Anno accademico 2011/2012 Gabriele Manduchi 611115

Costruzione di un framework semantico tuple-based Rdf-Jena per la coordinazione tra agenti.

Introduzione

La costruzione di un MAS e un'attività complessa, in cui le insidie da affrontare sono molteplici, e spaziano su vari ambiti; tra questi, uno dei più rilevanti, e al tempo stesso difficili, è rappresentato dalla coordinazione.
La coordinazione è quell'attività che agisce sullo spazio delle interazioni, imponendo regole e vincoli che gli agenti sono tenuti a rispettare. Grazie ad essa, un aggregato di componenti si trasforma in un vero e proprio sistema, in cui le parti collaborano e/o competono in modo sensato.
Con l'introduzione dei MAS, il problema della coordinazione e sempre più sentito, a causa del modo in cui sta evolvendo l'idea di software: dai programmi stand-alone isolati su un singolo computer, o su un singolo processo del sistema operativo, si sta passando inesorabilmente verso applicazioni di rete, in cui si prevede che il programma interagisca con qualche entità situata su un altro nodo, o addirittura che quel software esista solo in quanto distribuito su più nodi attivi. Uno dei modelli di coordinazione più famosi è rappresentato dagli spazi di tuple.

Un esempio di modello di coordinazione tuple-based è TuCSoN[1] ed è da esso che il mio lavoro ha preso ispirazione.
Tramite il lavoro di Nardini et Al [2] , ove ho partecipato alla conclusione

dell'implementazione e nella valutazione delle parformance, si è creata un estensione semantica del modello di coordinazione tuple-based, estendendo il concetto di tupla logica (la normale tupla linda[4] like)con quello di tupla semantica, ovvero una tupla che, sfruttando gli strumenti di reasoning (nel caso di TuCSoN semantico Pellet[5]), permette di gestire la comunicazione e l'interazione tra agenti mediante strutture dati più complesse e più esplicative.

Motivazioni

La creazione di un centro di tuple semantico ha messo in luce una grande problematica, la quale mi ha spinto ad intraprendere questo lavoro .
Essendo stato progettato Linda come framework di coordinazione utilizzante tuple logiche, si è pensato sempre a tuple di “piccolo peso” o comunque entità atomiche .
Con l'introduzione del concetto di tupla semantica, una tupla acquista un valore descrittivo innumerevolmente più alto, ma acquista anche una complessità molto elevata in quanto essa è poi collegata con tutto il resto dell'ontologia del sistema.
La primitiva che più risulta “pesante” in questa ottica è quella di In ; consideriamo un caso pratico.
Una tupla semantica rappresenta un individuo, ed è formata da tutti i suoi dati anagrafici ; ora questo individuo compie gli anni, per aggiornare la proprià età, utilizzando TuCSoN si deve :

Prelevare l'intera tupla dallo shared space. (eliminando le informazioni dall'ontologia, qui scatta un controllo di “consistenza”)
Aggiornare l'età
Reinserire la tupla semantica nello shared space.

Mentre tale operazione per una tupla logica è , in termini di performance , un'operazione banale , non si può dire la stessa cosa per una tupla semantica.

Il bisogno del concetto di “update” di una tupla semantica ci ha portati nella direzione di pensare ad un nuovo framework di coordinazione e , potente dalle conoscenze acquisite nel progetto di TuCSoN semantico , si è pensato di utilizzare il framework Jena .

Outline

Il lavoro svolto è suddiviso in 3 principali parti :

- Approfondimento della conoscenza
- Sviluppo delle primitive base del framework
- Test di performance.

Nel seguente articolo al punto 1 introdurrò gli strumenti utilizzati e le loro caratteristiche principali che mi hanno portato alla costruzione delle primitive.
Uno studio generale sulla coordinazione è stato seguito da uno più approfondito sugli strumenti da utilizzare per la costruzione delle primitive quali RDF, OWL ,JENA , SparQL e Sparul.

Nel punto 2 vi sarà un analisi approfondita delle primitive costruite con le valutazioni relative riguardo alle performance, primitive base per l'operaizione con le tuple semantiche quali Insert, Update , Delete, Match .

Nel punto 3 andrò ad analizzare le performance , utilizzando come infrastruttura per i test l'ontologia delle LSA del progetto SAPERE. Ogni singola primitiva è stata testa all'aumentare del numero delle LSA , il case-study utilizzato rappresenta già un'ontologia con un modesto grado di complessità, avente tuple con alberi a più livelli (introduce quindi il problema di ricorsività dei dati).

1. Premesse teoriche e strumenti utilizzati

1.1. Coordinazione nei sistemi software

La coordinazione nei sistemi tradizionali si basa sullo scambio di messaggi (qualunque cosa si intenda con il termine messaggio), aventi un particolare formato, conosciuto da tutte le entità partecipanti; la conoscenza del formato è necessaria per poter comprendere i dati contenuti.
Nei sistemi chiusi, questo è ammissibile; infatti tutti gli attori sono sviluppati e programmati dalla medesima organizzazione, perciò è facile fare in modo che tutti loro siano dotati di questa conoscenza.
L'introduzione dei sistemi distribuiti ha causato la nascita di nuovi tipi di sistemi,: i componenti non sono più situati su una unica macchina , ma sono dislocati su più nodi connessi per mezzo della rete. Questa rivoluzione ha visto il fiorire di innumerevoli tecnologie abilitanti, in modo molto più rapido rispetto alla formalizzazione di opportuni meta-modelli per i sistemi multi agente.
La nuova dimensione a cui bisogna prestare più attenzione nella modellazione è una : l'apertura.
Un sistema si dice aperto se non è dato conoscere a priori quali agenti ne entreranno a fare parte, durante il periodo di vita del sistema stesso.
Si tratta quindi di un sistema aperto e dinamico dal punto di vista del numero di partecipanti, nel quale gli agenti possono entrare a farne parte, oppure uscire a loro piacimento.
Questo vale anche per agenti che non fanno originariamente parte del sistema, e che sono stati sviluppati da terze parti; in altre parole, agenti sconosciuti.
L'assenza di informazione pregressa può valere da entrambi i punti di vista:

- il sistema potrebbe non conoscere gli agenti che ne entrano a fare parte
- un agente potrebbe non conoscere un sistema, ed entrarne a farvi parte.

Il concetto di apertura è legato in modo reciproco con quello di distribuzione:
Entità reciprocamente ignote sono tipicamente gestite da organizzazioni distinte, e collocate spazialmente su nodi diversi della rete; la scoperta dinamica è allora ipotizzabile se vi è la possibilità di sfruttare la rete per esplorare i dati e i servizi accessibili. Questo non significa che la distribuzione coincide con l'apertura: infatti vi possono essere sistemi, composti da numerose entità remote, tali da ammettere unicamente la presenza di entità conosciute.
I sistemi aperti presentano alcune problematiche nuove da affrontare, tra le quali:

- sicurezza;
- fiducia;
- comprensione dell'informazione.

La questione della sicurezza si pone perchè, consentendo l'accesso da parte di agenti di cui sono sconosciute le origini e le reali intenzioni, occorre proteggersi dal caso in cui essi intendano danneggiare o interrompere le funzionalità del sistema.
La fiducia ha anch'essa a che fare con agenti malevoli, ma che agiscono in modo più subdolo: puntando non ad effettuare atti distruttivi, ma ad alterare il corretto funzionamento del sistema tramite l'immissione deliberata di informazioni false;
da questo ne deriva l'esigenza di assegnare un tasso di credibilità agli agenti e alle informazioni da essi inserite.
I sistemi software più moderni sono sempre più orientati verso la distribuzione, la pervasività e l'eterogeneità dei componenti. In tali scenari, la dimensione in cui si rivela più ardua l'attività di ingegnerizzazione dei sistemi certamente non e la computazione, ma piuttosto l'interazione e coordinazione. Per poter impiegare l'usuale strategia di interpretazione sintattica dei dati, occorre che tutti gli agenti siano a conoscenza del formato utilizzato.

1.2. Tuple semantiche

Una tupla semantica deve necessariamente fare riferimento ad un'ontologia, la quale stabilisce quali sono i concetti e le proprietà definite in un particolare dominio.
Grazie alla semantica gli agenti non sono tenuti a conoscere in anticipo il formato esatto dei dati e il significato dei rispettivi elementi. Questo non significa però che un agende per poter operare tramite tuple semantiche non debba conoscere nulla in anticipo: infatti deve essere istruito sul significato degli elementi (concetti, proprietà) dell'ontologia.
In altre parole, la semantica da sola non porta allo sviluppo di agenti “intelligenti", cioè agenti che siano autonomamente in grado di comprendere la modellazione di un dominio, tramite l'analisi di un'ontologia.
Un essere umano e in grado di fare ciò, ma un agente software (come lo intendiamo ai giorni nostri) no. Ma allora che cosa cambia, con l'introduzione della semantica? Ci sono realmente dei vantaggi?
I vantaggi sono reali, e si basano soprattutto su due elementi:

- Il supporto implicito al ragionamento (reasoning);
- Il mapping tra ontologie differenti.

