Università degli Studi di Bologna
Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in

Informatica

Sperimentazione e Sviluppo di Strumenti per la Gestione di Metadati

Tesi di Laurea di

Domenico Lordi
18 luglio 2002

Relatore:

Chiar.mo Prof. Andrea Asperti

Parole chiave:

Metadati, Risorse, RDF, Database, Linguaggi di Interrogazione

---

I Sessione
Anno Accademico 2001–2002

A Gregorio e Maria Cristina

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Web e metadati

Con le specifiche accomunate dal titolo '*Semantic Web*' il World Wide Web Consortium è intento in uno sforzo organizzativo che coinvolge molte tecnologie e standard largamente diffusi (HTTP, XML, ...) finalizzato alla realizzazione di un nuovo sistema di gestione delle informazione. Partendo dalla citazione fatta dai tre principali promotori di questa attività (Tim Berners-Lee, James Hendler e Ora Lassila): '*The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation*' [SemWeb] lo scenario prospettato dalla realizzazione del Semantic Web vede, insieme agli utenti umani, anche la presenza attiva di meccanismi automatici in grado di 'comprendere' (seppure limitatamente ad un certo contesto applicativo) il contenuto informativo dei dati che hanno a disposizione, e sulla base di questo, compiere scelte opportune. Le motivazioni che hanno portato alla nascita di questa attività sono riconducibili all'espansione del web registrata negli ultimi anni e alla possibilità di trasferire in esso un numero sempre maggiore di dati, situazione questa che comporta il problema di reperire le informazioni volute scartando i contenuti non desiderati.

Il notevole sviluppo di quelli che sono comunemente denominati 'motori di ricerca' (ad esempio Google, `http://www.google.com`) e che in parte soddisfano le necessità dell'utente finale, non rappresenta una soluzione efficiente al problema introdotto precedentemente: spesso le tecnologie di ricerca si basano su un insieme di parole chiave (a volte suggerite dall'autore delle pagine web, eventualmente tramite il tag `META` del linguaggio HTML) che si suppone siano rappresentative dei contenuti informativi di un sito o di una pagina. I meccanismi di ricerca, quindi, operano in un *dominio sintattico*, in cui i parametri di ricerca vengono espressi come parole (*item*) da cercare e la bontà del risultato è espressa come ugualianza fra item e parole chiave. Ovviamente, e come spesso è possibile constatare, non sempre tale meccanismo fornisce risultati utili: questo comporta che l'utente operi una ulteriore scelta fra le alternative presentate dal motore di ricerca oppure che riformuli la '*query*' di ricerca tipicamente sostituendo o aggiungendo altri item. Inoltre, l'esempio proposto non è che un caso particolare di ricerca: se si pensa al gran numero di linguaggi e tecnologie capaci di presentare le informazioni nei modi più diversi (XHTML per visualizzazione di ipertesti, Voice XML per la trasmissione audio, il protocollo WAP e il linguaggio WML per i dispositivi wireless come cellulari e palmari) risulta evidente che un approccio come quello descritto, oltre che impreciso, ha anche un limitato campo di azione.

Tuttavia, nonostante queste limitazioni, le tecniche utilizzate nei motori di ricerca sono utili per comprendere in che direzione orientare gli sforzi per ottenere soluzioni migliori.

I metadati I motori di ricerca focalizzano la loro attenzione non direttamente sui dati a cui è interessato l'utente finale, bensì su un insieme di informazioni che descrivono gli stessi (le parole chiave): una strada sicuramente interessante per conseguire l'obiettivo proposto può essere quella di estendere questo principio e costruire un insieme di dati che descrivano le informazioni reperibili nel modo più preciso e puntuale possibile. Questo procedimento porta alla realizzazione di *metadati*, (letteralmente, dati sui dati) ossia

informazioni necessarie alla descrizione di altre informazioni. Nell'esempio dei motori di ricerca affrontato in precedenza, le principali limitazioni sono costituite dal fatto che i possibili metadati sono troppo semplici per dare descrizioni precise del contenuto informativo di una risorsa e, come detto, non viene preso in considerazione il significato di ciò che si vuole cercare, ma si compie unicamente una ricerca tramite match di stringhe. La realizzazione di un sistema che vada incontro alle esigenze del progetto Semantic Web deve, inoltre, tenere in considerazione che il web ha una natura fortemente decentralizzata: le informazioni reperibili in esso sono facilmente pubblicabili da chiunque, con formati e semantiche non necessariamente conformi ad un unico modello. Per eliminare queste incompatiblità è necessario non solo un formato comune per la rappresentazione delle metainformazioni, ma tale formato deve permettere ad un generico utilizzatore di ricostruire il significato originario delle metainformazioni senza possibilità di fraintendimenti dovuti, ad esempio, alla scelta di termini ambigui o dal doppio significato (si pensi alla parola inglese '*sentence*' che può essere tradotta in italiano sia come '*frase*' che come '*sentenza*' riferita ad un atto giuridico).

Il formato RDF Alla luce di questi problemi, il W3C ha approvato uno standard per la rappresentazione di metainformazioni denominato *RDF* (Resource Description Framework) il cui duplice obiettivo è quello di fornire da un lato un formato comune (tipicamente basato su XML) per la rappresentazione di metadati e dall'altro massimizzare l'interoperabilità fra sistemi sviluppati in modo indipendente da più parti. In particolare, per raggiungere questo secondo obiettivo, il W3C ha affiancato al framework RDF un'ulteriore specifica denominata *RDF Schema* il cui obiettivo è proprio quello di permettere all'utente di definire un insieme di termini utilizzati nella descrizione dei propri metadati e di specificare per ciascuno di essi il significato che assumono quando impiegati all'interno di un modello RDF. In questo modo, anche per un utente che non conosce il significato dei termini impiegati nella descrizione dei dati, l'interrogazione e l'analisi dello schema permettono di

ricostruire il significato delle metainformazioni.

Dal punto di vista pratico, l'utilizzo del formato RDF si suddivide nella stesura di due tipi di documenti:

1. preparazione di un *vocabolario* di termini specifici del contesto che si vuole rappresentare; in terminologia RDF questo richiede la preparazione di uno o più **schemi**

2. realizzazione della base di metadati che, utilizzando i termini definiti nello schema RDF, descrive le risorse del contesto applicativo a cui si riferiscono; questo secondo tipo di documenti RDF viene denominato **modello** e, al suo interno, vengono specificati (secondo le modalità previste nell'uso dei namespace XML [XMLNS]) quali siano gli schemi RDF cui si riferiscono.

Le problematiche esposte che hanno portato alla nascita del formato RDF si presentano in tutti quegli ambienti aperti e distribuiti in cui si sia presente un sistema per la trasmissione di informazioni. In particolare, l'utilizzo del World Wide Web come piattaforma per lo scambio di dati pone problemi della stessa natura per qualsiasi applicativo che decida di interfacciarsi ad esso.

1.2 Il progetto HELM

Il progetto HELM si colloca nel campo dell'automazione dei ragionamenti formali e della meccanizzazione della matematica cercando di eliminare il divario esistente fra i numerosi strumenti specifici di questo settore come i proof-assistant (che essendo stati ideati e realizzati prima dell'avvento del WWW non fanno propri concetti come la condivisione di risorse e l'interoperabilità con applicativi diversi), e le potenzialità offerte dalla rete Internet e in particolare dal Web in cui lo scambio di risorse fra nodi distinti rappresenta uno dei maggiori punti di forza. In particolare, l'obiettivo principale

del progetto HELM consiste nella realizzazione di una libreria distribuita di conoscenza matematica realizzata tramite l'uso del linguaggio XML e dei linguaggi di mark-up che da esso derivano.

1.2.1 Il formato della libreria elettronica

Alla base di qualsiasi formalizzazione della matematica ci sono *teoremi* e *definizioni*, che rappresentano le più piccole entità direttamente identificabili. La rappresentazione di singole unità in oggetti distinti (tipicamente file in formato XML) è il modo più naturale di organizzare le informazioni, ma spesso è utile poter avere una visione d'insieme di un certo gruppo di definizioni e teoremi per formare quelle che vengono definite teorie. Una *teoria* organizza in modo opportuno teoremi e definizioni in modo da rispettare le dipendenze logiche fra gli elementi che contiene e mettendo in evidenza eventuali legami come l'ereditarietà o l'inclusione. Questo concetto viene sfruttato anche nell'implementazione di applicazioni come i proof-assistant che internamente organizzano le informazioni di cui hanno bisogno in *moduli*, ossia una collezione di definizioni e teoremi correlati.

Sebbene il concetto di teoria permetta di organizzare le informazioni di cui si ha bisogno nel modo più strutturato possibile, spesso differenti applicazioni usano differenti formati per i moduli, rendendo impossibile lo scambio di dati fra programmi diversi. Per ovviare a questo problema, l'utilizzo delle teorie viene sostituito dall'utilizzo di *documenti*, la cui caratteristica è quella di avere una struttura meno rigida di quella di un modello: in un documento teoremi e definizioni vengono utilizzati tramite referenziazione, e sebbene il documento in sé non pone in evidenza i legami di dipendenza fra gli elementi che contiene, è possibile affidare questo compito al meccanismo di reference.

1.2.2 Scenario

Una libreria elettronica a contenuto matematico si compone di un elevato numero di documenti che, assieme, formano una base di dati sulla quale si è tipicamente interessati a compiere operazioni di ricerca, mentre operazioni di cancellazione o di aggiornamento rappresentano funzionalità di minore interesse: infatti i dati salienti per la descrizione di un documento sono le dipendenze logiche degli oggetti che contiene rispetto ad altri elementi della base di dati e tali dipendenze sono immutevoli poiché una dimostrazione corretta esprime legami fissi fra teoremi e definizioni coinvolti nella prova. Questa caratteristica consente di concentrare l'analisi e gli sforzi implementativi unicamente nella direzione di ottimizzare l'esecuzione di query che interrogano la base di dati senza alterarne il contenuto.

Sono quindi ipotizzabili due distinti scenari:

1. da un lato è possibile immaginare un semplice utente che tramite un client (ad esempio, un browser web o un applicativo specifico) vuole compiere ricerche all'interno della libreria per trovare informazioni su argomenti (teoremi o dimostrazioni) di suo interesse; in questo caso l'accesso alla libreria elettronica si può paragonare all'utilizzo di un tradizionale motore di ricerca per trovare informazioni nel web e si può immaginare che il sistema preposto alla gestione di ricerche di questo tipo operi con simili modalità;

2. un secondo scenario vede invece come utente principale della libreria elettronica non una persona, ma un'applicativo come un proof-assistant che compie interrogazioni mirate e necessita di avere risultati esatti alle proprie query.

Le differenze fra i due use-case presentati sono:

- **informative**: un utente può essere interessato a compiere ricerche legate ad informazioni di natura personale (ad esempio, avere la lista

dei documenti che sono stati redatti da un certo autore) oltre che indagare la struttura interna di un documento (ad esempio, sapere quali sono le dipendenze fra due elementi di una teoria); un proof-assistant compie unicamente ricerche del secondo tipo;

- **qualitative**: mentre nel primo caso può essere sufficiente fornire un risultato che si avvicini a quello desiderato dall'utente (il che significa che eventualmente può includere riferimenti a documenti non richiesti, in analogia con quanto accade nell'utilizzo di un motore di ricerca per pagine HTML), nel secondo è necessario costruire un risultato esatto;

- **prestazionali**: la precisione nella ricerca deve essere accompagnata dalla maggiore velocità possibile nell'esecuzione di una query quando l'utente della libreria è un software. Fermo restando che ottimizzazioni in questo senso sono comunque desiderabili, va sottolineato che l'accesso alla libreria elettronica da parte di un applicativo è tipicamente finalizzato all'esecuzione di altre operazioni (nel caso di un proof-assistant, la continuazione di una dimostrazione): il tempo per l'esecuzione di una query in questo caso deve rappresentare la minima percentuale di tempo utilizzato dal sistema software per concludere le proprie operazioni, richiedendo quindi ottimizzazioni più spinte.

L'utilizzo di metadati per la ricerca e l'espressione di relazioni di dipendenza risulta quindi essere una scelta inevitabile; in [HELM] è possibile constatare come il supporto per le URI e l'utilizzo del formato RDF siano già stati previsti come sistema di supporto per l'accesso alla libreria elettronica.

1.3 Presentazione della tesi

Sebbene l'utilizzo di RDF come formato di rappresentazione e scambio risulti essere uno standard sufficientemente potente e diffuso (viene impiegato in molti progetti per la memorizzazione di informazioni di svariata natura, dalle

preferenze personali di Mozilla [Mozilla] alla descrizione del set Dublin Core [DublinCore]) da poterlo considerare un punto fermo nella trattazione delle problematiche per la gestione di metadati, non esistono ancora strumenti o approcci nel trattamento di documenti RDF che si possano considerare altrettanto stabili e consolidati. Per trattamento di documenti RDF si intende l'insieme di operazioni che si desidera si possano eseguire una volta che la base di metadati raccolta sia stata organizzata in modelli e schemi RDF; tipicamente questo significa avere a disposizione un meccanismo di interrogazione, composto da un sistema di supporto ai metadati e da un linguaggio di interrogazione.

Il presente lavoro di tesi è volto ad analizzare strumenti che permettano la gestione di metadati in formato RDF con riferimento alle problematiche specifiche del progetto HELM. Nel capitolo 2 viene presentato in dettaglio lo standard RDF come framework per la rappresentazione di metadati; nel capitolo 3 si evidenziano gli aspetti principali che un sistema per la gestione di metadati deve avere; nel capitolo 4 viene presentata una suite completa di prodotti indirizzata all'arichiviazione e interrogazione di modelli e schemi RDF; il capitolo 5 discute lo sviluppo del sistema di gestione metadati per il progetto HELM.

Capitolo 2

Lo standard RDF

Lo standard RDF nasce come evoluzione della tecnologia PICS (Platform for Internet Content Selection) [PICS] che consente di descrivere il contenuto di una pagina Web in termini di *rating*, ossia come indice di gradimento da parte dell'utente: un server web che adotta questa tecnologia invia al client (browser) le informazioni sul contenuto delle pagine che sono state richieste e il client decide se la pagina web debba essere scartata perché non soddisfa i criteri dell'utente, oppure possa essere considerata interessante. RDF amplia il campo di applicazione di PICS sia perché può essere utilizzato per descrivere risorse che non sono necessariamente pagine web, sia perché permette di dare una migliore struttura, stabilendo una gerarchia fra gli elementi che vengono descritti.

2.1 Modelli RDF

I modelli RDF si basano sul concetto di *risorsa*: per i fini a cui ambisce lo standard RDF è stato scelto di definire una risorsa come qualsiasi cosa (concreta o astratta) a cui si possa associare un identificativo univoco. Questa definizione è molto generica e virtulamente comprende qualsiasi cosa immaginabile per cui è necessario un meccanismo di identificazione sufficientemente

potente da poter referenziare non solo oggetti reali o espressi in formato digitale, ma anche concetti astratti (come, ad esempio, il diritto d'autore) che non trovano rappresentazione in entità concrete. Per tutti questi motivi lo standard RDF prevede che l'identificazione di risorse avvenga tramite l'utilizzo di URI (come descritte in [URI]) che permettono esattamente di costruire identificatori in grado di referenziare qualsiasi tipo di oggetto, concreto o astratto.

Un modello RDF si compone di *statement* che costituiscono l'elemento base per fornire delle descrizioni sulle risorse. La struttura di uno statement è molto simile a quella di una frase in linguaggio naturale: ogni statement ha un *soggetto* (la risorsa che si vuole descrivere), un *predicato* (l'attributo della risorsa che si vuole mettere in evidenza) e un *oggetto* (il valore dell'attributo). La possibilità di assegnare a qualsiasi oggetto una URI fa si che ciascuna delle tre componenti di uno statement sia interpretabile come una risorsa; per comodità, lo standard RDF prevede che il valore dell'oggetto di uno statement possa anche essere un tipo semplice come definito dallo standard XML [XML]. In questo modo si può decidere fino a che livello di dettaglio si vuole dare una descrizione dei dati, scegliendo che il valore di un oggetto sia una risorsa quando si vuole, ad esempio, esprimere ulteriori proprietà su di esso (ad esempio, l'autore di un libro può essere descritto oltre che dal nome anche dalla nazionalità, indirizzo mail, ...), oppure assegnargli un valore semplice (ad esempio, una stringa) quando non si voglia dettagliare ulteriormente la descrizione dei dati. Questa caratteristica mette in evidenza un parallelo fra gli statement RDF e le strutture dati ad albero: possiamo pensare che il soggetto di uno statement sia un nodo da cui esca un arco orientato (il predicato) diretto verso l'oggetto, che può essere un nodo intermedio quando il suo valore è espresso da una risorsa, oppure una foglia quando si scelga un valore di tipo semplice. Il paragone è sicuramente appropriato, ma la generalità con cui è possibile costruire statement RDF non vieta di avere legami molto più complessi, in cui i nodi soggetto e oggetto possono scambiarsi di ruolo in statement diversi.

La rappresentazione che emerge da un insieme di statement RDF è, quindi, strutturata come un *grafo con archi orientati*, in cui nodi e archi possono essere etichettati in base a, rispettivamente, le risorse e le proprietà che rappresentano e i nodi a cui si è scelto di assgnare un valore semplice non hanno archi uscenti (solo le risorse possono avere delle property). Inoltre non tutti i nodi del grafo sono necessariamente etichettati: infatti lo standard RDF prevede la possibilità di creare risorse 'in-place' (ossia, direttamente all'interno di un modello), referenziate unicamente nel documento in cui vengono create; in questo caso il nodo resta anonimo e ovviamente non può essere referenziato da altri statement poiché non è possibile associare ad esso una URI.

