UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
FACOLTÀ DI INGEGNERIA "ENZO FERRARI"

Corso di Laurea Specialistica
in Ingegneria Informatica

Insieme

Progetto e sviluppo di un prototipo software per la gestione semantica di file

Relatore:
Prof. Sonia Bergamaschi

Correlatore:
Dott. Ing. Serena Sorrentino

Candidato:
Michele Stawowy

Anno Accademico 2009/10

Parole chiave:
semantica
ontologia
file system
python

Ringraziamenti

Desidero ringraziare la Professoressa Sonia Bergamaschi e, in particolare, l'Ingegnere Serena Sorrentino per il prezioso aiuto fornitomi durante la realizzazione di questa tesi.

I miei ringraziamenti vanno alla mia famiglia, che ha creduto e crede in me,
a mio padre che mi ha dato l'emozione della scoperta e lo spirito dell'ingegnere,
a mia madre che mi ha dato la forza di credere alle proprie idee,
a mia sorella che mi ha sostenuto e tirato i capelli quando ero piccolo,
a mia nonna che mi ha sempre circondato di affetto e di cure,
a mio nonno che mi ha mostrato la fatica del lavoro.

Non posso non ringraziare anche i miei amici, che, pur pensando che mai mi sarei laureato,
mi hanno incoraggiato e non mi hanno mai fatto mancare l'euforia del divertimento.
Prima fra tutti la mia ragazza, Daniela, che con il suo amore
mi ha seguito, rincuorato e soprattutto sopportato.
Serena, che mi è sempre stata vicina, ed il suo alter-ego Miss Lampone,
per le avventure creative vissute insieme.
Irina, per la pazienza e compagna di innumerevoli chiaccherate verso Modena,
nonostante le mancate promesse dolciarie.
Alex ed Ivan, per aver passato con me estenuanti "sedute di studio" di Unreal.
I miei amici dell'università, per aver condiviso con me questo percorso,
ed in particolare Giorgia, per avermi fatto da segretaria suo malgrado.
Daniele, il mio barman preferito e sovrano indiscusso della Canaletto.
La compagnia del Parkez, per essere stati sempre presenti quando c'era da festeggiare.
Sonia, Federica, Chiara (I Kill You), Erica (Etta) e Giulia, per le risate
ed i tantissimi momenti passati assieme in questi ultimi due anni.
I miei amici del Liceo, per avermi regalato cinque anni splendidi che mai dimenticherò.

Indice generale

Indice delle illustrazioni

Introduzione

Gli esseri umani basano la propria conoscenza ed esperienza sulla memoria degli eventi da loro vissuti: i ricordi vengono immagazzinati nel cervello e la persona è in grado di recuperarli nella loro interezza anche partendo da una parte di essi o a seguito di uno stimolo legato ad essi.

Tuttavia, fin dall'alba dei tempi l'uomo ha cercato di creare dei supporti su cui riversare i propri ricordi, mezzi in grado di custodire la conoscenza acquisita e capaci di tramandarla facilmente ad altri esseri umani, basti pensare alla portata storica che il libro ed il processo di stampa hanno avuto per il sapere umano.
Nell'ultima metà del secolo scorso si è fatta strada un'altra invenzione che sta rivoluzionando il nostro modo di vivere: il calcolatore elettronico e la sua capacità di memorizzare dati sotto forma di bit. Oggigiorno milioni di persone usano quotidianamente il computer in diversi ambiti, da quello lavorativo a quello personale. Ciò che accomuna tali operazioni è la produzione di contenuti che vengono salvati sulla memoria non volatile del computer, i cosiddetti *file*.

Oramai è possibile digitalizzare quasi tutti gli aspetti di un'esperienza vissuta (attraverso foto, video, commenti testuali, audio, messaggi), creando perciò un equivalente digitale dei ricordi che sono nel cervello. Tuttavia, le similitudini per ora si fermano qui, mentre ciò che è completamente diverso è il modo in cui i "contenuti" vengono gestiti ed usati.
I computer odierni sono basati sul cosiddetto *paradigma a cartelle* che impone l'utilizzo di una gerarchia ad albero statica per la catalogazione dei file, ed il cui recupero avviene inserendo il percorso esatto attraverso l'albero; nella mente, invece, i ricordi vengono recuperati in base al *contenuto* e non sono inseriti in una struttura fissa.
Da questa considerazione consegue che nel paradigma a cartelle si può rappresentare al più una relazione tra file, cioè raggruppare file che hanno una caratteristica comune (come ad esempio il salvare tutte le canzoni mp3 nella cartella "Musica") rinunciando a tutte le altre possibili: ciò che manca è la possibilità di legare tra loro i file con più associazioni differenti ed usare tali associazioni per generare una gerarchia dinamica o per la ricerca dei dati stessi, mantenendoli quindi isolati tra loro.
Esistono software che cercano di ovviare a questo stato di cose prefiggendosi come obiettivo quello di amministrare i contenuti ed i legami tra essi, come ad esempio i sistemi di gestione integrata dei documenti[1], o i software dedicati alla gestione della libreria musicale attraverso i cosiddetti tag delle canzoni[2], ma i contenuti rimangono comunque confinati, senza che avvenga un vero scambio di informazioni tra programmi diversi.
Vi è dunque l'esigenza di una modifica radicale a come i computer gestiscono i file, una modifica che intervenga sullo strato più basso per poter creare un ambiente di lavoro che sia "consapevole" dei contenuti e delle relazioni tra essi.

Il lavoro descritto in questa tesi si propone di rispondere a questa necessità analizzando i cosiddetti *file system semantici*, cioè, in sintesi, file system che abbandonano il paradigma a cartelle in favore di una rappresentazione dei file dinamica, basata sui legami semantici esistenti tra i contenuti.
L'obiettivo della tesi è quello di sviluppare un progetto volto alla creazione di un file system semantico installabile su un calcolatore e fruibile da un utente qualsiasi, senza particolari conoscenze in ambito informatico.

La tesi è organizzata nei seguenti capitoli:

- *Capitolo 1: Cenni teorici.* Si affrontano gli aspetti teorici necessari a comprendere il tema dei *file system semantici* ed utilizzati all'interno di questo documento
- *Capitolo 2: Lo stato dell'arte.* Si studiano i progetti esistenti che già hanno cercato di sviluppare un file system semantico, analizzando pregi e difetti delle soluzioni adottate

1 Un esempio: DocuWare (http://www.docuware.com/main.asp?sig=com_emp&lan=it&loc=it)
2 Un esempio: iTunes (http://www.apple.com/it/itunes/what-is/)

- *Capitolo 3: Il progetto Insieme*. Si descrive, dal punto di vista teorico, il progetto del file system semantico Insieme, esponendone la struttura interna ed i componenti funzionali
- *Capitolo 4: Descrizione del prototipo*. Si analizza il codice del prototipo di Insieme sviluppato in base alle specifiche esposte nel Capitolo 3
- *Capitolo 5: Esempio di utilizzo*. Viene presentato un esempio di uso pratico del prototipo, commentando le operazioni eseguite ed i risultati
- *Capitolo 7: Conclusioni*. Si riassumono gli esiti del progetto, esponendone i punti deboli e proponendo i possibili sviluppi futuri

1. Cenni teorici

Per affrontare l'argomento dei *file system semantici* è necessario introdurre alcune nozioni che verranno riprese nel corso del presente lavoro, in quanto fondamentali per la teoria e lo sviluppo del progetto.
Di seguito verranno trattati i seguenti temi:

- breve introduzione ai *file system*, in particolar modo al *Virtual File System* di Linux
- il campo di ricerca denominato *Rappresentazione della Conoscenza* e gli standard *RDF*, *RDFS* e *OWL* introdotti dal W3C
- gli aspetti teorici propri dei *file system semantici*

1.1 File system

Banalmente, un *file system* [1] è quella parte del sistema operativo incaricata di salvare e recuperare i file dell'utente da un generico dispositivo di archiviazione.

Entrando nel dettaglio, un qualsiasi *file system* ha innanzitutto il compito di gestire un dispositivo di archiviazione, tenendo traccia dei file salvati, amministrando lo spazio a disposizione per evitare perdite o sovrapposizioni di dati e rendendo trasparente all'utente tutte queste operazioni.
Un dispositivo di archiviazione è generalmente un supporto fisico quale hard disk, cd-rom, chiavetta usb, etc. dotato di un array di blocchi di memorizzazione di dimensione fissa (chiamati *settori*) che il file system utilizza per la scrittura dei file. Esistono anche file system che in realtà non utilizzano un supporto fisico, come ad esempio *NFS* (Network File System) che permette di vedere i dati presenti in un host remoto come se fossero locali: *NFS* quindi si appoggia ad un file system esistente per recuperare i file, preoccupandosi invece della comunicazione attraverso la rete e l'interfaccia locale.

Un *file system*, per tenere traccia dei file salvati, utilizza generalmente o una *tabella di allocazione* dei file (File Allocation Table o FAT) che è alla base dell'omonimo file system *FAT* sviluppato da Microsoft [2], oppure sfrutta gli *inode*, tipici dei file system stile Unix [3]. L'utente però non si serve di questi strumenti per accedere ai dati, ma utilizza un paradigma comune a tutti i file system più moderni e conosciuti: il *paradigma a cartelle*.
Il paradigma a cartelle prevede che ogni file sia identificato attraverso l'unione di due elementi:

1. un nome, che può essere seguito dall'estensione del file che ne descrive il tipo di contenuto
2. una locazione, cioè la posizione all'interno di un albero gerarchico (l'albero delle cartelle)

L'unione di questi due elementi crea un *percorso* che deve essere *univoco* per ogni file: non è possibile avere due file con lo stesso nome e la stessa posizione. Dato il percorso univoco, questo viene utilizzato per mappare i file presenti o nella tabella di allocazione o con gli inode.

Linux offre una soluzione particolarmente efficace per la gestione dei file system, di particolare interesse per chi vuole cimentarsi nello sviluppo di un proprio file system e soprattutto per la sperimentazione di file system semantici.
Riprendendo l'immagine presente nell'articolo di M. Tim Jones [4], si osserva come Linux non dialoghi direttamente con un determinato file system, ma passi per il cosiddetto *Virtual File System* (o *VFS*): questo componente espone delle interfacce astratte, che i vari file system riempiono con le loro specifiche funzioni.

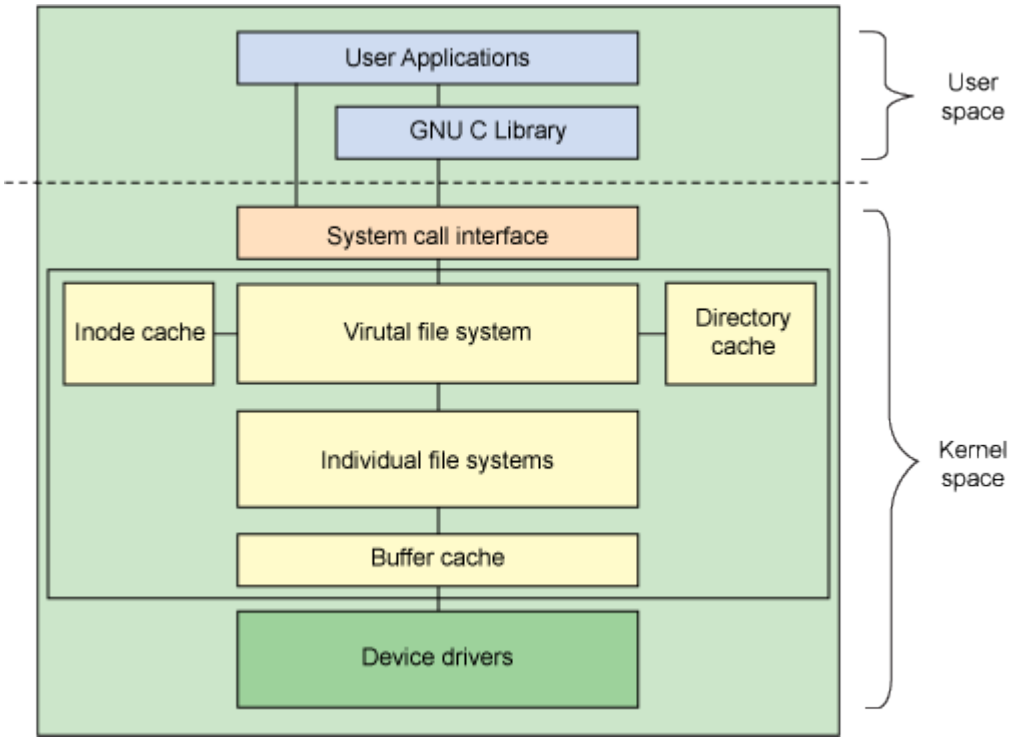

Illustrazione 1: Gestione dei file system in Linux

Linux offre anche un altro tool, *FUSE*[3] (o *Filesystem in USErspace*), che si occupa di re-indirizzare le richieste da VFS, che si trova nello spazio del Kernel, nuovamente nello spazio utente: questo permette all'utente di sviluppare ed usare un file system completo di ogni funzionalità senza necessitare dei privilegi di amministratore, altrimenti richiesti.

1.2 Rappresentazione della conoscenza

Il dizionario Zingarelli [5] definisce "semantico" come:

> *"**semàntico**: agg.*
> ***1** Che concerne il significato delle parole*
> *[...]"*

La creazione di un *file system semantico*, quindi, prevede per sua stessa definizione l'immagazzinamento di informazioni aggiuntive, *informazioni semantiche* riguardanti, appunto, il contenuto a cui si riferisce. Tali informazioni, chiamate a volte meta-dati, rappresentano la conoscenza che l'utente ha a riguardo i propri contenuti e, in quanto dati binari, possono essere utilizzate dal calcolatore come punto di partenza per una loro elaborazione: è proprio questo uno degli obiettivi della cosiddetta *Rappresentazione della Conoscenza.*

3 http://fuse.sourceforge.net/

La Rappresentazione della Conoscenza [6] è una branca dell'Intelligenza Artificiale che ha come obiettivo ultimo la formalizzazione della conoscenza umana in modo che anche i calcolatori la possano comprendere ed elaborare, generando nuova conoscenza in maniera autonoma.
Tale obiettivo è molto ambizioso e presenta numerosi problemi, tutti interconnessi tra loro. Dalla pagina di Wikipedia relativa alla Rappresentazione della Conoscenza [7] si citano i seguenti:

- qual'è la natura della conoscenza?
- come rappresentano la conoscenza le persone?
- quanto è espressivo uno schema di rappresentazione o un linguaggio formale?
- uno schema di rappresentazione dovrebbe essere generico o focalizzarsi su un particolare dominio?
- lo schema dovrebbe essere dichiarativo o procedurale?

Ai quali si aggiungono anche:

- quale linguaggio usare per rappresentare una conoscenza su un calcolatore?
- quali operazioni definire per verificare correttezza e congruenza dei contenuti?
- quali algoritmi usare per inferire nuovi contenuti?
- quali linguaggi ed algoritmi usare affinché le operazioni siano computazionalmente valide ed eseguibili?

Come si vede i problemi sono tutt'altro che banali e le soluzioni proposte sono molte, ognuna con i propri pregi ed i propri difetti. Per l'intento di questo lavoro, tuttavia, ci si sofferma solamente su due di questi quesiti, cioè su quale schema adottare per la rappresentazione della conoscenza e il linguaggio da usare per le operazioni logiche.

L'avvento del *Web Semantico* [8] ha dato un notevole impulso allo sviluppo di un modello per la rappresentazione della semantica dei contenuti delle pagine web, atto a far sì che i computer possano capire tali metadati, elaborarli e scambiarli tra loro. Questo modello è stato sviluppato (e viene tuttora aggiornato) dal *W3C*[4] e l'illustrazione [5] seguente ne rivela la struttura:

4 http://www.w3.org/
5 Tratta dalle slide di T.B. Lee (http://www.w3.org/2000/Talks/1206-xml2k-tbl/Overview.html)

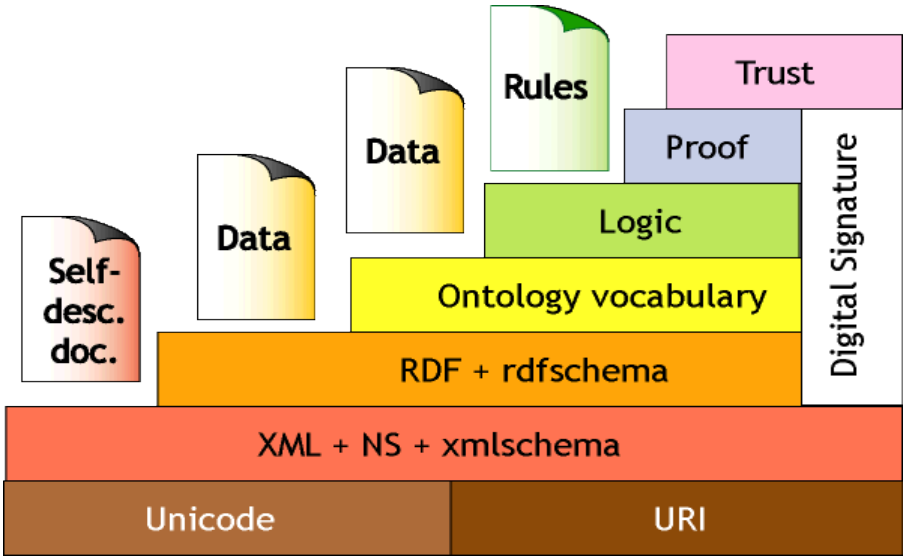

Illustrazione 2: Struttura del Web Semantico

Come si vede, la struttura pensata dal W3C poggia su *RDF* ed *RDF Schema* (RDFS) per la descrizione dei contenuti, e su *OWL* (Web Ontology Language) per creare un'ontologia vera e propria, dotata di una *logica proposizionale*.

1.2.1 Resource Description Framework

RDF, acronimo di *Resource Description Framework*, permette di "dire qualcosa" rispetto ad una risorsa qualsiasi utilizzando il costrutto detto *tripla*, cioè l'unione di 3 elementi:

soggetto - predicato - oggetto

Così, ad esempio, la frase "il prato è verde" viene rappresentato come:

soggetto: "il prato"
predicato: "è"
oggetto: "verde"

Il soggetto rappresenta la risorsa di cui si vuole annotare qualcosa, ed il predicato esprime l'aspetto che lega il soggetto all'oggetto. L'oggetto può essere a sua volta un'ulteriore risorsa (si pensi all'affermazione "Marco conosce Alex", in cui Alex è un'altra persona descritta in qualche altra tripla), oppure semplicemente un letterale, cioè una stringa o un valore numerico.

