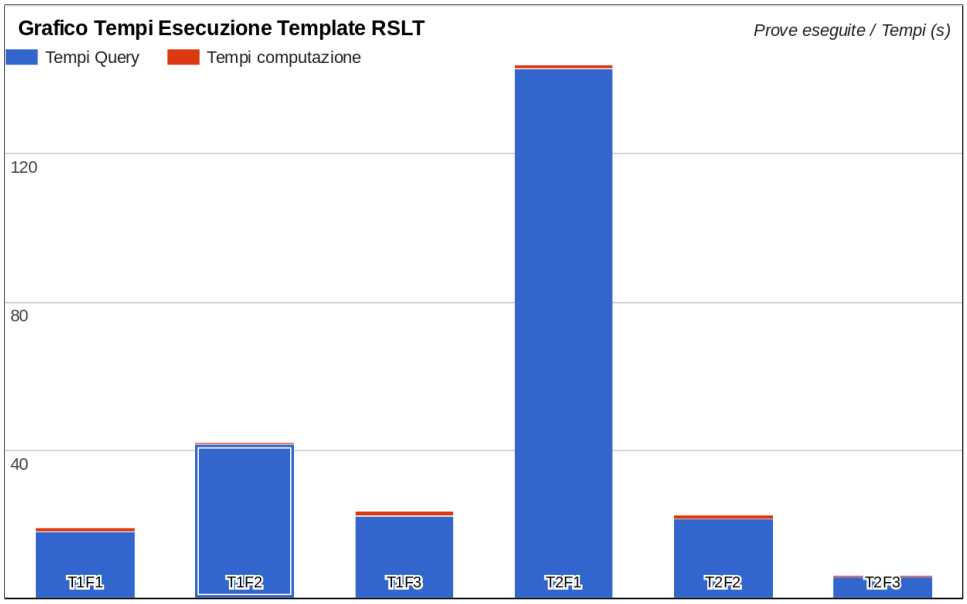


Figura 4.1: Grafico dei tempi richiesti dalle query e di esecuzione di RSLT
Le etichette usate all'interno del grafico indicano quale template (T) e quale filtro (F) siano stati utilizzati per ottenere quei valori.

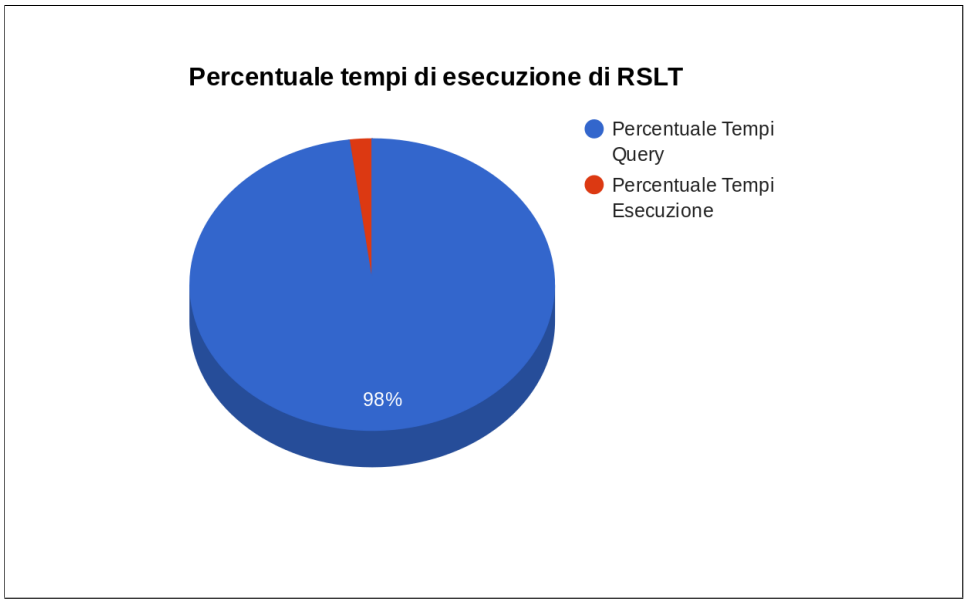


Figura 4.2: Grafico delle percentuali complessive dei tempi richiesti dalle query e di esecuzione di RSLT

Come ben visibile in figura 4.2, è inequivocabile che la maggior parte del tempo di trasformazione dei dati (98%) è dovuta all'attesa di risposta delle query eseguite: considerando che RSLT parallelizza tutte le richieste, e non essendoci una sintassi che permetta un'ottimizzazione delle richieste, non è possibile migliorare questi tempi di attesa in maniera significativa.
Per tale motivo, RSLT raggiunge già un ottimo risultato riguardo l'efficienza della computazione dei template e riesce, quindi, ad essere utilizzato all'interno di un'applicazione interattiva. Ogni eventuale tempo di attesa presente nell'utilizzo di RSLT è dovuto solo all'efficienza del triplestore da cui si recuperano i dati necessari all'applicazione dei template.