Il grafo, o meglio, il grafo oreintato è quindi la struttura dati in base alla quale qualsiasi metainformazione viene rappresentata in RDF. Il fatto che attraverso le URI si possano referenziare sia risorse che proprietà permette l'uso di una stessa URI come nodo e come arco e questo permette di costruire strutture come l'auto-referenzialità in un modo molto simile a quanto avviene nei linguaggi naturali.

L'organizzazione fisica degli statement RDF segue due formati distinti: infatti il W3C ha previsto una formato basato su XML, che a sua volta prevede la possibilità di usare due sintassi, una estesa (detta *serializazion syntax*) e una abbreviata (introdotta per facilitare la scrittura manuale di documenti RDF) e un formato più semplice definito *N-triple* (triple notation, [NTRIPLE]). La diffusione del linguaggio XML e la possibilità di avere a disposizione molti strumenti già collaudati per questo formato (parser, validatori, ecc...) fa generalmente propendere per la prima alternativa, sebbene il formato N-triple sia altrettanto espressivo e permetta di costruire modelli RDF analoghi a quelli ottenibili utilizzando XML.

Sintassi XML

Gli statement RDF sono generalmente racchiusi fra una coppia di tag `RDF`. Associato a questo tag si trovano le definizioni dei namespace utilizzati nel modello (ad esempio, per identificare gli schemi necessari per l'interpretazione degli statement), seguendo le stesse regole previste per l'uso dei namespace all'interno di documenti XML. Tipicamente vengono definiti i namespace dei due documenti che descrivono i modelli e gli schemi RDF in modo da poter utilizzare i tag base per la realizzazione di statement; nel seguito si farà riferimento ad essi con il prefisso `rdf` e `rdfs` rispettivamente.

Statement Il tag XML utilizzato per delimitare uno statement RDF è `rdf:Description`; con questo tag si possono:

- dare descrizioni di risorse già esistenti; in tal caso è necessario utilizzare l'attributo `rdf:about` il cui valore è la URI della risorsa che forma il soggetto dello statement;
- descrivere nuove risorse; omettendo l'attributo `rdf:about` lo statement che si genera non riguarda alcuna risorsa esistente, ma si costruisce una nuova risorsa di cui si può fornire un identificatore tramite l'attributo `rdf:ID`;
- fornire descrizioni 'locali' ad un modello, ossia non indicare nessuno dei due precedenti attributi per fornire uno statement non visibile dall'esterno poiché non avendo alcun `rdf:ID` associato non può essere referenziato tramite URI. Lo scopo principale di queste descrizioni consiste nel fornire un maggiore dettaglio su alcuni aspetti di una risorsa senza introdurne di nuove; questo tipo di statement prende il nome di *descrizioni anonime*.

La parte delimitata dai tag di apertura e di chiusura di `rdf:Description` è destinata a contenere le proprietà e i rispettivi valori che si voglio mettere in evidenza.

Nel seguente modello RDF vengono date due descrizioni: la prima riguarda il sito `http://foo.org` di cui si indicano le due proprietà `webmaster` e `webdesigner` i cui valori sono due risorse, una definita esternamente, l'altra definita nello stesso modello; la seconda descrizione introduce una nuova risorsa (etichettata `ID01`) di cui viene data la property `authorname` come literal e la proprità `company` come risorsa. Si noti che per quest'ultimo caso si è utilizzata una descrizione annidata poiché si è voluto fornire anche la property `email`, riferita non al webdesigner ma piuttosto all'azienda per cui lavora. Tutte le proprietà citate sono definite in uno schema esterno (`http://webdescription/schema`) che viene referenziato tramite il meccanismo dei namespace XML:

```
<rdf:RDF
  xmlns:rdf="http://www.w3c.org/1999/02/22-rdf-syntax-ns#"
  xmlns:web="http://webdescription/schema#">
    <rdf:Description rdf:about="http://foo.org/">
        <web:webmaster
          rdf:resource="http://foo.org/people/john"/>
        <web:webdesigner rdf:resource="#ID01"/>
    </rdf:Description>

    <rdf:Description rdf:ID="ID01">
        <web:authorname>bar</web:authorname>
        <web:company>
          <rdf:Description>
            <web:email>yourweb@yourweb.org</web:email>
          </rdf:Description>
        </web:company>
    </rdf:Description>
</rdf:RDF>
```

La sintassi di questi modelli è definita *serialization syntax*. Essa presenta le descrizioni delle metainformazioni nel modo più chiaro possibile; tuttavia, per comodità, è possibile definire gli statement in un modo più conciso utilizzando la *sintassi abbreviata*. La principale regola che consente di abbreviare le descrizioni degli statement RDF riguarda l'uso di property il cui valore sia un literal: se la proprietà non è ripetuta più volte all'interno dello statement, è possibile trasformarla in un attributo del tag `Description`. Nell'esempio presentato precedentemente il secondo statement diventa:

```
<rdf:Description
  rdf:ID="ID01">
  web:authorname="bar"/>
```

Container Una risorsa talvolta si presenta come una raccolta di elementi di uno stesso tipo: gli studenti di un corso, i capitoli di un libro e le voci di un menù di pop-up sono alcuni esempi di organizzazioni di questo tipo. In RDF è possibile descrivere una risorsa come collezione di elementi di questo tipo; per questo scopo vengono definiti tre tipi di containter destinati a contenere risorse o literal:

- **Bag**: rappresenta una lista non ordinata (può essere paragonato ad un insieme in cui sono ammesse ripetizioni di un elemento); il tag XML relativo è `Bag;`
- **Sequence**: lista ordinata; in questo caso viene fornita una enumerazione degli elementi contenuti; il tag XML è `Seq;`
- **Alternative**: lista di opzioni; un container di questo tipo rappresenta le possibili alternative (di cui la prima è considerata di default) fra cui scegliere un valore; il tag XML corrispondente è `Alt`.

In tutti i casi presentati l'appartenenza di un elemento ad un container viene indicata dal tag `rdf:li`; il seguente codice RDF evidenzia l'uso di un container per descrivere le sezioni di un libro:

```
<rdf:RDF
  xmlns:rdf="http://www.w3c.org/1999/02/22-rdf-syntax-ns#"
  xmlns:web="http://webdescription/schema#">
    <rdf:Description rdf:about="http://library/books/qumran">
        <rdf:Seq>
            <rdf:li rdf:resource="http://qumran/manoscritti"/>
            <rdf:li rdf:resource="http://qumran/santi"/>
        </rdf:Seq>
    </rdf:Description>
</rdf:RDF>
```

Auto-referenzialità Descrizioni che riguardano altri statement rappresentano statement di *ordine superiore*. La possibilità di costruire descrizioni di questo tipo si basa sulla realizzazione di un modello di dati che rispecchi la struttura di uno statement come presentata all'inizio di questa sezione, con soggetto, predicato e oggetto. Analogamente a quanto accade per le descrizioni presentate precedentemente, la presenza di statement di ordine superiore non implica l'esistenza nel modello RDF delle descrizioni che questi statement modellano (così come la presenza di uno statement tradizionale non implica la presenza delle risorse che descrive).

Le proprietà RDF necessarie per descrivere uno statement di ordine superiore sono:

`type` : lo scopo di questa proprietà è indicare il tipo di descrizione che si vuole effettuare; il suo uso non è limitato all'auto-referenzialità, ma può essere utilizzato anche in contesti diversi (ad esempio per indicare che una risorsa è di un certo tipo container). Nel caso di descrizione di statement è richiesto che il valore assunto sia `rdf:Statement`;

`subject` : la risorsa che viene descritta;

`predicate` : la proprietà utilizzata per descrivere il soggetto;

`object` : il valore della proprietà.

Nel frammento RDF presentato precedentemente veniva presentata la descrizione di un sito web; l'auto-referenzialità permette di descrivere gli statement creati in precedenza trattandoli come risorse:

```
<rdf:RDF
  xmlns:rdf="http://www.w3c.org/1999/02/22-rdf-syntax-ns#"
  xmlns:web="http://webdescription/schema#">
    <rdf:Description rdf:ID="REIFICATION01">
        <rdf:type rdf:resource="rdf:Statement"/>
        <rdf:subject rdf:resource="http://foo.org/"/>
        <rdf:predicate rdf:resource="web:webmaster"/>
        <rdf:object
          rdf:resource="http://foo.org/people/john"/>
    </rdf:Description>
</rdf:RDF>
```

Formato N-triple

Il formato N-Triple si basa su una rappresentazione puramente testuale degli elementi RDF, in cui non sono presenti tag di mark-up e in cui ogni linea di testo rappresenta uno statement. La stessa descrizione fatta in precedenza con la sintassi XML viene rappresentata in questo formato come:

```
<http://foo.org, web:webmaster, http://foo.org/people/john>
<http://foo.org, web:webdesigner, #ID01>
<#ID01, web:authorname, "bar">
```

2.2 RDF Schema

Gli schemi RDF vengono usati per descrivere i tipi delle proprietà che si vogliono evidenziare in un certo contesto (la descrizione di una tradizionale

biblioteca richiede informazioni, e quindi un insieme di proprietà, diverse da quelle necessarie per descrivere un'automobile). Attraverso gli schemi RDF si possono definire le proprietà valide in un modello e inoltre è possibile specificare per ciascuna di esse alcune caratteristiche utili per comprenderne il significato: ad esempio, si vorrebbe limitare il dominio di applicabilità della propietà 'numero di targa' ed evitare che una risorsa 'libro' possa avere questo attributo. Come visto in precedenza, i modelli RDF consentono solo di esprimere degli statement, e quindi di fornire delle descrizioni, ma non permettono di costruire nuove proprietà: gli schemi vengono introdotti proprio per realizzare un *vocabolario* di proprietà specifiche del contesto in cui vengono fatte le descrizioni. Inoltre in uno schema è possibile specificare **classi di risorse** a cui associare una determinata proprietà in un modo non troppo dissimile da quello che caratterizza i linguaggi *object-oriented*. In definitiva, con uno schema RDF è possibile fornire un sistema di tipi per i modelli RDF che lo utilizzano con tutti i vantaggi che ne derivano, come ad esempio la possibilità di evidenziare eventuali violazioni nell'utilizzo di un'attributo.

Confrontando RDF Schema con analoghi sistemi di validazione, come l'uso di DTD o XML Schema, è possibile sottolineare quali siano le potenzialità di questo approccio: mentre l'uso di un Document Type Definition permette di esprimere unicamente vincoli sintattici sulla costruzione di documento (indicando, ad esempio, che i possibili valori di un attributo devono essere dei caratteri `PCDATA`) e XML Schema, seppur rinforzando questo tipo di vincoli, si muove nello stesso ambito, RDF Schema costituisce un sistema di validazione semantica di un modello di dati RDF che focalizza la propria attenzione sul significato che assumono proprietà e risorse quando vengono legate assieme in uno statement.

Come anticipato, il sistema di tipi di uno schema RDF somiglia sotto molti aspetti a quello di un linguaggio object-oriented. Tuttavia, mentre in un linguaggio *OO* l'enfasi maggiore è riservata alle classi, che al loro interno contengono metodi e attributi, e la definizione di nuovi tipi avviene per unicamente per ereditarietà fra classi, gli schemi RDF consentono di descrivere

nuove proprietà (logicamente assimilabili ai metodi di una classe) senza il bisogno di 'includerle' in una classe di risorse: anziché definire classi come insieme di attributi, si definiscono le proprietà come relazioni fra classi, di cui si indica il dominio di applicazione e il range dei possibili valori. L'attenzione si sposta dalle classi (che comunque possono essere definite per ereditarietà a partire da altre classi) alle proprietà: è ad esempio possibile costruire proprietà come specializzazione di proprietà già esistenti con un meccanismo di sub-property. Questo modo di procedere avvicina ulteriormente gli statement RDF alle frasi dei linguaggi naturali, in cui gli oggetti del discorso non hanno un insieme prestabilito di caratteristiche di cui si possa discutere, ma chiunque può mettere in evidenza aspetti e proprietà che definisce introducendo nuovi legami. Ad esempio, la descrizione di una canzone può essere fatta sotto molti punti di vista:

- *tecnicamente* si è interessati a descrivere le proprietà fisiche del suono, come la frequenza e l'ampiezza;
- *artisticamente* si vogliono evidenziare aspetti diversi di matrice tipicamente stilistica, come ad esempio il genere musicale o, più audacemente, quali sensazioni evoca nell'ascoltatore;
- per esigenze di *catalogazione* si è interessati, ad esempio, all'autore del brano o all'anno in cui è stato composto.

Un approccio orientato alle classi di oggetti dovrebbe prevedere in partenza quali siano tutti gli aspetti che una certa risorsa può avere, pena la ristrutturazione della gerarchia di classi che ne derivano; la possibilità di introdurre nuove proprietà (slegate da questa o quella classe) che formano una gerarchia a sè stante rappresenta, invece, un approccio più flessibile ed elegante per la rappresentazione di dati: si pensi ad esempio ai problemi sorti nel passaggio dal formato 1.1 del formato ID3 al formato 2.4.x che voleva rappresentarne una estensione per descrivere più compiutamente risorse audio e multimediali (`http://www.id3.org`).

Interoperabilità Più volte si è messo in evidenza come la possibilità di avere un meccanismo in grado di favorire lo scambio di informazioni fra comunità diverse sia stato un obiettivo cruciale nella formulazione dello standard RDF: lo scambio di informazioni non consiste, ovviamente, solo nella possibilità tecnica di trasferire fisicamente una stringa di byte da un utente all'altro, ma anche nella trasmissione del significato che quella sequenza di `0` e `1` assume. La comprensione di uno schema RDF è necessaria per poter in seguito interpretare i modelli RDF che usano quello schema per esprimere delle caratteristiche sulle risorse. Questa operazione è facilitata dal modo in cui nuovi schemi vengono costruiti a partire da schemi esistenti, ossia per *ereditarietà*: se immaginiamo che un utente (umano o artificiale) abbia compreso il significato della risorsa 'veicolo' e in uno schema viene presentata una nuova risorsa 'bicicletta' definita come sotto-classe della prima, seppure non siano note le peculiarità di questa seconda classe, è comunque possibile associare ad essa tutte le proprietà di cui dispone la risorsa padre 'veicolo'. Analogamente, nuove proprietà possono essere interpretate come specializzazione di proprietà già esistenti da cui le prime derivano.

2.2.1 Descrizione

La creazione di nuovi schemi avviene utilizzando un insieme predefinito di risorse e proprietà che costituiscono la base per la creazione di nuovi elementi. Le classi previste dalle specifiche RDF Schema [RDFS] sono:

- `rdfs:Resource` per la rappresentazione di risorse;
- `rdf:Property` che rappresenta le property RDF. Una proprietà è una risorsa, per cui questa classe è definita come una specializzazione di `rdfs:Resource`;
- `rdfs:Class` che definisce un insieme di risorse con caratteristiche comuni.

La terminologia può creare confusione: una *classe* è un concetto astratto, mediato dalla teoria dei tipi della programmazione *object-oriented*, con cui è possibile fornire una *catalogazione* degli oggetti presi in considerazione. Tale catalogazione può essere fatta scegliendo il livello di dettaglio che meglio si adatta alle specifiche esigenze; in particolare, in RDF si sceglie di creare una classe per la rappresentazione di risorse (`rdfs:Resource`), una classe per le proprità (`rdf:Property`) (che sono risorse speciali) e una classe per le classi di risorse (`rdfs:Class`). In base a questo meccanismo qualsiasi classe ha tipo `rdfs:Class` (inclusa la stessa `rdfs:Class`) e qualsiasi risorsa è una specializzazione di `rdfs:Resource` poiché deriva da essa. La possibilità di utilizzare valori corrispondenti ai tipi semplici di XML è garantita dall'esistenza della classe `rdfs:Literal`.

Si può quindi vedere che la costruzione di nuovi elementi RDF avviene tramite *istanziazione* oppure tramite *ereditarietà*. Per il primo caso esiste un'unica property di RDF Schema destinata a descrivere questo legame: essa è `rdf:type` che esplicita esattamente il tipo di una risorsa. Se la costruzione di una nuova risorsa avviene per specializzazione, lo standard RDF Schema prevede due distinte proprietà: `rdfs:subClassOf` per la definizioni di nuove classi e `rdfs:subPropertyOf` per la definizione di nuove proprietà.

Vincoli in RDF

La possibilità di esprimere dei vincoli sull'utilizzo di risorse e proprietà è fondamentale per raggiungere gli obiettivi che stanno alla base di RDF. L'importanza di un sistema per esprimere dei vincoli è rimarcato dai constraint indicati nelle specifiche RDF Schema:

- il valore di una proprietà deve essere una risorsa di una classe;
- le proprietà possono essere utilizzate solo su risorse di classe non literal;
- il meccanismo di costruzione per ereditarietà non deve formare cicli.

L'espressione di vincoli tramite l'uso di proprietà è delegato alla classe `rdfs:ConstraintProperty` (sottoclasse di `rdf:Property`): qualsiasi proprietà appartenente a questa classe ha il compito di indicare un constraint e in particolare, per la definizione di nuove property, sono definite `rdfs:range` e `rdfs:domain` che vengono usate per indicare rispettivamente il range e il dominio di applicazione di un'attributo.