Attraverso tale costrutto, *RDF* implementa un modello di annotazione semplice ma molto potente, un modello i cui principi base sono:

- qualunque cosa può essere identificata da un *Universal Resource Identifier* (abbreviato in *URI*)
- si utilizza il linguaggio meno espressivo per definire qualunque cosa
- qualunque cosa può dire qualunque cosa su qualunque cosa

Come specificato al punto 1, l'*URI* è l'identificativo univoco di una risorsa e si presenta nella maggior parte dei casi sotto forma di un URL, solitamente così formato:

url vocabolario o ontologia + nome risorsa

Questa suddivisione permette di sostituire all'URL del vocabolario un *prefisso*, in modo da rendere più leggibile le annotazioni (ma ininfluente per il calcolatore, che comunque utilizzerà l'URI intero). Lo stesso RDF, in quanto vocabolario contenente altre risorse per la costruzione di triple, è dotato di un URI:

http://www.w3.org/1999/02/22-rdf-syntax-ns#

al quale si aggiunge il nome della risorsa specifica che si vuole usare. Ad es.:

http://www.w3.org/1999/02/22-rdf-syntax-ns#type

Se poi si utilizza *rdf* come *prefisso* per l'URL di RDF, diventa:

rdf:type

RDF mette a disposizione alcuni elementi necessari all'annotazione, di cui i due più importanti sono:

- *type*: è un predicato, usato per specificare di che tipo è la risorsa annotata. Linguisticamente corrisponde a "... è del tipo ..."
- *Property*: rappresenta la classe delle proprietà e, usata come oggetto in una affermazione in concomitanza con il predicato rdf:type, specifica che il soggetto è una proprietà, utilizzabile come predicato in altre triple

Un esempio di tripla RDF corretta può quindi essere:

<http://localhost#Mark> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .

in cui si afferma che la risorsa locale "Mark" è una persona.

L'unione delle triple, poi, può essere visto come un grafo connesso, in cui i nodi sono le risorse usate come soggetto ed oggetto, e gli archi le proprietà che li legano. Ad esempio sul sito del W3C[6] sono riportati vari esempi:

6 Immagini prese agli indirizzi http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#figure12 e http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#figure17

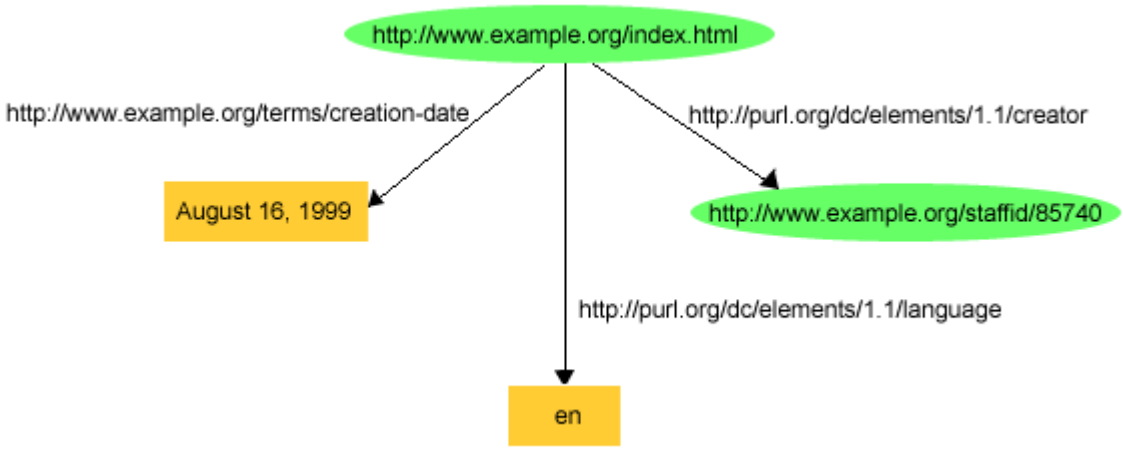

Illustrazione 3: Esempio di grafo RDF

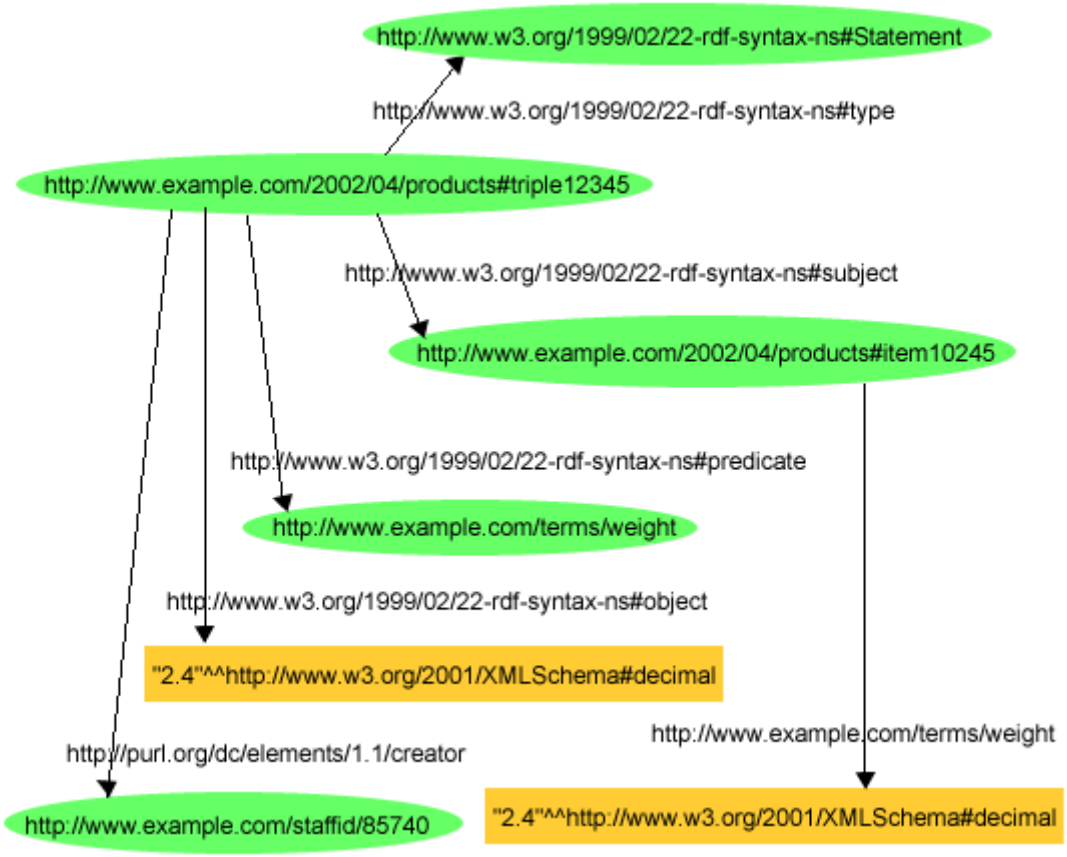

Illustrazione 4: Esempio di grafo RDF

La rappresentazione testuale, usata per lo scambio di dati tra calcolatori, prevede un metodo ufficiale scelto dal W3C, più altri implementati da terzi che hanno comunque avuto un buon successo e per questo molto utilizzati: il metodo ufficiale per la rappresentazione di un grafo RDF è l'utilizzo del linguaggio di

markup *XML*, anche se il tipo MIME[7] è "application/rdf+xml".
Il formato XML risulta però essere pesante e scarsamente leggibile da una persona, pertanto sono stati introdotti altre "serializzazioni", che vengono riportate di seguito, senza però entrare nel dettaglio di ognuna:

- *Notation3*: solitamente abbreviato in N3 e sviluppato da Tim Barnes Lee. Include anche la possibilità di definire regole basate su RDF (costrutti non presenti, però, nello modello standard RDF)
- *Turtle*: sottoinsieme di N3, non va oltre il modello definito da RDF
- *N-Triples*: sottoinsieme di Turtle
- *TRiG*
- *TRiX*
- *RDFa*

1.2.2 RDF Schema e OWL

Il passaggio da un semplice linguaggio di annotazione quale *RDF* ad una ontologia vera e propria avviene attraverso *RDFS* e *OWL*.

RDFS (RDF Schema) introduce dei concetti assenti in *RDF* e necessari per descrivere la struttura di una ontologia [9]. Tra i concetti introdotti, quelli più significativi sono:

- *rdfs:Class*: rappresenta il concetto di classe, e specifica una risorsa come classe per altre risorse
- *rdfs:subClassOf*: permette di creare delle gerarchie di classi
- *rdfs:Domain*: permette di specificare il dominio di una proprietà
- *rdfs:Range*: permette di specificare il codominio di una proprietà
- *rdfs:subPropertyOf*: permette di creare delle gerarchie di proprietà
- *rdfs:Literal*: rappresenta la classe dei valori letterali quali stringhe e numeri interi
- *rdfs:Datatype*: rappresenta la classe dei tipi di dato

ove *rdfs* è il prefisso per l'URI di RDFS, cioè *http://www.w3.org/2000/01/rdf-schema#*.

Con questi costrutti si è in grado di definire una gerarchia di risorse, come esemplificato dall'illustrazione tratta dal sito del W3C[8], che rappresenta una possibile gerarchia minima dei veicoli a quattro ruote:

7 http://www.iana.org/assignments/media-types/
8 http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#figure18

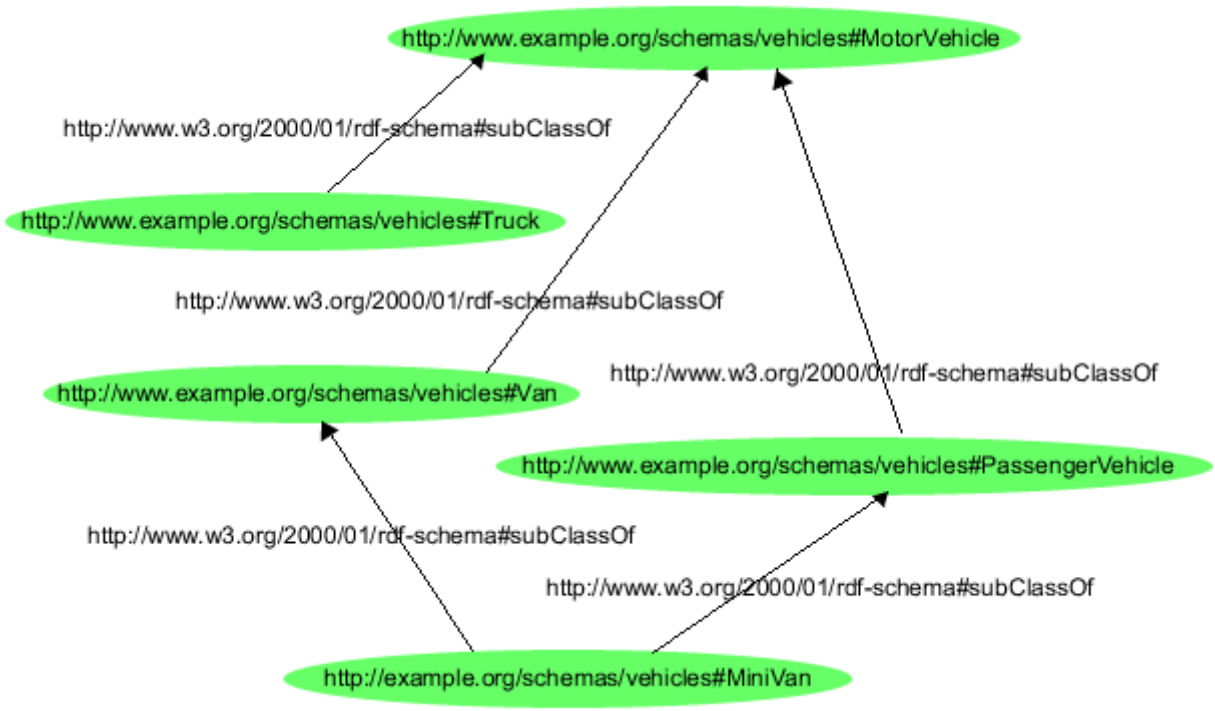

Illustrazione 5: Esempio di gerarchia RDFS

OWL (acronimo di *Web Ontology Language*), invece, apporta delle modifiche molto più sostanziali e significative, in quanto è grazie ad esso che si possono creare delle ontologie complete e funzionali. Ecco una breve definizione riportata dal sito del *W3C* [10] a proposito di *OWL2*, successore di *OWL*:

> *"The W3C OWL 2 Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL is a computational logic-based language such that knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit. OWL documents, known as ontologies, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies."*

cioè:

> *"Il Web Ontology Language (OWL) OWL2 del W3C è un linguaggio del Web Semantico progettato per rappresentare concetti estesi e complessi riguardo le cose, gruppi di cose e relazioni tra cose. OWL è un linguaggio computazionale basato sulla logica tale che la conoscenza espressa in OWL può essere elaborata da un programma sia per controllarne la consistenza, sia per rendere la conoscenza implicita, esplicita (ndr: cioè operare delle inferenze). I documenti OWL, conosciuti come ontologie, possono essere pubblicati nel World Wide Web e fare riferimento a e/o essere riferiti da altre ontologie OWL."*

L'elemento più importante che OWL introduce è quindi il *linguaggio computazionale*, in grado di generare nuova conoscenza: il W3C ha formalizzato diversi linguaggi per OWL, detti *profili* [11], che differiscono tra loro per la capacità espressiva e, di conseguenza, costi computazionali. Il profilo più espressivo è rappresentato da *OWL2 DL*, e poi ci sono 3 profili, tutti sottoinsiemi di OWL2 DL:

- *OWL2 EL*: pensato per applicazioni che usano ontologie in cui è specificato un grande numero di proprietà e/o classi
- *OWL2 QL*: pensato per applicazioni con grandi volumi di istanze e in cui il tempo di risposta alle query è critico
- *OWL2 RL*: pensato per applicazioni in cui l'aspetto del ragionamento è importante, senza però dover rinunciare all'espressività

Rimane comunque compito dello sviluppatore costruire la logica di controllo rispettando le direttive date dal W3C.

1.3 File system semantici

Esporre gli aspetti teorici specifici per i file system semantici risulta problematico, in quanto, nonostante i numerosi progetti e tentativi (vedi Capitolo 3 sullo stato dell'arte), è tuttora un campo di ricerca sperimentale, in cui non esiste una teoria consolidata ed accettata. Le cause possono essere individuate in:

- la ricerca nel campo della *Rappresentazione della Conoscenza* sta facendo passi avanti, ma non è giunta ad un punto fermo, e ciò si riflette inevitabilmente nello studio dei *file system semantici*
- gli esseri umani non hanno un unico modo di pensare, ma ognuno possiede delle sfumature derivanti dalle esperienze personali e questo crea visioni diverse rispetto agli stessi concetti: ciò rende la creazione di un modello unico molto difficoltoso
- il problema di trovare una nuova soluzione alla gestione dei contenuti, seppur sentito, non è ancora ritenuto primario e questo influisce direttamente sia sul numero di persone interessate, sia sul numero di risorse spese a tal proposito

Secondo l'autore di questo lavoro, esistono però dei punti chiave che un *file system semantico* deve soddisfare al fine di potersi definire tale:

1. innanzitutto, in quanto semantico, definire un metodo di annotazione dei contenuti, cioè dare la possibilità all'utente di correlare i propri contenuti con la conoscenza che lui possiede
2. utilizzare un metodo di annotazione della conoscenza che lasci la più ampia libertà all'utente, cioè evitare l'uso di schemi preconfezionati e fissi per l'annotazione

3. dare la possibilità all'utente di modellare contenuti virtuali, cioè contenuti che non hanno dati ma sono la rappresentazione virtuale di oggetti reali: sono esempio di questa categoria le persone, i luoghi, gli eventi, etc. Tali elementi non sono dei dati quali possono essere documenti o foto, ma sono non di meno necessari ad una corretta e più completa rappresentazione della conoscenza dell'utente
4. utilizzare un formato di annotazione di facile condivisione e conversione con altri utenti e/o calcolatori, cercando di adottare standard internazionali riconosciuti (come ad es. RDF e OWL del W3C prima descritti) e open source
5. utilizzare un formato di annotazione su cui sia possibile eseguire operazioni sia di controllo della consistenza e della correttezza, sia di ragionamento
6. gestire direttamente i supporti fisici e lo spazio disponibile, anziché appoggiarsi su file system tradizionali: oltre alla ovvia osservazione che non sarebbe un vero file system nel senso stretto della parola, si aggiunge un overhead nelle operazioni da eseguire avendo uno strato software in più inutile (quello del file system su cui si poggia)

Un punto aggiuntivo, dovuto però allo status di campo di ricerca, è l'implementazione di una logica che garantisca la compatibilità con i sistemi e, soprattutto, i programmi esistenti, che continuano a ragionare con il paradigma a cartelle.

1.3.1 Comparazione con i file system tradizionali

Ribadendo la mancanza di una teoria consolidata e prodotti software diffusi, si può comunque fare una comparazione teorica tra i *file system tradizionali* e quelli *semantici*, evidenziando i vantaggi di ognuno.

File system tradizionale:

- viene salvato solo il contenuto, senza informazioni aggiuntive, occupando quindi meno spazio di memorizzazione
- le operazioni relative ai dati risultano più veloci, in quanto non si deve tener conto di nessun metadato aggiuntivo o relazione con altri contenuti
- la gestione dei contenuti in una gerarchia è più semplice a livello di software, e quindi computazionalmente meno onerosa
- tutti i programmi esistenti sono pensati per interagire con un file system di tipo tradizionale

File system semantico:

- i contenuti, uniti ai metadati relativi, sono in diretta relazione con la conoscenza dell'utente, e non semplicemente dati isolati

- i contenuti sono gestiti con una metodologia più vicina a come opera il cervello umano
- è possibile eseguire ricerche in base al contenuto, alle relazioni che ha con altri contenuti o in base a proprietà che non sono relative al contenuto vero e proprio (si pensi all'autore di una foto, l'autore non fa parte dell'immagine)
- creare delle gerarchie dinamiche dei contenuti in base alle proprietà scelte dall'utente
- il computer può essere in grado di "capire" il contenuto, anche se ciò dipende da come è strutturata la parte semantica
- l'utente può condividere con la propria rete sociale non solo i contenuti, ma anche la conoscenza relativa ad essi

2. Stato dell'arte

Il tema dei *file system semantici* non è certamente nuovo, essendoci stati studi e progetti su tale tema fin dagli anni '90[9]. Non è neppure un tema considerato di nicchia, visto che cercando "semantic file system" su Google vengono restituiti oltre 900.000 risultati, e tra questi è possibile trovare le pagine di decine di progetti software, indice che il problema di una gestione più efficace dei contenuti è sentito da molti. Impensabile, dunque, cercare di sviluppare una soluzione software per la gestione semantica del contenuto a livello desktop senza confrontarsi e studiare il lavoro già svolto da altri.