1.2.1. supporto implicito al ragionamento

Le attività di ragionamento fornite dalle tecnologie semantiche, provenienti per lo più dal campo della logica descrittiva [7], conferiscono alle basi di conoscenza caratteristiche altrimenti non ottenibili (quanto meno in modo automatico):

- il controllo di consistenza,
- l'organizzazione delle informazioni in una gerarchia di concetti e di proprietà.

Ma quale potenza ha un'attività di ragionamento automatico rispetto alle attività attualmente presenti di gestione dati ?

I principali strumenti utilizzati sono il framework Jena , Sparql , Sparul (estensione di sparql contenente i concetti di INSERT UPDATE e Delete) , Pellet (valutato poi scartato per problemi gravi di performance).

1.2.2 Mapping tra ontologie differenti.

Le ontologie possono essere definite da chiunque, sia che si intenda renderle disponibili pubblicamente, sia che siano destinate ad un uso privato. Ad ogni modo, e molto frequente che concetti analoghi siano definiti in più di una ontologia, tipicamente con nomi diversi, e probabilmente anche all'interno di gerarchie differenti; supponiamo, ad esempio, di avere due ontologie Ont1, Ont2 : in Ont1 abbiamo il concetto Fridge, ed in Ont2 Refrigerator; i due concetti sono equivalenti. Siccome ciò rappresenta in un certo senso una dispersione di sforzi e risorse, allora ci si e posti il problema dell'integrazione di ontologie eterogenee. In pratica, si tratta di definire dei mapping tra elementi (concetti, proprietà), di modo che il motore semantico sia in grado di considerare un insieme di ontologie, inizialmente separate, come un'unica grande ontologia, avente certi gruppi di elementi equivalenti tra loro. Per capire l'importanza di questo aspetto, si pensi al fatto che un agente deve essere preventivamente istruito sul significato di un'ontologia.

1.3 Strumenti utilizzati

Per ognuno degli strumenti che mi appresto ad elencare è stato speso del tempo attuo ad ottenere una solida base di conoscenza prima di addentrarsi nell'implementazione vera e propria.
Prima di parlare di Jena e sparql è necessario però introdurre RDF (lo strumento base proposto da W3C per la codifica, lo scambio e il riutilizzo di metadati strutturati) , OWL (linguaggio di markup per

rappresentare esplicitamente significato e semantica di termini con vocabolari e relazioni tra gli stessi)

1.4 RDF [6]

Rdf è un linguaggio utilizzato per la gestione di metadati strutturati; Qualunque cosa descritta da RDF è detta risorsa.
Una risorsa è reperibile in una locazione che è specificata da un URI[7] il quale funge anche da identificatore per la risorsa.

Il modello di dati RDF è formato da risorse, proprietà e valori. Le proprietà sono delle relazioni che legano tra loro risorse e valori, e sono anch'esse identificate da URI. Un valore , invece può essere :

- Un valore scalare (con tipo definito dall'ontologia)
- Un URI (riferimento ad un altra risorsa)
- Un blank node (risorsa non identificata).

L'unità base per rappresentare un'informazione in RDF è lo statement. Uno statement è una tripla del tipo "{*Soggetto–Predicato–Oggetto}*, dove il soggetto è una risorsa, il predicato è una proprietà e l'oggetto è un valore (e quindi anche un URI che punta ad un'altra risorsa).

Il data model RDF permette di definire un modello semplice per descrivere le relazioni tra le risorse, in termini di proprietà identificate da un nome e relativi valori. Tuttavia, RDF data model non fornisce nessun meccanismo per dichiarare queste proprietà, né per definire le relazioni tra queste proprietà ed altre risorse. A tale compito è definito da RDF Schema [9]

1.4.1 Blank node

Un blank node rappresenta una risorsa RDF alla quale non è stata associata un URI , esso è utilizzato per la gestione di strutture RDF complesse, formate ad albero; un esempio di struttura complessa, espressa tramite notazione N3 [9] è il seguente:

```
ex:obs485713
      a sensing:Observation, sapere:LSA ;
      sapere:updateTime "2012-04-14T11:00:00"^^xsd:dateTime ;
      sensing:observedBy ex:ubisenseofficeA ;
      sensing:about ex:ubisensetagZ11dx ;
      sensing:value
      [ a location:3DCoordinatePosition ;
        location:z [ :numericalValue "3.85" ;
                     :measuredIn ucum:meter ] ;
        location:y [ :numericalValue "41.80" ;
                     :measuredIn ucum:meter ] ;
        location:z [ :numericalValue "1.27" ;
                     :measuredIn ucum:meter ] ;
        sensing:confidence "0.7"^^xsd:double .
```

Tramite notazione N3 il blank node è identificato dai caratteri “[“ “]” , ovvero la proprietà value ha come oggetto un blank node, il quale è una risorsa composta a sua volta da altre proprietà ed altri blank node; se si dovesse fare il grafico ad albero di tale risorsa ogni oggetto “scalare” rappresenta una foglia, mentre un blank node rappresenta una diramazione .

1.5 OWL [10]

Lo scopo di OWL è descrivere delle basi di conoscenze, effettuare delle deduzioni su di esse attraverso il reasoning.

Grazie a OWL in futuro sarà possibile, ad esempio, effettuare delle ricerche estremamente complesse nel web evitando i problemi di omonimia e ambiguità presenti nelle normali ricerche testuali. Altro

scopo di OWL è permettere alle applicazioni di effettuare delle deduzioni sui dati.

La rappresentazione dei termini e delle relative relazioni è chiamata ontologia.

Insieme ad RDF, di cui è un'estensione, OWL fa parte del progetto, ancora *in itinere*, del Web semantico.

Per scegliere un sottoinsieme della logica del prim'ordine che sia decidibile si è utilizzata la logica proposizionale aumentandone la potenza aggiungendo delle logiche rappresentate per convenzione con delle sigle .

1.6 Sparql [11]

SPARQL, linguaggio di interrogazione dell'RDF, anch'esso facente parte degli standard W3C, è stato accolto entusiasticamente come l'agognato ultimo tassello per l'edificazione del Web semantico[12].

Sparql si interpone come il linguaggio di interrogazione di RDF anch'esso basato sulla tipologia di interrogazione “soggetto predicato oggetto”.

soggetto	**predicato**	**oggetto**
id1234	anno	1994

Si nota come è facilmente "traducibile" come segmento di riga della tabella di un database relazionale nella forma "chiave - nome colonna - valore colonna":

ID	**anno**	**.....**
1234	1994	.....

SPARQL adotta la sintassi Turtle[13], un'estensione di N-Triples, alternativa estremamente sintetica e intuitiva al tradizionale RDF/XML.

Si considerino le seguenti triple RDF, che saranno utilizzate nel corso dell'articolo come riferimento per le query d'esempio:

```
@prefix cd: <http://example.org/cd/>
@prefix: <http://example.org/esempio/>
:Permutation cd:autore "Amon Tobin".
:Bricolage cd:autore "Amon Tobin".
:Amber cd:autore "Autechre".
:Amber cd:anno 1994.
```

Le asserzioni sono espresse in concise sequenze soggetto-predicato-oggetto e delimitate da un punto fermo.”@prefix” introduce prefissi e namespace; i due punti senza prefisso (seconda riga) definiscono il namespace di default. Gli URI sono inclusi tra parentesi uncitate. I letterali di tipo stringa sono contrassegnati da virgolette.
Le query SPARQL si basano sul meccanismo del "pattern matching" e in particolare su un costrutto, il "triple pattern", che ricalca la configurazione a triple delle asserzioni RDFfornendo un modello flessibile per la ricerca di corrispondenze.

```
?titolo cd:autore ?autore.
```

In luogo del soggetto e dell'oggetto questo "triple pattern" prevede due variabili, contrassegnate con ?.

Le variabili fungono in un certo senso da incognite dell'interrogazione,cd:autore funge invece da costante: le triple RDF che trovano riscontro nel modello associeranno i propri termini alle variabili corrispondenti.