Esempio

Le caratteristiche delineate in precedenza vengono riassunte nel seguente esempio di schema, in cui si definiscono le classi `Veicolo` e `Automobile` e una proprietà (`numeroTarga`) associata a quest'ultima classe di risorse:

```
<rdf:RDF
  xmlns:rdf="http://www.w3c.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3c.org/2000/01/rdf-schema#">
    <rdf:Description rdf:ID="Veicolo">
        <rdf:type rdf:resource="rdfs:Class"/>
        <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
    </rdf:Description>

    <rdf:Description rdf:ID="Automobile">
        <rdf:type rdf:resource="rdfs:Class"/>
        <rdfs:subClassOf rdf:resource="#Veicolo"/>
    </rdf:Description>

    <rdf:Description rdf:ID="numeroTarga">
        <rdf:type rdf:resource="rdf:Property"/>
        <rdfs:domain rdf:resource="#Automobile"/>
        <rdfs:range rdf:resource="rdfs:Literal"/>
    </rdf:Description>
</rdf:RDF>
```

Capitolo 3

Strumenti per gestire metadati

3.1 Analisi

Con il termine 'gestione di metadati' ci si riferisce al trattamento di metainformazioni finalizzato a:

- *generare* un insieme di documenti (modelli ed eventualmente schemi) descrittivi di un set di dati; per definizione, i metadati sono costruiti su altri dati già esistenti per cui la generazione è una operazione che tipicamente richiede di conoscere il formato dei dati di partenza, e sulla base di questa informazione, costruire i metadati associati;

- scegliere un'efficiente *formato di memorizzazione* delle metainformazioni; la presentazione dei modelli RDF fatta in 2.1 ha già individuato due possibili standard per la rappresentazione di documenti RDF utilizzando file. Tuttavia l'uso di file, in particolare quando la base di documenti a disposizione ha notevoli dimensioni, pone problematiche organizzative che possono portare alla scelta di formati differenti. I formati di archiviazione sono discussi in 3.2.

- estrarre le informazioni di interesse da uno o più documenti; tipicamente questa operazione consiste nell'individuare uno o più statement

che soddisfano particolari requisiti. Questo tipo di operazione porta alla realizzazione di un *linguaggio di interrogazione* tramite il quale è possibile esprimere vincoli sugli statement di interesse; i linguaggi di interrogazione sono presentati in 3.3;

La dipendenza dal formato dei dati da descrivere impedisce di analizzare una soluzione sufficentemente generica per la generazione di metadati: nella maggior parte dei casi questa operazione è svolta manualmente; solamente se i dati da descrivere sono espressi in un formato idoneo è possibile realizzare strumenti in grado di generare automaticamente i metadati relativi: ad esempio, la durata di una brano MP3 può essere calcolata analizzando i singoli frame che componogono il file, ma l'autore del brano deve essere indicato manualmente dall'utente.

Le restanti operazioni possono invece essere svolte da strumenti costruiti sfruttando le specifiche RDF per la rappresentazione di metadati, tenendo conto che l'uso di questo formato consente il trattamento di metadati a differenti livelli:

- **sintatticamente**: in questo caso si è interessati unicamente ad esplorare la sintassi del documento RDF così come viene presentato, senza investigare ulteriormente sulla struttura dello stesso. Ad esempio, un operatore di pattern matching fra stringhe opera in questo livello di analisi, poiché è in grado di interrogare un insieme di dati e fornire un risultato unicamente sulla base della sintassi utilizzata. Un meccanismo più completo che compie questo livello di analisi è XPath ([XPath]) che consente di specificare percorsi all'interno di un documento XML esaminando la sintassi dei tag e il loro annidamento;

- **strutturalemente**: a differenza dell'analisi sintattica presentata in precedenza, il supporto per l'analisi strutturale consente di riconoscere e gestire correttamente gli statement RDF intesi come triple formate da soggetto, predicato e oggetto;

- **semanticamente**: l'analisi semantica è sicuramente la più complessa in quanto richiede la comprensione del documento RDF così come le specifiche RDF Schema (presentate in 2.2) impongono. In questo livello di analisi non è sufficiente fare affidamento sul nome utilizzato per descrivere un legame fra due risorse, né può bastare analizzare gli statement che fisicamente compaiono in un modello per comprendere il significato delle informazioni descritte.

3.1.1 Sintassi

Come visto in 2.1, lo standard RDF prevede un formato XML e un formato N-triple per la rappresentazione di statement. L'analisi sintattica in quest'ultimo caso è estremamente semplice: ogni statement viene rappresentato da una linea di un file di testo e ha un formato che ne consente l'immediata individuazione delle componenti utilizzando strumenti disponibili come singole utility (ad esempio, il diffuso comando `grep` dei sistemi Unix). Tuttavia a tale formato viene tipicamente preferita la sintassi XML poiché, consentendo l'utilizzo di marcatori, agevola la lettura degli stessi statement.

Il formato XML prevede già un meccanismo standard per l'identificazione di sezioni di un documento: tale tecnologia è rappresentata da XPath, un formalismo che consente di utilizzare i nomi dei tag come elementi per la costruzione di percorsi all'interno di un oggetto XML con una sintassi simile a quella utilizzata nei file-system Unix-like. L'utilizzo di questo formalismo, però, sembra non centrare l'obiettivo per quanto riguarda la sua applicazione a contenuti RDF: XPath, infatti, è pensato per un dominio di applicazione molto generico (qualsiasi codifica XML, non solo RDF), quindi non solo non cattura le particolarità espressive di RDF (ad esempio, la reificazione degli statement), ma il riconoscimento delle strutture di base (le triple) è affidato interamente all'utente XPath che dovrà costruire percorsi per il riconoscimento di statement senza alcun supporto da parte del formalismo utilizzato. Ad esempio, XPath consente di utilizzare il percorso `/` all'interno di un do-

cumento RDF/XML, ma tale percorso non rappresenta nè uno statement nè parte di esso. Inoltre, l'uso di tag per la costruzione di percorsi rende di difficile realizzazione la distinzione fra statement, i quali usano uno stesso tag iniziale (`rdf:Descripion`) e si differenziano solo per il valore dell'attributo `rdf:about`.

Queste considerazioni portano ad escludere strumenti in grado di compiere unicamente analisi sintattica poiché risulta limitativa per i fini del formato RDF.

3.1.2 Struttura

L'analisi strutturale è una specializzazione dell'analisi sintattica in cui si pone enfasi sulla struttura di base di dati RDF, ossia le triple che compongono gli statement. Strumenti in grado di compiere questo tipo di analisi prevedono come principale formato di rappresentazione una combinazione di soggetto, predicato e oggetto di statement RDF, forzando una rappresentazione più vicina al modello di dati di RDF di quanto non sia possibile fare con l'analisi sintattica.

Questo tipo di analisi non aggiunge espressività all'analisi sintattica: ne rappresenta piuttosto una versione di più semplice formulazione, introducendo costrutti che altrimenti dovrebbero essere realizzati ad un più basso livello. Come conseguenza, restano ancora irrisolti i problemi derivanti dalle pecuiliarità del formato RDF.

3.1.3 Semantica

L'analisi semantica è giustificata dalle potenzialità espressive di RDF e rappresenta sicuramente lo stadio più completo di gestione di un documento RDF. La presenza di tag come `rdfs:subPropertyOf` all'interno di uno schema non implica soltanto la presenza di un marcatore per delimitare una porzione del documento RDF (analiasi sintattica), né può essere considerata

una proprietà qualsiasi che lega soggetto e oggetto di uno statement (analisi strutturale); esso rappresenta un meccanismo per la costruzione di una gerarchia di proprietà i cui componenti sono legati dalle note relazioni padre-figlio derivanti da qualsiasi costruttore di tipi per ereditarietà. In particolare, una classe figlio può essere utilizzata ovunque sia richiesto l'uso della classe padre.

Strumenti in grado di compiere analisi semantica devono essere in grado di riconoscere questo tipo di relazioni e consentirne un trattamento che tenga conto di legami per ereditarietà fra tipi di dato.

3.2 Archiviazione

L'organizzazione di una notevole quantità di metadati richiede un efficiente sistema per la loro memorizzazione che consenta di eseguire i comandi del linguaggio scelto per l'interrogazione dei metadati nel minor tempo possibile. Il più semplice sistema di memorizzazione consiste nell'utilizzare file strutturati eventualmente in una gerarchia di directory che rifletta l'organizzazione interna dei documenti al fine di facilitare le operazioni di ricerca. In questo modo una base di metadati iniziale viene partizionata scegliendo un opportuno criterio di suddivisione; l'interrogazione dei metadati può avvalersi di questo partizionamento per ottimizzare le relative operazioni di ricerca restringendo il numero di file da consultare.

Questo sistema presenta il vantaggio di non richiedere particolari elaborazioni sui metadati una volta che questi siano generati: infatti, è sufficiente collocare un documento nell'opportuno ramo di directory affinchè sia immediatamente consultabile. A fronte di questa semplicità, però, è necessario constatare che l'interrogazione di file richiede di effettuarne il parsing ogni volta che si voglia accedere alle informazioni contenute; sebbene una attenta scelta del formato dei file e degli strumenti per effettuarne il parsing consenta di ottenere tempi ridotti in questa fase, bisogna comunque osservare che esistono strumenti molto più performanti per la memorizzazione di gran-

di quantitativi di dati (ad esempio, l'utilizzo di DBMS). Inoltre la scelta di strutturare i metadati in una gerarchia di directory risulta essere troppo rigida: con questo sistema è al più possibile ottimizzare un tipo di interrogazioni, ma per le restanti tipologie di query non si hanno evidenti vantaggi. Ad esempio, l'organizzazione dei dati di una biblioteca può essere fatta scegliendo di suddividere i file di metadati in directory che rappresentano gli autori: in questo modo la ricerca per autori è semplificata, ma la ricerca per parametri diversi (titolo, anno) non ha migliori prestazioni rispetto al caso in cui la gerarchia di directory sia del tutto assente. La ristrutturazione della gerarchia per parametri diversi, inoltre, richiede tempi di esecuzione che annullerebbero il vantaggio di un partizionamento dei dati.

Queste considerazioni portano alla scelta di un database per l'archiviazione dei dati: in questo modo è possibile fare affidamento su strumenti specifici per la gestione di grandi quantità di dati, i cui principali vantaggi sono:

- **meccanismi di indicizzazione**: la possibilità di costruire indici su più parametri garantisce una maggiore flessibilità nella scelta delle ottimizzazioni da eseguire; nell'esempio della biblioteca, è possibile avere indici su qualsiasi dato si voglia interrogare;

- **memorizzazione persistente dei dati**: non sono richiesti tempi di parsing per i file ogni volta che si voglia accedere al loro contenuto. Ovviamente anche un sistema di database adotta un proprio formato per la codifica interna dei dati, ma tale formato è ottimizzato per il reperimento delle informazioni che contiene, a differenza di altri formati il cui obiettivo può essere la semplicità di scrittura (N-triple) o la garanzia di interoperabilità (XML).

L'adozione di un database prevede due possibilità: da un lato è possibile utilizzare un DBMS, che rappresenta un sistema completo per la gestione dei dati (così come un sistema operativo fornisce una gestione completa del-

l'hardware e delle risorse di un computer), dall'altro si può ipotizzare l'uso di una libreria per la manipolazione di database.

3.2.1 DBMS

L'utilizzo di un sistema di basi di dati tipicamente si realizza tramite un *DBMS* (DataBase Management System) relazionale che oltre ad avvantaggiarsi delle caratteristiche elencate in precedenza, presenta anche altri aspetti di interesse:

- si basa su un modello di dati semplice: la struttura base di un DBMS relazionale sono le *relazioni*, ossia tabelle in cui le colonne sono gli attributi di un oggetto che si vogliono descrivere e le righe (tuple) rappresentano i singoli oggetti contenuti all'interno della relazione;
- le operazioni sui dati sono formalizzate nella teoria dell'*algebra relazionale* che ne specifica la semantica esatta;
- offre strumenti di supporto alla memorizzazione dei dati, fra cui:
  - utilità per la gestione di database utilizzabili immediatamente senza la necessità di fare riferimento alle API del DBMS: tali utilità includono non solo gli strumenti per la manipolazione diretta di database (creazione, modifica e cancellazione), ma anche tool per creare utenti validi della base di dati, in modo da poter impostare diritti di accesso e di modifica ai dati contenuti, con modalità e funzioni non troppo dissimili da quelle previste da un moderno file-system;
  - un linguaggio di interrogazione per eseguire query all'interno di una base di dati: la presenza di un linguaggio di query ad alto livello rappresenta una ulteriore semplificazione nell'utilizzo della base di dati poiché evita di formulare richieste di interrogazione attraverso le complicate funzioni della API. Le query vengono

espresse in un linguaggio tipicamente conforme allo standard SQL, che utilizza espressioni molto simili alle frasi della lingua inglese, in cui non si forniscono i passi necessari per raggiungere il risultato finale, ma piuttosto si esprimono le caratteristiche che tale risultato deve avere.

3.2.2 Librerie per la gestione di database

In alternativa all'utilizzo di un DBMS, è possibile optare per la scelta di una libreria per l'archiviazione di informazioni in database: a differenza di un DBMS che rappresenta un sistema completo per la gestione di basi di dati, una libreria di archiviazione non dispone di utilità accessibili immediatamente dall'utente finale, ma espone le proprie funzionalità unicamente nella API. Analogamente, non viene fornito alcun linguaggio di interrogazione di alto livello: la possibilità di interrogazione è offerta da funzioni che rispecchiano il modello di dati adottato internamente.

L'uso di una libreria per la gestione di basi di dati ha il grande vantaggio di offrire performance generalmente superiori a quelle ottenibili da un DBMS se il formato dei dati che si vuole memorizzare nella base di dati è simile a quello adottato dalla libreria: ad esempio, i tempi per il parsing dei comandi del linguaggio di interrogazione sono annullati dalle chiamate di funzioni per l'accesso alle strutture dati interne. Legato a questo aspetto c'è però anche lo svantaggio di avere una minore flessibilità nella progettazione della base di dati. Ad esempio, la libreria BerkleyDB ([BerkleyDB]) adotta come struttura dati interna le tabelle hash, facilitando la gestione di dati quando questi si presentano come coppie *(nome, valore)*, ma rendendo più difficoltosa la gestione di dati che abbiano un formato più complesso. Da questo punto di vista l'uso di relazioni permette di trattare con maggiore facilità dati di cui si vogliano esprimere attributi multipli, come le triple RDF.

3.3 Linguaggi di interrogazione

L'analisi dei linguaggi di interrogazione viene fatta presentando due proposte per l'analisi strutturale e l'analisi semantica di documenti RDF:

- *SquishQL*, proposto dell'Institute for Learning and Research Technology (ILRT) Semantic Web Group per la realizzazione di un semplice linguaggio di interrogazione per il framework RDF ([SquishQL]); il linguagggio SquishQL segue un'approccio strutturale;

- *RDFPath*: proposta dell'RDFPath group ([RDFPath]) per la realizzazione di un linguaggio di navigazione all'interno di documenti RDF. Il processo di definizione e sviluppo di questo linguaggio non è ancora completato, ma l'approccio di base è quello di compiere un'analisi semantica dei modelli e schemi RDF tralasciando gli aspetti strutturali.

Come anticipato in precedenza, l'analisi sintattica si basa su sistemi e metodologie più vicine ad altri formalismi (XML) e pertanto non viene considerata in questa trattazione.

3.3.1 Squish (SQL-ish)/SquishQL/RDQL

SquishQL si basa sul concetto di navigazione all'interno del grafo costruito dagli statement RDF in cui il meccanismo di query è il confronto fra sottografi: una query Squish viene infatti interpretata come un grafo i cui nodi devono soddisfare le eventuali condizioni aggiuntive e l'esecuzione della query viene implementata come l'esecuzione di un algoritmo di graph matching.

Il nome del linguaggio deriva dall'adozione dei comandi SQL per l'interrogazione di una base di dati; infatti, una tipica query Squish è la seguente (da cui emerge una stretta somiglianza sintattica con lo statement `SELECT` del linguaggio SQL):

```
SELECT ?webmaster, ?webdesigner
```

```
FROM http://foo.org/sample1, http://foo.org/sample2
WHERE
  (web::webmaster ?doc ?webmaster)
  (web::webdesigner ?doc ?webdesigner)
AND ?webmaster ~ john
USING web FOR http://webdescription/schema
```

SquishQL prevede due tipi di strutture per selezionare il risultato finale:

- la clausola `WHERE`, che determina come realizzare il grafo con cui fare matching all'interno del modello RDF da interrogare (i modelli sono indicati dalla clausola `FROM`); questa parte della query viene definita *generativa* poiché costruisce una struttura dati da utilizzare all'interno della query stessa;
- la clausola `AND` che permette di costruire un *filtro* per selezionare solo i nodi che soddisfano condizioni aggiuntive.

Il linguaggio RDQL, sviluppato dai laboratori Hewlett-Packard nell'ambito del progetto Jena [RDQL], è una implementazione di SquishQL fatta in Java. La realizzazione di questo linguaggio è stata fatta principalmente come ponte di unione fra le specifiche RDF-API per l'accesso a documenti RDF da applicazioni Java e un linguaggio ad alto livello che consenta di interrogare in modo semplice un modello RDF.