Di seguito vengono riportati in tabella i progetti trovati in rete con alcuni dati sulla loro natura:

9 Come è possibile vedere dai progetti riportati sulla pagina http://www.objs.com/survey/OFSExt.htm

Nome	*Sito*	*Attivo*	*OS*
WinFS	http://blogs.msdn.com/b/winfs/	No	No
Spotlight	http://www.apple.com/macosx/what-is-macosx/spotlight.html	Sì	No
Cluug	http://www.cluug.com	Sì	No
DeepaMetha	http://www.deepamehta.de/	Sì	Sì
Iris	http://www.openiris.org/	No	Sì
TagFS	http://www.aifb.kit.edu/web/Inproceedings1140	No	NC
Insight	http://www.dmi.me.uk/code/insight/	No	NC
GLScube	http://www.glscube.org	No	Sì
SemFS	http://www.uni-koblenz-landau.de/koblenz/fb4/AGStaab/Research/systeme/SemFS/index_html/	No	NC
SemFS		Sì	Sì
X-Cosim	http://www.uni-koblenz-landau.de/koblenz/fb4/AGStaab/Research/ontologies/x-cosim/index_html	No	Sì
NHFS	http://rffr.de/nhfs	No	Sì
Gnowsis	http://gnowsis.opendfki.de/	No	Sì
Organise Framework	http://www.organise-fw.org/	No	Sì
DBFS	http://tech.inhelsinki.nl/dbfs/	No	Sì
Sedar	http://www.hpl.hp.com/techreports/2002/HPL-2002-199.html	No	NC
Lifestreams	http://cs-www.cs.yale.edu/homes/freeman/lifestreams.html	No	NC
SemDAV	http://www.semdav.org	No	NC
Nepomuk	http://nepomuk.semanticdesktop.org	Sì	Sì
Haystack	http://groups.csail.mit.edu/haystack/index.html	Sì	Sì
Tagsistant	http://www.tagsistant.net	Sì	Sì

Tagxfs	http://tagxfs.sourceforge.net/	No	Sì
Dhtfs	http://code.google.com/p/dhtfs/	No	Sì
Leaftag	http://www.chipx86.com/w/index.php/Leaftag	No	Sì
TransparenTag FS	http://transparentag.sourceforge.net/	No	Sì
Storage	http://people.gnome.org/~seth/storage/	No	Sì
FenFire	http://fenfire.org/	No	Sì
Tracker	http://projects.gnome.org/tracker/index.html	Sì	Sì

Legenda:

- : Progetti sviluppati da software house a scopo commerciale
- : Progetti universitari quali tesi di laurea o dottorato
- : Progetti derivanti da collaborazione tra istituti e/o aziende
- : Progetti non commerciali e/o personali
- OS : open source. Vengono considerati open source i progetti che rilasciano il codice sorgente delle soluzioni implementate
- NC : Nessun Codice, cioè non è stato trovato né codice sorgente, né un applicativo utilizzabile

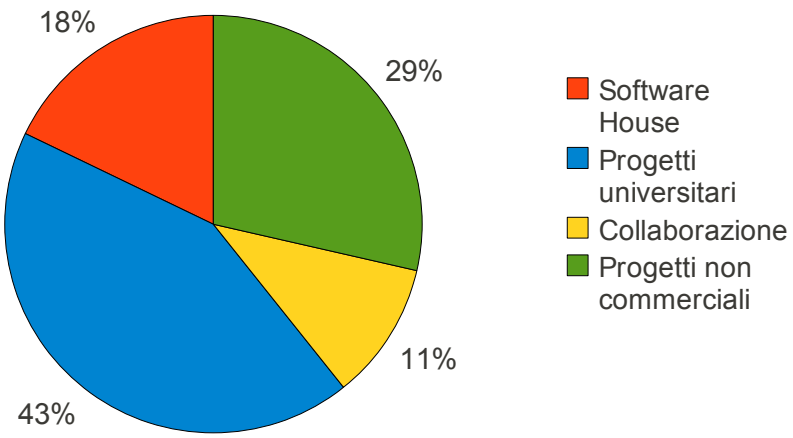

Illustrazione 6: Provenienza dei progetti

Analizzando la tabella ed osservando il grafico riportato nell'illustrazione 6, è possibile vedere che la provenienza dei progetti esistenti è in prevalenza universitaria (12 progetti su 28), seguiti da progetti a carattere non commerciale e/o personale: ciò è indice di come il problema di un nuovo paradigma di gestione

dei contenuti non sia ancora visto come primario da parte delle grandi aziende. Ciò può essere dovuto non tanto ad una scarsa considerazione o perché ritenuto superfluo, ma perché risulta essere un problema complesso che richiede molte risorse al fine di avere un prodotto completo. Emblematico è il caso di Microsoft e del suo WinFS, presentato come il prossimo file system di casa Redmond ed arrivato addirittura alla seconda versione beta per poi essere abbandonato in favore di altri progetti.

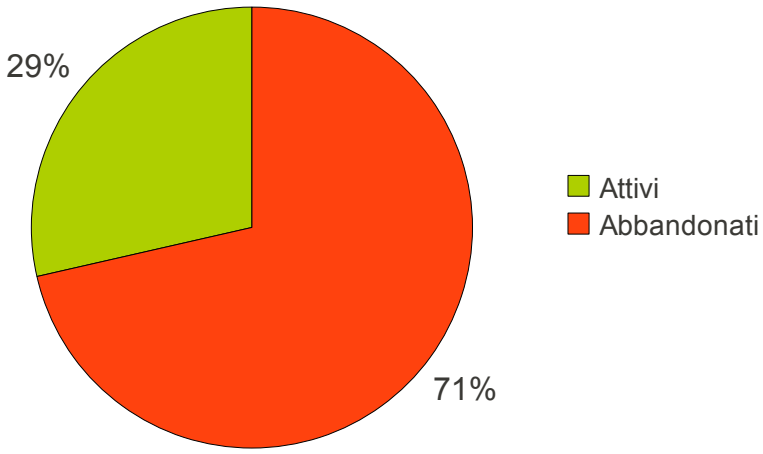

Illustrazione 7: Progetti ancora attivi

La considerazione precedente svela quindi la mancanza di uno degli attori principali in termini di capacità di ricerca e sviluppo e soprattutto di risorse spendibili sul problema: a parte alcuni casi, come ad esempio Nepomuk che è stato finanziato dall'Unione Europea con 11 milioni di €, tutti gli altri progetti sono progetti individuali sviluppati senza risorse esterne o finanziamenti. Se uniamo tale dato alla caratteristica, purtroppo ricorrente, che la maggioranza di questi progetti risulta al giorno d'oggi abbandonata e non più sviluppata (illustrazione 7), possiamo dire a ragion veduta come la ricerca di una soluzione efficace richieda risorse e tempistiche non affrontabili da un singolo individuo.

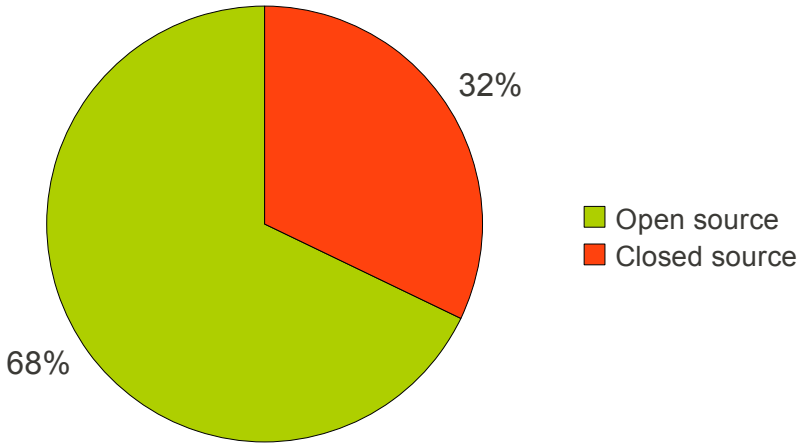

Illustrazione 8: Progetti open source

Una nota positiva è data dal fatto che la maggior parte dei progetti software analizzati è open source (ben 19 progetti su 28, illustrazione 8), e questo permette, a chi volesse sviluppare un proprio programma, di studiare a fondo il codice implementato e scoprire così quali sono le soluzioni trovate ed i loro punti di forza. Ovviamente le scelte adottate da ogni progetto sono diverse, alcune molto simili tra loro e altre con differenze più marcate, ma è possibile comunque delineare degli aspetti comuni:

- utilizzo di un database, non necessariamente relazionale, per la gestione dei metadati
- tendenza ad usare semplici tag anziché ontologie complete
- sviluppo in ambiente Linux attraverso FUSE per la creazione di un file system virtuale
- utilizzo di un file system normale per il salvataggio dei file veri e propri
- ricerca della compatibilità con i sistemi e programmi esistenti attraverso la riscrittura delle funzioni di apertura di un file e della sua ricerca (finestre di dialogo "Apri file" e file manager)

Di seguito vengono analizzati in dettaglio alcuni dei molti progetti, quelli ritenuti più significativi: per ogni progetto viene data una breve descrizione, i pro e i contro delle soluzioni adottate e, anche se non per tutti, la citazione di una frase presente nella documentazione ufficiale ritenuta interessante e di particolare valore.

A conclusione, si può affermare che al giorno d'oggi una risposta concreta e fruibile dall'utente comune non è stata trovata, in quanto nessuno dei progetti studiati ha trovato spazio nei sistemi operativi più diffusi, e rimangono software di nicchia, adottati da una ristrettissima cerchia di persone. La ricerca, dunque, continua.

2.1 TagFS

TagFS si presenta come un filesystem virtuale scritto in Java ed esplorabile dall'utente attraverso un'interfaccia WebDAV[10]. TagFS, come suggerisce il nome, organizza i file in base a delle tag assegnate a questi ultimi: ogni file può avere un numero indefinito di tag, le quali vengono viste come insiemi di oggetti (i file appunto). Le operazioni di ricerca vengono eseguite digitando le tag a cui l'utente è interessato e restituendo l'insieme composto dall'intersezione degli insiemi delle tag specificate.

Osservazioni

Pro

- È compatibile con i file system ed i programmi esistenti
- Intuitivo e facile nell'utilizzo
- Permette ricerche abbastanza dinamiche
- Le tag sono salvate sfruttando RDF

Contro

- È legato al concetto di file
- Le tag non rappresentano un'ontologia strutturata
- Le tag hanno significato solo per l'utente, non per il computer

A pagina 4 di [12], si legge:

> *"A study on personal file management states a strong preference of users to guess file locations and browse for a file in a list rather than using keyword search tools. Often they find their files in the first or a few tries."*

cioè:

> *"Uno studio sulla gestione personale dei file afferma la marcata preferenza degli utenti ad indovinare il luogo in cui è salvato un file e cercare un file in una lista piuttosto che utilizzare degli strumenti di ricerca di parole chiave. Spesso trovano i loro file ai primi tentativi."*

Questa osservazione sembrerebbe affermare la preferenza categorica dell'utente alla gestione tradizionale dei file, ma si potrebbe intendere come una preferenza dettata in realtà dalla mancanza di strumenti adatti alla ricerca semantica dei file. Il paradigma delle cartelle è conosciuto e di facile apprendimento: è necessario quindi che gli strumenti di ricerca semantica si dotino di un'interfaccia grafica ottimizzata e di semplice utilizzo, in grado di poter competere in quanto a velocità con il paradigma a cartelle. È la velocità di utilizzo che interessa all'utente, quindi è preferibile uno strumento di ricerca che ritorni più risultati di quelli richiesti ma

10 http://en.wikipedia.org/wiki/WebDAV

velocemente, che uno molto preciso con la ricerca ma lento.
Altro dato che si può estrarre dalla frase sopra citata, è la necessità di automatizzare il più possibile l'estrazione di dati semantici dai file, richiedendo all'utente un intervento minimo, solo al fine di completare le informazioni.

2.2 Storage

Storage è una soluzione per il desktop envirorment Gnome che salva tutte le informazioni su un database SQL, anche se non è chiaro dalla documentazione come tali informazioni vengano ricevute dal sistema. Permette all'utente di fare delle ricerche utilizzando il linguaggio naturale, come ad es. "tutti i film girati da Spielberg."

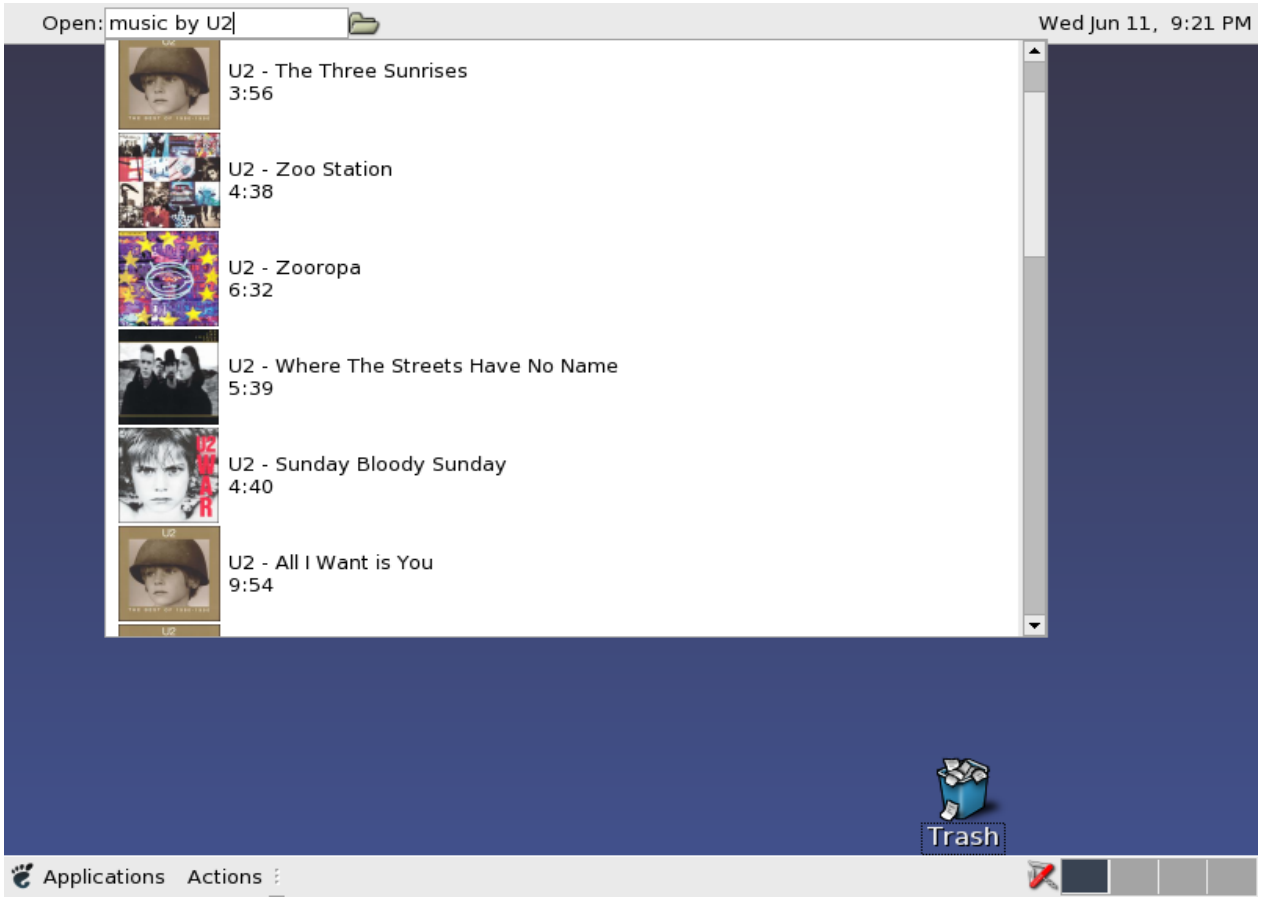

Illustrazione 9: Schermata di Storage

Osservazioni

Pro

- Uso di linguaggio naturale per le ricerche

Contro

- Non utilizza nessuno standard quale RDF o OWL
- Nessun aspetto di condivisione dell'informazione con i propri conoscenti

A pagina 6 di [13], si legge:

> *"The "organize my files" goal never takes precedence over the more highly valued immediate goals (usually) involved in using a computer: to get something done. This in spite of the fact that organizing files frequently would probably improve overall production.*
>
> *Consequently, unmanagable numbers of documents accrue in people's primary computer work space."*

cioè:

> *"L'obiettivo "organizzare i miei file" non prende mai la precedenza sul più immediato e prezioso obiettivo che si ha nell'uso del computer: portare a termine qualcosa. Questo nonostante organizzare i file frequentemente potrebbe probabilmente aumentare la produttività globale.*
>
> *Conseguentemente, un numero ingestibile di documenti si accumula nello spazio di lavoro delle persone."*

Ciò indica la necessità di adottare misure e strategie che inducano naturalmente l'utente a catalogare al meglio i propri file, come ad es. imporre all'utente l'inserimento di almeno 3 termini descrittivi del contenuto quando viene salvato un file.

2.3 DBFS

DBFS [14] si presenta come uno strato aggiuntivo al file system normale che si occupa della gestione dei file personali dell'utente (lasciando perciò fuori tutti i file di sistema e dei programmi). DBFS maschera il livello dei file sostituendosi alle consuete finestre di salvataggio e apertura di un file, ed implementando un proprio file manager: con questi strumenti permette all'utente di catalogare i propri files attaccandogli delle keywords (concettualmente equivalenti alle tag) e ricercarli in basi a tali keywords.

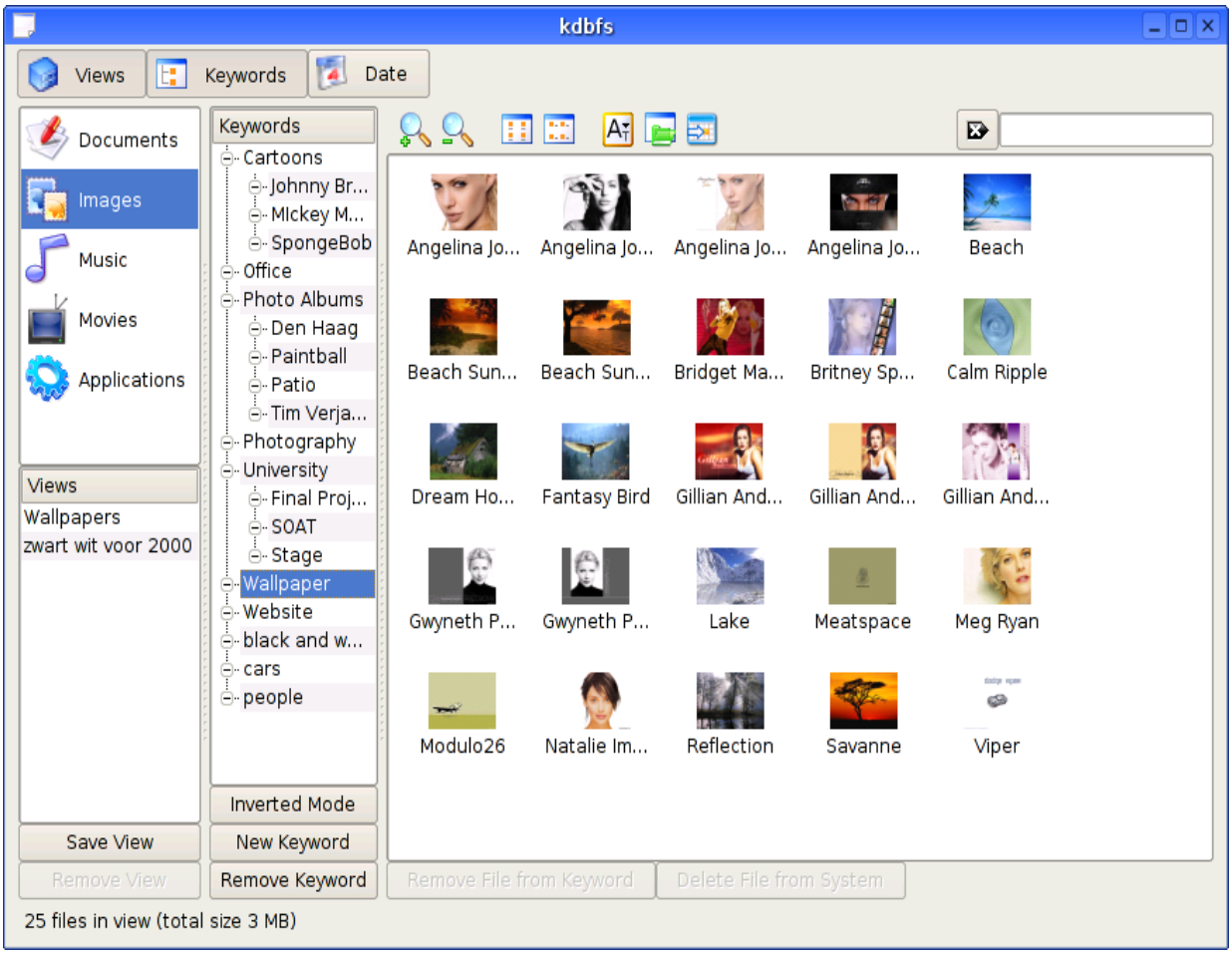

Illustrazione 10: Schermata di DBFS

Osservazioni

Pro

- Si integra nel sistema, nascondendo all'utente la gestione dei file

Contro

- Non utilizza nessuno standard quale RDF o OWL
- Nessun aspetto di condivisione dell'informazione con i propri conoscenti

Gorter, l'autore di DBFS, ha eseguito alcuni test di usabilità di DBFS rispetto ai file system tradizionali, raccogliendo dati ed impressioni utili, dai quali si può partire per sviluppare uno strumento più efficace nell'interazione con l'utente.