Per chiarire, ecco una semplice query di selezione SPARQL:

```
PREFIX cd: <http://example.org/cd/>
SELECT ?titolo ?autore ?anno
FROM  <http://cd.com/listacd.ttl>
WHERE {?titolo cd:autore ?autore.
            ?titolo cd:anno ?ann .
}
```

L'analogia con SQL è lampante, nel seguente elenco si possono vedere le parolechiavi di sparql :

- PREFIX dichiara prefissi e namespace.
- SELECT definisce le variabili di ricerca da prendere in considerazione nel risultato (nell'esempio: titolo, autore e anno).
- FROM specifica il set di dati su cui dovrà operare la query
- WHERE definisce il criterio di selezione specificando tra parentesi graffe uno o più "triple patterns" separati da punto fermo.
- UNION esprime un OR logico: la query non si limita pertanto alle triple che soddisfano entrambi i triple patterns, ma cattura sia quelle che soddisfano il primo, sia quelle che soddisfano il secondo.

```
PREFIX cd: <http://example.org/cd/>
SELECT ?titolo ?autore ?anno
FROM  <http://cd.com/listacd.ttl>
WHERE{
                {?titolo cd:autore ?autore}
                UNION
                {?titolo cd:anno ?anno}


}
```

1.6.1. Restrizioni sui valori

È possibile porre restrizioni sui valori da associare alle variabili. Ad esempio:

```
PREFIX cd: <http://example.org/cd/>
SELECT ?titolo ?anno
FROM  <http://cd.com/listacd.ttl>
WHERE {?titolo cd:anno ?anno.
            FILTER (?anno > 2000).
}
```

In questo caso, la restrizione è effettuata mediante l'operatore di confronto > : il filtro esclude i termini che non soddisfano la condizione definita tra le parentesi tonde: il risultato in questo caso è nullo (il dataset di riferimento non prevede valori maggiori di 2000 per la proprietà anno).

1.6.2 Funzioni base e custom

Dal paragrafo precedente si è visto come è possibille specifiare pattern di filtraggio , SPARQL prevede anche la funzionalità di richiamare in questo costrutto FILTER funzioni che possono essere di due tipologie:

- Default → già inserite direttamente da SPARQL
- Custom → create ad hoc , nel nostro caso in Java.

```
PREFIX cd: <http://example.org/cd/>
SELECT ?titolo ?autore
FROM   <http://cd.com/listacd.ttl>
WHERE  { ?titolo cd:autore ?autore .
            FILTER regex(?autore, "^au", "i")
}
```

In questo esempio il filtro seleziona, senza riguardo per maiuscole o minuscole, solo gli autori che iniziano per "au", utilizzando una funzione default.

titolo	autore
"Amber"	"Autechre"

Oltre al costrutto filter per le funzioni è possibile utilizzare il costrutto BIND, il quale assegna un valore ad una variabile.

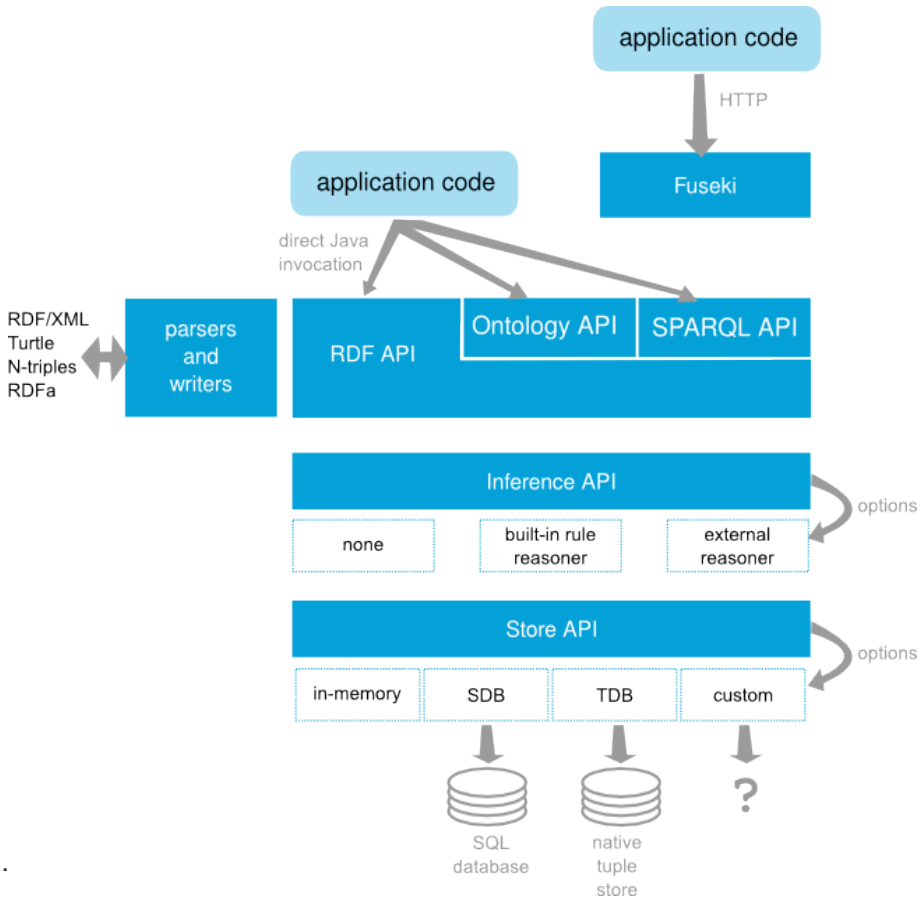
1.7 Jena

Si tratta di un progetto open source, che ha beneficiato dell'apporto di professionalità offerto dai laboratori HP.

Jena è stato inizialmente sviluppato per la realizzazione di applicazioni, che supportassero RDF e RDFS, ma successivamente esteso verso i linguaggi per le ontologie tra i quali DAML+OIL e OWL.

Principalmente il framework permette e contiene:

- RDF API;
- la lettura e scrittura RDF in RDF/XML, N3 e N-Triples;
- OWL API;
- la memorizzazione persistente e *In-Memory*;
- un motore di query SPARQL
- un motore di reasoning interno



.

Ci sono vari tipi di linguaggi per le Ontologie che vanno dal più espressivo, l'OWL Full a quello più semplice RDFS, il framework Jena mira a fornire, tramite l'Ontology API, una interfaccia di programmazione coerente, indipendentemente dal linguaggio utilizzato. Ciò è realizzato mettendo a disposizione una interfaccia comune a tutte le tipologie di linguaggio, l'interfaccia OntModel.

2. Implementazione delle primitive

A seguito di un approfondito studio dei vari componenti per la creazione del framework è seguita la fase di implementatazione vera e propria, nel dettaglio verranno riportati la procedura di creazione di una query custom e i dettagli implementativi di ogni funzione , con le relative valutazioni di efficienza.
La grande innovazione relativa al lavoro svolto è quella di , attraverso le query custom , dare maggiore potenza al motore di query, estendendo il concetto di funzione da un semplice “filtrao” di dati ad una vera e propria componente dell'intelligenza del sistema.
Difatti è stata riscontrata la possibilità di effettuare dei sides-effect sul modello attraverso queste query, la possibilità di crearle in Java ci permette di operare anche con un modello condiviso e da una query farne scaturire altre a cascata o qualsivoglia operazione , il tutto tenendo conto dei problemi di consistenza relativi alle modifiche “on – line” sull'ontologia.
Difatti non si ha una totale autonomia ed indipendenza.
Supponiamo di essere all'interno di una select, non è possibile eliminare fisicamente triple RDF che fanno match con essa attraverso una funzione perchè ciò renderebbe inconsistente la query stessa.
Questi “vincoli” comunque non ci hanno impedito la creazione delle primitive base.

2.1 Collegamento Sparql – Java:funzioni custom .

Sparql, attraverso jena, offre la funzionalità di richiamare funzioni create anche con diversi linguaggi , i passi necessari per poter sfruttare questa funzionalità sono essenzialmente 3 :

- Definizione della funzione
- Registrazione della funzione
- Richiamo della funzione all'interno dei costrutti “BIND” o “FILTER” .

2.1.1 Definizione della funzione

è possibile creare una classe che diventi poi funzione richiamabile da SPARQL derivando dalla classe FunctionBaseN (dove N sta per il numero di parametri che si vogliono fornire alla funzione), è necessario poi eseguire un override del metodo exec , un esempio molto semplice è il seguente:

```
public class customFunction extends FunctionBase1 {

public customFunction (){
      super();}
public NodeValue exec(NodeValue nv){
      if(nv.isString() && nv.toString().contains("Jones"))
            return NodeValue.TRUE;
      return NodeValue.FALSE;
}
```

In questo caso è stata creata una funzione che accetta un parametro (functionBase1) , i valori di ingresso e uscita sono sempre del tipo NodeValue .

Nota bene :

per tutte implementazioni è stato utilizzato un model condiviso , ovvero una classe statica che fornisce il model jena per poter lavorare sullo stesso store di dati sia “user side” che “server side”.