3.3.2 RDFPath

L'obiettivo di questo linguaggio è quello di localizzare le informazioni contenute in un grafo RDF in modo simile a quanto avviene nel linguaggio XPath [XPath]. Una espressione RDFPath è costruita utilizzando, in sequenza, tre costrutti di base:

1. *primary selection*, il cui scopo è quello di selezionare un insieme di nodi (risorse) di partenza da un grafo RDF. Esistono due tipi di costruttori per effettuare una selezione:

- `resource()` per specificare una risorsa tramite URI: ad esempio, `resource('http://foo.org')` individua la risorsa associata alla URI fornita come parametro;
- `literal()` per identificare valori literal (`literal('john')`);

2. *location step*, ossia l'insieme dei nodi che è possibile raggiungere dall' insieme ottenuto con la primary selection in un solo passo. Riprendendo gli analoghi costruttori di XPath, un location step può essere espresso come figlio (`child()`), padre (`parent()`) o come il nodo stesso (`self()`) associato ad un elemento della primary selection. Esistono inoltre delle specializzazioni di `child()` e `parent()` per l'uso dei tipi contenitore di RDF: `element()` permette di selezionare gli elmenti di un contenitore, mentre `container()` può essere utilizzato in uno di tali nodi elemento per avere un riferimento al nodo padre.

3. utilizzo di *filtri* per esprimere condizioni booleane che devono essere soddisfatte dalle risorse (la sintassi e la sematica dei filtri è la stessa prevista da XPath).

La principale differenza fra RDFPath e XPath riguarda l'*operatore di estensione* `^`. Tale operatore consente di estrarre da un documento non solo gli statement fisicamente presenti, ma anche quelli che derivano dall'utilizzo delle proprietà `rdfs:subClassOf` e `rdfs:subPropertyOf`. Ad esempio, nel grafo rappresentato in figura 3.1 gli archi non tratteggiati indicano gli statement espliciti, mentre l'arco tratteggiato esprime una relazione che deriva dalla gerarchia di classi di risorse: una corretta elaborazione semantica del documento RDF richiede che si possano individuare anche questo tipo di legami e questa possibilità è offerta dall'operatore di estensione. La relativa query RDFPath è: `resource()[child(^ rdf:type)=B]`.

Lo scopo di RDFPath è quindi quello di fornire tecniche generiche per la costruizione di percorsi all'interno di grafi RDF che, per i motivi mostrati in precedenza, non rispecchia la struttura delle triple con cui sono costruiti gli

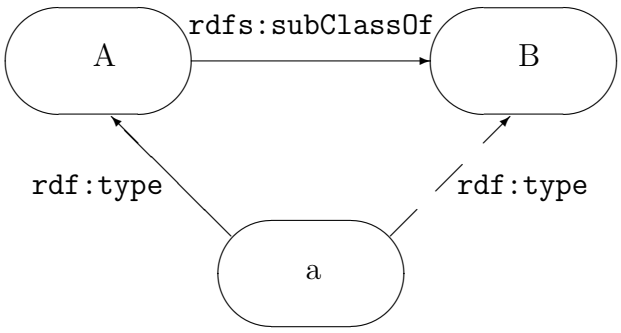

Figura 3.1: RDF: statement impliciti

statement, ma consente di compiere una analisi approfondita degli statement tramite l'operatore di estensione.

Capitolo 4

RDFSuite

La suite RDFSuite [RDFSuite] è un pacchetto per la gestione di documenti RDF che fornisce tre distinti applicativi; essi sono:

VRP (Validating RDF Parser): un parser validante per schemi e modelli RDF espressi in formato XML. VRP opera in due passi:

1. in un primo stadio viene effettuata una operazione di *verifica sintattica* del documento RDF in cui gli statement sono confrontati con le specifiche per la sintassi dei modelli RDF;
2. in seguito il parser compie un'*analisi semantica* del documento RDF accertando la compatibilità con le specifiche per la definizione di schemi RDF.

Le funzionalità del parser possono essere impostate dall'utente che ha a disposizione un insieme di opzioni per attivare o disattivare l'esecuzione di controlli sui documenti RDF: in questo modo è possibile, ad esempio, eseguire controlli con vincoli stringenti per assicurare la massima conformità dei propri documenti RDF agli standard W3C, oppure scegliere di avere controlli meno rigidi per adattare l'uso del parser alle proprie esigenze, in perfetta analogia con quanto accade

nella scelta delle opzioni di compilazione di un tradizionale linguaggio di programmazione;

RSSDB (RDF Schema-SPecific Data Base): tool per organizzare schemi e modelli RDF in un database relazionale. Il principale obiettivo che si è voluto raggiungere nella realizzazione di uno strumento di persistent-storage di metadati è la separazione fra gli schemi e i dati del Resource Description Framework: la tecnica utilizzata consiste nello strutturare il database in base alle classi o alle proprietà presenti negli schemi RDF in modo che durante la fase di caricamento dei modelli RDF siano già presenti tutte le tabelle necessarie per la memorizzazione degli statement RDF.

Uno dei punti di forza nella realizzazione del pacchetto RSSDB è la possibilità di effettuare operazioni di memorizzazione dei dati in modo *incrementale*: se un modello viene aggiornato dopo che è stato inserito all'interno del database (ad esempio aggiungendo degli statement) il pacchetto RSSDB effettua un'analisi del contenuto della base di dati per determinare se il modello è già presente e aggiornare solo i contenuti più recenti.

Interprete RQL (RDF Query Language): implementazione di un linguaggio di query orientato all'interrogazione di modelli RDF. RQL è un linguaggio dichiarativo tipato, il cui interprete si compone di tre moduli:

1. un **parser**, per l'analisi sintattica delle query;
2. un **graph constructor** che cattura la semantica di una query in base ai tipi e alle interdipendenze delle espressioni coinvolte;
3. **evaluation engine** che esegue la query attraverso la formulazione di statement SQL. L'evaluation engine ottimizza l'esecuzione di una query lasciando che il DBMS sottostante esegua la maggior parte delle operazioni coinvolte in modo da sfruttare le ottimiz-

zazioni specifiche del database (indici e interpreti SQL performanti)

Note sulla versione

Nel momento in cui sono stati eseguiti i test su questo prodotto l'ultima versione disponibile è la release 1.5, che tuttavia presenta problemi per quanto riguarda la validazione degli schemi RDF impedendo la memorizzazione all'interno del database. Tutti i test sono stati quindi eseguiti utilizzando la precedente versione 1.0 che sebbene presenti alcune limitazioni in termini di compatibilità con lo standard RDF (in particolare, il mancato supporto della property `type`) permette di analizzare le peculiarità di questo pacchetto.

4.1 VRP

4.1.1 Esigenze

La realizzazione del parser VRP è stata dettata da richieste specifiche del framework RDF che, sebbene trovino corrispondenza in analoghe esigenze per il linguaggio XML, presentano peculiarità che non vengono completamente risolte dagli strumenti sviluppati per quest'ultimo formalismo.

Il primo parser specifico per RDF è stato SiRPAC [SiRPAC]. SiRPAC non è un parser validante (quindi non verifica che un modello sia conforme agli schemi cui si riferisce), ma effettua solo una analisi sintattica del documento RDF garantendo la conformità con lo standard [RDF] per la sintassi e i modelli. Si basa su SAX (quindi richiede la presenza di un parser XML che supporti questa interfaccia) e il principio di funzionamento consiste nell'estrarre gli statement RDF dall'albero costruito dal parser XML.

I principali difetti di questo parser sono:

- non verifica che i modelli rispettino i vincoli degli schemi: il parser,

infatti compie unicamente un'analisi sintattica sul modello RDF, quindi se, ad esempio, i vincoli per il domain o il range di una proprietà non vengono rispettati il parser SiRPAC non riesce a catturare questa situazione di errore;

- dipende da un parser XML: questa dipendenza limita le possibilità di SiRPAC alle caratteristiche del parser XML utilizzato; ad esempio, il supporto per Unicode, indispensabile quando si voglia raggiungere una concreta interoperabilità, è demandato al parser XML;

- non c'è distinzione fra gli oggetti di tipo `rdfs:Literal` (che quindi fanno riferimento ad un tipo di dato semplice) e gli oggetti di tipo `rdfs:Resource` (che utilizzano URI); internamente SiRPAC rappresenta entrambe le tipologie di oggetti nello stesso formato impedendo di distinguere, ad esempio, URI da stringhe di caratteri.

Il parser VRP è realizzato esattamente per sopperire a questi svantaggi; infatti:

- effettua la validazione di schemi RDF sia direttamente che indirettamente: nel primo caso l'input del parser è lo schema da validare, nel secondo, l'input può essere un modello (o un diverso schema) che tramite namespace referenzia uno schema da validare;

- utilizza strumenti standard (CUP e JFlex) per la costruzione di una grammatica e di un parser aderenti alle specifiche per la sintassi RDF: non è quindi richiesta l'installazione di un parser XML;

- è in grado di distinguere le diverse classi di risorse: sulla base della validazione degli schemi da cui dipende un modello tiene separati literal da risorse referenziate tramite URI.

L'implementazione di VRP è fatta nel linguaggio Java e offre pieno supporto per i tipi di dato semplici definiti in XML Schema.

4.2 RSSDB

La memorizzazione di un modello RDF nella base di dati organizzata da RSSDB richiede, come anticipato, la preventiva preparazione del database fatta attraverso la creazione delle tabelle destinate a contenere gli statement. I parametri per la preparazione di tali tabelle sono stabiliti dall'analisi degli schemi RDF da cui dipendono i modelli che si vogliono memorizzare.

La struttura di base comprende la creazione e inizializzazione delle seguenti tabelle:

`generator` : utilizzata per generare identificatori numerici all'interno del database. Questa tabella contiene un solo valore `INTEGER` (valore dell' ultimo identificatore generato): la tipica interrogazione di questa tabella consiste nell'aggiornare (incrementando di 1) il valore contenuto;

`subclass` : esprime le relazioni di sottoclasse fra risorse di uno schema. Il formato di questa tabella prevede tuple di due colonne: in una viene riportato l'identificatore di una classe di risorse, nell'altra l'identificatore della classe padre. Questa tabella è accompagnata da due indici, uno per ogni colonna;

`subproperty` : è l'equivalente della precedente relazione per le proprietà; anche in questo caso le tuple contengono due attributi, su ciascuno dei quali viene costruito un indice;

`namespaces` : contiene i namespace individuati all'interno degli schemi. Ha due colonne per rappresentare un identificatore univoco e la URI completa del namespace. I namespace che fanno riferimento agli standard RDF per i modelli e gli schemi, nonché il namespace per i tipi base previsti da XML Schema vengono allocati di default; la seguente tabella mostra il contenuto di questa relazione non appena viene creata:

`property` : definizione di proprietà RDF. Il formato di questa relazione prevede 4 attributi principali per il riconoscimento di una proprietà:

0	null
1	http://www.w3.org/1999/02/22-rdf-syntax-ns#
2	http://www.w3.org/2000/01/rdf-schema#
3	http://www.w3.org/1999/XMLSchema-datatypes#

Tabella 4.1: RSSDB: contenuto iniziale della tabella `namespaces`

- identificatore univoco ottenuto come nei casi precedenti consultando la tabella `generator`;
- identificatore del namespace in cui la property è definita;
- nome della property;
- identificatore della classe di risorse domain;
- identificatore della classe di risorse range;

Su questa relazione vengono costruiti due indici, uno sul nome della proprietà e l'altro sull'identificatore.

`class` : equivalente della tabella precedente per le classi di risorse. In questo caso il formato si semplifica poiché non sono richiesti gli identificatori di domain e range. Anche in questa relazione gli indici definiti riguardano nome e identificatore.

`literal` : rappresentazione di un literal RDF. Al fine di distinguere il caso in cui l'oggetto di uno statement RDF è un tipo base dal caso in cui sia una risorsa, questa tabella elenca i possibili literal riconosciuti all'interno degli schemi. Così come la relazione per i namespace viene inizializzata con valori di default, anche per i literal la relazione non appena creata, viene anche aggiornata inserendo i valori mostrati nella tabella 4.2 Il formato prevede che il primo attributo sia l'identificatore del literal, il secondo l'identificatore del namespace in cui è definito

30	3	string
31	3	integer
32	3	float
33	3	boolean
34	3	dateTime

Tabella 4.2: RSSDB: contenuto iniziale della tabella `literal`

(si noti, consultando la tabella 4.1, che i literal previsti di default appartengono tutti al namespace che fa riferimeno a XML Schema) e infine il nome del literal.

Se si presentano descrizioni prive sia dell'attributo `rdf:about` che dell'attributo `rdf:ID` (descrizioni anonime) ad esse viene associato un identificatore interno che sarà utilizzato per referenziare la descrizione solo nel database, mentre all'esterno (quindi per l'utente) la descrizione continuerà ad essere anonima. Il formato dell' identificatore è `id`*`n`* dove *`n`* è costruito interrogando la tabella `generator`.

L'inserimento all'interno del database degli schemi RDF comporta, come detto, la formattazione della base di dati, ottenuta tramite la creazione una relazione (tabella) per ogni classe di risorse o proprietà definita negli schemi. Il nome delle relazioni segue un formato standard: per ogni classe di risorse o proprietà si ottiene un identificatore univoco (all'interno del database) *`n`* consultando la tabella `generator`; a questo punto la relazione prende il nome `t`*`n`*. Per le tabelle di questa forma che si riferiscono a classi di risorse la relazione presenta un solo attributo che contiene l'identificativo RDF della risorsa (`rdf:about` o `rdf:ID`) oppure l'identificativo generato internamente da RSSDB come descritto in precedenza. Relazioni associate a classi di risorse prevedono la costruzione di un indice associato all'unico attributo presente.

Per le tabelle che si riferiscono a proprietà, la relazione prevede due colonne per consentire di legare una risorsa subject ad una risorsa oggetto; in questo caso la tabella viene indicizzata su entrambi gli attributi presenti.

4.2.1 Implementazione

Il pacchetto RSSDB è stato realizzato interamente in Java, utilizzando il DBMS PostgreSQL per la gestione delle relazioni e la loro interrogazione. L'accesso alle funzioni del database viene garantita dall'interfaccia JDBC (Java DataBase Connectivity).

RSSDB viene fornito sia come applicativo *stand-alone*, che tramite interfaccia GUI permette all'utente di gestire l'inserimento di schemi e modelli nella base di dati, sia esportando una API per il linguaggio Java.

4.2.2 Funzionamento

Il caricamento di un documento RDF nella base di dati avviene tramite due passi:

1. analisi del documento: in questa fase si confronta il contenuto del documento RDF con quello del database al fine di controllarne la consistenza. Ad esempio, se il documento referenzia uno schema già presente nella base di dati, vengono effettuati i controlli sulle proprietà (vincoli domain e range) per garantire un uso coerente all'interno della base di dati. In questa fase è anche possibile riconoscere quali elementi (schemi o singole proprietà di uno schema) sono già presenti nel database e quali necessitano di essere introdotti: solo questi ultimi vengono effettivamente memorizzati nelle relazioni;

2. caricamento nel DBMS: l'effettiva memorizzazione degli elementi RDF avviene in questa fase, in cui prima vengono preparate le relazioni destinate a contenere le risorse o le proprietà di un documento (per ciascuna

di esse viene creata una tabella `tn`) e in seguito tali tabelle vengono popolate con i valori estratti dal parser VRP.

4.3 Il linguaggio RQL

RQL è un linguaggio di interrogazione dichiarativo; è tipato e segue un approccio di tipo funzionale, ricalcando le orme di XQuery, tuttavia, a differenza di quest'ultimo, gestisce le peculiarità dei documenti RDF, in particolare permette di esaminare le proprietà come entità separate delle classi di risorse a cui sono associate. Questa caratteristica consente di formulare query RQL scegliendo come punto di partenza non i singoli statement di un modello ma le proprietà che vi compaiono. I comandi di questo linguaggio consentono inoltre di gestire in modo trasparente schemi e modelli RDF: è infatti possibile interrogare sia gli statement di un modello per estrarre informazioni sui metadati contenuti, sia interrogare gli schemi RDF per conoscere l'organizzazione dei metadati all'interno dei modelli ed espolrarne le gerarchie di classi.

4.3.1 Descrizione

Interrogazione di schemi

Il supporto per l'interrogazione di schemi RDF è offerto da comandi che consentono di

- interrogare gerarchie di classi di risorse (`subClassOf`) e di proprietà (`subPropertyOf`): ad esempio, lo statement

  ```
  subClassOf(Veicoli)
  ```

  restituisce tutte le sotto-classi di risorse della classe `Veicoli` secondo una relazione transitiva. A questi comandi è possibile postporre il

simbolo ^ per ottenere unicamente gli oggetti in relazione diretta con la classe di risorse o proprietà fornita come parametro;

- estrarre informazioni sul dominio (`domain`) e range (`range`) di una property;
- avere informazioni sulle classi di risorse/proprietà radice (`topclass`/`topproperty`) o foglia (`leafclass`/`leafproperty`) di una gerarchia;
- identificare qualsiasi classe (`Class`) o proprietà (`Property`) di uno schema.

`SELECT` A questi comandi di base si affianca l'uso dello statement `SELECT` che consente di specificare vincoli e legami fra i diversi elementi di uno schema. L'utilizzo di `SELECT` consente anche l'introduzione di variabili necessarie per formulare query su generiche proprietà o classi di risorse. Il linguaggio RQL prevede tre tipi di variabili:

- variabili riferite a proprietà identificate dal prefisso `@`;
- variabili riferite a classi di risorse in senso stretto (prefisso `$`); questo tipo di variabili può assumere valori corrispondenti alle classi definite in uno schema, ma non valori literal corrispondenti ai tipi base delle specifiche XML Schema;
- variabili riferite a classi di risorse generiche (prefisso `$$`); a differenza delle precedenti, questo tipo di variabili può contenere qualsiasi tipo di classe di risorse.