2.4 DeepaMehta

DeepaMehta si propone come una piattaforma software per la gestione a 360° della conoscenza prodotta da un utente sfruttando la rappresentazione grafica delle mappe mentali, cioè un grafo i cui nodi possono avere collegamenti con altri nodi. L'ambiente di lavoro è unico, cioè la modellazione del grafo, modifica dei contenuti e ricerca avvengono nel medesimo programma.

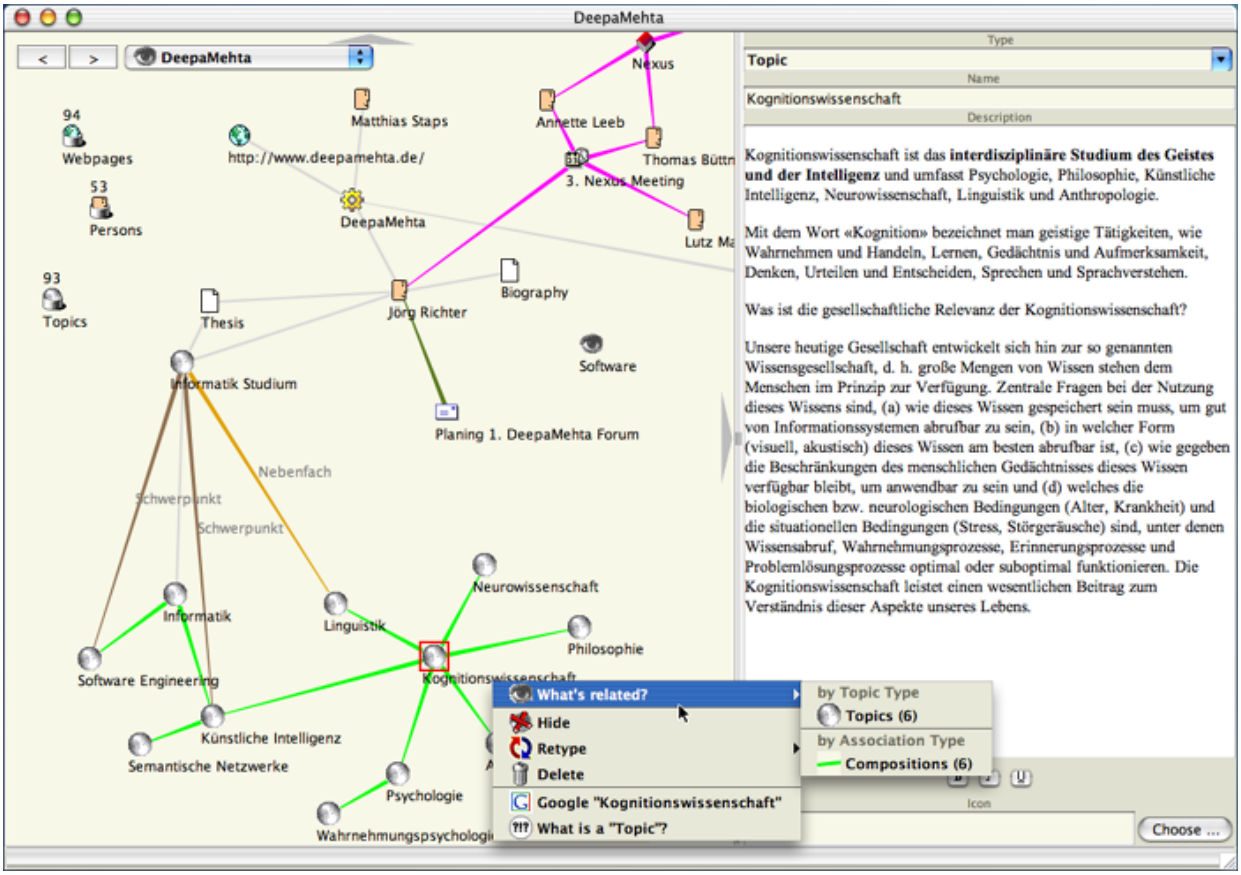

Illustrazione 11: Schermata di DeepaMehta

Osservazioni

Pro

- Cerca di slegarsi il più possibile dalla gestione tradizionale dei file
- Per la modellazione dei dati segue alcuni standard ISO

Contro

- L'interfaccia risulta poco intuitiva e di difficile utilizzo
- Il voler concentrare tutte le funzioni, tra cui la modifica del contenuto stesso dei nodi, in un unico posto risulta perdente
- L'utente è costretto ad usare le classi definite dal programma, non può creare una propria gerarchia

In [15] si legge:

> *"1) Missing Relations*
>
> *The meaningful relations between information resources are not represented in the computer. Thus, they neither can be displayed on-screen nor be exploited for navigation. There is no relation between e.g. the author of a Word document and its entry in the address book application.*
>
> *(...)*
>
> *2) Missing Work Context*
>
> *The greatest problem which makes using a computer so cumbersome is that the user's work context is not represented on-screen. (...) The user has to switch between several applications on a regular basis. Every switch changes the display as well the usage rules abruptly. What is displayed in one moment has vanished (resp. is obscured) in the next, the usage rules that are applicable in one moment are not so in the next."*

cioè:

> *"1) Relazioni mancanti*
>
> *Le relazioni significative tra le risorse informative non sono rappresentate nei computer. Perciò non possono essere visualizzate sullo schermo, né essere usate per la navigazione. Non c'è nessuna relazione ad es. tra l'autore di un documento Word e la sua scheda nell'applicazione che gestisce i contatti.*
>
> *(...)*
>
> *2) Contesto lavorativo mancante*
>
> *Il più grosso problema che rende i computer così scomodi è che il contesto di lavoro dell'utente non viene rappresentato sullo schermo. (...) L'utente deve passare attraverso varie applicazioni regolarmente. Ogni passaggio cambia bruscamente sia il contesto visivo che le regole di utilizzo. Quello che viene visualizzato in un dato momento svanisce in quello dopo, le regole di utilizzo applicabili in un dato momento non sono valide in quello dopo."*

I due punti riportati colgono due problemi fondamentali, di cui solo il primo riguarda i file system semantici: la mancanza di relazioni tra risorse è proprio ciò a cui un file system semantico cerca di sopperire.
Il secondo punto si inserisce in un discorso ancora più ampio, in quanto richiede una riscrittura del modo in cui i programmi interagiscono con il sistema operativo e anche tra di loro: affinché si possa creare un ambiente virtuale cosciente del contesto di lavoro è necessario che il sistema operativo sottostante offra ai programmi degli strumenti per lo scambio di informazioni e metadati.

2.5 NHFS

NHFS [16] si presenta come un file system virtuale utilizzabile dall'utente come un normale file system per le usuali operazioni, tra cui aprire un file salvato in NHFS con il relativo programma per la modifica del contenuto. La peculiarità di NHFS risiede nel fatto che un file può risiedere in un numero non precisato di cartelle, eliminando quindi la verticalità gerarchica imposta dai file system tradizionali. I dati riguardanti i file e le cartelle vengono salvati con SQLite, mentre i file veri e propri risiedono su una partizione normale a cui l'utente ha accesso.

Osservazioni

Pro

- Mantiene la compatibilità con i sistemi ed i programmi esistenti.

Contro

- Si basa comunque sul concetto di cartelle e file
- Non utilizza nessuno standard quale RDF o OWL
- Nessun aspetto di condivisione dell'informazione con la propria rete sociale

2.6 Insight

Concettualmente simile a TagFS, Insight [17] permette all'utente di classificare i propri file utilizzando delle parole chiave, inserendoli quindi in insiemi logici. Mantiene la compatibilità con i sistemi attuali costruendo un file system virtuale in cui il percorso al file viene inteso come unione degli insiemi identificati dalle keyword e permettendo, attraverso l'uso di caratteri speciali, delle query più complesse in grado di restituire delle ricerche più raffinate. Il salvataggio dei file e delle keyword sfrutta la struttura degli inode presenti nei sistemi operativi Unix.

Osservazioni

Pro

- Mantiene la compatibilità con i sistemi ed i programmi esistenti.

Contro

- Si basa comunque sul concetto di cartelle e file
- Non utilizza nessuno standard quale RDF o OWL
- Nessun aspetto di condivisione dell'informazione con la propria rete sociale

2.7 Sedar

Sedar è, usando la definizione dei suoi creatori, un sistema di archiviazione semantico che offre interessanti strumenti per la gestione dei contenuti. Per mantenere la compatibilità con i sistemi ed i programmi esistenti, viene sviluppato per essere visto dal sistema operativo come un NFS (Network File System). I contenuti vengono salvati nel *Sedar distributed storage*, cioè un sistema di memorizzazione P2P (Peer to Peer) in grado di scalare orizzontalmente. I metadati vengono estratti automaticamente dai file attraverso un sistema di analizzatori dedicati, in cui ogni analizzatore si dedica esclusivamente ad un certo tipo di contenuto: ogni analizzatore genera un cosiddetto Vettore Semantico (o Semantic Vector, abbreviato SV). Gli SV dei contenuti vengono usati per memorizzare gli stessi e trovare contenuti affini, usati per calcolare la "distanza semantica" tra essi. Tale distanza viene poi usata per la gestione delle versioni dei file, in quanto Sedar tiene in memoria le varie modifiche, implementando un cosiddetto Versioning System.

Osservazioni

Pro

- Uso di analizzatori specifici per vari tipi di contenuto
- Analisi ed estrazione automatica di caratteristiche semantiche da un contenuto

Contro

- Non utilizza nessuno standard quale RDF o OWL
- Nessun aspetto di condivisione dell'informazione con la propria rete sociale
- Non sembra possibile che l'utente aggiunga dati semantici propri

A pagina 6 di [18], si legge:

> *"Human brains remember objects based on their contents or features. When you run into a friend in elementary school, you may not remember her name, but you can recognize her by features like the round face and shiny smile. We call these features semantics."*

cioè:

> *"Il cervello umano ricorda gli oggetti basandosi sui loro contenuti o caratteristiche. Quando incontri un'amica delle scuole elementari, potresti non ricordarti il suo nome, ma sei in grado di riconoscerla da caratteristiche quali la faccia rotonda e il sorriso smagliante. Noi chiamiamo queste caratteristiche semantica."*

Nella frase riportata vi sono osservazioni simili, se non uguali, a quelle espresse nell'introduzione di questo lavoro: è chiaro che il paradigma a cartelle è orientato

a come lavora internamente il computer, non a come lavora la mente umana. Elaborare un metodo di gestione dei contenuti che si avvicini il più possibile a come il cervello ragiona è comunque un compito arduo, visto che, mentre tutti i computer del mondo (con le dovute differenze) seguono le stesse regole logiche, lo stesso non si può dire degli umani e del loro modo di pensare: è necessario quindi che le soluzioni adottate siano il più generiche possibili, evitando di imporre all'utente finale un metodo a lui scomodo, ma aiutandolo a creare il proprio.

2.8 Win FS

WinFS (Windows Future Storage) era un progetto targato Microsoft per la realizzazione di un file system basato su un database relazionale. La peculiarità di WinFS [19] stava nel fatto che i file potevano avere degli schemi XML che ne descrivevano i campi salienti (ad es. la proprietà "nome" e "telefono" per un file di tipo "contatto") e che permettevano la correlazione tra file diversi e ricerche in base ai contenuti. Inoltre WinFS esponeva gli schemi dei file alle applicazioni che ne avessero richiesto i meta-dati, creando la possibilità di uno scambio di informazioni che prima non era possibile.

Illustrazione 12: Schermata di WinFS

Osservazioni

Pro

- Permette a terzi di sviluppare schemi specifici per un tipo di file

Contro

- Non utilizza nessuno standard quale RDF o OWL
- Nessun aspetto di condivisione dell'informazione con la propria rete sociale

2.9 Nepomuk

NEPOMUK (Networked Environment for Personalized, Ontology-based Management of Unified Knowledge) [20] è un ambizioso progetto sviluppato da un consorzio di università ed aziende e finanziato in larga parte dall'Unione Europea. Obiettivo del progetto è la realizzazione di un framework composto da metodi, strutture dati e tool software atto a trasformare il desktop in un desktop socio-semantico: tale obiettivo viene raggiunto dando la possibilità di annotare semanticamente qualsiasi risorsa attraverso l'utilizzo del NEPOMUK Representational Language (NRL). NRL è un linguaggio basato su RDFS, ma a differenza di questi ultimi presuppone che la conoscenza in possesso si riferisca ad un mondo chiuso. I metadati creati vengono salvati in un cosiddetto RDF-store.

Osservazioni

Pro

- Permette l'annotazione di una qualsiasi risorsa
- Prevede lo scambio di conoscenza tra utenti diversi

Contro

- Soluzioni sviluppate non troppo intuitive a livello di interfaccia grafica

Nepomuk si presenta come il progetto più vasto e completo dal punto di vista teorico: leggendo l'abbondante documentazione, è possibile vedere come gli sforzi profusi siano stati indirizzati alla costruzione di un framework il più generale possibile, basato su standard approvati quali RDF e OWL e che tiene conto anche dell'aspetto sociale e di scambio di informazioni tra persone della stessa rete sociale.

3. Il progetto Insieme

Il nome del software che si intende creare è *Insieme*, che si propone come un *file system semantico e distribuito.*

In estrema sintesi, gli obiettivi che si vogliono raggiungere con *Insieme* sono:

1. capacità di gestire i contenuti dell'utente e la conoscenza relativa ad essi
2. capacità di controllo sulla correttezza e consistenza dei dati semantici inseriti e le relazioni tra essi
3. struttura orientata ad aiutare l'utente nella generazione di contenuti semantici
4. capacità di importare ed utilizzare ontologie esterne (come ad es. *FOAF*[11])
5. dare una visione unica dei contenuti dell'utente anche se questi sono sparsi su più dispositivi quali lettori mp3, chiavette usb, altri computer, etc.
6. gestione della rete sociale dell'utente ed interazione con essa per lo scambio di contenuti
7. struttura in grado di utilizzare servizi internet per il recupero di informazioni e contenuti
8. utilizzo più ampio possibile di strumenti e protocolli standard ed open-source

11 http://www.foaf-project.org/

Gli obiettivi, come si vede, sono ambiziosi ed impegnativi, e coprono aspetti distanti tra loro: tutti però hanno come denominatore comune il lavorare su contenuti non più isolati come succede nei file system tradizionali, ma su contenuti dotati di metadati ed appartenenti ad una rete di legami che ne ampliano esponenzialmente il contenuto informativo.

La prima parte di questo capitolo si concentra sull'analisi del contenuto e di come si intende rappresentare la conoscenza dell'utente all'interno del progetto *Insieme*, mentre il resto del capitolo vede l'esposizione della struttura di *Insieme* e l'analisi delle parti che lo compongono.

3.1 I contenuti

Come anticipato brevemente nell'introduzione, *Insieme* è incentrato sui contenuti. Vengono di seguito descritti in dettaglio il concetto di *contenuto* all'interno di *Insieme* e, senza entrare nelle scelte tecniche fatte per l'implementazione del progetto, a come vengono rappresentati i contenuti e la conoscenza legata ad essi.

Viene considerato *contenuto* qualsiasi *dato contenente informazioni di qualche interesse per l'utente*, distinguendo però sulla loro natura. Si hanno perciò:

- *contenuti primari*: ciò che ha significato anche se considerato in un contesto isolato. Sono *contenuti primari* quelli considerati tradizionalmente file, come lo sono i documenti elettronici, le foto, le canzoni, etc.
- *contenuti semantici*: ciò che è atto ad aumentare il contenuto informativo di altri contenuti, sian primari che semantici. Perdono di significato se considerati in un contesto isolato

Ad esempio, si consideri l'annotazione "Questa foto è relativa alle vacanze del 2009": se mostriamo all'utente solamente la foto, è in grado comunque di capirne il significato ed estrarre delle informazioni (anche se non necessariamente è in grado di ricavare l'annotazione presa in esame). La foto, dunque, ha significato anche se presa in un contesto isolato. L'annotazione, d'altro canto, se data all'utente senza però mostrare la foto, perde il proprio punto di riferimento (il soggetto), lasciando l'utente stesso con una frase priva di contenuto informativo.

Dalla definizione di *contenuto* si può dire che qualsiasi cosa che possa essere digitalizzata, può divenire *contenuto*: questa considerazione, di per sé banale, è invece fondamentale in quanto il contenuto digitalizzato è a tutti gli effetti *la conoscenza del calcolatore*: se l'utente è in grado di rappresentare nella maniera più completa la propria conoscenza, allora il calcolatore diventa uno specchio abbastanza preciso di come l'utente percepisce la realtà.

3.1.1 Gerarchia minima

In *Insieme* i *contenuti* vengono modellati e dotati di proprietà standard per aiutare l'utente nella gestione degli stessi. Tale modellazione, però, non obbliga assolutamente l'utente ad uno schema predefinito, ma rimane il più generico possibile: ogni persona ha un modo di pensare e ragionare diverso, e sarebbe quindi una scelta perdente quella di obbligare tutti a conformarsi ad un modello unico. Una ragione su tutte è data dalla banale constatazione che ogni persona coltiva, nella propria vita, interessi diversi: si prenda ad esempio due persone, una appassionata di calcio, ed un'altra appassionata di vini. È presumibile che sui propri computer queste due persone abbiano dei contenuti relativi al proprio interesse (una tutte le foto, i video ed il calendario delle partite delle squadre di calcio, l'altra le schede tecniche dei vini e la mappa con le migliori cantine d'Italia) e che vorranno modellarli in base alla propria conoscenza sull'argomento. Difficilmente, allora, è possibile creare un modello unico che colga tutti i particolari in possesso dei due utenti sui rispettivi interessi.