2.1.2 Registrazione di una funzione java in sparql.

Il framework jena ha implementato un registro statico chiamato FunctionRegistry , il quale contiene tutte le funzioni create da Jena per l'ausilio a SPARQL .
Per registrare una nuova funzione occorre eseguire la seguente operazione :

```
FunctionRegistry.get().put("http://example.org/function#cus
Func", customFunction.class)
```

Dove il metodo put accetta 2 parametri : l'URI che si utilizzerà successivamente per richiamare la funzione e la classe ove quest'ultima è implementata.

2.1.3 Richiamo della funzione custom

Terminata la fase di implementazione e registrazione ora la funzione è utilizzabile nei costrutti BIND e FILTER come specificato nel paragrafo 1.6 .
Si riporta ora un esempio di utilizzo per il costrutto FILTER.

```
PREFIX cd: <http://example.org/cd/>
PREFIX function: <http://example.org/function#>
SELECT ?titolo ?autore
FROM   <http://cd.com/listacd.ttl>
WHERE  { ?titolo cd:autore ?autore .
            BIND function:cusFunc(?autore)
}
```

2.2 Implementazione primitive.

Le primitive di base di qualsiasi framework che debba manipolare qualsiasi tipo di informazione sono sempre le stesse ovvero :

- Inserimento di dati
- Eliminazione
- Modifica

A queste è stato aggiunto , dato il contesto semantico nel quale il prototipo sarà sfruttato , una quarta primitiva : il match .
Per match si intende quel procedimento atto ad individuare le offerte che meglio rispondono, o che meglio combaciano (*match-ano*), anche solo in parte, con una richiesta effettuata da un soggetto.
Nel caso del nostro framework il match, stante l'architettura dati prevista tramite ontologia , sarà di tipo semantico .

2.2.1 Match semantico

Il match semantico è stato sviluppato in 3 differenti modalità :

- inclusione → ovvero un match è ritenuto tale se tutte le proprietà del soggetto A sono anche nel soggetto B
- corrispondenza → un match è ritenuto tale se tutte le proprietà del soggetto A sono esattamente quelle del soggetto B (quindi il soggetto B non ne può avere altre)
- match per rdf type

La complicazione principale incorsa nello sviluppo è dovuta alla struttura delle tuple semantiche dell'infrastruttura.
Difatti una tupla è formata da una radice(Id) e N diramazioni che possono essere foglie (quindi valori literal) oppure Bnode (quindi ulteriori diramazioni).
Tramite SPARQL non è stato possibile ottenere query che ricorsivamente andassero ad indagare sulla struttura del grafo, quindi si è sfruttato anche il framework jena.

Dettaglio implementativo → prima versione .

Lato “utente” la funzione di match è una semplice funzione da richiamare , essa ritorna true se il
soggetto A matcha con il soggetto B (quindi , nel caso di inclusione , è il soggetto A che deve essere incluso in B e non viceversa).
Un esempio di utilizzo è il seguente:

```
PREFIX sapere: <http://sapere.org/function#>
prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
Select DISTINCT ?s
       WHERE {?s ?p ?o.
              FILTER(sapere:match(?
s,<http://somewhere/JohnSmith>))
             }
```

La seguente funzione seleziona tutti i soggetti che fanno match con il soggetto “JohnSmith” .

Lato server la funzione è divisa per step :

- Tramite select sparql vengono individuate tutte le proprietà del soggetto A
- Per ogni proprietà si controlla se sono literal (foglie) o Resource (Bnode)
- Se è literal allora si controlla che vi sia una proprietà medesima nel soggetto B (via SPARQL)
- Se è resource si richiama una funzione ricorsiva che :
  - Punta al Bnode tramite l'id di quest'ultimo.
  - Ricava da esso tutti le proprietà
  - per ogni proprietà controlla se è una foglia o una risorsa
  - Se è foglia controlla che nel nodo paritario del Soggetto B vi sia la stessa proprietà
  - Se è resource la funzione richiama se stessa sul livello più basso.

Osservazioni :

è stato riscontrato, tramite le successive prove di performance, che richiamare query SPARQL all'interno di funzioni custom ha un costo in termini di performance maggiore rispetto all'utilizzo del framework jena esplorando l'albero RDF , sono state quindi riscritte la funzione di match semantico e tutte le successive funzioni utilizzando , nel possibile, esclusivamente il framework jena.

Dettaglio implementativo → seconda versione

La funzione lato server , tenendo conto della stessa logica espressa nella prima versione , è stata riscritta con un unica funzione ricorsiva che percorre l'albero dalla root della Tupla semantica a tutte le foglie; per incrementare le performance al primo risultato negativo viene restitutito l'insuccesso senza aspettare la fine del controllo sull'albero.

```
private boolean RecursiveSemanticMatchInBlankNodes(Model
model,Resource principal,Resource secondary)
      {
               Selector selPrimary = new
SimpleSelector(principal,null,(RDFNode)null);
               Selector selSecondary=new
SimpleSelector(secondary,null,(RDFNode)null);
               StmtIterator iterPrimary=
model.listStatements(selPrimary);
               boolean result =true;
               while(iterPrimary.hasNext() && result){
                     boolean find=false;
                     Statement stm=iterPrimary.nextStatement();
                     Property predicate= stm.getPredicate();
                     RDFNode object = stm.getObject();
                     selSecondary=new
SimpleSelector(secondary,predicate,(RDFNode)null);
                     StmtIterator
iterSecondary=model.listStatements(selSecondary);
                     while(iterSecondary.hasNext() && !find){
// controllo se nel soggetto secondario ci sono le proprietà
prelevate dal primario.
                           Statement stmSecondary=
```

```
                            iterSecondary.next();
                            Property predSecondary=
                            stmSecondary.getPredicate();
                            RDFNode objectSecondary=
                            stmSecondary.getObject();
          if(predicate.toString().equals(predSecondary.toString())){
              if(object.isResource() &&
        object.asNode().isBlank()){
                      // controllo se ho un altro blank node

        if(objectSecondary.isResource())

find=RecursiveSemanticMatchInBlankNodes(model, object.asResource(),
objectSecondary.asResource()); //ripeto operazio
______else
              return false;
        }else if(object.isLiteral()){// non è una risorsa quindi è un
literal
        find=object.asLiteral().equals(objectSecondary.asLiteral());
        }
        else //resource di tipo "non blank"
        {
              find=
object.asResource().getURI().equals(objectSecondary.asResource().getU
RI());
        }

        }

        }

result&=find;
}
return result;
}
```

La funzione è relativa al match inclusivo, ovvero tutto “l'albero” del soggetto A deve essere presente nel soggetto B : per quello che riguarda il match esatto la funzione è la medesima con l'aggiunta di un controllo sull numero di statement totali di ogni “livello” , se esso corrisponde si prosegue.

2.2.2 Copia di una priopietà

Questa funzionalità offre la possibilità di prendere parte di una Tupla semantica e copiarla in un altra, in particolare la funzione copia una proprietà da un soggetto ad un altro.

Esempio: *Lato client*

```
"PREFIX sapere: <http://sapere.org/function#>
prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
Insert {<http://somewhere/AllBanchSmith>
<http://somewhere/peopleInfo#parents> ?o;}

WHERE  { BIND(sapere:CopyGraph(<http://somewhere/JohnSmith>,
       <http://somewhere/peopleInfo#parents>) as ?o)}";
```

Viene quindi inserita nella tupla semantica “AllBanchSmith” la proprietà “parents” .

Lato server la funzionalità prende spunto da quanto già sfruttato nel caso del match semantico, ed anche qui la copia viene effettuata in modalità ricorsiva nel caso l'oggetto da copiare sia un Bnode.
In questo caso è stato ritenuto conveniente aggiungere una query SPARQL per il controllo dell'effettiva esistenza della proprietà all'interno della tupla.
L'oggetto è costruito con “ricorsione a destra” ,ovvero si scorre l'albero fino alle foglie e da esse poi viene costruito tutto l'oggetto fino a ritornarne l'id iniziale che viene restituito dalla funzione.

```
private RDFNode CheckProprietiesAndCopy(Model model, String
URIPrincipal,String predicate){
try {
    String query= "prefix vcard: <http://www.w3.org/2001/vcard-
rdf/3.0#> Select ?o WHERE { "+URIPrincipal+" "+predicate+" ?o.}";

    Query queryJena= QueryFactory.create(query);
    QueryExecution qe =
```

```
QueryExecutionFactory.create(queryJena,model );
    ResultSet results = qe.execSelect();
    if(results.hasNext())
    {
      QuerySolution solution= results.next();
      RDFNode object=solution.get("o");
      if(object.isResource() && object.asNode().isBlank())
      {
            return
RecursiveCopyBlankNodes(model,object.asResource());
      }
      else // è un literal
      {
            return object;
      }
    }
   }
   catch(Exception ex){
      ex.printStackTrace();
      System.out.print("error");
      }
return null;
}


private RDFNode RecursiveCopyBlankNodes(Model model,Resource
principal)
{
   Selector selPrimary = new SimpleSelector(principal,null,
(Object)null);
   StmtIterator iterPrimary= model.listStatements(selPrimary);
   boolean result =true;
   Resource objectCopy=model.createResource();
   while(iterPrimary.hasNext() && result){
   boolean find=false;
   Statement stm=iterPrimary.nextStatement();
   Property predicate= stm.getPredicate();
   RDFNode object = stm.getObject();
   if(object.asNode().isBlank()){
objectCopy.addProperty(predicate,RecursiveCopyBlankNodes(model,object
.asResource()));
   }
   else{
```

```
        objectCopy.addProperty(predicate,object);
         }
    }
    return objectCopy;
}
```