In uno schema in cui sia definita una proprietà `numeroTarga` un primo esempio di query che utilizza lo statement `SELECT` è la seguente:

```
SELECT $D, $R
  FROM {$D}numeroTarga{$R}
```

Con questa query vengono richieste le classi dominio e range della proprietà `numeroTarga`. Come è possibile notare, la clausola `FROM` non specifica una sorgente fisica di dati; piuttosto in essa sono forniti gli elementi (proprietà o classi di risorse) che si desidera interrogare. Questa caratteristica è di notevole importanza rispetto ai linguaggi presentati precedentemente: anziché basarsi sulla struttura del formato RDF o sulla sua rappresentazione XML, il linguaggio RQL pone alla base dei propri costrutti unicamente il concetto di risorsa RDF (nella sua eccezione più ampia, comprendente classi e proprietà) consentendo una completa analisi semantica di tale primitiva.

Il linguaggio RQL prevede al suo interno i tipi di dato semplici definiti in XML Schema; ad esempio, tramite la seguente query:

```
SELECT @P
  FROM {;Veicolo}@P{;string}
```

vengono richieste tutte le proprietà di tipo stringa di cui dispone la classe `Veicolo`.

Analogamente a quanto accade in altri linguaggi, il comando `SELECT` prevede una clausola `WHERE` che consente di imporre dei vincoli di tipo booleano al risultato della clausola `FROM` permettendo quindi la realizzazione di filtri. In RQL le condizioni booleane possono riguardare sia i tipi semplici (ad esempio, `2 > 1`) che le gerarchie di classi di risorse. Uno schema in cui si definisca una classe di risorse `Veicolo` e una sua sotto-classe `Bicicletta` crea una relazione di ereditarietà fra la classe padre e la classe figlio; tale relazione viene catturata dagli operatori `<=` e `>=`. La seguente query:

```
SELECT C
  FROM Class{C}
  WHERE C >= Veicolo
```

permette quindi di individuare tutte le classi di risorse che derivano dalla classe `Veicolo`. Lo stesso risultato si può ottenere utilizzando il comando `subClassOf` presentato inizialmente (la query iniziale si semplifica in:

`subClassOf(Veicolo)`), tuttavia la possibilità di utilizzare gli operatori presentati permette la realizzazione di query più sofisticate che, ad esempio, prevedono ulteriori vincoli sugli statement RDF.

Interrogazione di risorse

RQL consente di interrogare modelli RDF tramite le classi di risorse e le proprietà definite nei corrispondenti schemi.

Risorse e proprietà L'interrogazione di risorse avviene specificando il nome della risorsa stessa: per conoscere tutte le istanze appartenenti alla classe di risorse `Veicolo` è ad esempio sufficiente la seguente query:

```
Veicolo
```

Più dettagliatamente, tale query fornisce le risorse (specificate all'interno dei modelli RDF) che sono di tipo `Veicolo` o di una sua sotto-classe. La semplicità con cui è possibile specificare query di questo tipo rappresenta un aiuto nella scrittura degli statement di interrogazione; la precedente query è equivalente a:

```
SELECT X
  FROM Veicolo{X}
```

Si noti che l'utilizzo di variabili per istanze di risorse (`X` nell'esempio) non è soggetto a particolari vincoli sintattici.

Analogamente a quanto visto per l'interrogazione di schemi, è possibile postporre il simbolo apice (`^`) ad una classe di risorse o proprietà per ottenere unicamente le risorse che sono istanze dirette.

Vincoli La clausola `WHERE` del comando `SELECT` permette di esprimere vincoli sui risultati che si vogliono ottenere; oltre ai casi previsti per l'interrogazione di schemi, è previsto un operatore `like` che consente di specificare un pattern per il riconoscimento di stringhe utilizzando i tradizionali caratteri speciali delle espressioni regolari.

Navigazione all'interno del grafo orientato L'operatore . (dot) consente di costruire percorsi all'interno del grafo ottenuto dagli statement RDF. Esso non aggiunge maggiore espressività al linguaggio, piuttosto, similmente a quanto visto in precedenza, rappresenta una semplificazione nella costruzione di query.

La seguente query RQL:

```
SELECT T
  FROM Automobile{X}.numero_targa{T}
  WHERE X like "*ferrari*"
```

restituisce il valore della proprietà `numero_targa` associata alle risorse di classe `Automobile` che soddisfano il vincolo imposto nella clausola `WHERE`. Questa query può essere riscritta scegliendo di non utilizzare l'operatore . introducendo un ulteriore vincolo:

```
SELECT T
  FROM Automobile{X}, {Z}numero_targa{T}
  WHERE X like "*ferrari*"
  AND X = Z
```

4.3.2 Implementazione

L'implementazione dell'interprete RQL è fatta in C++ e le piattaforme testate per cui viene distribuito l'interprete sono Sun Solaris e Red Hat Linux. L'accesso ai documenti RDF si realizza unicamente tramite l'interrogazione di una base di dati organizzata dal pacchetto RSSDB secondo le modalità illustrate in precedenza.

L'interprete RQL è utilizzabile tramite un client testuale che, emulando l'interprete `psql` di PostgreSQL, una volta effettuata la connessione al database, consente all'utente di introdurre interattivamente query in input, fornendo il risultato delle query come `Bag` RDF. Viene anche esportata una

API per il linguaggio C++ che permette l'esecuzione di query da proprie applicazioni.

Capitolo 5

Sperimentazione nel progetto HELM

La realizzazione di un sistema per il supporto ai metadati nell'ambito del progetto HELM ha richiesto una fase di studio dei metadati a supporto della libreria elettronica presentata in 1.2.1. In seguito è stato necessario trovare un efficiente formato di archiviazione scegliendo di utilizzare un DBMS e, nel contempo, realizzare un interprete per il linguaggio di interrogazione al fine di poter misurare le performance generali del sistema di memorizzazione.

5.1 Struttura dei metadati

Nel momento in cui si è sviluppato il sistema di supporto per i metadati sono presenti due categorie di metainformazioni:

- per le dipendenze logiche; questi metadati servono per descrivere le dipendenze esistenti fra i diversi elementi (teoremi e definizioni) che compongono un documento;
- Dublin Core, ossia informazioni sull'autore di un documento, sui diritti di copyright e sul linguaggio utilizzato per i metadati.

Il set di metadati Dublin Core è descritto compiutamente in [DublinCore]; essi vengono supportati come insieme di attributi (proprietà RDF) assegnabili a singoli documenti oppure a intere teorie: la possibilità di referenziare l'uno o l'altro è offerta dalla presenza di una gerarchia di classi di risorse.

Le dipendenze logiche sono rappresentate tramite la creazione di opportune proprietà RDF che modellano la semantica di dipendenza logica attraverso due tipi di relazioni: da un lato è necessario descrivere quali elementi sono richiesti per la costruzione di una dimostrazione che porta alla definizione di un nuovo termine (dipendenza *forward*); dall'altro si vuole modellare anche la dipendenza inversa, ossia sapere quali elementi di una teoria dipendono dal termine che si vuole descrivere (dipendenza *backward*). Entrambi i tipi di dipendenza devono indicare anche il contesto in cui un termine viene utilizzato poiché è necessario distinguere l'utilizzo di un oggetto nell'ipotesi di un teorema piuttosto che nella tesi.

5.1.1 Descrizione dello schema

Lo schema utilizzato per la definizione degli oggetti RDF necessari al supporto dei metadati si compone di una gerarchia di classi di risorse alla cui radice si trova `MathResource`: questa risorsa vuole essere la rappresentazione di qualsiasi risorsa di tipo matematico descritta all'interno della libreria. In particolare, la sottoclasse `Object` modella un singolo termine (definizione o teorema) di una teoria, mentre `DirectoryOfObject` rappresenta un raggruppamento dei singoli oggetti per formare teorie.

Le property definite nello schema riguardano sia i metadati Dublin Core che le dipendenze logiche: per le prime vengono definite le corrispondenti proprietà (`author`, `title`, ...), mentre per le seconde le dipendenze forward sono espresse da `refObj` e le dipendenze backward vengono indicate da `backPointer`. Per modellare correttamente il concetto di dipendenza e di contesto si intruduce una ulteriore classe di risorse `Occurrence` che tramite le proprietà `position` e `occurrence` consente di esprimere queste

informazioni. I valori assunti dalle proprietà presentate sono di tipo stringa, tuttavia il valore dell'attributo `position` può assumere valori unicamente in un insieme di stringhe prestabilito che rappresenta i possibili contesti in cui si può presentare una dipendenza:

- `MainHypothesis`: ipotesi principale del teorema;
- `InHypothesis`: ipotesi del teorema;
- `MainConclusion`: tesi principale del teorema;
- `InConclusion`: tesi del teorema;
- `InBody`: corpo della dimostrazione di un teorema.

5.2 Il linguaggio di interrogazione MathQL

L'accesso della libreria elettronica viene realizzato tramite il linguaggio MathQL, specifico per l'interrogazione dei metadati presentati in precedenza. Il risultato di una query MathQL è un'insieme di elementi della libreria (URI) che presentano determinate caratteristiche; tali caratteristiche possono riguardare il formato della URI, l'esistenza di dipendenze logiche, particolari valori del set di metadati Dublin Core oppure una combinazione di questi. Il linguaggio MathQL viene presentato suddividendo i comandi fra quelli che richiedono un accesso al database sottostante e quelli che operano unicamente su strutture dati presenti in memoria.

5.2.1 Accesso al database

`PATTERN` L'interrogazione della libreria prevede che alla base di ogni query vi sia almeno una URI di partenza di cui si vogliono estrarre determinate informazioni; la URI viene specificata tramite il comando `PATTERN` p che consente di costruire una lista di URI che soddisfano il pattern p fornito. Il pattern si basa sulla suddivisione della URI in tre componenti:

1. protocollo, la parte della URI che precede i caratteri '`:/`'. Il protocollo può o meno essere specificato nel pattern: se presente esso si riferisce ad uno specifico protocollo previsto nei metadati, in caso contrario (protocollo assente) viene referenziato qualsiasi protocollo della base di dati;

2. fragment identifier: rappresentazione del frammento XPointer eventualmente presente in una URI; sitatticamente è individuato da tutto ciò che segue l'ultimo carattere '`#`'. I possibili fragment identifier della libreria elettronica di HELM seguono un formato rigido: essi sono al più triple di numeri interi separati dal carattere speciale slash ('`/`'); sul fragment identifier sono definiti due caratteri 'jolly':

   - `*`: per fare match con qualsiasi sequenza di caratteri che non contenga lo slash; in questo modo è possibile specificare il formato del frammento XPointer (singoletto, coppia o tripla) senza imporre vincoli sul valore di ciascuna componente;
   - `**`: per fare match con qualsiasi sequenza di caratteri, compreso lo slash e quindi avere la massima flessibilità nell'identificazione del fragment identifier.

3. corpo: parte della URI compresa fra protocollo e fragment identifier. Il carattere slash (`/`) assume particolare rilievo per la costruzione del pattern: sono infatti definiti tre tipi di caratteri jolly sul corpo di una URI:

   - `?`: qualsiasi carattere del corpo della URI che non sia uno slash;
   - `*`: qualsiasi sequenza di caratteri del corpo della URI che non contenga uno slash;
   - `**`: qualsiasi sequenza di caratteri, incluso lo slash.

È quindi possibile utilizzare il comando `PATTERN` per specificare esattamente una URI (`cic:/Nancy/FOUnify/nat_complements/n_False.con`) oppure, tramite caratteri '*jolly*', un insieme di URI (`cic:/Nancy/*`).

`USE/USEDBY` Per l'individuazione dei legami di dipendenza logica sono presenti due comandi specifici:

`USE` *`uri_list`* `POSITION` *`context`* : lo scopo di questo comando è restituire la lista di elementi che utilizzano un elemento di *`uri_list`* nella posizione (contesto) *`context`*, permettendo quindi di selezionare le dipendenze di tipo backward. La grammatica del linguaggio MathQL prevede che il valore del parametro contesto non sia specificato direttamente ma che al suo posto si utilizzi una variabile su cui imporre eventuali condizioni tramite comandi diversi;

`USEDBY` *`uri_list`* `POSITION` *`context`* : questo comando seleziona le dipendenze forward con le stesse modalità del comando precedente.

Funzioni Il linguaggio MathQL definisce un insieme di funzioni base il cui scopo è estrarre attributi Dublin Core associati ad una URI. Sebbene vengano definite tante funzioni quanti sono gli attributi DC indicati nello schema RDF (`TITLE`, `DATE`, ...) tutte seguono la stessa sintassi *`func uri`* per restituire il valore della property associata alla URI.

5.2.2 Accesso in memoria

Filtri: il comando `SELECT` *`uri`* `IN` *`uri_list`* `WHERE` *`condition`* Il comando `SELECT` consente la costruzione di filtri (clausola `WHERE`) sull'insieme risultato (clausola `IN`) in cui ogni URI viene referenziata tramite *`uri`*. I filtri vengono espressi come condizioni booleane che possono riguardare:

- l'ugualianza di stringhe: operatore `IS`; le stringhe possono essere costruite a partire dalla variabile della clausola `SELECT`, da una espressione

quotata con gli apici semplici (') oppure utilizzando una costante stringa. Le costanti stringa sono definite per i possibili valori assunti dalla proprietà `position` dello schema RDF.

- l'ugualianza di insiemi (operatore `SETEQUAL`) e relazione di inclusione insiemistica (operatore `SUBSET`): in questi casi gli insiemi possono essere generati a partire dal risultato di una query e pertanto sono insiemi di URI.

oppure una combinazione di questi tramite le tradizionali operazioni logiche `AND`, `OR` e `NOT`. Tramite questo comando è, ad esempio, possibile specificare le condizioni sul parametro contesto dei comandi `USE`/`USEDBY` trattandolo come un valore di tipo stringa.

Variabili: `LET` *uri* `BE` *uri_list* `IN` *command* Con il comando `LET` è possibile assegnare alla variabile *uri* l'insieme risultato *uri_list* nella valutazione della query *command*. All'interno della clausola `IN` sarà possibile fare riferimento all'insieme *uri_list* tramite la variabile *uri*.

Ordinamento: il comando `SORTEDBY` Un insieme risultato può essere ordinato in base al nome degli elementi che vi compaiono oppure in base al valore di una determinata property Dublin Core che ciascuno di essi assume: nel primo caso viene definita la funzione `NAME` per estrarre il nome di un elemento da una URI; nel secondo si utilizzano le funzioni Dublin Core viste precedentemente. Si noti che sebbene la funzione `NAME` venga presentata in questo contesto insieme alle funzioni Dublin Core e con esse condivida anche la sintassi, se ne differenzia sostanzialmente poiché per essere eseguita richiede unicamente l'analisi della URI come stringa e non un accesso al database sottostante.

Il comando utilizzato per ordinare un insieme risultato segue il formato *uri_list* `SORTEDBY` *func* `(ASC | DESC)` in cui l'insieme *uri_list* viene

ordinato in base ai valori della funzione *func* per ordinamento crescente (ASC) o decrescente (DESC).

Operatori insiemistici Le operazioni di unione, interesezione e differenza fra insiemi sono fornite dagli operatori UNION, INTERSECT e DIFF la cui generica sintassi è *A op B* in cui *A* e *B* rappresentano due insiemi di URI.

5.3 Memorizzazione dei metadati

La realizzazione del sistema di gestione di metadati è stata inizialmente improntata sull'uso del pacchetto RSSDB come strumento per l'archiviazione dei metadati. Tuttavia, dopo le prime prove significative eseguite scegliendo questo sistema di memorizzazione, si sono rilevati tempi di esecuzione per le query troppo alti affinchè l'uso di un DBMS sottostante potesse essere di utilità. Questo tipo di problemi è da imputarsi alla struttura scelta per la rappresentazione degli statement: come visto in 4.2 le relazioni utilizzate non consentono un immediato indirizzamento alle componenti di uno statement, favorendo piuttosto la rappresentazione delle informazioni suddivise in classi di risorse o proprietà. Si è scelto comunque di continutare ad utilizzare PostgreSQL [PostgreSQL] come DBMS per la sperimentazione di un differente formato per la base di dati che potesse adeguarsi meglio alle richieste del linguaggio di interrogazione.

5.3.1 Organizzazione del database

Il formato di archiviazione delle metainformazioni è stato influenzato dalle richieste del linguaggio di interrogazione, il quale:

- prevede comandi precisi per la ricerca delle relazioni di dipendenza logica: tali comandi, che sono utilizzati con frequenza, devono necessariamente accedere alla base di dati e pertanto devono garantire le migliori prestazioni;

- basa ogni query su una URI (corrispondente ad un elemento della libreria di cui si vogliono trovare delle informazioni): questa richiesta impone di mantenere la lista delle URI di cui esistono descrizioni nella base di dati in modo che sia immediatamente accessibile e in modo che si possano conoscere le proprietà ad associate ad una URI nel minor tempo possibile.

Si è deciso, pertanto, di improntare la memorizzazione dei metadati sui documenti RDF, scegliendo un'organizzazione degli statement in modo da riflettere la loro collocazione in file i quali raccolgono le informazioni di dipendenza logica relative a documenti diversi in file distinti.

Il formato di archiviazione prevede che le proprietà di un modello RDF siano mantenute in tabelle separate al fine di partizionare il più possibile l'insieme di dati della libreria elettronica per ottenere migliori prestazioni nelle operazioni di ricerca. Ogni modello viene quindi rappresentato da una relazione le cui tuple (righe) indicano il tipo di proprietà che si vuole esprimere e i valori ad essa associati. Nel caso specifico delle relazioni di dipendeza logica (rappresentate dalle property RDF `refObj` e `backPointer`) il valore è espresso da due campi: uno mantiene l'informazione associata al contesto, ossia la posizione in cui compare un riferimento ad un oggetto esterno, l'altro indica la URI dell'oggetto referenziato. Gli attributi delle relazioni di questo tipo sono:

- `prop_id`: identificatore della proprietà;
- `context`: valore della posizione;
- `uri`: URI dell'oggetto referenziato.