RSLT, quindi, riesce a trasformare dati RDF velocemente; tutti i tempi di attesa sono dovuti al triplestore specifico con cui comunica. Per mostrare, ora, l'effettiva capacità della libreria di essere utilizzata all'interno di applicazioni interattive, nella prossima sezione se ne mostra una funzionante, che permette l'interazione continua da parte dell'utente con l'output generato dai template.

4.2 LANCET Data Browser (for SPARQL 1.0)

Per mostrare come sia realmente possibile integrare RSLT 1.1 all'interno di una web application, ho modificato il LANCET Data Browser, un'applicazione che era già basata su RSLT 1.0, adattandola alla nuova versione della libreria e riscrivendone i template.

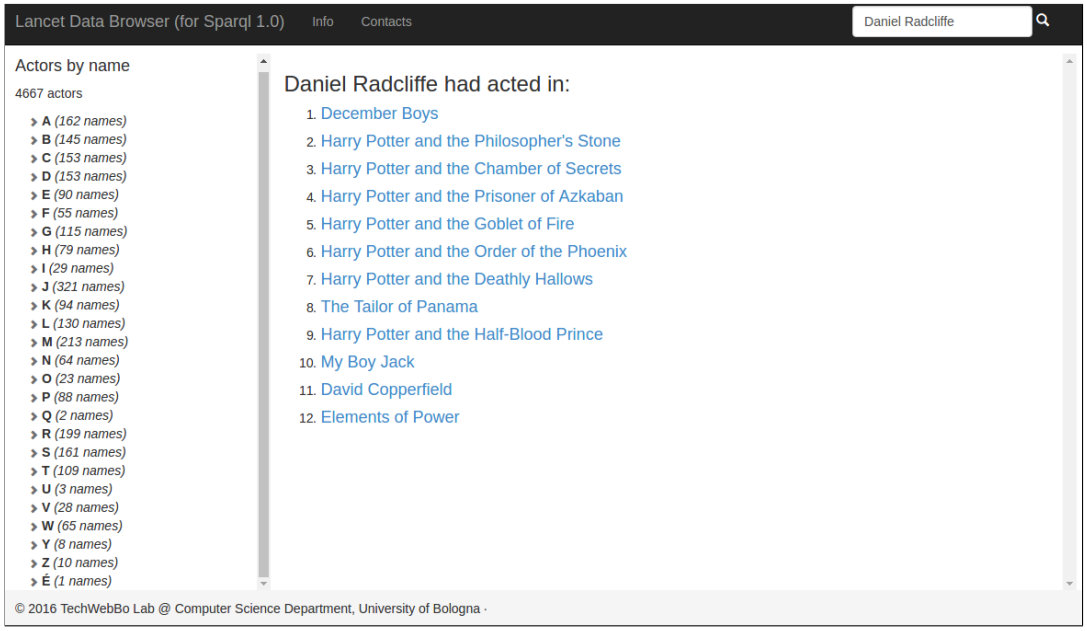


Figura 4.3: Schermata iniziale di LANCET

LANCET è un browser di informazioni dinamiche: permette di navigare tra i dati contenuti in uno SPARQL point sfruttando i template definiti con RSLT. Inizialmente sviluppato per informazioni riguardanti la pubblicazione di trattati scientifici, è stato adattato per gestire informazioni riguardanti attori, film, registi e produttori presenti all'interno di LinkedMDB.
L'applicazione offre, a sinistra, la lista di tutti gli attori conosciuti, dando anche la possibilità di cercarne uno nell'apposita barra di ricerca, in alto a destra. Una volta selezionato un attore, viene visualizzato l'elenco dei film in cui ha partecipato: cliccando sul titolo di uno di essi, è possibile visualizzarne ulteriori dettagli. Le informazioni disponibili nel dettaglio di un film sono:

- Elenco dei registi.