2.2.3 Clone di tupla semantica.

Avendo appena implementato la funzionalità di copia di proprietà il passo alla copia dell'intero soggetto sembrava breve .
Difatti la funzione di clone , in una prima implementazione , richiamava a sua volta la funzione di copia proprietà per ognuno degli attributi della tupla semantica.
Questa operazione però , dato che per richiamare la funzione di copia del grafo è necessaria una query per ogni ramo della tupla , è poco perfomante; ne è conseguita la riscrittura totale utilizzando il solo framework jena .

```
public NodeValue exec(NodeValue indir) {
       Model model= modelJena.getInstance();
          String URIsel= indir.getNode().getURI();
          String URIToAdd=URIsel.substring(0,URIsel.length())
+"_Clone";
          Resource cloneInd=model.createResource(URIToAdd);
          ArrayList<Property> propList=new ArrayList<Property>();
           ArrayList<RDFNode> objList=new ArrayList<RDFNode>();
             try {

                          Selector selPrimary = new
SimpleSelector(model.getResource(URIsel),null,(RDFNode)null);
                            StmtIterator iterPrimary=
model.listStatements(selPrimary);
                            boolean result =true;
                            while(iterPrimary.hasNext() && result){
                                  Statement
stm=iterPrimary.nextStatement();
                                  propList.add(stm.getPredicate());
                              objList.add(stm.getObject());
                            }
```

```
                        for(int i=0;i<objList.size();i++)
                        {
                              RDFNode object=objList.get(i);
                              Property prop=propList.get(i);
                              if(object.isResource() &&
object.asNode().isBlank())
                           {

                              model.add(cloneInd,prop,
RecursiveCopyBlankNodes(model,object.asResource()));
                           }
                           else if(object.isResource())// è un
resource valorizzato
                           {
                              model.add(cloneInd,prop,object);
                           }
                           else // è un literal
                           {

      model.addLiteral(cloneInd,prop,object.asLiteral());
                           }
                        }

                  }
                catch(Exception ex){
                  ex.printStackTrace();
                  System.out.print("error");
                }

      return NodeValue.makeString(URIToAdd);
 }


private RDFNode RecursiveCopyBlankNodes(Model model,Resource
principal)
{
        Selector selPrimary = new SimpleSelector(principal,null,
(RDFNode)null);
        StmtIterator iterPrimary= model.listStatements(selPrimary);
        Resource objectCopy=model.createResource();
        ArrayList<Property> propList=new ArrayList<Property>();
          ArrayList<RDFNode> objList=new ArrayList<RDFNode>();
        while(iterPrimary.hasNext()){
```

```
                Statement stm=iterPrimary.nextStatement();
                propList.add(stm.getPredicate());
            objList.add(stm.getObject());
          }
          for(int i=0;i<objList.size();i++){
                    RDFNode object=objList.get(i);
                    Property prop=propList.get(i);

                if(object.asNode().isBlank()){
                      objectCopy.addProperty(prop,
RecursiveCopyBlankNodes(model,object.asResource()));  //

                }
                else{
                      objectCopy.addProperty(prop,object);
                }
          }
        return objectCopy;
}
}
```

2.2.4 Delete di una tupla semantica.

Data la struttura delle tuple semantiche prevista ad albero, la semplice DELETE utilizzata da SPARUL (sparql-update) non è sufficente.
Difatti il soggetto verrebbe si eliminato, ma all'interno del model resterebbero “orfani” tutti i Bnode collegati ad essa, che andrebbero quindi a “sporcare” l'RDF store .
Per far si che l'eliminazione sia completa è stata implementata la funzione di “CascadeDelete” , che ricorsivamente elimina (con ricorsione a destra, quindi bottom-up) anche i Bnode collegati ad essa.

Esempio : lato Client

Lato client l'istruzione è una semplice Delete :

```
PREFIX sapere: <http://sapere.org/function#>
prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
DELETE {<http://somewhere/JohnSmith> ?p ?o}
WHERE { <http://somewhere/JohnSmith> ?p ?o .
FILTER(sapere:CascadeDelete(<http://somewhere/JohnSmith>))}";
```

Lato server il comportamento è del tutto analogo alla funzione ricorsiva della clone .

```
private boolean RecursiveDeleteBlankNodes(Model model,Resource
principal)
      {
            try{
              Selector selPrimary = new
SimpleSelector(principal,null,(Object)null);
        StmtIterator iterPrimary= model.listStatements(selPrimary);
              boolean result =true;
              ArrayList<Statement> stmtToDel=new
ArrayList<Statement>(); // lista di statement da eliminare.
              while(iterPrimary.hasNext() && result){
                    Statement stm=iterPrimary.nextStatement();
                    stmtToDel.add(stm);
              }
              for(Statement stm:stmtToDel){
                    if(stm.getObject().asNode().isBlank()){
RecursiveDeleteBlankNodes(model,stm.getObject().asResource());
                    }
                    model.remove(stm);
              }
            }catch(Exception ex)
            {
             ex.printStackTrace();
            }
      return true;
      }
```

Una differenza implemetativa è che in questo caso prima vengono selezionati tutti i blankNodes da eliminare , poi vengono eliminati successivamente (è indispensabile per non incorrere in problematiche di concorrenza tra la Select dei nodi e l'eliminazione di questi).

2.2.5 NewId.

La funzione di NewId è una funzione che centralizza l'attribuzione dei nominativi delle tuple semantiche , l'idea base è quella che parte della tupla (quella propria del nodo) venga impostata tramite UNIQUEIDENTIFER, mentre la restante parte viene richiesta alla base di conoscenza (o chiamata a servizio, da valutare).
Essa restituisce la risorsa che funge da Soggetto, non la inserisce nello store.

```
public NodeValue exec(){
        Model model= modelJena.getInstance();
        String
newId="http://sapere.org"+"/"+modelJena.getCurrentNodeName()
+"_"+UUID.randomUUID().toString();
        System.out.println("newId--->"+newId);
        Resource res= model.createResource(newId);
        return NodeValue.makeNode(res.asNode());
 }
```

3. Benchmark

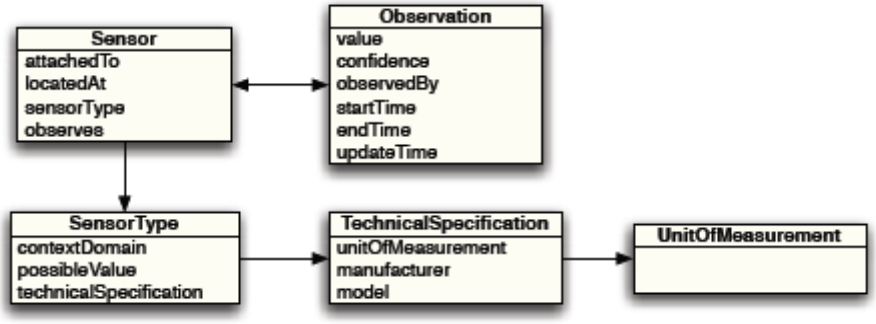
Per una migliore visione d'insieme dei possibili scenari nei quali il framework semantico possa essere utilizzato si è effettuato un Benchmark un particolare scenario reale , utilizzando quindi un ontologia concreta con un buon grado di complessità.
Il caso reale studiato è il progetto Sapere[15] e la propria ontologia di riferimento.
Il Benchmark come gia detto e stato per lo piu incentrato sulle tempistiche , nodo dolente di tutta l'architettura, quindi non considera al momento la quantita di memoria utilizzata.
Per ogni Benchmark si e studiata la reattivita , la rapidita di risposta delle operazioni fondamentali quali rd , in e out , all'incrementare del numero di individui presenti all'interno.
Oltre ad esse e stato effettuato benchmark anche sull'instance check e sulla sussunzione, queste ultime solo sull'ontologia Lumb.
Le prove sono state eseguite dalla seguente macchina :

- Intel (R) Core(TM) i5
- 4 GB of RAM ddr 3
- Eclipse Helios
- SO → Microsoft Windows 7

Gli strumenti utilizzati per il calcolo delle tempistiche sono i timer interni di java , richiamndo la funzione System.CurrentTimeMillis ; essi però risultano non efficenti in quanto le tempistiche stimate in alcuni casi sono dell'ordine di qualche millisecondo.
Per ovviare a tal problematica ogni funzione è stata ripetuta N volte , preso il valore totale e diviso per N .
N nel nostro caso è stato attribuito a 100.
Utilizzando la procedura appena descritta si perde l'informazione riguardo situazioni a freddo, ovvero quando i meccanismi di caching dei dati ancora non sono attività (difatti la prima esecuzione di una qualsiasi query impiega più tempo delle sue successive), ma ciò non è stato considerato rilevante in questa fase di analisi.