Per poter interrogare la tabella relativa ad uno specifico modello è necessario mantenere anche una ulteriore relazione il cui scopo è associare la URI di un documento al nome della tabella relativa. Tale relazione (nominata `registry`) presenta quindi due campi:

- `uri`: URI del modello;

- `id`: identificativo univoco associato alla URI.

in cui l'identificativo è tipo numerico e viene generato progressivamente: un nuovo modello viene inserito nella base di dati con un identificatore il cui valore è il successore del maggiore `id` presente nella relazione `registry`.

Il binding fra identificativo numerico e la relativa tabella viene realizzato scegliendo di nominare la relazione con il formato `t`n dove n è la rappresentazione stringa dell'identificatore presente nella tabella `registry`.

5.3.2 Caricamento della base di dati

L'archiviazione della base di dati è fatta in due passi:

1. trasformazione dei modelli RDF: in questa prima fase i modelli RDF sono trasformati in un formato testuale al fine di facilitare l'inserimento degli statement all'interno del database. La trasformazione (fatta applicando un foglio di stile XSLT ai documenti RDF) crea un nuovo formato contenente gli statement RDF suddivisi nelle componenti soggetto, predicato e oggetto, in cui il predicato originale, ossia il nome della proprietà RDF, viene sostituito dall'identificativo che assumerà all'interno del database;

2. caricamento nel database: una volta trasformati i metadati RDF nel nuovo formato, questi vengono elaborati da uno script PERL che prepara i comandi da fornire all'interprete SQL di PostgreSQL al fine di inserire correttamente (realizzando il binding fra la relazione `registry` e le singole tabelle corrispondenti ai modelli) i dati nel database.

L'illustrazione dei passi necessari per il caricamento dei modelli RDF nella base di dati viene fornito nell'appendice B.

5.4 Intepretete MathQL

Nel presente lavoro di tesi è stato implementato il *back-end* per il linguaggio MathQL il cui compito è l'esecuzione delle query formulate in questo linguaggio. Le principali responsabilità del back-end sono:

- accedere al database per recuperare le informazioni richieste;
- mantenere opportune strutture dati in memoria per l'esecuzione delle query.

Al *front-end* del linguaggio è affidata unicamente la parte di effettuare il parsing di una query MathQL al fine di fornire al back-end la rappresentazione della stessa tramite costruttori e strutture dati opportune definite dall'intefaccia dell'*engine* MathQL.

L'interprete MathQL è stato realizzato nel linguaggio OCaml [OCaml] al fine di favorire l'integrazione con il software già sviluppato all'interno del progetto HELM. L'interfacciamento con il DBMS PostgreSQL è stato realizzato tramite il pacchetto *Postgres* [Postgres] che consente l'esecuzione di query SQL e la gestione dei risultati generati mantenendo l'approccio funzionale e a oggetti del linguaggio OCaml.

5.4.1 Implementazione del back-end MathQL

I principali problemi che si sono affrontati nella realizzazione dell'interprete hanno riguardato la gestione del formato interno del risultato di una query: come visto in 5.2.1 i comandi `USE`/`USEDBY` non ritornano immediatamente un insieme di URI che costituisce il risultato finale. Ad esempio, nella seguente query MathQL:

```
SELECT uri
  IN
    USE PATTERN "cic:/Algebra/**" POSITION $a
  WHERE
```

```
$a IS INBODY
```

che viene vista dal back-end come l'albero della figura 5.1

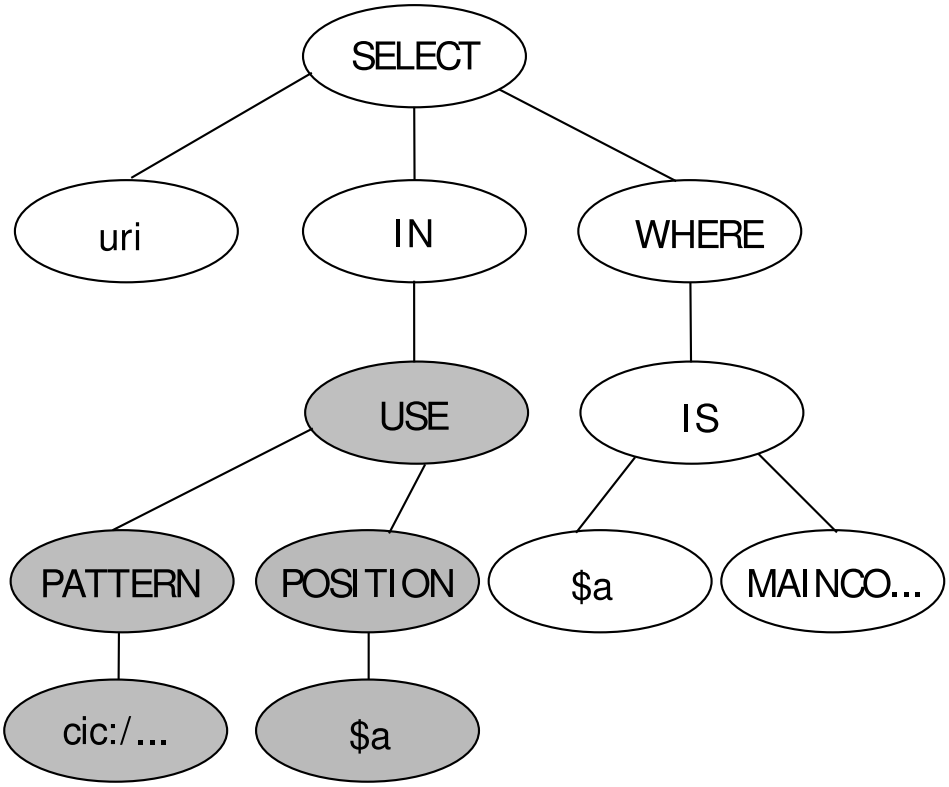

Figura 5.1: Query MathQL

Al termine dell'esecuzione dello statement `USE` (sotto-albero grigio) non è possibile sapere quali URI faranno parte del risultato finale poiché le condizioni booleane sono note solo nel ramo `WHERE` e non nel ramo `IN`. Tuttavia le condizioni espresse nel filtro dello statement `SELECT` si basano su valori del parametro contesto che sono visibili quando si interroga il database (poiché sono memorizzate all'interno di una opportuna relazione) ma che in seguito non sono più note.

La soluzione a questo problema è stata la realizzazione di una struttura dati che potesse mantenere, oltre al valore di una URI, anche un insieme di

coppie *(nome, valore)* ciascuna delle quali specifica il *valore* che la variabile *nome* deve assumere affinchè la URI possa far parte del risultato (*URI parametrizzata*). Tale struttura viene creata dagli statement `USE`/`USEDBY` poiché sono gli unici che possono conoscere il valore da assgnare ad una variabile contesto interrogando il database; la clausola `WHERE` del comando `SELECT` dovrà (eventualmente) accedere a tali coppie di valori per validare una URI e poterla includere nel set di risultati finali. La struttura è quindi:

- in prima istanza, una lista di URI per le quali siano forniti dei vincoli sugli attributi;

- più dettagliatamente, i vincoli sulle URI sono a loro volta una lista di coppie di stringhe. Un elemento del risultato memorizzato nel formato interno si compone quindi di una URI e di una lista di vincoli per essa.

Bisogna notare che una stessa URI può comparire più volte all'interno della struttura dati per la rappresentazione del risultato finale: questa condizione è dettata dalla possibilità di avere differenti valori di un parametro riferiti ad una stessa URI. Ad esempio, nel caso in cui il documento con URI `cic:/Algebra/Neg_anti_convert.con` venga referenziato in corrispondenza di valori diversi per il parametro `$a`, la lista mantenuta internamente avrà un contenuto simile a quello presentato nella tabella 5.1.

...	...
`cic:/Algebra/NEG_anti_convert.con`	`$a` = `MainConclusion`
`cic:/Algebra/NEG_anti_convert.con`	`$a` = `InConclusion`
...	...

Tabella 5.1: Risultato memorizzato nel formato interno

I passi necessari per l'esecuzione della query di esempio sono dettagliati nell'appendice C in cui viene mostrato come tale struttura viene aggiornata durante l'esecuzione dei comandi di una query MathQL.

Tutte le altre operazioni del linguaggio vengono implementate tenendo in considerazione la presenza di una struttura dati di questo tipo e adottando gli opportuni accorgimenti affinchè la query venga eseguita coerentemente. In particolare, gli operatori insiemistici vengono realizzati aggiungendo al test di appartenenza (ossia, che la URI appartenga ad entrambi gli insiemi che si vogliono intersecare affinchè possa appartenere all'insieme intersezione e condizioni analoghe per gli operatori di unione e differenza insiemistica) un controllo sulla coerenza degli attributi.

Esempio: il comando `INTERSECT` L'intersezione di due liste di URI parametrizzate deve garantire che una stessa URI non venga riportata nell'insieme risultato con valori discordi per uno stesso parametro; nell'esempio riportato nella tabella 5.2:

...	...	...	...
uri1	$a = a	uri1	$a = c
uri2	$a = b	uri2	$a = b
...	...	...	...

Tabella 5.2: Intersezione con valori discordi dei parametri

La URI `uri1` non può far parte dell'insieme intersezione poiché presenta valori diversi per uno stesso parametro; la URI `uri2`, invece, viene considerata valida e viene inclusa nell'insieme intersezione.

Un caso più frequente è rappresentato dall'intersezione di liste di URI parametrizzate in cui si utilizzino parametri con nomi diversi; in questo caso è necessario estendere la lista delle coppie *(nome, valore)* di ogni URI facente

parte del risultato finale poiché i vincoli imposti su di essa riguardano più attributi. Un esempio è presentato nella tabella 5.3

...	...	...	...
uri1	$a = a	uri1	$b = b
...	...	...	...

Tabella 5.3: Intersezione con più parametri

Il risultato dell'intersezione di queste due liste di URI è fornito nella tabella 5.4

...	...
uri1	$a = a; $b = b
...	...

Tabella 5.4: Risultato dell'intersezione con più parametri

5.5 Performance

5.5.1 Dimensioni della base di dati

L'insieme dei documenti RDF della libreria elettronica HELM è formato da `15244` modelli per la descrizione delle dipendenze logiche, per un totale di `80` MB di file su disco in formato XML; più dettagliatamente, vengono rappresentate `416699` descrizioni tramite statement RDF. Le misurazioni cronometriche riportate di seguito sono state osservate su PC con le seguenti caratteristiche:

- processore AMD Duron a 850MHz;
- 128 MB di memoria centrale;
- disco rigido da 15 GB/2 MB di cache/7200 rpm.

Al termine dell'inserimento nel DBMS PostgreSQL, l'organizzazione dei metadati, comprese strutture di supporto (indici, descrittori delle relazioni) ha richiesto uno spazio di 382 MB.

5.5.2 Tempi di caricamento

Il caricamento è avvenuto tramite la procedura descritta in precedenza. Il tempo totale richiesto per l'inserimento dei metadati è stato di 90 minuti di cui:

- 6 minuti per la prima fase, ossia la creazione di file intermedi per la rappresentazione degli statement in un formato di immediato utilizzo;
- 84 minuti per la seconda fase, cioé l'effettiva archiviazione dei metadati tramite script PERL.

5.5.3 Tempi di esecuzione: confronto con RSSDB

Query in RSSDB L'esecuzione del comando `USE` utilizzando il formato RSSDB richiede l'utilizzo di tre relazioni per estrarre le informazioni di cui necessita:

- relazione per la proprietà `backPointer`: questa relazione deve essere interrogata per conoscere quali oggetti sono referenziati dalla URI fornita. Le informazioni che legano un elemento all'insieme delle URI che utilizzano tale elemento è mantenuto in questa relazione, che quindi deve essere interrogata prima di eseguire altri comandi. Il risultato di questo primo accesso è rappresentato da una lista di identificatori che

rappresentano gli oggetti di tipo `Occurrence` legati alla URI fornita al comando `USE`;

- relazione per la proprietà `occurrence`: per poter conoscere quali URI vengono referenziate. Dal precedente accesso alla tabella relativa alla relazione `backPointer` sono noti gli oggetti `Occurrence` referenziati nei modelli RDF; la proprietà `occurrence`, che per ciascuno di questi esprime la URI dell'elemento utilizzato, deve essere esaminata per poterne ricavare le informazioni di interesse;

- relazione per la proprietà `position`: per poter estrarre le informazioni relative al contesto in cui si referenzia un oggetto. Analogamente al precedente accesso, è necessario fornire anche il valore del contesto in cui si verifica una dipendenza logica.

Se si immagina di voler conoscere le dipendenze di tipo backward legate alla risorsa la cui URI è `cic:/Algebra/NEG_anti_convert.con` la query SQL associata assume questa forma:

```
SELECT DISTINCT t69.att1, t60.att1
  FROM t69, t60, t59
  WHERE t59.att0 = 'cic:/Algebra/NEG_anti_convert.con'
  AND t59.att1 = t60.att0
  AND t59.att1 = t69.att0
  ORDER BY t69.att1 ASC
```

Come visto in 4.2, le tabelle il cui nome segue il formato `t`n sono relative a classi di risorse oppure proprietà; in questo caso tutte le tabelle sono relative a proprietà RDF e, nel dettaglio:

- `t59` contiene i legami della proprietà `backPointer`;

- `t60` contiene i legami della proprietà `position`;

- `t69` contiene i legami della proprietà `occurrence`.

Nel dettaglio, la query SQL:

1. seleziona dalla relazione `t59` gli statement RDF il cui soggetto sia la URI fornita: `t59.att0 = 'cic:/Algebra/NEG_anti_convert.con'`;

2. impone che per ogni statement che soddisfa questo primo vincolo si crei un legame (*join*) fra l'oggetto di tale statement e il soggetto degli statement relativi alle proprietà `occurrence` e `position`: `t59.att1 = t60.att0 AND t59.att1 = t69.att0`. Questo vincolo è necessario per poter selezionare solo le risorse `Occurrence` legate alla URI fornita inizialmente e per ciascuna di esse i valori delle proprietà richieste;

3. restituisce i valori della URI referenziata e della posizione in cui è stata individuata una dipendenza logica: `SELECT DISTINCT t69.att1, t60.att1`.

L'esecuzione di questa query da parte dell'interprete SQL causa l'esecuzione di join fra tabelle di notevoli dimensioni: infatti, il formato RSS-DB prevede che *ogni* coppia di valori legati da una stessa proprietà venga mantenuta in una stessa tabella. Nel caso della libreria del prgetto HELM ogni statement relativo alle dipendenze logiche utilizza una proprietà di tipo `refObj` o `backPointer` e per ciascuna di queste si fornisce un legame di tipo `position` e un legame di tipo `occurrence`, portando la dimensione delle relative relazioni pari al numero di statement dell'intera libreria elettronica.

Query nel formato proposto Lo stesso comando viene eseguito, utilizzando il formato proposto, tramite due query SQL:

1. ```
   SELECT id
     FROM registry
     WHERE uri='cic:/Algebra/NEG_anti_convert.con'
   ```

   questa prima query ha il compito di selezionare dalla tabella `registry` l'identificativo univoco associato alla URI di cui si vogliono conoscere le
   ```

dipendenze. L'identificativo verrà utilizzato per ricavare il nome della relazione nella quale sono mantenuti i legami di dipendenza logica.

2. `SELECT uri, context`
   `FROM t1`
   `WHERE prop_id='B' ORDER BY uri ASC`

   in seguito, tramite la query presentata, si accede alla tabella relativa alla URI fornita (nell'esempio, `t1`) e si estraggono le coppie (URI, contesto) che identificano una dipendenza logica.

Il formato proposto per l'archiviazione delle metainformazioni ha quindi una struttura più lineare, in cui le tabelle presentano immediatamente le informazioni richieste dal linguaggio di interrogazione relative ad uno specifico oggetto della libreria.

Confronto di performance Un confronto prestazionale con il formato di archiviazione del pacchetto RSSDB è stato eseguito tramite la realizzazione di una *stress suite*, ossia una applicazione OCaml in grado di eseguire un elevato numero di query orientate all'interrogazione della base di dati.

Nel dettaglio, l'applicativo realizzato ha effettuato `6040` query in cui veniva utilizzato il comando `USE`, secondo il formato:

```
USE PATTERN p POSITION $a
```

L'esecuzione di ogni query è stata realizzata cambiando di volta in volta il valore del pattern `p`. In questo modo è stato possibile esaminare le performance sia nell'esecuzione del comando `USE` che del comando `PATTERN`.

I risultati sono stati ottenuti impostando le facility per il debug di PostgreSQL affinchè riportassero il tempo richiesto per l'esecuzione di ciascuna query SQL; sono stati, inoltre, registrati anche eventi di page fault, sollevati dall'engine di PostgreSQL quando le strutture dati in memoria non sono sufficienti per completare l'esecuzione di una query ed è necessario effettuare un accesso al disco. Naturalmente, l'accesso al disco rappresenta un collo di

bottiglia per l'interprete SQL che è costretto a caricare in memoria le informazioni necessarie (tipicamente i dati di una relazione, oppure gli indici ad essa associati) con dispendio di tempo.