- Elenco dei produttori.
- Elenco degli attori e i personaggi che hanno interpretato.

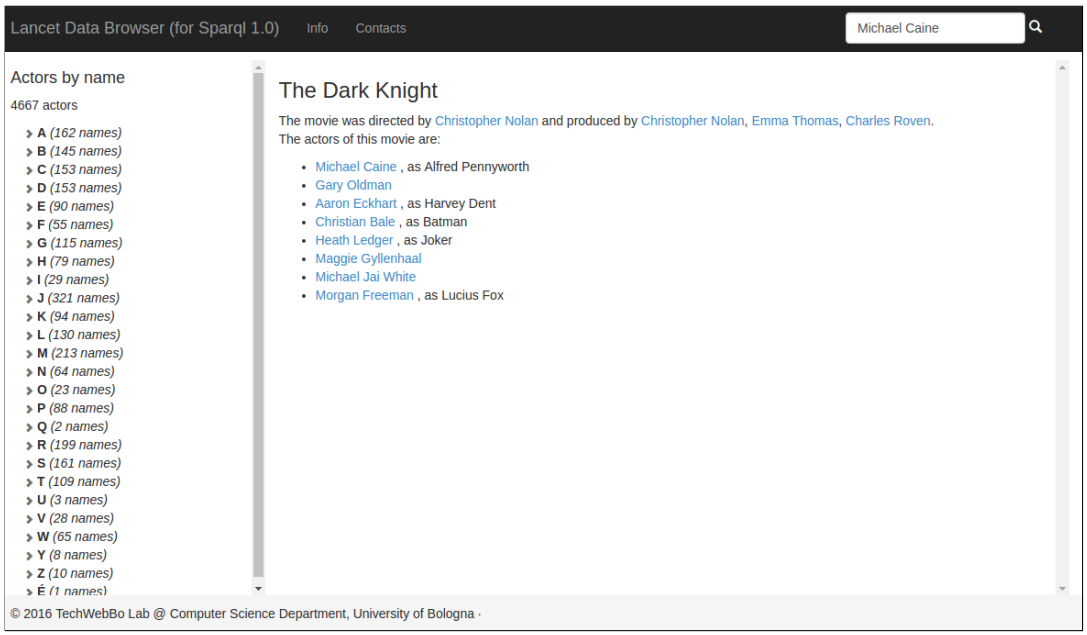


Figura 4.4: Schermata di LANCET dei dettagli di un film

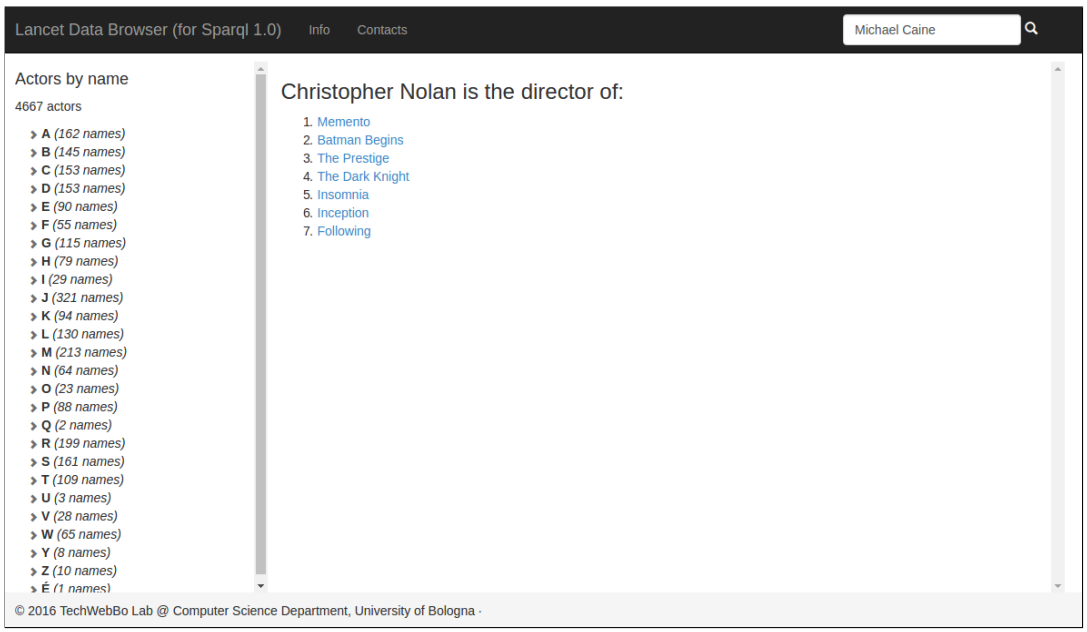


Figura 4.5: Schermata di LANCET dei dettagli di un regista

LANCET offre anche la possibilità di visualizzare l'elenco dei film di un particolare regista o produttore, andando a scandagliare le informazioni del triplestore e rendendole facilmente visualizzabili per l'utente. I template che utilizza sono stati pensati per gestire anche l'assenza di informazione, nel caso i dati siano mancanti o il triplestore non risponda. I seguenti template, presenti in LANCET, sono i responsabili dell'output in figura 4.4.

```
<template name="actor">
  <at select="??person movie:actor_name '{{$root.name}}'" preload="??movie
     movie:actor ?person.">
    <ifnone>
      <h3>We found no film of {{$root.name}}.</h3>
    </ifnone>
  </at>
</template>

<template match="?person -> movie:actor">
  <h3>{{person['movie:actor_name']}} had acted in:</h3>
        <ol>
```

```
12          <at select='??movie movie:actor ?person' mode='simple'></at>
13        </ol>
14</template>
15<template match="?movie -> movie:film" mode='simple'>
16  <li>
17    <a href="#movie/{{movie['uri'] | printable }}">
18      <h4>{{ movie['dcterms:title']}}<span ng-if="movie['dcterms:date']">
    ({{movie['dcterms:date'].substr(0,4)}})</span></h4>
19    </a>
20  </li>
21</template>
```

Codice 4.3: Template utilizzati da LANCET

La definizione dei template è davvero semplice e mostra quanto sia semplice definire visualizzazioni complesse tramite un numero arbitrario di template più semplici.

Abbiamo visto, quindi, come il problema prestazionale di RSLT non risieda all'interno della libreria, ma nelle prestazioni delle query, che a volte risultano lente e creano grandi lassi di tempo di totale inattività della libreria. Quest'inefficienza delle query non è risolvibile, in quanto non esiste una sintassi alternativa più performante, ma con la stessa capacità espressiva. Prova di ciò è il