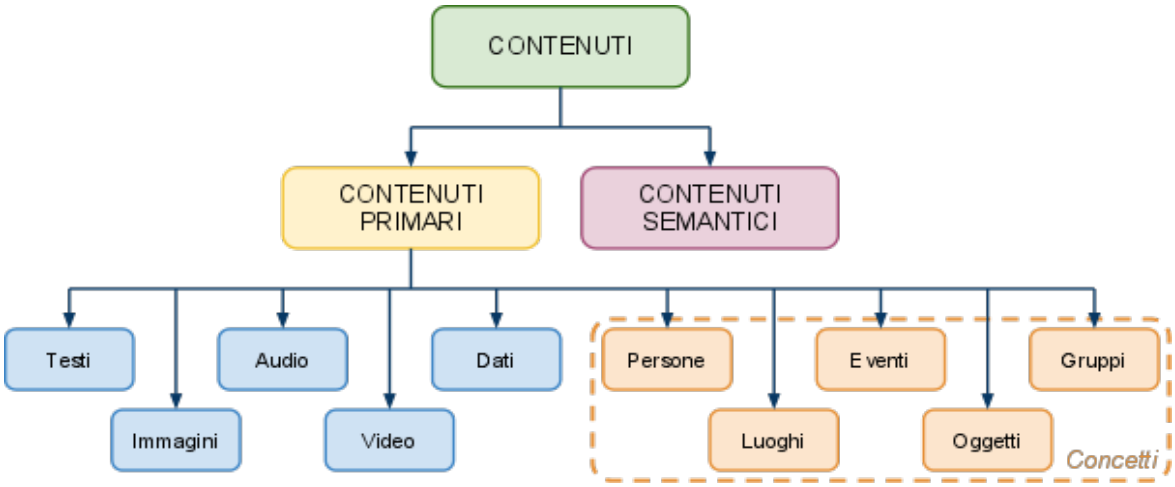

Illustrazione 13: Gerarchia minima di Insieme

Rimane comunque nell'interesse di *Insieme* aiutare l'utente nella catalogazione dei *contenuti*, e per questo cerca, là ove è possibile, di sostituirsi all'utente nella descrizione semantica degli stessi.
Innanzitutto, *Insieme* utilizza una *gerarchia minima* per i *contenuti primari*, che vengono suddivisi in 5 categorie:

- *testi*: contenuti rappresentati da stringhe di caratteri di senso compiuto appartenenti ad un linguaggio scritto
- *immagini*: contenuti rappresentati da sequenze di pixel con una codifica colore
- *audio*: contenuti rappresentati da onde sonore
- *video*: contenuti rappresentati da un'unione temporale di immagini e audio
- *dati*: contenuti rappresentati da sequenze di caratteri e/o numeri ma che, considerati da soli, non hanno senso compiuto

Queste classi servono, oltre che per avere una prima distinzione tra i contenuti e per distinguerli durante la ricerca degli stessi, anche come base per lo sviluppo di una eventuale *gerarchia* in cui viene specializzato il tipo di contenuto con ulteriori proprietà (ad esempio la classe *foto* è una specializzazione della classe *immagini* ed ha la proprietà *luogo*, non prevista invece per la classe *immagini*) oppure indicandone il modello seguito (ad esempio si può prevedere la classe *lettera* specializzazione della classe *testo* che risponde ad un modello indicante la struttura del testo stesso).

Tali classi però non bastano: come supposto in precedenza, si vuole dare all'utente la possibilità di legare i contenuti con le esperienze reali vissute dallo stesso: ad esempio, riprendendo il caso precedente, l'utente appassionato di calcio avrà bisogno di modellare entità quali i giocatori di calcio, cioè persone,mentre l'utente appassionato di vini vorrà utilizzare entità quali luoghi ed eventi delle fiere vinicole.
Si introduce pertanto un gruppo di classi, detti *concetti*, con le quali modellare quei contenuti non fisicamente salvabili sulla memoria del computer, ma dotati di significato anche in un contesto isolato, essendo perciò *contenuti primari*. Fanno parte di tale gruppo:

- *persone*
- *luoghi*
- *eventi*: sia passati che futuri
- *oggetti*: qualsiasi oggetto del mondo reale, come libri, automezzi, etc.

Un'altra classe speciale appartenente ai concetti è *gruppi*, cioè una collezione di *contenuti primari* uniti da una qualche proprietà comune, come ad esempio le foto di una festa, i contenuti relativi ad un progetto, un gruppo di amici, etc.

Si delinea quindi una gerarchia di base, sviluppabile dall'utente con ulteriori classi che reputa necessarie.

3.1.2 Proprietà di base

Un *contenuto primario* ha legato ad esso molteplici informazioni in grado di descriverlo ed identificarlo univocamente: pensando ad una foto, tali informazioni possono essere la data ed il luogo in cui è stata scattata, le persone presenti nella foto, i commenti lasciati da qualcuno riguardo la foto, etc.
Alcune informazioni sono dei semplici valori legati al contenuto (ad es. un commento è una stringa di testo), mentre altri sono a loro volta dei *contenuti primari* (ad es. quando una persona viene taggata in una foto, tale persona è un *contenuto primario* che ha tra le sue proprietà il nome, il telefono, la mail, etc.) definendo così dei collegamenti tra contenuti.

Utilizzando la gerarchia minima impostata in precedenza, ogni *contenuto primario* viene legato ad una classe, la quale può prevedere delle proprietà standard (alcune obbligatorie, altre no) per i suoi membri; suddette proprietà vengono attribuite in automatico e non richiedono l'intervento dell'utente.

Proprietà della classe *contenuto primario*, ereditate da tutte le altre classi:

- *classe del contenuto*: descrive la classe di appartenenza del contenuto
- *data di creazione*: data in cui è stato creato il contenuto
- *date di modifica*: date in cui il contenuto è stato modificato
- *proprietario del contenuto*: colui che detiene i diritti di proprietà del contenuto, non è detto che sia per forza l'utente del sistema
- *cancellato* (opzionale): indica che il contenuto è stato marcato per la cancellazione dall'utente. Serve per offrire la funzionalità del "cestino"
- *condiviso da/provenienza* (opzionale): indica chi ha condiviso con l'utente il contenuto o da dove proviene (ad es. chiavetta usb, sito web, etc.)
- *condiviso con* (opzionale): indica con chi l'utente ha condiviso il contenuto
- *genitori* (opzionale): indica quali sono i contenuti che hanno dato origine al contenuto attuale. Utile nel caso in cui un *contenuto primario* derivi direttamente da un altro (come nel caso di una duplicazione e successiva modifica) o l'autore voglia indicare da quali contenuti ha tratto origine (ad es. le foto di un collage)
- *figli* (opzionale): elenco dei contenuti che annoverano il *contenuto primario* corrente come genitore
- *appartiene a gruppo* (opzionale): elenco dei gruppi di cui il *contenuto primario* fa parte.

Proprietà della classe *testo*:

- codifica testo
- autore
- lingua del testo

Proprietà della classe *immagine*:

- codifica immagine
- dimensioni immagine
- autore

Proprietà della classe *audio*:

- codifica audio
- lunghezza temporale
- bpm (battiti per minuto)

- autore
- genere

Proprietà della classe *video*:

- codifica video
- lunghezza temporale
- autore

3.2 Struttura

Riprendendo le finalità che *Insieme* si prefigge, si analizza ora come il progetto viene strutturato.

Dato che gli obiettivi coprono aree tematiche diverse tra loro (si va dalla gestione fisica dei dati allo scambio di dati con host appartenenti alla rete sociale), risulta ottimale strutturare il progetto in macro-blocchi, ognuno con funzioni circoscritte. Ecco uno schema dei blocchi citati e le relazioni tra essi:

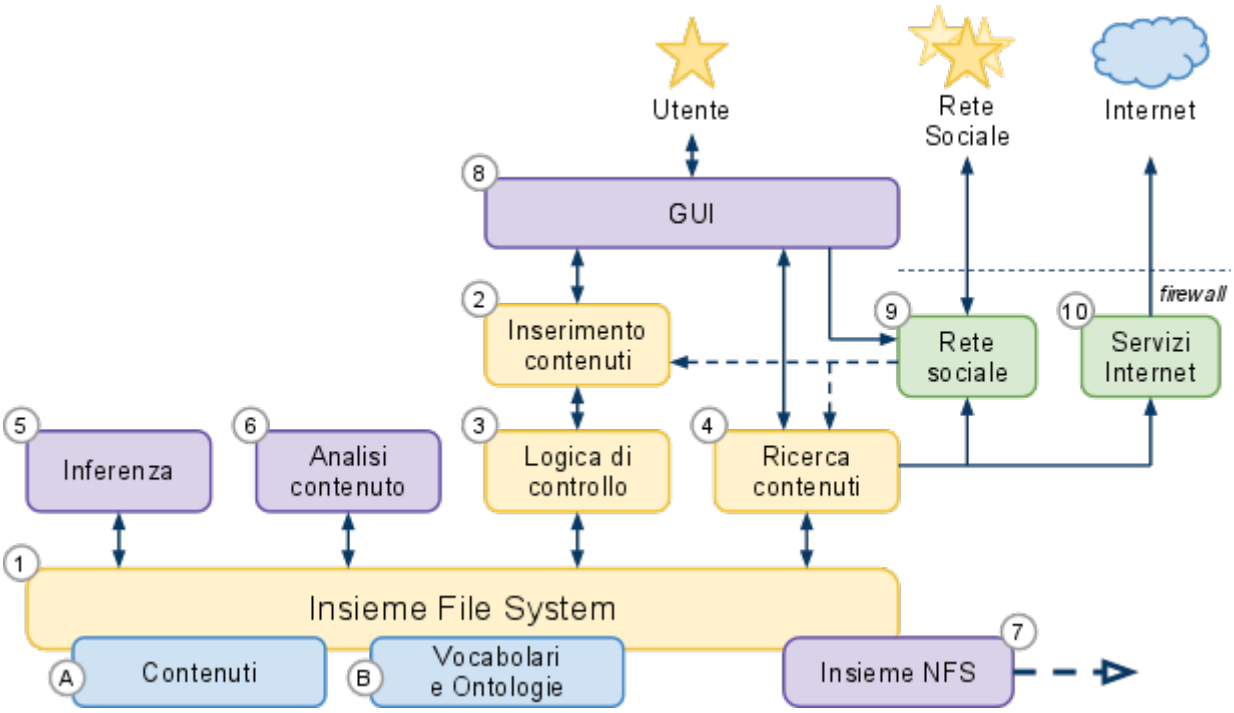

Illustrazione 14: Struttura di Insieme

con la seguente convenzione per i colori:
: blocchi necessari, rappresentano la base di Insieme
: blocchi opzionali, aggiungono funzionalità ad Insieme
: blocchi opzionali, aggiungono funzionalità orientate alla rete ad Insieme
: blocchi dati, rappresentano i contenuti salvati ed elaborati tramite Insieme

I blocchi sono pertanto:

1. Insieme File System
2. Inserimento contenuti
3. Logica di controllo
4. Ricerca contenuti
5. Inferenza (opzionale)
6. Analisi contenuto (opzionale)
7. Insieme NFS (opzionale)
8. Interfaccia grafica (GUI) (opzionale)
9. Rete sociale (opzionale)
10. Servizi Internet (opzionale)

Si vede dallo schema che l'utente è la "fonte" primaria del contenuto, ma bisogna chiarire fin da subito che non è l'unica: *Insieme* è pensato per accogliere più generatori di contenuto per assistere l'utente, come specificato al punto 3 degli obiettivi, nella creazione di ulteriore conoscenza.
Tale funzionalità è desiderabile per vari motivi, primo fra tutti il fatto che generalmente gli utenti vogliono spendere meno tempo possibile ad inserire dati e d'altra parte avere ciò che cercano subito (come anche suggerito da Seth Nickell [13] e riportato nel capitolo sullo Stato dell'arte): bisogna quindi prevedere l'utilizzo di strumenti che siano in grado di sostituirsi all'utente nell'inserimento di contenuti semantici corretti e riducendone di conseguenza il carico di lavoro richiesto. Strumenti di questo genere sono i cosiddetti *analizzatori*, in grado di estrarre delle informazioni da un contenuto primario analizzandone i dati (ad esempio trovando il numero di persone in una foto), ed i *reasoner*, capaci di inferire nuovi contenuti semantici partendo da altri.

Altra fonte di contenuto importante è quella data dai blocchi opzionali che danno ad *Insieme* funzionalità orientate alla rete, ed in particolare il blocco 9, per la gestione ed interazione con una *rete sociale*: attraverso tale rete, come spiegato in seguito, è possibile scambiare contenuti, aumentando il patrimonio informativo.

Si riprendono di seguito alcuni dei blocchi analizzandoli più a fondo.

3.2.1 Insieme File System

È il componente base su cui poggia tutto *Insieme*, essendo preposto al salvataggio dei contenuti sul dispositivo di memorizzazione.

Come i file system tradizionali, la funzione principale di *Insieme File System* è nascondere il livello fisico di salvataggio dei dati all'utente, presentando un vista virtuale degli stessi ed offrendo un set di operazioni standard con le quali è possibile agire sui contenuti: mentre la parte di salvataggio sul dispositivo di memorizzazione è quasi identica a quella dei file system tradizionali (se non altro

per l'ovvio fatto che i dispositivi sono gli stessi), è nell'interfaccia virtuale e negli strumenti che si offrono che vi sono differenze sostanziali.

Come spiegato nel capitolo dedicato alla teoria (Capitolo 1), i *file system tradizionali* offrono una vista dei contenuti salvati che è gerarchica (il paradigma a cartelle) e statica, mentre i *file system semantici* si discostano da questa metodologia, lasciando all'utente la possibilità di visualizzare i propri contenuti secondo i criteri che preferisce. Sotto molti punti di vista, il voler dare all'utente tali libertà nella gestione dei dati è in tutto e per tutto uguale a ciò che già fanno i *database management system* (sia relazionali che non), e difatti *Insieme* si propone di fondere le caratteristiche di un file system con le potenzialità offerte dai database in un unico componente; i dettagli tecnici, però, vengono lasciati a lavori successivi.

Dal punto di vista delle operazioni offerte all'utente per gestire i propri contenuti, *Insieme* mette a disposizione le seguenti:

- *Crea contenuto primario*: crea l'istanza di un nuovo contenuto primario. L'utente specifica il tipo di contenuto che vuole creare ed Insieme si occupa di salvare le informazioni di base del contenuto specificato
- *Modifica contenuto primario*: vengono modificati i dati del contenuto primario, lasciando inalterati i contenuti semantici relativi ad esso.
- *Crea contenuto semantico*: l'utente aggiunge nuovi contenuti semantici ed Insieme si preoccupa di aggiornare le relazioni o proprietà interessate
- *Modifica contenuto semantico*: viene aggiornato un contenuto semantico. A seconda della politica scelta, il modulo della logica di controllo aggiorna i contenuti semantici relativi ad esso
- *Elimina contenuto*: viene eliminato il contenuto. A seconda della politica scelta, il modulo della logica di controllo aggiorna i contenuti semantici relativi ad esso
- *Duplica contenuto*: duplica il contenuto selezionato nel caso l'utente voglia modificare una copia e mantenere l'originale. Viene salvato tra le proprietà il link al contenuto d'origine.
- *Recupera contenuto primario*: restituisce il contenuto primario specificato
- *Recupera contenuto*: restituisce il contenuto specificato e tutti i contenuti semantici che lo interessano direttamente
- *Recupera contenuti*: recupera tutti i contenuti che rispondono alle caratteristiche specificate (simile al comando che elenca i file contenuti in una cartella) assieme ai contenuti semantici che li interessano direttamente

Insieme File System, inoltre, mantiene in un'area riservata l'ontologia creata dall'utente (cioè lo schema con cui modella la propria realtà) e anche i vocabolari e le ontologie che l'utente desidera utilizzare, evidenziata nello schema generale di *Insieme* dal blocco B. Come detto in precedenza, si lascia libertà nella modellazione dei contenuti, senza l'imposizione di nessuno schema, ma questo

non vuole dire per forza che l'utente debba costruirsi la propria ontologia da zero: visto che è un procedimento che può risultare faticoso e complicato, si mettono a disposizione gli strumenti per importare vocabolari ed ontologie già fatte da altri ed utilizzarle per annotare i propri contenuti.

Un altro accorgimento adottato da Insieme è obbligare l'utente ad usare solamente il lessico disponibile nei vocabolari ed ontologie da lui salvate nel sistema: questo, oltre ad evitare errori di digitazione da parte dell'utente nell'inserimento di contenuti, permette di considerare il vocabolario globale come definito e chiuso. Con questa supposizione Insieme può "indovinare" quale termine l'utente sta inserendo anche se viene utilizzato solo il nome e non l'identificativo unico, rendendo anche la sintassi usata per l'inserimento di dati più vicina al linguaggio umano. Nel caso Insieme non riuscisse ad identificare univocamente il termine usato dall'utente, presenta una lista dei termini possibili, lasciando a quest'ultimo il compito di selezionare quello voluto.

L'inserimento dei vocabolari e delle ontologie esterne passa sempre attraverso il blocco della *logica di controllo*, per evitare conflitti con definizioni già esistenti e che andrebbero a creare inconsistenze a livello di contenuto.

Insieme NFS

È oramai consuetudine per le persone avere più dispositivi digitali (computer, netbook, cellulari, etc.) attraverso i quali esegue le operazioni più disparate:

- scrivere documenti
- navigare sul web
- fare foto e condividerle
- chattare e scambiare mail con i propri conoscenti
- etc.

Mentre fino a poco tempo fa le operazioni di elaborazione dei contenuti erano relegate al computer di casa o di lavoro, ora si hanno molteplici strumenti in grado di generare e salvare contenuto. Ciò porta alle seguenti considerazioni:

- un utente singolo ha i propri dati sparsi su più dispositivi
- spesso tali dati sono ripetuti (basti pensare alle canzoni che l'utente copia sul lettore mp3)
- all'utente manca una visione unica dei propri dati, rischiando inoltre una loro perdita accidentale
- taluni dispositivi sono in grado di generare contenuti semantici in quanto sono a diretto contatto con l'utente e riuscirebbero a monitorarne le azioni (sempre riprendendo l'esempio del lettore mp3, questo potrebbe salvare le preferenze sulle canzoni ascoltate)

Si introduce *Insieme NFS* proprio per ovviare ai problemi citati, offrendo all'utente la possibilità di "collegare" ad *Insieme File System* tutti i dispositivi che possiede, e creando dei riferimenti virtuali quando suddetti dispositivi non sono fisicamente connessi. Per l'utente ciò significa essere in grado di:

- avere una vista unica dei contenuti in suo possesso
- accedere ai contenuti quando il dispositivo è collegato e, talora non lo fosse, vedere comunque un riferimento ai contenuti non disponibili e le informazioni sul dispositivo in cui sono presenti
- sapere quali contenuti sono duplicati e dove, ma vederli comunque una sola volta nelle ricerche
- aggiornamento automatico dei contenuti ed inserimento dei contenuti semantici generati dai dispositivi nella conoscenza generale salvata sul calcolatore

Tale modulo è opzionale, in quanto non essenziale per il funzionamento base di *Insieme*; i dettagli tecnici vengono lasciati a lavori successivi.

3.2.2 Logica di controllo

È un modulo di vitale importanza, in quanto è quello che assicura la consistenza e la coerenza logica dei contenuti salvati con *Insieme File System*, in special modo quelli semantici: questo è una logica conseguenza del fatto che, mentre nei *file system tradizionali* i contenuti sono isolati, nei *file system semantici* si cerca di salvare anche la conoscenza dell'utente rispetto a tali contenuti. Ciò crea una dipendenza fortissima tra i *contenuti primari* ed i *contenuti semantici* ad essi relativi e, attraverso legami tra *contenuti semantici*, anche con gli altri *contenuti*: i *contenuti semantici*, infatti, esprimono dei concetti rispetto ad uno specifico *contenuto* (sia esso primario che semantico), e se tale contenuto viene modificato o rimosso, il concetto espresso perde la sua validità, e con lui tutti i contenuti collegati.