Tempi di esecuzione La tabella 5.5 mostra il tempo medio di esecuzione delle query `USE` e `PATTERN` espresso in secondi nei due formati di archiviazione RSSDB e HELM DB (ossia, il formato sviluppato nell'ambito del progetto HELM):

	Statement `USE`	Statement `PATTERN`
HELM DB	`0.0217`	`0.0010`
RSSDB	`1.8228`	`0.0029`

Tabella 5.5: Tempo medio di esecuzione dei comandi `USE` e `PATTERN`

Si noti come il comando `PATTERN` presenti tempi dello stesso ordine di grandezza in entrambi i formati, mentre il comando `USE` risulta essere notevolmente più veloce nel formato HELM DB di circa due ordini di grandezza. Una ulteriore analisi dell'implementazione proposta mette in luce le caratteristiche delle due query richieste per l'esecuzione del comando `USE` nella tabella 5.6:

Accesso alla relazione `registry`	Interrogazione del modello
`0.0003`	`0.02139`

Tabella 5.6: Comando `USE`: tempi medi per query

È possibile constatare come l'accesso alla relazione `registry`, che ha il compito di partizionare la base di dati di documenti RDF, abbia un peso

minimo e permetta di eseguire lo statement MathQL `USE` interrogando la relazione relativa ad un modello in tempi molto bassi.

Le tabelle relative alla misurazione estesa dei tempi misurati nell'esecuzione dei test sono riportate nell'appendice D. Nell'immagine D.1 vengono evidenziati i page-fault registrati durante l'esecuzione della stress suite realizzata. Come è possibile notare, il formato RSSDB, dovendo gestire numerosi indici su molte tabelle, causa spesso una eccezione che richiede l'accesso al disco per bufferizzare porzioni di dati necessari all'esecuzione della query. Conseguenza di questa richiesta sono i tempi di esecuzione della query, notevolmente più alti nel caso in cui si utilizzi il formato RSSDB.

Capitolo 6

Conclusioni

La gestione di informazioni attraverso l'introduzione di metadati presenta notevoli vantaggi per quanto riguarda la descrizione delle informazioni e permette la realizzazione di strumenti ad-hoc per la loro gestione. L'utilizzo del formato RDF riesce ad ovviare ad una serie di problemi di interoperabilità che costituiscono il principale ostacolo nello scambio di informazioni dell'era Internet.

Il presente lavoro ha mostrato come il trattamento di metadati tramite strumenti tradizionali per l'archiviazione e l'interrogazione degli stessi presenti ancora alcuni aspetti per i quali gli strumenti a disposizione non sempre possono essere efficaciemente utilizzati; in particolare, la scelta di utilizzare un formato espressivo per la memorizzazione di metainformazioni che consenta di evidenziare tutti gli aspetti del formato RDF (sia sintattici che semantici) è generalemente in contrasto con requisiti prestazionali che impongono un efficiente sistema di reperimento dei dati. Il fatto che non vi sia un formato standard per l'archiviazione di metadati RDF utilizzando DBMS relazionali costituisce una riprova del fatto che differenti esigenze portano a diversi formati e scelte di memorizzazione.

Il formato proposto rappresenta una soluzione a questo problema quando le di operazioni di ricerca siano orientate alla interrogazione di proprietà di

un oggetto noto: la possibilità di suddividere gli statement RDF in strutture separate consente di individuare, per ogni oggetto, la relazione che raccoglie le proprietà ad esso riferite. In questo modo si evita di considerare l'intero contenuto informativo della base di dati (che può avere dimensioni notevoli) focalizzando l'attenzione su un insieme ristretto di informazioni.

6.1 Sviluppi futuri

La realizzazione del sistema di supporto per la gestione di metadati nel progetto HELM, sebbene come evidenziato in precedenza rappresenti un efficiente strumento per il reperimento di metadati, si presta a future migliorie e ampliamenti, fra i quali i più interessanti sono:

- possibilità di inserire documenti RDF di qualsiasi origine: la base di dati è attualmente realizzata basandosi su documenti RDF con uno specifico formato (dipendenze backward e forward). Un interessante sviluppo del formato di memorizzazione può riguardare l'inserimento di metadati RDF di qualsiasi origine, gestendo le peculiarità di ognuno. Ad esempio, il formato di metadati Dublin Core della libreria matematica associa prevalentemente metainformazioni a insiemi di documenti (referenziati dalla classe `DirectoryOfObjects`): è possibile immaginare l'esistenza di alcuni metadati per una gerarchia di documenti (ad esempio `cic:/Algebra`) e di altri metadati per specializzazioni della gerarchia (`cic:/Algebra/Basics`). In questo caso si presenta il problema di associare a tutte le gerarchie di documenti che appartengono ad una directory 'padre' i metadati definiti per essa, eventualmente estesi o modificati da metainformazioni specifiche per una determinata gerarchia.

  Questa tipologia di problemi implica la corretta realizzazione di strutture in grado di essere interrogate per *ereditarietà*: ad esempio, la URI `cic:/Algebra/Basic/NEG_anti_convert.con`, appartenente alla

directory `cic:/Algebra` eredita da questa le metainformazioni legate alla descrizione dei metadati Dublin Core o di qualsiasi tipo di informazione che si voglia rendere ereditabile; quindi, anche se non vengono fornite proprietà associate direttamente alla risorsa in esame, deve comunque essere possibile ricavarne i corretti valori interrogando risorse di livello più alto;

- arricchire l'espressività della base di dati: questa caratteristica rappresenta la principale differenza fra il formato scelto per la memorizzazione del progetto HELM e il formato offerto dal pacchetto RSSDB. In particolare, la memorizzazione di opportune strutture dati per l'interrogazione delle gerarchie di classi di risorse e proprietà rappresenta un interessante obiettivo nel momento in cui si decidesse di inserire all'interno della libreria elettronica documenti con maggiore complessità (in termini di relazioni fra classi di risorse e proprietà) rispetto a quelli presenti attualmente;

- realizzazzione di strumenti automatici per l'inserimento di metadati: attualmente l'inserimento all'interno della base di dati viene fatto in più passi, richiedendo la presenza di un utente che segua il processo di archiviazione. La memorizzazione di notevoli quantità di dati può richiedere tempi molto elevati: in questo caso è necessario realizzare strumenti in grado di gestire automaticamente il processo di caricamento, riportando eventuali condizioni di errore.

Di pari passo, lo sviluppo dell'interprete MathQL resta aperto all'introduzione di nuovi comandi specifici per il trattamento di particolari formati di metainformazioni: analogamente agli statement `USE`/`USEDBY` che sono orientati all'interrogazione di particolari attributi, l'introduzione di nuovi metadati può richiedere l'ampliamento dei comandi del linguaggio per gestire correttamente nuove proprietà.

Appendice A

Statement SQL

A.1 Inizializzazione

L'inizializzazione del database avviene tramite la creazione della relazione `registry`; l'attributo uri costituisce chiave primaria poiché qualsiasi operazione di accesso alla base di dati (`PATTERN` o `USE`/`USEDBY`) richiede l'individuazione di una URI; lo statement SQL relativo alla inizializzazione è:

```
CREATE TABLE registry
  (uri VARCHAR PRIMARY KEY, id INT NOT NULL UNIQUE)
```

A.2 Aggiunta di un nuovo modello

L'aggiunta di un nuovo modello avviene tramite due query SQL:

1. ```
   INSERT INTO registry
     VALUES (uri_doc, n)
   ```

   aggiornamento della relazione `registry` per memorizzare il legame fra la nuova URI inserita e la relazione creata in precedenza.
   ```

2. `CREATE TABLE tn`
   `(prop_id INT, context VARCHAR, uri VARCHAR)`

   creazione di una relazione destinata a contenere le proprietà legate al nuovo modello;

A.3 Interrogazione

A.3.1 Comando `PATTERN`

Il comando `PATTERN` viene eseguito tramite una sola query SQL:

```
SELECT uri FROM registry
  WHERE uri ~ pattern
  ORDER BY registry.uri ASC
```

Note

- `pattern` rappresenta una espressione regolare conforme allo standard SQL ottenuta a partire da un pattern MathQL;
- l'ordinamento sulle URI è richiesto dalle specifiche del linguaggio MathQL.

A.3.2 Comandi `USE/USEDBY`

Per l'esecuzione dei comandi `USE/USEDBY` sono richieste due distinte query SQL:

1. `SELECT id FROM registry`
   `WHERE uri = u`

Note

- `u` rappresenta la URI del documento di cui si vogliono estrarre le metainformazioni; a differnza della query per il comando `PATTERN` in questo caso viene utilizzato un operatore di ugualianza fra stringhe (`=`) e non un operatore di pattern match (`~`) poiché la URI è espressa per esteso (senza caratteri jolly).

2.
```
SELECT uri, context FROM t1
  WHERE prop_id = p
  ORDER BY uri ASC
```

Note

- `p` rappresenta l'identificativo della property RDF che si desidera interrogare.

Appendice B

Caricamento: esempio

Il seguente modello rappresenta un esempio di metadati della libreria elettronica gestita da HELM:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF xml:lang="en"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:h="http://www.cs.unibo.it/helm/schemas/mattone.rdf#">
  <h:Object
   rdf:about="cic:/Nancy/FOUnify/nat_complements/n_False.con">
    <h:refObj>
      <h:Occurrence>
        <h:position>MainConclusion</h:position>
        <h:occurrence>
          cic:/Coq/Init/Logic/not.con
        </h:occurrence>
      </h:Occurrence>
    </h:refObj>
    <h:refObj>
      <h:Occurrence>
        <h:position>InConclusion</h:position>
```

```
        <h:occurrence>
          cic:/Coq/Init/Logic/False.ind#(1/1)
        </h:occurrence>
      </h:Occurrence>
    </h:refObj>
  </h:Object>
</rdf:RDF>
```

L'elaborazione di questo modello con il foglio di stile XSLT produce un formato intermedio di puro testo in cui il subject degli statement viene riportato una sola volta nella prima riga poiché costante, mentre il resto del file contiene il tipo di dipendenza logica e relativi valori di contesto e URI per ogni proprietà separati dal carattere 'pipe':

```
cic:/Nancy/FOUnify/nat_complements/n_False.con
F|MainConclusion|cic:/Coq/Init/Logic/not.con
F|InConclusion|cic:/Coq/Init/Logic/False.ind#(1/1)
```

Il modello espresso in questo formato temporaneo viene archiviato nella base di dati prima aggiornando la relazione `registry`, in cui viene inserita una tupla come quella mostrata in tabella B.1:

cic:/Nancy/FOUnify/nat_complements/n_False.con	321

Tabella B.1: Tupla della relazione `registry` relativa al modello in esempio

in seguito viene creata la relazione `t321` il cui contentuto è mostrato nella tabella B.2:

prop_id	context	uri
F	MainConclusion	cic:/Coq/Init/Logic/not.con
F	InConclusion	cic:/Coq/Init/Logic/False.ind#(1/1)

Tabella B.2: Il contenuto della relazione relativa al modello in esempio

Appendice C

Interrogazione: esempio

L'esecuzione della query