3.1. Creazione della struttura dati.

Appoggiandosi sulla struttura dati sapere, in particolare l'ontologia relativa ai sensori – osservatori, è stato sviluppato un case study concreto nel quale sono stati inseriti N set di individui relazionati tra loro, in modo da dare complessità all'intero sistema.
L'architettura del modello utilizzato è la seguente :



L'architettura sopra descritta è la sola parte di ontologia utilizzata per il test, ma ne è stato caricato il modello completo.
I dati sono stati inseriti sotto forma di tuple LSA attraverso il model jena, un esmempio è il seguente :

```
//ex:ubisenseofficeA
            Resource ubisenseOfficeA=
model.createResource(newIdForTest("ubisenseOfficeA",i));
            TestUtils.IncCountIndividuals();
//          a sensing:Sensor, sapere:LSA ;
            Resource sensor=
model.getResource(sensing+"sensor");
            model.add(ubisenseOfficeA,RDF.type,sensor);
//          sensing:sensorType ex:Ubisense ;
            Property sensorTypeProp=
model.getProperty("sensorType");
```

```
      model.add(ubisenseOfficeA,sensorTypeProp,sensorType);
//            sensing:locatedAt ex:officeA .
              Property locatedAt =
model.getProperty(sensing+"locatedAt");
              Resource
officeA=model.getResource(sensing+"OfficeA");
              model.add(ubisenseOfficeA,locatedAt,officeA);
```

è stato quindi inserito un piccolo set di dati cosi composto: (notazione N3).

```
ex:Ubisense
a sensing:SensorType, sapere:LSA ;
sensing:contextDomain location:3DCoordinatePosition ;
sensing:valueType sensing:numeric ;
sensing:technicalSpecification
[ a sensor:TechnicalSpecification ;
sensing:manufaturer "Ubisense" ;
sensing:model "IP30" ;
sensing:unitOfMeasurement ucum:metres ;
] .
ex:ubisenseofficeA
a sensing:Sensor, sapere:LSA ;
sensing:sensorType ex:Ubisense ;
sensing:locatedAt ex:officeA .
ex:ubisensetagZ11dx
a sensing:Sensor, sapere:LSA ;
5
sensing:attachedTo ex:bob .
ex:obs485713
a sensing:Observation, sapere:LSA ;
```

```
sapere:updateTime "2012-04-14T11:00:00"^^xsd:dateTime ;
sensing:observedBy ex:ubisenseofficeA ;
sensing:about ex:ubisensetagZ11dx ;
sensing:value
[ a location:3DCoordinatePosition ;
location:z [ :numericalValue "3.85" ;
:measuredIn ucum:meter ] ;
location:y [ :numericalValue "41.80" ;
:measuredIn ucum:meter ] ;
location:z [ :numericalValue "1.27" ;
:measuredIn ucum:meter ] ;
sensing:confidence "0.7"^^xsd:double .
```

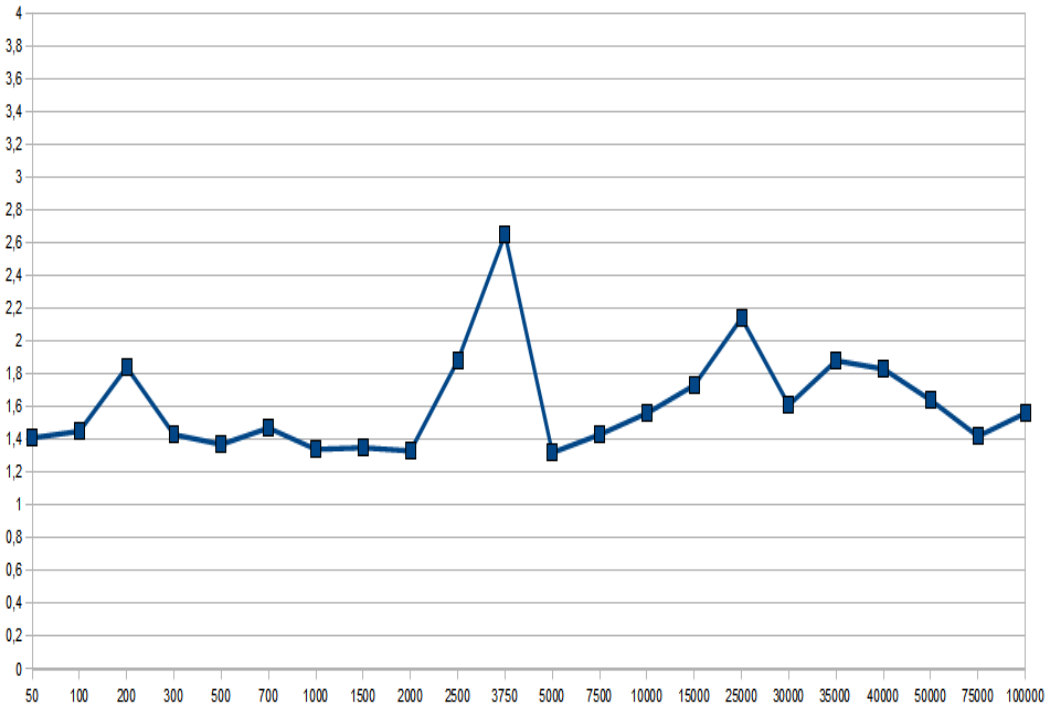
Il metodo di inserimento è stato poi inserito in un ciclo in modo tale da inserire (a multipli di 5 che sono il minimo degli individui per gestire tutte le relazioni) un numero a piacere di individui.

3.2 Benchmark Simple Query

Come primo test di performance è stata realizzata una semplice query sparql , in modo tale da essere di paragone con le altre primitive implementate.
Questo soprattutto per valutare il rapporto tra una semplice query ed il carico aggiunto da jena.
La query utilizzata è la seguente :

```
" SELECT ?x WHERE { ?x  rdf:type  <"+lsa.getURI()+">}";
```

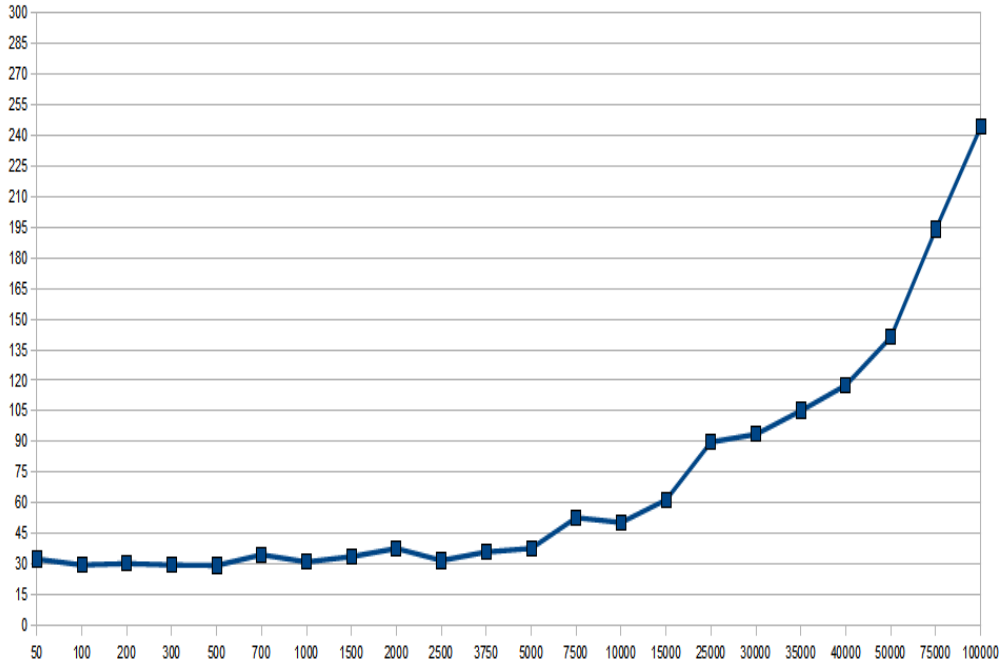
ovvero richiede tutte le LSA presenti all'interno del model .