Il problema della consistenza e coerenza si presenta quando viene svolta una delle seguenti operazioni:

- inserimento di nuovi contenuti da parte dell'utente o da fonti esterne ad *Insieme* (come ad esempio la rete sociale)
- modifica di un contenuto primario
- modifica di un contenuto semantico
- rimozione di un contenuto

Mentre l'inserimento di un nuovo contenuto (soprattutto semantico) è abbastanza facile da gestire, la modifica e la rimozione richiedono politiche molto attente. L'inserimento è ritenuto semplice perché se l'utente cerca di inserire contenuti non compatibili con lo stato della conoscenza salvata in *Insieme*, basta avvisarlo

con un messaggio di errore, chiedendogli di modificare ciò che non va e rigettando quindi l'inserimento.

La modifica o la rimozione di un contenuto richiedono invece l'analisi di tutti i contenuti ad esso collegati per vedere come tale modifica/rimozione li influenza. Si pensi alle affermazioni "Marco è un amico di Alex" e "Le foto della festa sono condivise con gli amici di Alex", ne segue che Marco ha accesso alle foto; se viene modificata la prima affermazione in "Marco non è amico di Alex" oppure eliminata, allora bisogna aggiornare i permessi di accesso alle foto e anche la definizione di "amici di Alex", cioè i membri del gruppo "amici di Alex".
Un altro esempio è dato nell'affermare che un contenuto *x* appartiene a due gruppi, A e B, dove A è sottogruppo di C e B è sottogruppo di D: se in un secondo momento si afferma che C è disgiunto da D, allora l'affermazione "x appartiene ad A e B" diventa errata e bisogna correggere. In casi più complessi è possibile che si formi una catena di elementi da aggiornare, richiedendo un'analisi ancora più approfondita.

3.2.3 Inferenza ed analisi del contenuto

I blocchi 5 e 6 (illustrazione 14), denominati "inferenza" e "analisi contenuto", sono particolari in quanto non implementano nessuna funzione specifica, ma sono pensati come interfacce virtuali per strumenti sviluppati da terzi. Gli strumenti a cui si fa riferimento sono:

- gli *analizzatori di contenuto*: sono software in grado di analizzare un certo tipo di contenuto (ad esempio testuale piuttosto che un'immagine) e di estrarne delle proprietà descrittive. Sono software che coprono discipline molto differenti, in quanto tali strumenti sono specifici per contenuti diversi, ed ognuno di essi sfrutta metodologie e tecniche adatte ai dati analizzati. Ad esempio per l'analisi di contenuti testuali si possono sfruttare gli analizzatori di testo, in grado di estrarre parole chiave e statistiche relative ai vocaboli usati sfruttando algoritmi propri dell'*Elaborazione del linguaggio naturale*[12]; le immagini, d'altro canto, necessitano di un'analisi che fa riferimento al campo della *Visione Artificiale*[13], ad esempio riuscendo a riconoscere quante persone sono presenti in una foto e, basandosi su un database di volti, associarvi le persone (intese come *contenuti primari* di *Insieme*)
- i *reasoner*: tradotti in italiano come *ragionatori automatici*, i reasoner sono, citando Wikipedia[14], "... software in grado di svolgere dei ragionamenti su delle basi di conoscenza adeguatamente formalizzate."
  Tali ragionatori, partendo quindi da asserzioni o assiomi formalizzati secondo un linguaggio logico (come ad esempio OWL), sono in grado di generare nuove asserzioni coerenti e conseguenti ai dati di partenza.

12 http://it.wikipedia.org/wiki/Elaborazione_del_linguaggio_naturale
13 http://it.wikipedia.org/wiki/Visione_artificiale
14 http://it.wikipedia.org/wiki/Ragionatore_automatico

Come si capisce dalla breve descrizione data, gli strumenti citati possono essere dei validi sostituti dell'utente nell'inserimento di contenuti semantici aggiuntivi, sempre nell'ottica di un aiuto attivo da parte del sistema nella creazione e gestione della conoscenza.

Esistono già varie soluzioni software, complete e mature, ed è quindi poco consigliabile svilupparne di nuovi per *Insieme*: per sfruttare suddette soluzioni, quindi, si crea un'interfaccia virtuale attraverso la quale strumenti di terze parti possono accedere ai dati, analizzarli e salvare i risultati di tale elaborazione.

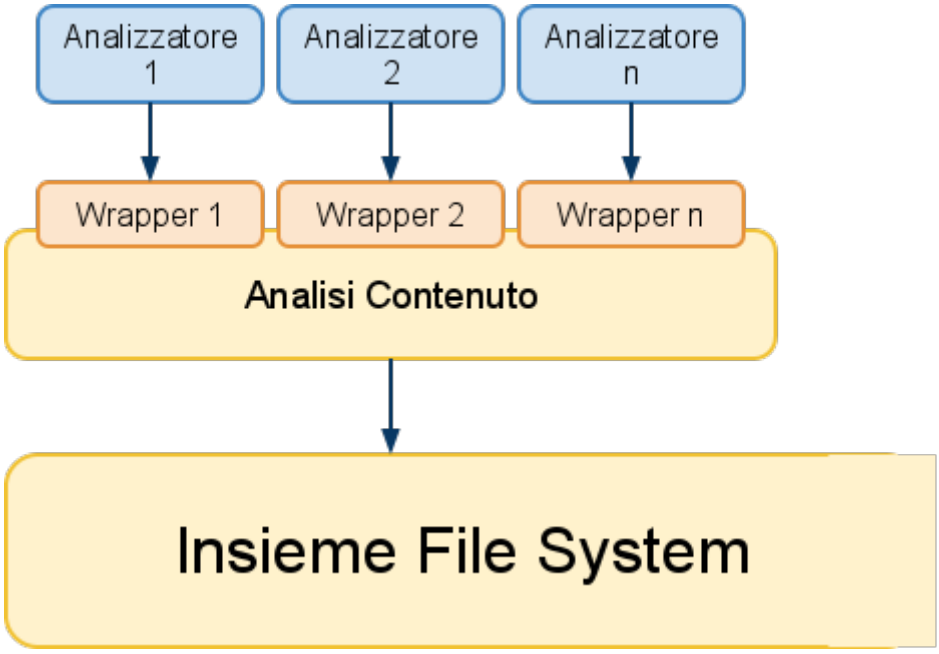

Illustrazione 15: Schema blocco "Analisi contenuto"

L'illustrazione 15 visualizza bene come i vari software scelti per l'analisi del contenuto (ma il discorso è analogo per i ragionatori automatici) si innestano nel sistema: ognuno di essi deve essere dotato di un wrapper che riempie le funzioni vuote messe a disposizione dal blocco *Analisi Contenuto*, facendo quindi da ponte tra *Insieme File System* e l'analizzatore. Il wrapper infatti si occupa di tradurre i dati in maniera che siano compatibili con il formato dell'analizzatore e, viceversa, trasforma i risultati dell'analisi nel formato interno di *Insieme*.

3.2.4 Rete sociale

Una persona, nella vita reale, interagisce con molte altre persone, le quali possono essere catalogate in base al tipo di rapporto (famiglia, amici, colleghi di lavoro, etc.), generando la cosiddetta "*rete sociale*" dell'utente (illustrazione 16).

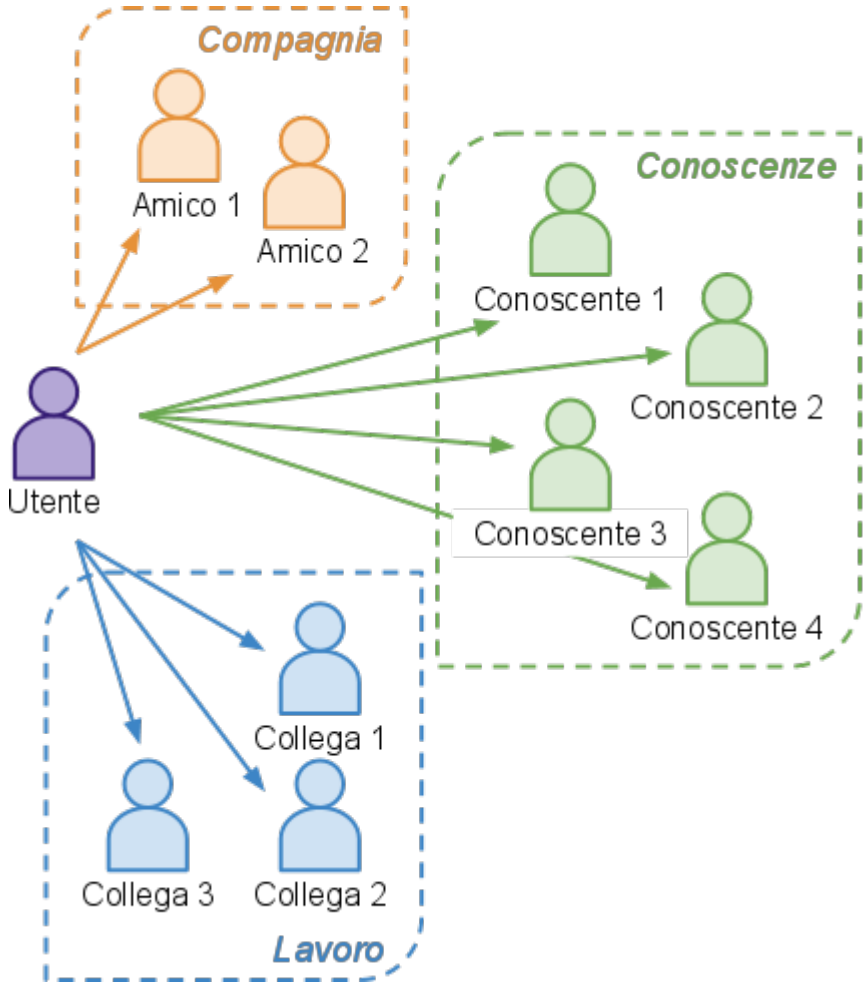

Illustrazione 16: Esempio di rete sociale

Con le persone della propria *rete sociale*, l'utente condivide esperienze di vita reali, dalle quali scaturiscono contenuti digitali (il report aziendale del gruppo di lavoro di cui l'utente fa parte, le foto con gli amici, etc.): è quindi indubbio che confrontando i contenuti dell'utente con quelli di un suo amico, si troverebbe che i due posseggono foto, video e documenti identici provenienti da esperienze vissute insieme. Questi contenuti sono perciò potenzialmente replicati su altri computer e dotati di contenuti semantici aggiunti dall'amico dell'utente: tali contenuti semantici potrebbero differire da quelli presenti sul computer dell'utente (ad esempio perché l'amico ha annotato informazioni che l'utente non ricorda o per altri motivi), rappresentando quindi una facile occasione per aumentare il patrimonio informativo di quest'ultimo.

Quello appena descritto non è l'unico vantaggio possibile:

- come descritto in precedenza, integrare i contenuti semantici dell'utente con quelli di altre persone della rete sociale
- condividere facilmente i contenuti solamente con le persone interessate
- visualizzare il contenuto condiviso da altre persone
- possibilità di non salvare un contenuto primario remoto in locale, ma tenerne solamente traccia, visualizzandoli comunque come risorse locali
- eseguire ricerche semantiche anche su contenuti remoti
- dare un peso diverso a contenuti provenienti da persone che hanno un rapporto considerato diverso dall'utente: ciò significa che vengono considerati più attendibili contenuti provenienti da un amico stretto che quelli di un conoscente
- aggiungere contenuto semantico direttamente sui contenuti delle persone della rete sociale (sempre che queste abbiano dato il permesso)

La rete sociale è ciò che si definisce una *Friend-to-Friend network* (abbreviato in F2F) [21], cioè una rete Peer-to-Peer in cui l'utente si connette direttamente ai propri conoscenti e solo a loro, ma con una differenza sostanziale: le reti definite dall'utente valgono solamente per l'utente stesso. Infatti la catalogazione che l'utente fa dei propri conoscenti non è valida per i conoscenti stessi, i quali possono usare catalogazioni diverse, dando origine a permessi diversi a seconda del punto di vista: è una differenza sostanziale che richiede soluzioni alternative a quelle proposte dai software pensati per la creazione di reti F2F.

Essendo tale modulo opzionale, in quanto non essenziale per il funzionamento base di *Insieme*, i dettagli tecnici vengono lasciati a lavori successivi.

3.2.5 Servizi internet

Oggigiorno avere una connessione internet sempre attiva è considerato abbastanza normale, quindi l'utente ha accesso ad una mole di dati imponente, contenuti relativi agli argomenti più disparati e servizi che vanno dal social sharing alla gestione del conto bancario: sono tutti contenuti che potenzialmente interessano l'utente.
È però impensabile salvare permanentemente, nella memoria locale, tutti i contenuti che l'utente incontra durante la navigazione in rete, ma d'altro canto in molti casi non è neppure necessario. Vi è infatti una "piramide" di valore dei contenuti: considerando i dati relativi al conto bancario e la ricerca fatta per sapere quando è nato un personaggio famoso, si capisce bene il motivo dell'affermazione precedente.

Se l'utente reputa di alto valore il contenuto a cui ha acceduto durante la navigazione, può salvarlo (se il proprietario di tale contenuto ne da il diritto)

all'interno di *Insieme*, mantenendo comunque un link alla risorsa originale. *Insieme* aiuta l'utente in tale processo mettendo a disposizione un'architettura su cui innestare dei wrapper dedicati a trasformare il contenuto preso dalla rete in un contenuto compatibile con *Insieme File System*. Solitamente i dati che si trovano in rete o sono dati grezzi, senza nessuna annotazione semantica, oppure utilizzano formati di annotazione diversi: per tali motivi servono strumenti ad hoc per la conversione.

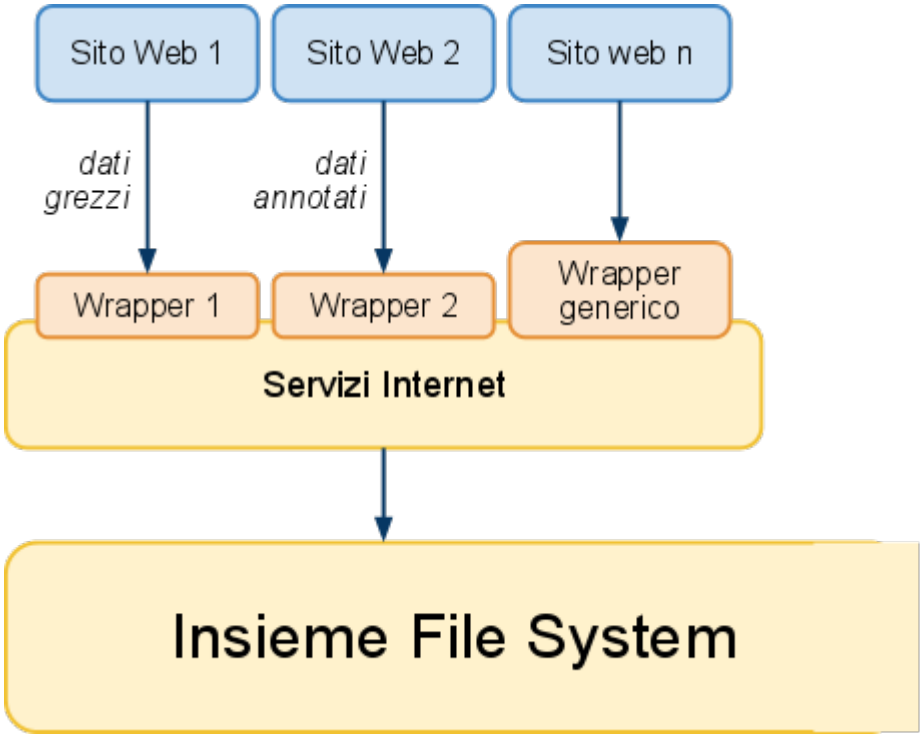

Illustrazione 17: Schema blocco "Servizi internet"

Quando invece l'utente reputa il contenuto trovato in rete non abbastanza importante da giustificare il suo salvataggio, è possibile comunque generare un contenuto virtuale, in cui viene salvato solo il riferimento al contenuto originale, ma annotabile come tutti gli altri contenuti salvati in *Insieme*. Tale opzione è giustificata anche dal fatto che alcune fonti di dati possono essere considerate "sempre a disposizione", sempre "rintracciabili": sono quei servizi internet di chiara fama mondiale e perciò difficilmente passabili di eliminazione o di spostamento a livello di indirizzo web. Fonti di questo tipo possono essere considerati siti quali *Wikipedia*[15], *YouTube*[16], *Facebook*[17], *Google*[18], etc.

15 http://www.wikipedia.org/
16 http:/www.youtube.com
17 http:/www.facebook.com
18 http:/www.google.com

4. Descrizione del prototipo

Questo capitolo è dedicato all'implementazione software di *Insieme*, con l'analisi delle scelte effettuate, gli strumenti utilizzati e le difficoltà incontrate durante lo sviluppo.
Si specifica innanzitutto che, come descritto nel capitolo precedente (Capitolo 3), *Insieme* risulta essere un progetto vasto e complesso, per cui si sono concentrati gli sforzi solamente su alcuni dei componenti strutturali: più nello specifico, sono stati implementati, a livello di prototipo, i componenti essenziali, lasciando quelli opzionali a sviluppi futuri. Tale scelta è dovuta anche alla necessità di acquisire una maggiore conoscenza e consapevolezza delle difficoltà presentate da progetti del genere, insieme ad una sperimentazione iniziale di soluzioni software possibili atta ad individuare la più promettente.

Di seguito viene riportata l'illustrazione della struttura di Insieme con evidenziati gli elementi realizzati per il prototipo.