LANCET Data Browser, prima applicazione basata su RSLT 1.1, che permette di navigare fra i dati di un triplestore, dando la possibilità di interagire ad un utente con essi. Ogni singola informazione viene mostrata in un linguaggio naturale, facilmente comprensibile e indispensabile per la comprensione dei dati stessi.

Capitolo 5

Conclusioni

Il problema della trasformazione di dati RDF in un testo in linguaggio naturale, utile per consentire all'uomo di poter comprendere semplicemente tali dati, è molto importante. La crescita del Semantic Web sta facendo aumentare sensibilmente la mole di questi dati, ed ogni giorno sono sempre più importanti. Le soluzione che abbiamo discusso nel capitolo 1 consentono di avvicinarsi ad una soluzione del problema, ma non ne permettono la risoluzione. Ognuna di tali proposte offre grosse funzionalità, ma ha anche alcuni limiti. In questa dissertazione abbiamo discusso, nel dettaglio di RSLT, introdotta nel capitolo 1.3.2, mostrando come tale libreria sia la più vicina ad una soluzione semplice per la generazione di template dichiarativi per dati RDF, che consenta dinamismo nella generazione dell'output, in grado di gestire qualsiasi linguaggio (integrandolo all'interno di codice HTML) e adatto ad un utilizzo nell'ambito delle web application. Il mio contributo, che ha aggiunto a RSLT il supporto a SPARQL 1.0, ristrutturato il codice, migliorato la stabilità e risolti alcuni bug, ha permesso al progetto di avvicinarsi al suo scopo di consentire la visualizzazione in un linguaggio naturale di dati semantici.
I dati mostrati nel capitolo 4, mostrano come il problema principale dell'ultima versione di RSLT, ovvero la lentezza nella trasformazione dei dati, è dovuto alle prestazioni del triplestore e ai tempi di risposta alle query, e non alla computazione vera e propria. Il grafico 4.2 mostra come il 98% del tempo di esecuzione dei template sia passato ad attendere le risposte dalle query. RSLT ha quindi un implementazione abbastanza efficiente, e l'unico problema che la rallenta è la sintassi, per SPARQL 1.0, delle query: la sintassi