In ordinata il tempo impiegato in millisecondi , in ascissa il numero di individui presenti all'interno dell'ontologia.
Si nota una crescità pressochè nulla , con valori inferiori o quasi ai 2 ms fino ad un limite di 100000 individui, dopo tale soglia l'efficenza della query tende a degradare in modalità meno che lineare.
Si è deciso di fermarsi a 100.000 individui come soglia massima di test, perchè rappresentante già un ottimo valore.

3.3 Benchmark Clone

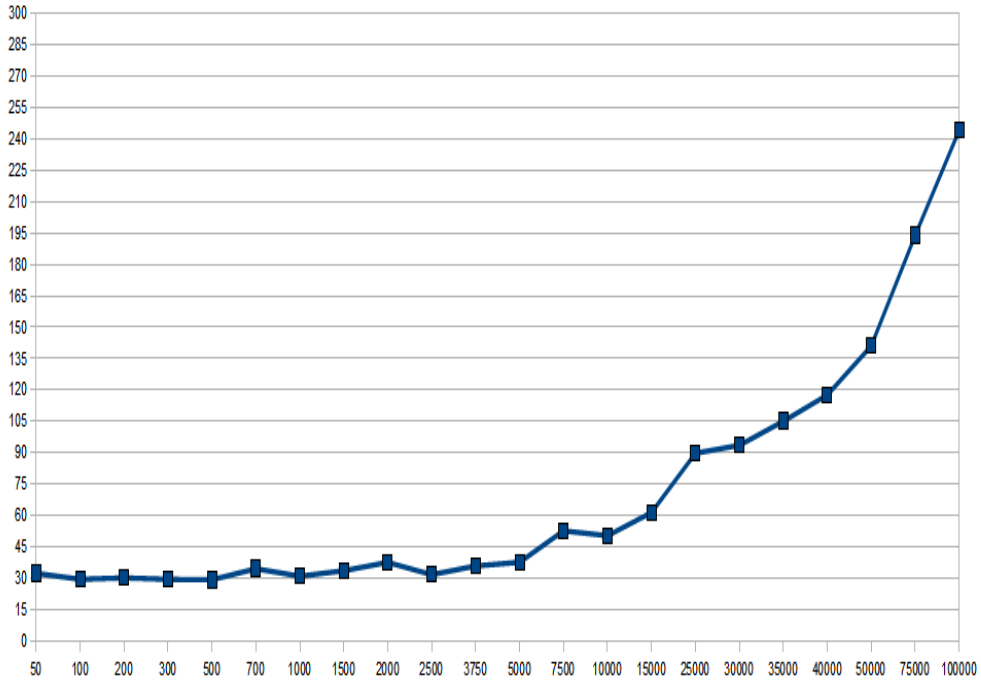
L'operazione di clone di una tupla semantica è risultata tra le più onerose come da aspettative.
Difatti il bisogno di dover percorrere ricorsivamente tutto l'albero della tupla per poi aggiungere ogni foglia nel nuovo “individuo” comporta un notevole stress per il sistema.
I risultati ottenuti, sempre in modalità grafica , sono i seguenti:



In ordinata il tempo impiegato in millisecondi , in ascissa il numero di individui presenti all'interno dell'ontologia.
Si nota una crescità pressochè nulla , con valori inferiori o quasi ai 45 ms fino ad un limite di 10000-15000 individui, dopo tale soglia l'efficenza della query tende a degradare in modalità meno che lineare.
Basti pensare che se con 10000 individui ho un tempo medio di 50 ms, se la crescità fosse lineare con 100.000 individui il tempo medio dovrebbe essere attorno ai 500ms, mentre si riscontrano poco più di 240 ms .

3.4 Benchmark CopyGraph

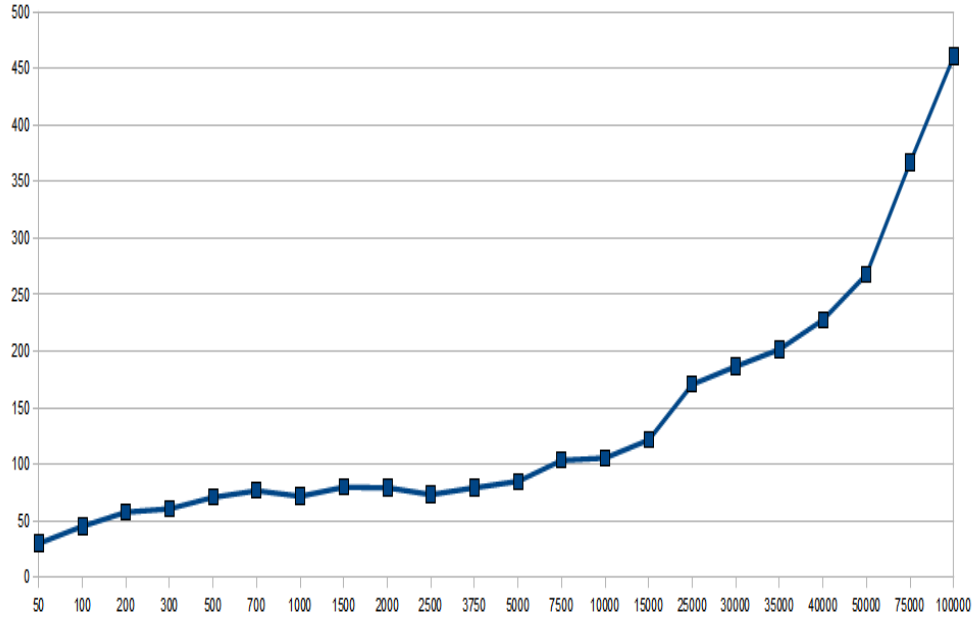
La funzione chiamata CopyGraph, la quale copia una proprietà della tupla semantica con il relativo albero connesso (se esso è presente) ,è una versione “light” della ben più articolata clone, come da aspettattive quindi l'efficenza di tale primitiva è maggiore rispetto alla sua “sorella maggiore” .



Proprio come per la clone si nota il punto di inizio degrado intorno ai 10.000 individui , con le stesse considerazioni riguardo alla crescita me no che lineare.

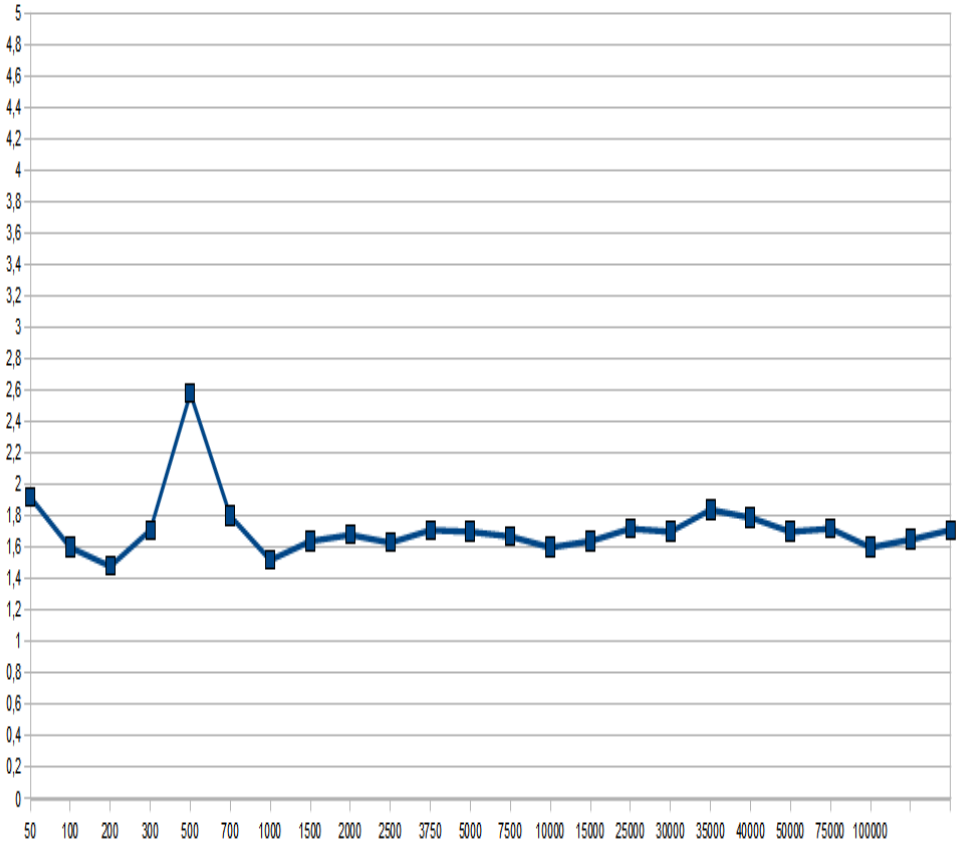
3.5 Benchmark Cascade Delete

L'operazione più pesante in assoluto è la Delete a cascata di una tupla semantica difatti , data la struttura ad albero prevista per la tupla, non basta solo eseguire una delete della risorsa ,bensì occorre eliminare con ricorsione a destra tutte le foglie dell'albero.
Tale operazione ha quindi lo stesso peso di un'operazione di clone o di match semantico inclusivo, con l'aggiunta dell'operazione di remove che di per se è tra le più onerose.