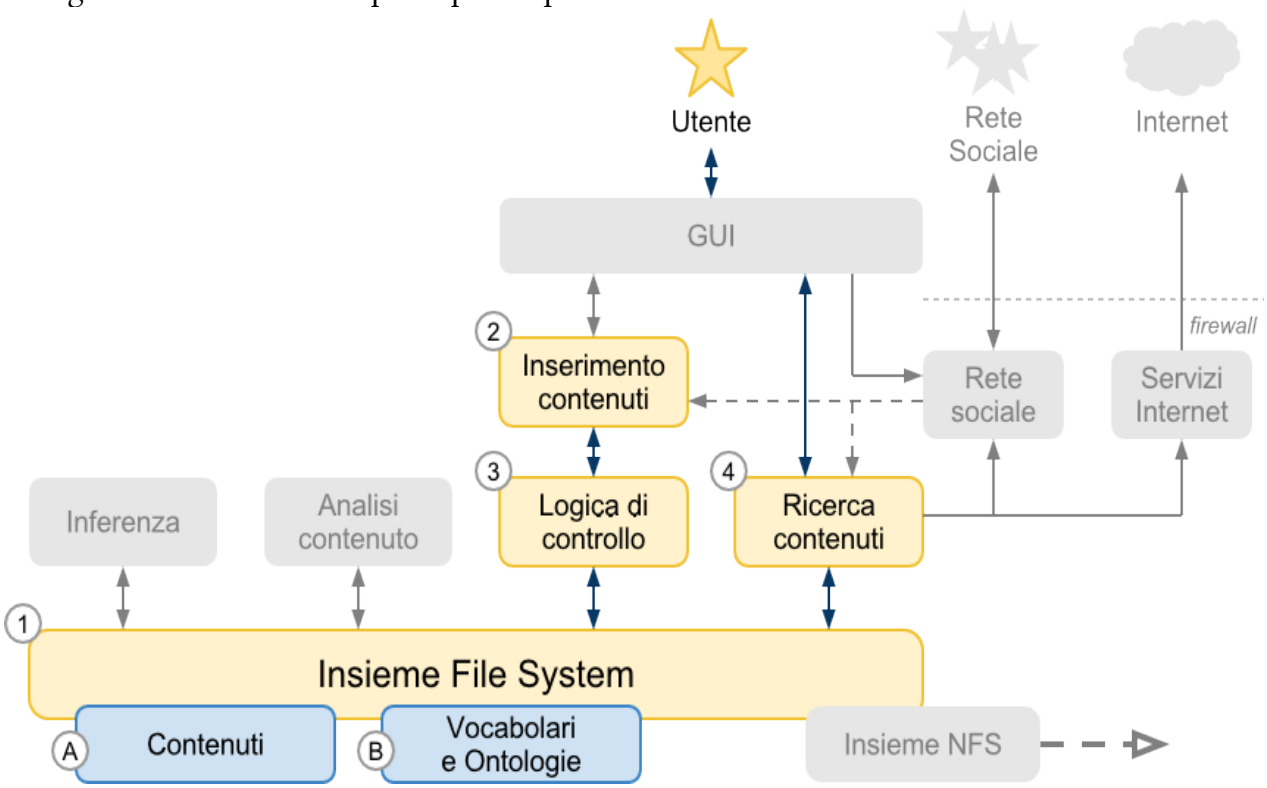

Illustrazione 18: Schema dei blocchi sviluppati

Come in un qualsiasi progetto, sono necessarie alcune scelte di carattere tecnico su cui impostare tutto il lavoro successivo, e che andranno ad influenzare le decisioni che verranno prese in seguito. Per il progetto *Insieme* tali scelte sono state, in sintesi, le seguenti:

- uso di *Python 3* come linguaggio di programmazione
- modellazione dei contenuti in base al paradigma offerto da *RDF*
- utilizzo della libreria *RDFLib*[19]

4.1 Il linguaggio di programmazione

La scelta del linguaggio per la codifica dei componenti è ricaduta su *Python*[20], più precisamente sulla versione 3. *Python*, come si legge nella homepage del sito ufficiale, è

> *"... a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs."*

19 http://www.rdflib.net/
20 http://www.python.org/

cioè

> *"... un linguaggio di programmazione che permette di lavorare più velocemente ed integrare i sistemi più efficacemente. Si può imparare Python e riscontrare benefici pressoché immediati nella produttività e minori costi di manutenzione."*

e, riprendendo in parte la lista riportata sul sito ufficiale[21], i suoi punti di forza che ne hanno deciso l'utilizzo sono:

- sintassi molto chiara e leggibile
- forti capacità di introspezione[22]
- orientato agli oggetti
- piena modularità, supporto alla gerarchia dei moduli
- gestione degli errori basata sulle eccezioni
- ampio catalogo di librerie standard e moduli di terze parti per ogni necessità
- estensioni e moduli facilmente scrivibili in C, C++, ma anche Java e .NET
- utilizzabile all'interno di applicazioni come interfaccia di scripting
- uso di UTF-8 come default per tutti i dati testuali (le stringhe), e quindi supporto a tutte le lingue
- uso dell'indentazione del codice per identificare i blocchi logici, pertanto non opzionale, garantendo una leggibilità maggiore del codice
- linguaggio pseudo-compilato, rendendo portabile il codice su diverse piattaforme (come Java)
- open source

4.2 Il modello dei dati

L'ultimo dei vantaggi elencati per Python (l'essere open source) risponde al punto 8 degli obiettivi di *Insieme*, ed è guardando sempre a questo obiettivo che si è scelto di strutturare i contenuti che verranno elaborati seguendo il *modello usato da RDF*. *RDF* (come spiegato nel Capitolo 2) è uno standard sviluppato dal *W3C*, un organismo internazionale il cui lavoro influenza pesantemente (se non decide) una serie di protocolli e linguaggi che sono alla base del Web (si pensi al codice di mark up HTML): ultimamente il *W3C* si è concentrato sul concetto di "web semantico" e ha prodotto a questo proposito, oltre a *RDF* e alla sua estensione *RDFS*, anche l'ontologia *OWL*, tutti strumenti adottati da moltissime terze parti per sviluppare le proprie ontologie. Per rendere *Insieme* compatibile con questa realtà e renderlo facilmente integrabile con altri software (almeno a livello di scambio dati), si è deciso di basarsi sulle specifiche *RDF* per la rappresentazione della conoscenza.

21 http://www.python.org/about/

22 Si legga a tale proposito il 3° Capitolo di "Dive into Python" (http://it.diveintopython.org/power_of_introspection/index.html)

Secondo la *logica dei predicati*, le informazioni sono esprimibili con *asserzioni* (*statement* in inglese); tali asserzioni sono delle *triple* formate da *soggetto*, *predicato* e *oggetto*, ed è proprio così che RDF (ma in generale tutta la suite di linguaggi e ontologie messe a punto dal W3C) esprime i concetti. Quello che introduce RDF rispetto alla logica dei predicati è la necessità di identificare gli elementi con un URI, che risulta essere un vantaggio nella gestione dei dati in quanto si può usare tale URI come identificativo anche all'interno del sistema, senza doversi preoccupare di creare un identificativo diverso.

4.3 Lavorare coi dati

Le specifiche RDF sono abbastanza estese, per cui si è deciso di sfruttare una libreria già implementata per poter modellare e lavorare con i dati secondo il paradigma scelto. La libreria adottata è *RDFLib*, una libreria open source scritta in *Python*, dopo un'attenta ricerca dei tool disponibili online: tale ricerca viene riportata per intero nel prossimo capitolo.

La struttura base di *RDFLib* è la classe *Graph*, un set non ordinato nel quale vengono messe le triple, che sono, a loro volta, dei set (questa volta, però, ordinati) costruiti utilizzando delle classi che modellano i termini RDF. Tali classi sono:

- *URIRef*: modella il concetto di risorsa dotata di un URI
- *Literal*: modella il concetto di letterale, cioè un valore quali stringhe e numeri, preoccupandosi di convertirle dallo standard XSD[23] ad oggetti Python
- *BNode*: modella il concetto di Blank Node, cioè una risorsa di cui si conosce l'esistenza, ma che comunque non è definita da un URI univoco

L'aggiunta di triple avviene attraverso il metodo *add()* della classe *Graph*, passando come argomento il set composto da soggetto, predicato ed oggetto, cioè gli elementi di una tripla; non è però l'unico modo per popolare un grafo. Attraverso il metodo *parse()*, sempre della classe *Graph*, è possibile importare direttamente un intero grafo salvato in una delle annotazioni supportate (RDF/XML, Notation 3, N-Triples), sia esso salvato in un file locale, sia che venga specificata una locazione remota.

Una funzionalità messa a disposizione da RDFLib per facilitare l'inserimento degli URI è la gestione dei *Namespace*: in molti formati di annotazione di RDF (ad esempio nel formato RDF/XML) tale funzionalità corrisponde ai *prefissi* che si trovano all'inizio del file e il cui significato è stato spiegato nel Capitolo 1. RDFLib offre una gestione automatica dei namespace, rendendo l'uso dei prefissi uguale al metodo di accesso degli attributi di un oggetto: tramite il metodo *bind()* della classe *Graph* è possibile associare ad un grafo un prefisso univoco per l'URI di un

23 http://www.w3.org/TR/xmlschema11-2/

vocabolario od ontologia. Tale prefisso viene trasformato in un'istanza della classe *Namespace*, dotata al suo interno di un metodo per generare dinamicamente gli URI dei termini appartenenti al vocabolario od ontologia. Se ad esempio associamo l'URI del vocabolario RDF ad un grafo (codice 14 - Appendice A), si crea un oggetto *Namespace* chiamato 'RDF', utilizzabile per rappresentare dinamicamente i vocaboli di RDF.
Ad esempio il codice 15 (Appendice A) equivale all'URI completo del vocabolo "type" di RDF, cioè *http://www.w3.org/1999/02/22-rdf-syntax-ns#type*.

Utilizzare la classe *Graph* non comporta automaticamente il salvataggio dei dati inseriti: tale funzionalità è demandata alla classe *Store*, che, attraverso una struttura a plug-in, permette di utilizzare ed implementare wrapper per soluzioni di storage delle triple differenti. Di base *RDFLib* offre solamente uno store basato sulla memoria temporanea, privo di alcune funzionalità necessarie (prima fra tutte la possibilità di recuperare i dati salvati in sessioni di lavoro precedenti), per cui per il progetto di Insieme si è scelto di affidarsi al database non relazionale *CouchDB*, il cui wrapper, sviluppato ad-hoc, viene descritto in seguito.

Si è resa necessaria la riscrittura della libreria *RDFLib* in quanto il codice usato è tuttora di una versione di *Python* precedente alla 3, che è quella scelta per il progetto *Insieme*: la versione 3 di Python, pur mantenendo la stessa sintassi, introduce funzionalità nuove che modificano profondamente le dinamiche interne di lavoro, rendendolo non retro-compatibile. La riscrittura si è rivelata essere da un lato abbastanza semplice grazie alla chiarezza ed immediatezza di Python ed alla documentazione ufficiale molto completa, ma dall'altro anche difficoltosa, in quanto il codice della libreria *RDFLib* è scarsamente annotato, rendendo oneroso il processo di reverse-engineering degli errori e la loro correzione.

4.4 Librerie e strumenti esistenti

In questo sotto-capitolo vengono riportati le librerie ed il software analizzati per un loro possibile uso all'interno di Insieme; l'obiettivo della ricerca è stato trovare tutti quegli strumenti, possibilmente scritti in Python, utili ad elaborare dati in RDF o con funzioni affini. Per ognuno viene specificato:

- nome
- sito internet
- breve spiegazione delle funzionalità offerte
- pro e contro in un'ottica di utilizzo all'interno del software sviluppato

4.4.1 Database specifici per RDF

Esistono varie soluzioni per il salvataggio dei dati semantici ed in particolare per i dati espressi sotto forma di triple RDF. I cosiddetti *triple stores* sono database specializzati nella gestione delle triple RDF e quasi tutti supportano uno o più

linguaggi di interrogazione, di cui il più comune è SPARQL[24]. Di seguito presentiamo un elenco delle soluzioni disponibili, anche se non ci si soffermerà sulle caratteristiche, né sui pregi ed i difetti, in quanto non è obiettivo primario del progetto l'analisi di tali database.

Triple store disponibili:

- 4store (http://4store.org/)
- Sesame (http://www.openrdf.org/)
- OWLIM Semantic Repository (http://www.ontotext.com/owlim/index.html)
- AllegroGraph RDFStore 4.0 (http://www.franz.com/agraph/allegrograph/)
- Mulgara (http://www.mulgara.org)

4.4.2 Librerie specifiche per RDF ed OWL

RDFLib[25]

Libreria scritta in Python per lavorare con RDF, in cui le triple RDF vengono presentate attraverso un grafo. Mette a disposizione vari parser, serializzatori e plug-in per vari triple store.

Pro	*Contro*
Supporta tutti i formati di annotazione	Scritto per versioni di Python precedenti alla 3
Struttura a plug-in, anche per la memorizzazione e le query	Pensato solo per RDF e non per un'ontologia completa come OWL
Documentazione buona	
Progetto ancora attivo	
Usato come base per altri progetti	

SuRF[26]

Libreria scritta in Python in cui i dati RDF vengono trattati con uno stile orientato agli oggetti.

Pro	*Contro*
Supporta più metodi per scrivere i dati su un triple store	Scritto per versioni di Python precedenti alla 3
Query eseguite con SPARQL	Pensato solo per RDF e non per un'ontologia completa come OWL
Progetto ancora attivo	Orientato a lavorare solo con

24 http://www.w3.org/TR/rdf-sparql-query/
25 http://www.rdflib.net/
26 http://code.google.com/p/surfrdf/

	ontologie già fatte senza modificarle

ORDF Python Library[27]

ORDF è una libreria Python basata su RDFLib. Prevede l'uso di ragionatori e il supporto al clustering.

Pro	*Contro*
Supporta più metodi per scrivere i dati su un triple store Supporta OWL Supporta l'uso di ragionatori Progetto ancora attivo	Scritto per versioni di Python precedenti alla 3

Pyrple[28]*, Purple*[29]*, RDFAlchemy*[30]*, Sparta*[31]*, Ontopy*[32]*, SETH*[33]

Sono tutte librerie per la manipolazione di dati RDF, che però risultano essere o incomplete o addirittura abbandonate dagli sviluppatori. Non presentano funzionalità di particolare rilievo, ma è possibile comunque analizzarne il codice per capire se certe soluzioni possano essere adottate.
L'unica nota di interesse è per SETH, che è una libreria specifica per lavorare con dati espressi in OWL, però anch'essa di fatto non più sviluppata.

4.4.3 Librerie specifiche per triple store

Sparrow[34]

Sparrow è una libreria pensata per offrire una API di alto livello per i Database RDF, uniformando le differenze esistenti tra le varie soluzioni di salvataggio e dando una visione unitaria allo sviluppatore.

Pro	*Contro*
Supporta più serializzazioni delle triple Maschera il livello di comunicazione col database	Scritto per versioni di Python precedenti alla 3

27 http://ordf.org/
28 http://infomesh.net/pyrple/
29 http://www.deelan.com/dev/purple/
30 http://www.openvest.com/trac/wiki/RDFAlchemy
31 http://github.com/mnot/sparta/
32 http://github.com/afternoon/ontopy/
33 http://seth-scripting.sourceforge.net/
34 http://pypi.python.org/pypi/sparrow/1.0b4

SPARQL Endpoint interface to Python[35]

Interfaccia per dialogare con un SPARQL Endpoint generico.

HTTP SPARQL server and client for twisted and rdflib[36]

Libreria pensata per eseguire query SPARQL attraverso il protocollo http su host remoti. Contiene sia un client che un server, ma può dialogare con altri client e server SPARQL.

Pro	*Contro*
Pensato per query remote Progetto ancora attivo	Scritto per versioni di Python precedenti alla 3

4.4.4 Ragionatori automatici

Fuxi[37]

Fuxi si presenta come un ragionatore scritto in Python che lavora con grafi serializzati in N3; necessita di RDFLib per funzionare.

Pro	*Contro*
Unico ragionatore trovato per Python Progetto ancora attivo Supporta RDFS e OWL	Scritto per versioni di Python precedenti alla 3

4.4.5 Librerie specifiche per grafi

Python-graph[38]

Libreria molto completa per lavorare con grafi di qualunque tipo. Mette a disposizione numerosi algoritmi di ricerca ed operazioni specifiche per i grafi.

Pro	*Contro*
Specifico per i grafi Molto completo dal punto di vista delle funzionalità Progetto ancora attivo	Scritto per versioni di Python precedenti alla 3

35 http://sparql-wrapper.sourceforge.net/
36 http://projects.bigasterisk.com/sparqlhttp/
37 http://code.google.com/p/fuxi/
38 http://code.google.com/p/python-graph/ e http://www.linux.ime.usp.br/~matiello/python-graph/docs/

GraphPath[39]

GraphPath è una libreria che analizza dati strutturati come grafi, specialmente quelli RDF. Offre anche un risolutore di query ed un generatore di inferenze.

Pro	*Contro*
Pensato per l'analisi dei dati Possiede un risolutore di query Possiede un generatore di inferenze	Scritto per versioni di Python precedenti alla 3

Graph.py[40]

Semplice libreria per la gestione di grafi diretti e non. Non ha particolari pregi e risulta essere non più sviluppata.

4.4.6 Librerie varie

MoPy[41]

Libreria pensata specificatamente come interfaccia alla Music Ontology, permette di leggere e scrivere elementi RDF con uno stile orientato agli oggetti.

Golem[42]

Golem è un'ontologia unita ad un set di strumenti per processare dati annotati con il Chemical Markup Language (o CML) ed è quindi pensata specificatamente per l'ambito chimico.

ConceptNet[43]

ConceptNet è una particolare libreria che vuole dare accesso ai computer ad una conoscenza di senso comune, ottenuta con i contributi dati da utenti qualsiasi attraverso siti come Open Mind Common Sense[44].

Pro	*Contro*
Obiettivo del progetto in sintonia con quelli di Insieme	Scritto per versioni di Python precedenti alla 3

39 http://www.langdale.com.au/GraphPath/
40 http://robertdick.org/python/graph.html
41 http://www.omras2.org/PythonTutorial e https://motools.svn.sourceforge.net/svnroot/motools/
42 http://www.lexical.org.uk/science/golem/
43 http://csc.media.mit.edu/conceptnet
44 http://openmind.media.mit.edu/

Luminoso[45]

Luminoso è un analizzatore di testo scritto in Python.

Pro	*Contro*
Utile per l'analisi dei testi	Scritto per versioni di Python precedenti alla 3 Tool solamente grafico, non offre una libreria

4.5 Sviluppo soluzione memorizzazione

Insieme, in quanto pensato per essere innanzitutto un file system, deve necessariamente salvare i contenuti che l'utente inserisce in maniera permanente e, conseguentemente, recuperarli per elaborazioni successive.

La scelta di utilizzare *RDFLib* come base per la manipolazione di dati RDF permette la costruzione di wrapper qualsiasi soluzione di memorizzazione venga scelta, a patto di implementare le funzioni astratte della classe *Store*. Graficamente l'operazione di salvataggio dei dati attraversa i seguenti blocchi software:

45 http://csc.media.mit.edu/docs/luminoso/walkthrough.html

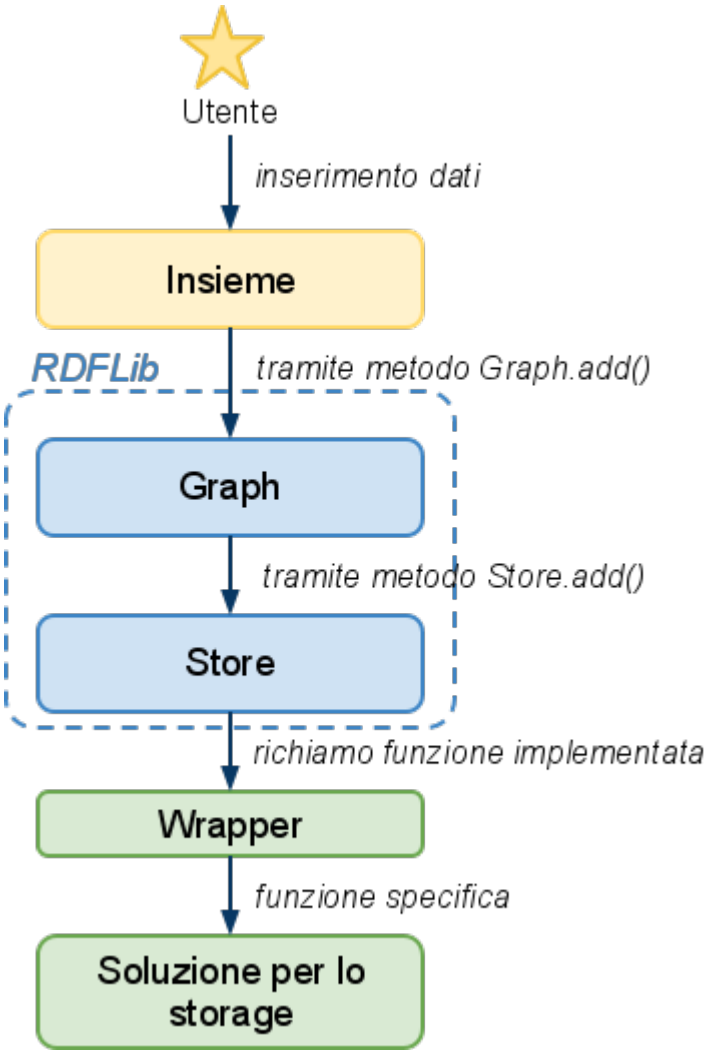

Illustrazione 19: Schema dell'operazione di salvataggio dati

4.5.1 Sesame2

Inizialmente la scelta della soluzione per lo storage è ricaduta su *Sesame2* di *OpenRDF*[46], in quanto database open-source specializzato nella gestione di triple RDF, ma tale decisione si è rivelata in seguito affetta da un problema che ne ha causato l'abbandono. Come visto in precedenza nel sotto-capitolo dedicato alla libreria RDFLib, vi sono tre elementi con cui costruire una tripla: *URIRef*, *Literal* e *BNode*.