```
SELECT uri
  IN
    USE PATTERN "cic:/Algebra/**" POSITION $a
  WHERE
    $a IS INBODY
```

viene realizzata eseguendo prima il comando all'interno della clausola `IN` e applicando poi al risultato così ottenuto i vincoli della clausola `WHERE`; il comando contenuto nella clausola `IN` a sua volta contiene il comando `PATTERN` di MathQL (che rappresenta la base di partenza per ogni query espressa in questo liguaggio). L'esecuzione della query viene pertanto realizzata nei seguenti passi

1. comando `PATTERN`: l'esecuzione di questo comando comporta la realizzazione della lista raffigurata nella tabella C.1. Si noti come l'esecuzione di un comando `PATTERN` non implichi la creazione valori per i parametri, ma solo la selezione di URI con uno specifico formato;

2. comando `USE`: il comando `USE` estrae le dipendenze logiche relative ad ogni URI resitituita dal comando `PATTERN`. Ogni URI può avere un

...	...
cic:/Algebra/Basics/NEG_anti_convert.con	
cic:/Algebra/CFields/cf_div.con	
...	...

Tabella C.1: Risultato del comando `PATTERN`

qualsiasi numero di dipendenze che, unitamente al valore dell'attributo posizione, costituiranno la nuova lista di risultati intermedi raffigurata nella tabella C.2;

...	...
cic:/Algebra/Basics/min_convert_is_NEG.con	$a = InBody
...	...
cic:/Algebra/Reflection/xexpr_ind.con	$a = InHypothesis
...	...

Tabella C.2: Risultato del comando `USE`

3. clausola `WHERE`: l'applicazione del filtro specificato dalla clausola `WHERE` porta all'esclusione dalla lista risultato delle URI il cui valore del parametro `$a` non sia quello richiesto. La tabella C.3 rappresenta questa situazione;

4. termine della query: al termine dell'esecuzione della query è necessario restituire solo la lista delle URI che fanno parte del risultato, senza i valori per i parametri contesto. Il risultato finale è presentato nella tabella C.4

...	...
cic:/Algebra/Basics/min_convert_is_NEG.con	$a = InBody
...	...

Tabella C.3: Applicazione del filtro

...
cic:/Algebra/Basics/min_convert_is_NEG.con
...

Tabella C.4: Risultato della query

Appendice D

Tabelle dei test comparativi

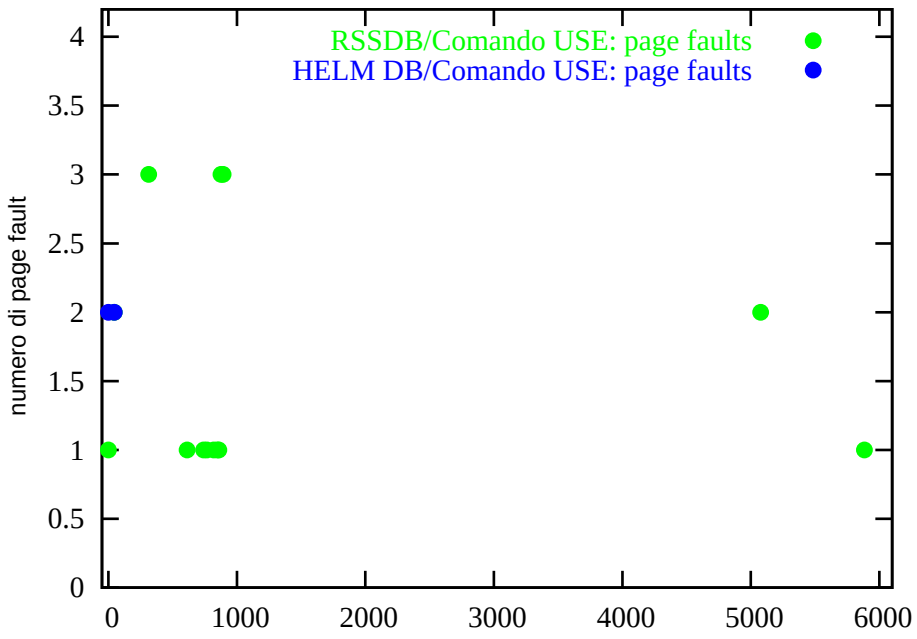

Figura D.1: Confronto fra i page-fault del comando `USE`

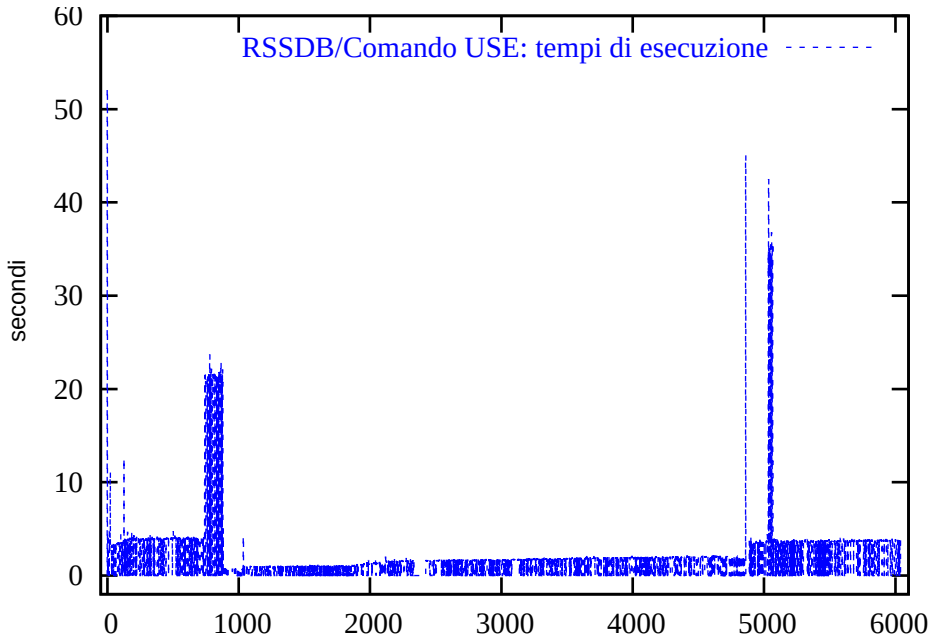

Figura D.2: Tempi di esecuzione del comando `USE` (formato RSSDB)

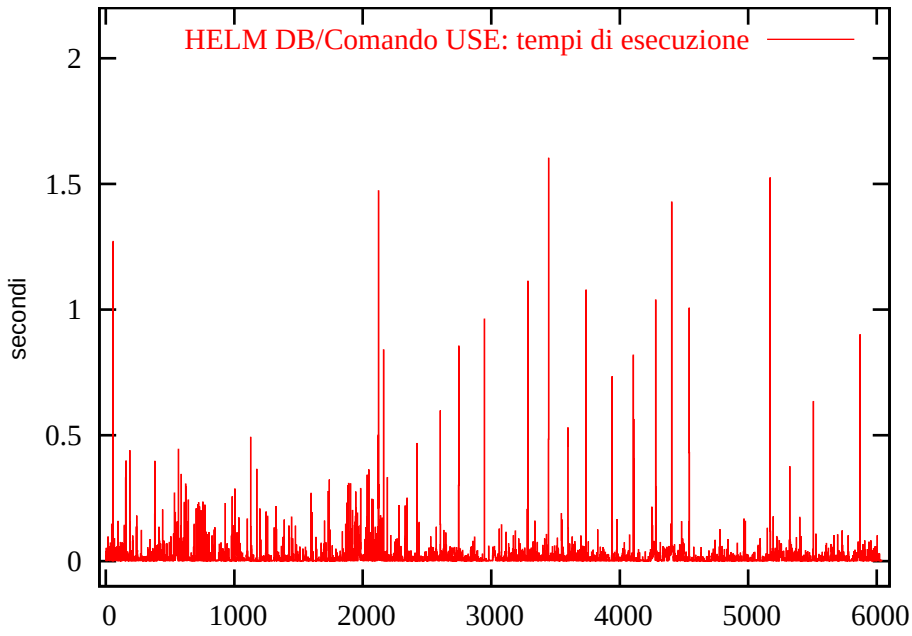

Figura D.3: Tempi di esecuzione del comando `USE` (formato proposto)

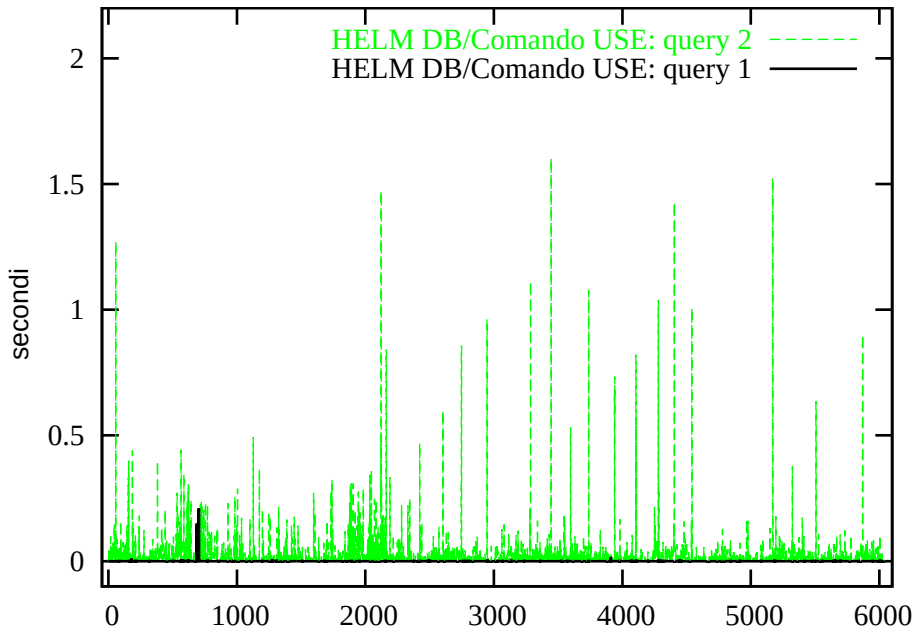

Figura D.4: Tempi di esecuzione delle due query che compongono il comando `USE` (formato proposto)

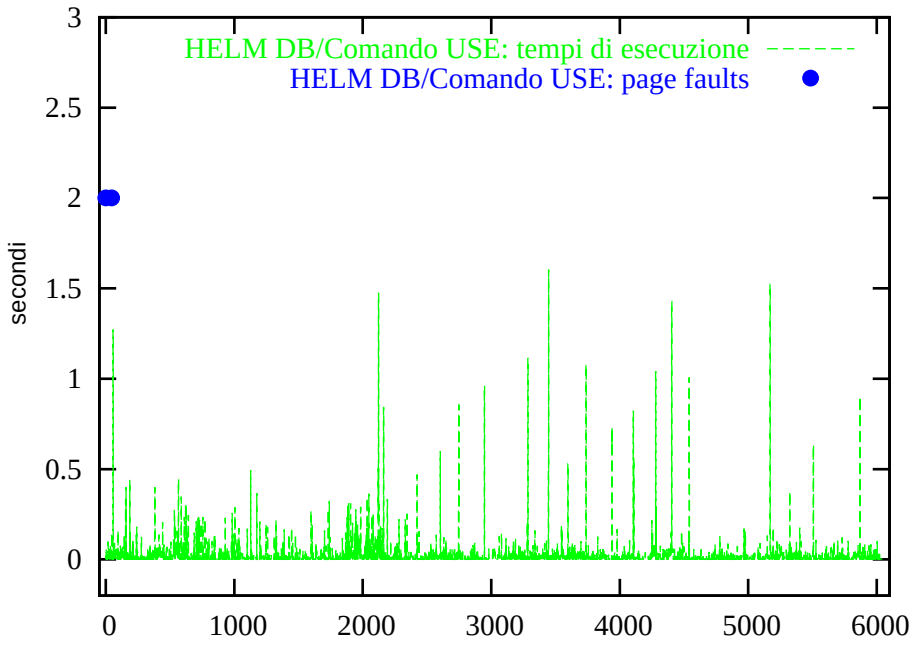

Figura D.5: Tempi di esecuzione e page-fault per il comando `USE` (formato proposto)

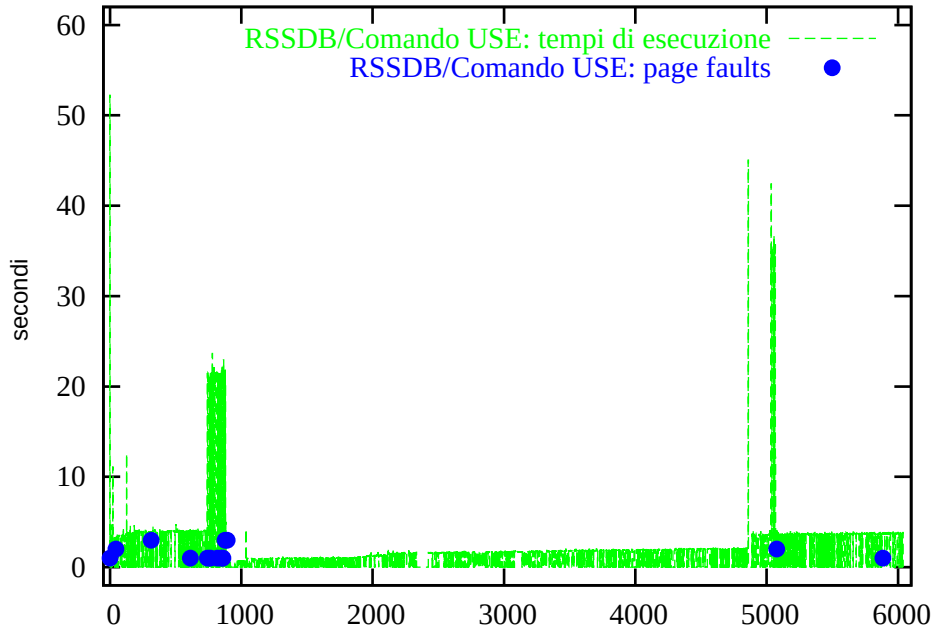

Figura D.6: Tempi di esecuzione e page-fault del comando `USE` (formato RSSDB)

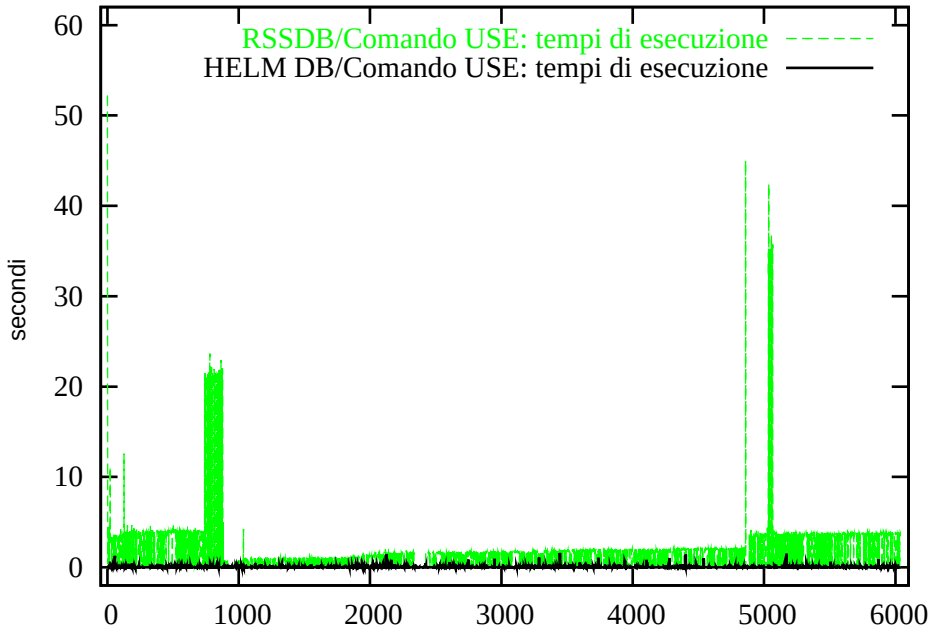

Figura D.7: Confronto fra i tempi di esecuzione del comando `USE`

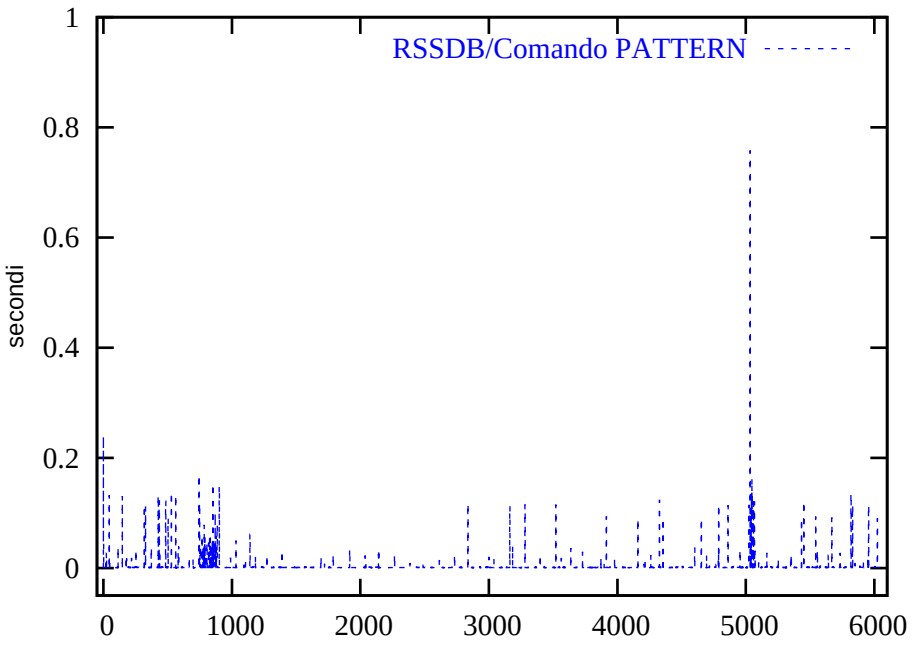

Figura D.8: Tempi di esecuzione del comando `PATTERN` (formato RSSDB)

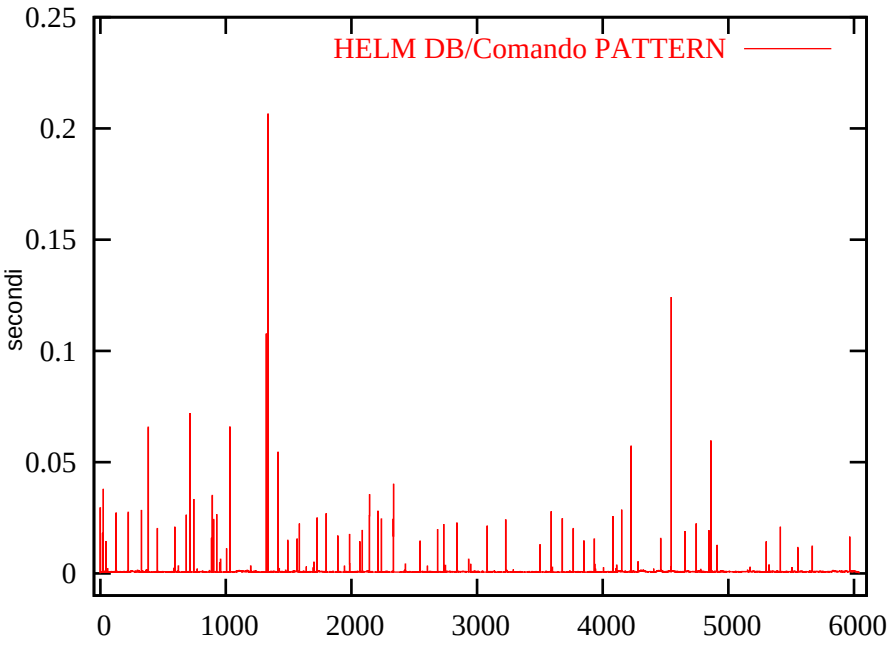

Figura D.9: Tempi di esecuzione del comando `PATTERN` (formato proposto)

Bibliografia

[SemWeb] Tim Berners-Lee, James Hendler, Ora Lassila, *Semantic Web*, Scientific American, maggio 2001

[URI] Tim Berners-Lee, R. Fielding, U. C. Irvine, L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax*, RFC2396, agosto 1998
`http://www.ietf.org/rfc/rfc2396.txt`

[RDFS] Dan Brickley, R. V. Guha, *Resource Description Framework (RDF) Schema Specification*, W3C Proposed Recommendation, 3 marzo 1999
`http://www.w3c.org/TR/PR-rdf-schema`

[RDF] Ora Lassila, Ralph R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification*, W3C Recommendation, 22 febbraio 1999
`http://www.w3c.org/TR/REC-rdf-syntax`

[DublinCore] *The Dublin Core Initiative*
`http://dublincore.org`

[PICS] *PICS Label Distribution, Label Syntax and Communication Protocols*, versione 1.1, W3C Recommendation, 31 ottobre 1996
`http://www.w3c.org/TR/REC-PICS-labels`

[Mozilla] *Mozilla Cross Reference*
`http://lxr.mozilla.org/seamonkey/source/rdf/base/idl/`

[XMLNS] Tim Bray, Dave Hollander, Andrew Layman, *Namespaces in XML*, W3C Recommendation, 14 gennaio 1999
`http://www.w3c.org/TR/REC-xml-names`

[XML] Tim Bray, J. Paoli, C. M. Sperberg-McQueen, *Extensible Markup Language (XML) 1.0*, W3C Recommendation, 10 febbraio 1998
`http://www.w3c.org/TR/REC-xml`

[XPath] James Clark, Steve DeRose, *XML Path Language (Xpath)*, W3C Recommendation, versione 1.0, 16 novembre 1999
`http://www.w3.org/TR/xpath`

[NTRIPLE] Art Barstow, Dave Beckett, *N-Triple*, W3C RDF Core WG Internal Working Draft, 6 settembre 2001
`http://www.w3c.org/2001/sw/RDFCore/ntriples`

[RDFPath] Stefan Kokkelink, *Transforming RDF with RDFPath*, 15 marzo 2001
`http://zoe.mathematik.uni-osnabrueck.de/QAT/Transform/RDFTransform.pdf`

[BerkleyDB] *SleepyCat BerkelyDB Library*
`http://www.sleepycat.com`

[SquishQL] *Inkling: RDF query using SquishQL*
`http://swordfish.rdfweb.org/rdfquery`

[SiRPAC] *RDF Validation Service*
`http://www.w3c.org/RDF/Validator`

[HELM] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Ferruccio Guidi, Irene Schena, *Mathematical Knowledge Management in HELM*, International Workshop on Mathematical Knowledge Management (MKM2001), settembre 2001
`http://www.cs.unibo.it/~sacerdot/`

[RDFSuite] ICS Forth, *RDFSuite*
`http://www.ics.forth.gr/proj/isst/RDF/`

[PostgreSQL] *PostgreSQL DBMS*
`http://www.postgresql.org`

[OCaml] *The Objective Caml system release 3.04*, Documentation and user's manual
`http://caml.inria.fr/ocaml/htmlman/`

[Postgres] Alain Frisch, *Postgres: OCaml bindings for PostgreSQL*
`http://www.eleves.ens.fr:8080/home/frisch/soft`

[RDQL] Hewlett Packard Laboratories, *Jena: A Java API for RDF*
`http://www.hpl.hp.com/semweb/`

Ringraziamenti

Ringrazio il Prof. Andrea Asperti e il team del progetto HELM con cui ho lavorato in questi mesi per il supporto che mi hanno offerto; in particolare, ringrazio il dott. Claudio Sacerdoti Coen per l'aiuto e il tempo che ha dedicato a risolvere i problemi sorti durante la fase di sviluppo del software realizzato.

Ringrazio tutti i miei amici e i miei compagni di corso per aver condiviso con me l'avventura universitaria: non è stata facile, ma d'ora in poi sarà anche peggio!

Il maggiore ringraziamento va ai miei genitori per la pazienza con cui hanno seguito il mio percorso di studi in questi anni e per il sostegno che non mi hanno mai fatto mancare: questa tesi è dedicata a loro.