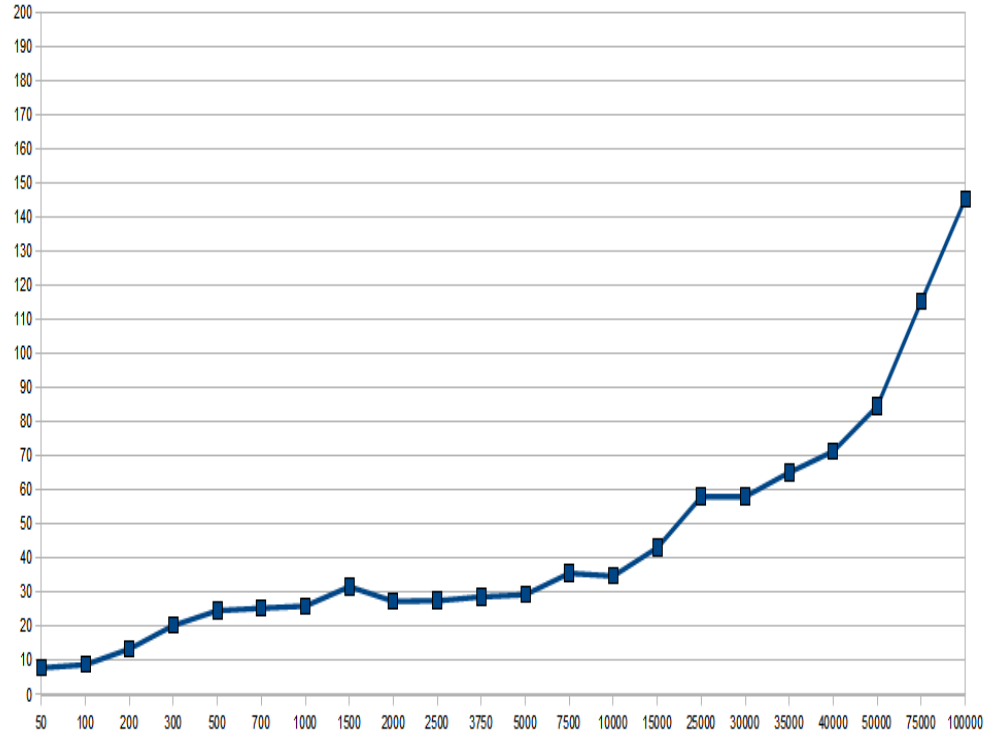
3.6 Benchmark Match Semantico

L'operazione di match semantico , inteso come appartenenza di due tuple ad una stessa classe OWL, è sicuramente agile e veloce , in quanto i dati forniti non hanno una grande specializzazione nelle classi fornite , è facile poi partendo dalla radice trovare il primo nodo comune.
Di conseguenza l'operazione è risultata tra le più veloci, le cui tempistiche sono addirittura inferiori a quelle del caso simple query.



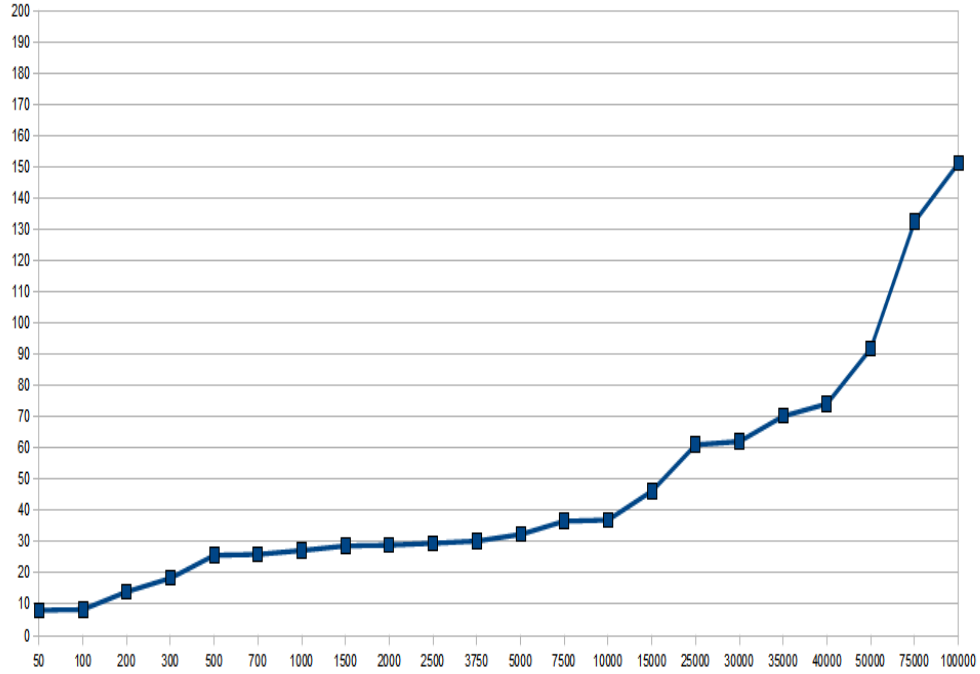
3.7 Benchmark Match Inclusivo

Il match inclusivo, data la necessità dello scorrimento dell'albero nella propria interezza , è risultata abbastanza onerosa anche se non ai livelli della clone e della delete .
Il caso peggiore ovviamente per questa primitiva è che vi sia effettivamente match tra le parti , in quanto costringe alla totale esplorazione della tupla , casi in cui non è presente match possono essere significativamente inferiori.
Il caso studiato inoltre rappresenta già uno tra i casi peggiori, in quanto il match vi è sempre e l'albero della tupla ha ben 3 livelli.



3.8 Benchmark Match Esatto.

L'operazione di match esatto , rispetto a quella del match inclusivo, aggiunge un ulteriore grado di complessità.
Il controllo dell'esatta corrispondenza fa si che si debba controllare per ogni “livello” che vi sia lo stesso numero di statements , quindi la primitiva di match esatto è , seppur di poco, meno performante di quella di match inclusivo.



4. Conclusioni

In questo lavoro è stato studiato un nuovo framework che introduce una sostanziale novità : il concetto di update di una tupla.
Spostandoci in un'ottica ove le tuple diventano strutture dati complesse ciò diventa assolutamente necessario.
Le primitive utilizzate inoltre hanno riscontrato, in termini di performance, un ottimo risultato che ci spinge a migliorare ed andare avanti con il nostro lavoro.
Difatti i risultati sono stati ottenuti dopo una prima fase di performance improving, la quale non è detto che non si possa migliorare ulteriormente .
Tramite funzioni custom inoltre è possibile aggiungere qualsivoglia tipo di primitiva , fornendo al framework massima scalabilità e flessibilità.

4.1 Sviluppi futuri

La crescità di SPARQL ha portato alla costrulzione di computed proprieties [16] o “volgarmente” dette magic proprieties.
Queste proprieties permettono di incamerare al loro interno della semantica oltre al semplice match con il sottografo della risorsa.
Quindi si potrebbe ipotizzare la costruzione di un framework ancora più potente , che utilizzi sparql nelle primitive svincolato dai limiti dell'utilizzo dei costrutti bind e filter ove, per il momento, è il solo luogo possibile per il richiamo delle primitive.

Bibliografia

[1] “The TuCSoN Coordination Model & Tecnology” A.Omicini S. Mariani

[2] “Semantic Coordination Through Programmable Tuple Spaces “ Nardini et All

[4] “Generative communication in Linda” Gelernter, D. (1985)

[5] http://clarkparsia.com/pellet

[6] http://www.w3.org/RDF/

[7] http://dl.kr.org/

[8] http://www.w3.org/DesignIssues/Notation3.html

[9] http://www.w3.org/TR/rdf-schema/

[10] http://www.w3.org/TR/owl-features/

[11] http://www.w3.org/TR/sparql11-query/

[12] "The semantic web is upon us, says Berners-Lee". Jonathan Bennett

[13] **http://www.w3.org/TeamSubmission/turtle/**

[14] **http://jena.apache.org/**

[15] http://www.sapere-project.eu/

[16] http://www.w3.org/wiki/SPARQL/Extensions/Computed_Properties