attualmente utilizzata è la migliore di quelle trovate, soprattutto grazie al meccanismo di invio delle richieste in parallelo, ma non si può essere certi che sia la più efficiente in assoluto. Essendo un problema noto, questo, RSLT cerca di poter ridurre tale problema al minimo, consentendo la personalizzazione del valore di LIMIT delle query.
Con LANCET, nel capitolo 4.2, è stato possibile dimostrare come l'utilizzo di tali template sia adatto alla costruzione di un browser di informazioni semantiche, consentendo una semplice rappresentazione di dati RDF. Tale applicazione, inoltre, dimostra come sia semplice creare applicazioni in grado di trasformare una buona varietà di dati senza dover scrivere codice applicativo.

In conclusione, RSLT 1.1 raggiunge, come obiettivo, la compatibilità con tutti le versioni dello standard SPARQL, avvicinando il prodotto al suo obiettivo di trasformare qualsiasi dato RDF tramite template dichiarativi, permettendo interattività e dinamicità dell'esecuzione degli stessi. RSLT può avere tempi di attesa di ricezione dei dati abbastanza elevati, ma tale problema è dovuto al triplestore dal quale si recuperano i dati, e non alla libreria stessa. Ottimizzare ulteriormente, ove possibile, la libreria, permetterebbe di diminuire i tempi di attesa per l'utente, ma essendo già attualmente il 98% del tempo passato in attesa di risposta dallo SPARQL point, la situazione non può subire miglioramenti sensibili, perché il problema risiede nelle performance dei triplestore e nella sintassi delle query eseguite. Questo limite non significa, però, che l'attuale versione di RSLT non offra moltissimi spunti per ulteriori migliorie e per l'aggiunta di nuove funzionalità. Di seguito verranno presentate alcune delle future modifiche che porteranno a migliorare la libreria a livello di prestazioni, funzionalità e versatilità.

- Rimozione di jQuery.
  jQuery, viene utilizzato in maniera molto marginale in RSLT e, nel dettaglio, solamente per stampare l'output dei template e per eseguire le query con chiamate sincrone. Queste particolari chiamate, però, sono presenti solo per motivi di debugging dei primi stadi di sviluppo: questa situazione le rende completamente deprecate, e rimovibili, lasciando l'unica funzionalità di jQuery la stampa di un template nell'elemento definito.
  Credo sia importante considerare ciò che il sito `http://youmightnotneedjquery.`

com/ cerca di spiegare, ovvero che jQuery è una libreria corposa e pesante, e che se viene utilizzata per pochissime e semplici funzionalità, si può usare JavaScript puro. Per questo motivo (e perché il sito stesso mostra come implementare un selettore di elementi della pagina HTML partendo da un selettore CSS), jQuery può essere completamente rimosso da RSLT.

- Aggiungere "first" nei template.
  In un template è possibile esprimere condizioni legate alla variabile booleana "last": essa risulta essere verificata solo quando si sta eseguendo il template sull'ultima entità ottenuta dal triplestore. Questa funzionalità è molto comoda, ed è per questo motivo che aggiungere il supporto anche a "first", permetterebbe di offrire più modularità ai template, permettendo una maggiore libertà espressiva, consentendo anche eventuali assenza di particolari proprietà, fondamentali invece per l'output del template.

- Ottimizzare gestione risposta SPARQL 1.0
  Attualmente, la risposta alle query con sintassi compatibile con SPARQL 1.0 (che è in formato XML) viene "parsata" e, per ogni tripla in XML, la si converte in un oggetto identico a quelli descritti nelle risposte alle query SPARQL 1.1 (che sono in formato JSON): solo dopo aver eseguito questa conversione, si fa il parsing dei dati ottenuti e li si salva all'interno del triplestore locale. Per grandi quantità di dati è ovvio che questa gestione sia lenta, in quanto ha un ordine di grandezza di $O(n^2)$. Si potrebbe ottimizzare questo meccanismo evitando la conversione di risposta da SPARQL 1.0 a 1.1, andando a salvare i dati nel triplestore direttamente dalla risposta in XML.

- Aggiungere supporto per file RDF
  Attualmente RSLT recupera i dati solo da SPARQL point con l'utilizzo di particolari query: potrebbero esserci situazioni in cui i dati RDF su cui si lavora sono statici, cioè subiscono cambiamenti molto raramente, e andarli ad ottenere tramite query rallenta, inutilmente, l'applicazione. Per questo motivo, aggiungere il supporto a file RDF (in diversi formati), può offrire prestazioni decisamente mi-

gliori, perché tutti i dati a disposizione sarebbero direttamente caricati durante l'inizializzazione di RSLT.

- Definizione dei template anche in RDF oltre che XML e JSON.
  Può essere utile, permettere la definizione di template, oltre che XML e JSON, in RDF, in modo da poter andare a poter utilizzare template disponibili su triplestore (ovviamente, ciò, richiederebbe la definizione di un ontologia apposita).

Bibliografia

[AL08] Faisal Alkhateeb e Sébastien Laborie. «Towards extending and using SPARQL for modular document generation». In: *Proceedings of the 2008 ACM Symposium on Document Engineering, Sao Paulo, Brazil, September 16-19, 2008*. A cura di Maria da Graça Campos Pimentel, Dick C. A. Bulterman e Luiz Fernando Gomes Soares. ACM, 2008, pp. 164–172. ISBN: 978-1-60558-081-4. URL: http://doi.acm.org/10.1145/1410140.1410174.

[BGH08] Diego Berrueta, Jose Emilio Labra Gayo e Ivan Herman. «XSLT+ SPARQL: Scripting the Semantic Web with SPARQL embedded into XSLT stylesheets». In: *4th Workshop on Scripting for the Semantic Web* (2008).

[BH09] Matt Brophy e Jeff Heflin. *OWL-PL: A presentation language for displaying semantic data on the web*. Rapp. tecn. Technical report, Department of Computer Science e Engineering, Lehigh University, 2009.

[Bis12] Stefan Bischof et al. «Mapping between RDF and XML with XSPARQL». In: *J. Data Semantics* 1.3 (2012), pp. 147–185. URL: http://dx.doi.org/10.1007/s13740-012-0008-7.

[CF14] Olivier Corby e Catherine Faron-Zucker. «SPARQL Template : un langage de Pretty Printing pour RDF». fr. In: (giu. 2014). https://hal.inria.fr/hal-01015267.

[CF15] Olivier Corby e Catherine Faron-Zucker. «STTL: A SPARQL-based Transformation Language for RDF». In: *11th International Conference on Web Information Systems and Technologies* (mag. 2015).

[CFG15] Olivier Corby, Catherine Faron-Zucker e Fabien Gandon. «A Generic RDF Transformation Software and its Application to an Online Translation Service for Common Languages of Linked Data». en. In: (ago. 2015). https://hal.archives-ouvertes.fr/hal-01187393.

[GB14] Ramanathan Guha e Dan Brickley. *RDF Schema 1.1*. W3C Recommendation. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/. W3C, feb. 2014.

[Gro13] The W3C SPARQL Working Group. *SPARQL 1.1 Overview*. W3C Recommendation. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/. W3C, mar. 2013.

[Kay07] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. W3C Recommendation. http://www.w3.org/TR/2007/REC-xslt20-20070123/. W3C, gen. 2007.

[LHL01] Tim Barners Lee, James Hendler e Ora Lassila. «The Semantic Web[A new form of Web content that is meaningful to computer will unleash a revolution of new possibilities]». In: *Scientific American* (mag. 2001).

[Pie06] Emmanuel Pietriga et al. *Fresnel: A Browser-Independent Presentation Vocabulary for RDF*. ARTCOLLOQUE. 2006. URL: http://hal.inria.fr/inria-00125465/en/.

[PS08] Eric Prud'hommeaux e Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. W3C, gen. 2008.

[PV15] Silvio Peroni e Fabio Vitali. «RSLT: R. D. F. Stylesheet Language Transformations». In: *Proceedings of the ESWC Developers Workshop 2015 co-located with the 12th Extended Semantic Web Conference (ESWC 2015), Portorož, Slovenia, May 31, 2015*. A cura di Ruben Verborgh e Miel Vander Sande. Vol. 1361. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 7–13. URL: http://ceur-ws.org/Vol-1361.

[Rob14] Jonathan Robie et al. *XQuery 3.0: An XML Query Language*. W3C Recommendation. http://www.w3.org/TR/2014/REC-xquery-30-20140408/. W3C, apr. 2014.