Tra i tre, l'elemento *BNode* è quello che risulta più problematico da gestire, in quanto, nonostante sia un nodo a tutti gli effetti del grafo, non è dotato di un *URI* univoco (come invece hanno i nodi di tipo URIRef) e bisogna quindi provvedere a dotare ogni istanza di BNode di un codice di identificazione: RDFLib si preoccupa

46 http://www.openrdf.org/

di generare una stringa alfanumerica ogni qualvolta viene istanziato un BNode, e tale codice viene poi usato nella creazione delle triple in cui tale BNode compare.
Il problema sorge quando le triple vengono passate a Sesame2 per il salvataggio: Sesame2 rimuove il codice del BNode generato da RDFLib e lo sostituisce con uno proprio, creando perciò dei riferimenti sbagliati quando le triple vengono recuperate. Inoltre il problema è aggravato dal fatto che la sostituzione di un codice generato da RDFLib non è univoca: per ogni tripla in cui compare il codice dello stesso BNode, viene generato un codice da Sesame2 diverso (avendo perciò più codici Sesame2 per ogni codice RDFLib), rendendo perciò impossibile la traduzione durante la fase di recupero.

4.5.2 CouchDB

Visti i problemi dati dall'utilizzo di *Sesame2*, si è deciso di cambiare drasticamente, implementando uno store per RDFLib basandosi su *CouchDB*[47].

Illustrazione 20: Logo di CouchDB

CouchDB viene descritto nell'home page del progetto come:

> *"Apache CouchDB is a document-oriented database that can be queried and indexed in a MapReduce fashion using JavaScript. [...]*
>
> *A CouchDB server hosts named databases, which store **documents**. Each document is uniquely named in the database, and CouchDB provides a RESTful HTTP API for reading and updating (add, edit, delete) database documents.*
>
> *Documents are the primary unit of data in CouchDB and consist of any number of fields and attachments. Documents also include metadata that's maintained by the database system. Document fields are uniquely named and contain values of varying types (text, number, boolean, lists, etc), and there is no set limit to text size or element count."*

47 http://couchdb.apache.org/

cioè:

> *"Apache CouchDB è un database orientato al documento che può essere interrogato ed indicizzato con uno stile MapReduce usando Javascript. [...]*
>
> *Un server CouchDB ospita dei database dotati di nome, che immagazzinano* ***documenti****. Ogni documento è identificato univocamente del database, e CouchDB mette a disposizione una RESTful HTTP API per leggere e aggiornare (aggiungere, modificare, eliminare) i documenti del database.*
>
> *I documenti sono l'unità dati primaria in CouchDB e consistono di un numero qualsiasi di campi e allegati. I documenti includono inoltre metadati mantenuti dal sistema del database. I campi dei documenti hanno nomi univoci e contengono valori di differente tipo (testo, numeri, booleani, liste, etc.), e non vi è limite alla dimensione del testo o al numero di elementi."*

CouchDB è un database del tipo *NoSQL*[48], cioè database studiati apposta per non richiedere schemi fissi di tabelle, evitare spesso le operazioni di unione (*join*) e, tipicamente, scalare orizzontalmente.

Vista la scelta di utilizzare le triple come elemento base all'interno di Insieme per la rappresentazione della conoscenza, si potrebbe obiettare la mancata adozione di un database relazionale SQL: le triple presentano una struttura (soggetto - predicato - oggetto) che ben si presta ad essere salvata in un database SQL, pensati per lavorare con tabelle e schemi.
Si è comunque optato per CouchDB come risultato delle seguenti considerazioni:
i database NoSQL sono tutti progetti abbastanza recenti e propongono interessanti soluzioni: si coglie quindi l'occasione per un loro studio, osservando pro e contro di una loro adozione per il progetto *Insieme.*
Essendo il software sviluppato solamente un prototipo, la struttura definitiva con cui modellare i contenuti potrebbe subire, in riscritture successive, profonde modifiche: l'utilizzo di un database che non basa il proprio lavoro su schemi predefiniti permette un'ottima flessibilità in caso di modifica, riducendo di molto gli sforzi da dedicare alla gestione del database.

48 http://en.wikipedia.org/wiki/NoSQL

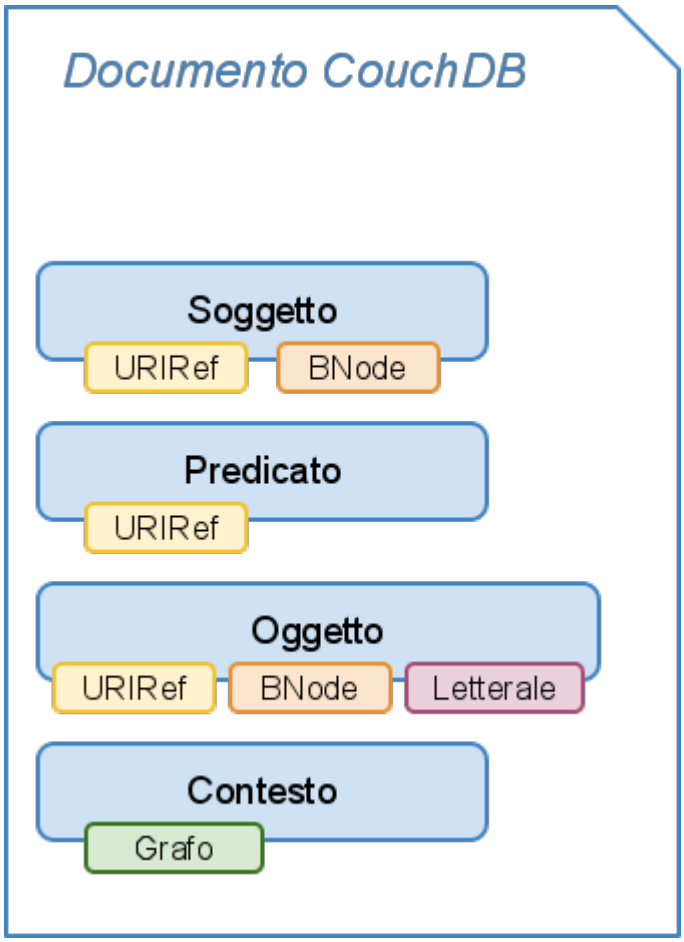

Illustrazione 21: Schema di un documento CouchDB

La modellazione delle triple in CouchDB (illustrazione 21) segue le seguenti regole:

1. ogni tripla viene salvata in un documento
2. l'identificativo del documento è formato unendo la stringa "triple_" ad una stringa alfanumerica casuale. Un esempio di identificativo può essere perciò:
   triple_006bbd8c8dd54f3cb38438c8eefb4034
3. ogni documento presenta 6 campi:
   1. *_id* : l'identificativo del documento, è un campo speciale di CouchDB
   2. *_rev* : il numero di revisione del documento, è un campo speciale di CouchDB
   3. *s* : rappresenta il soggetto della tripla
   4. *p* : rappresenta il predicato della tripla
   5. *o* : rappresenta l'oggetto della tripla
   6. *context* : rappresenta il grafo a cui appartiene tripla

Come si vede nell'immagine (illustrazione 21), i campi che compongono il documento (tralasciando quelli specifici di CouchDB) possono assumere come

valore la rappresentazione di una delle classi RDFLib per la costruzione di una tripla (cioè *URIRef*, *BNode* e *Literal*), mentre il contesto ha come valore la rappresentazione della classe *Graph*. Ognuna delle rappresentazioni delle classi RDFLib su CouchDB si ottiene tramite un dizionario, nel quale è sempre presente la chiave *type* per indicare di che elemento si tratta:

- "node" per indicare un oggetto *URIRef*
- "bnode" per indicare un oggetto *BNode*
- "literal" per indicare un oggetto *Literal*
- "graph" per indicare un oggetto *Graph*

Oltre alla chiave type, ogni dizionario ha delle chiavi con valori specifici per la classe rappresentata:

- *URIRef*
  - chiave *id* che assume come valore l'URI del nodo
- BNode
  - chiave *id* che assume come valore il codice del blank node assegnato da RDFLib
- Literal
  - chiave *value* che assume come valore il valore del letterale
  - chiave datatype che assume come valore il tipo di datatype XSDdel letterale
  - chiave *language* che assume come valore, nel solo caso in cui il letterale sia una stringa, la lingua in cui la stringa è scritta
- Graph
  - chiave *id* che assume come valore l'URI del grafo

Anche i *namespace* associati dall'utente al proprio grafo vengono salvati sul database in documenti separati, la cui struttura è la seguente:

1. l'identificativo del documento è formato unendo la stringa "namespace_" ad una stringa alfanumerica casuale. Un esempio di identificativo può essere perciò:
   namespace_4309764328314e8988b16ac823bb56b6
2. ogni documento presenta 4 campi:
   1. *_id* : l'identificativo del documento, è un campo speciale di CouchDB
   2. *_rev* : il numero di revisione del documento, è un campo speciale di CouchDB
   3. *namespace* : rappresenta l'URI base del vocabolario o ontologia
   4. *prefix* : il prefisso scelto per il vocabolario o l'ontologia

4.5.3 Analisi codice dello store

Lo store sviluppato per RDFLib e basato su CouchDB implementa tutti i metodi della classe *Store*, coprendo quindi tutte le funzionalità che RDFLib prevede per le operazioni di memorizzazione. Di seguito vengono riportati e spiegati i punti del codice di interesse maggiore.

Il primo punto riguarda la creazione dello store e l'istanziazione del relativo database: come è riportato nel codice 16 (Appendice A) appartenente al metodo *open()* della classe *CouchdbStore*, si controlla sempre se il nome scelto per il database (passato attraverso il dizionario *configuration*) non sia già usato da qualche database esistente (riga 1). In caso esista (riga 2), la si restituisce, altrimenti la si crea con la funzione *create()* sempre appartenente alla classe *CouchdbStore* (riga 5); se però si cerca di recuperare un database (attraverso il parametro *retrieve* passato alla funzione *open()*) e tale database non esiste, viene sollevata un'eccezione (riga 7).

Due funzioni basilari per il corretto funzionamento sono *element_to_db()* (codice 17 – Appendice A) ed *element_from_db*() (codice 18 – Appendice A), in quanto sono le funzioni che si occupano della conversione dei dati tra RDFLib e CouchDB, operando le necessarie trasformazioni per passare dai tipi di dato supportati da Python (cioè il formato usato da RDFLib) e JSON (che è il formato usato da CouchDB).

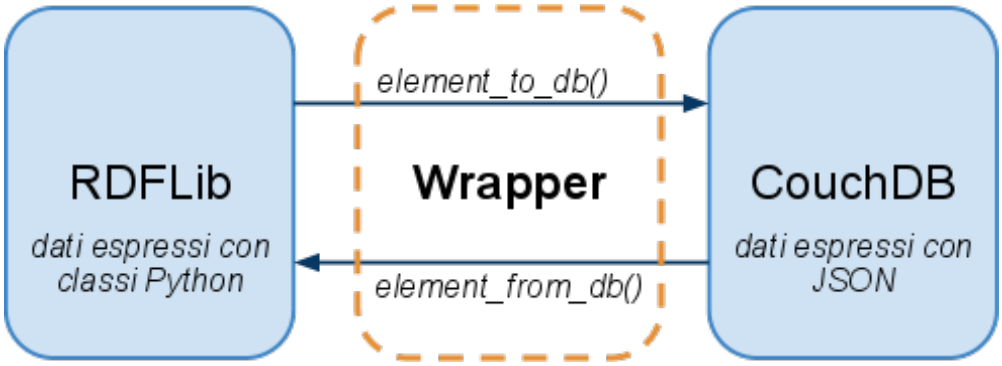

Illustrazione 22: Il wrapper per la traduzione dei dati

La funzione *element_to_db()* converte gli oggetti RDFLib in dizionari seguendo lo schema precedentemente esposto; la funzione *element_from_db()*, invece, richiama il costruttore apposito degli oggetti RDFLib in base al valore del campo "type" ed inserendo i parametri necessari per l'inizializzazione corretta dell'istanza creata.

Come si può osservare, il codice non è particolarmente complesso, permettendo quindi una conversione lineare e veloce tra i due formati e la ricostruzione delle triple avviene senza nessun problema anche in presenza di blank nodes.

L'ultima osservazione riguardo al codice dello store sviluppato riguarda non tanto un metodo di classe, quanto l'utilizzo di una funzionalità di CouchDB per rendere il codice più snello ed evitare operazioni onerose dal punto di vista computazionale: le *viste*[49].
CouchDB, così come i tradizionali database relazionali, implementa il concetto di *vista*, cioè la possibilità di salvare una particolare ricerca, in modo da poter accedere velocemente alla lista aggiornata dei risultati richiamando semplicemente il nome della vista, e di procedere all'esecuzione di ulteriori operazioni.

Le viste di CouchDB sono funzioni scritte in *Javascript*[50] e si possono raggruppare più funzioni vista in un documento speciale nel database a cui fanno riferimento; il nome dei documenti che raggruppano le viste sono costruiti unendo la parola riservata "_design" seguita dal carattere "/" e il nome del gruppo di viste. Ad esempio se vengono create alcune viste che si interessano di recuperare i documenti secondo condizioni relative ai soggetti delle triple, allora tali viste possono essere salvate nel documento "_design/soggetto". Di seguito vengono riportate alcune schermate di *Futon*[51], che è l'interfaccia grafica di CouchDB, relative alle viste ed anche il codice esempio di una vista.

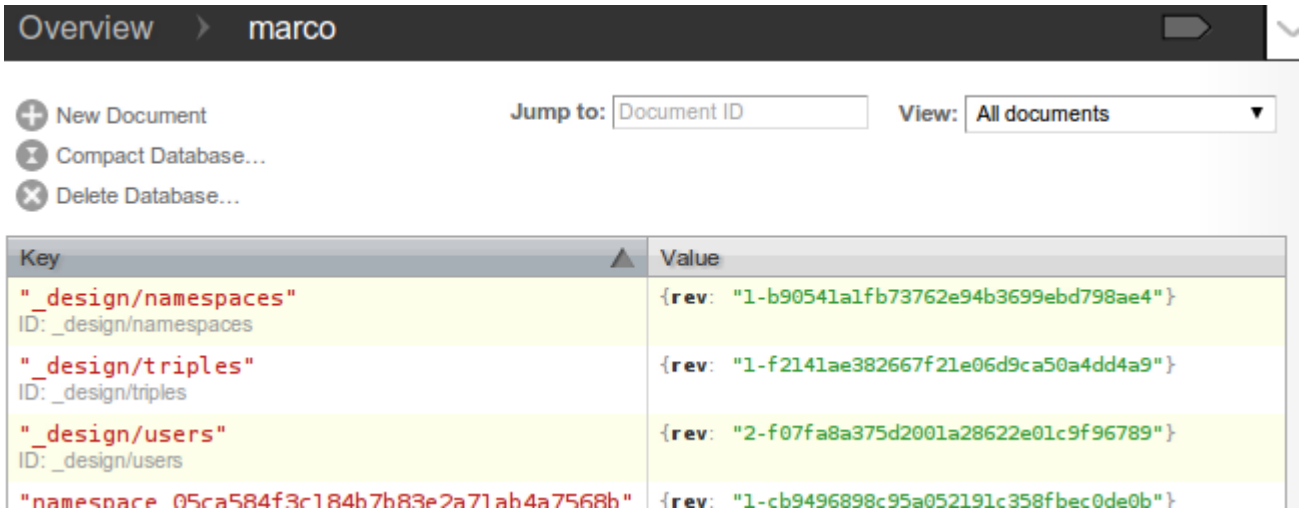

Illustrazione 23: I documenti speciali di CouchDB

49 http://couchdb.apache.org/docs/overview.html
50 http://www.ecma-international.org/publications/standards/Ecma-262.htm
51 http://answers.oreilly.com/topic/1395-introduction-to-couchdbs-futon-administration-interface/

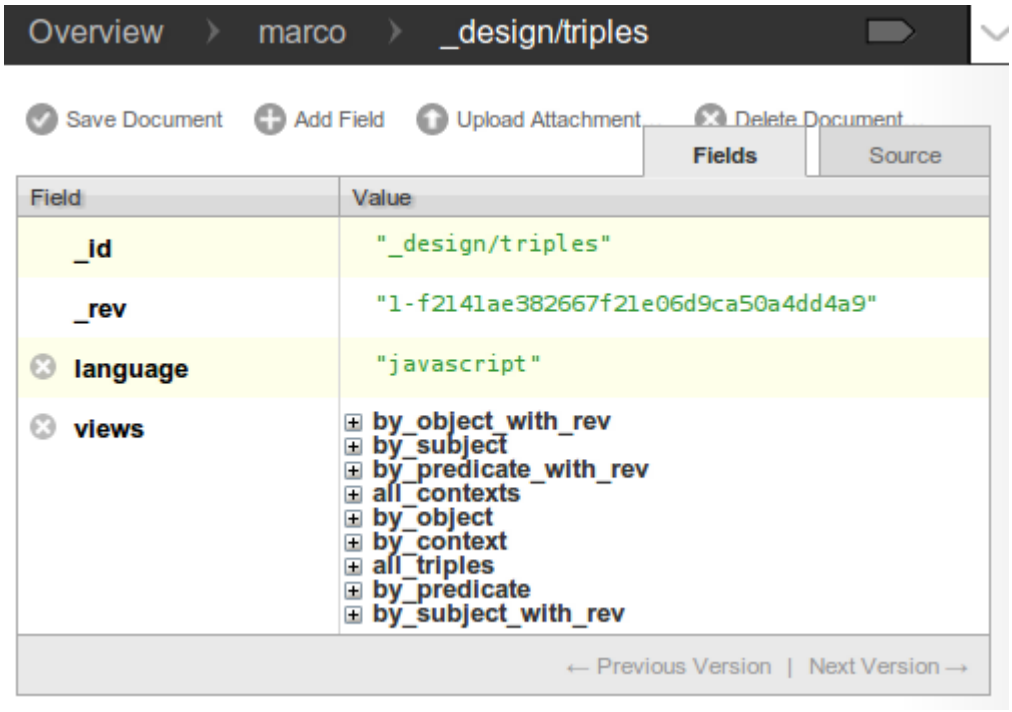

Illustrazione 24: Documento CouchDB contente delle viste

Il codice 19 (Appendice A) rappresenta una vista che recupera tutte le triple e le ordina rispetto al soggetto.

Le viste, come detto in precedenza, vengono usate all'interno dello store per agevolare alcune operazioni che altrimenti risulterebbero onerose, operazioni legate soprattutto al controllo dell'esistenza di una tripla, alla ricerca di una o più triple dato un elemento (ad esempio tutte le triple con un dato soggetto): le triple permettono di avere subito i risultati voluti, evitando di dover scorrere tutto il database e ripetere le stesse operazioni migliaia di volte. Un esempio banale è dato dal recupero delle triple appartenenti ad un certo grafo: mentre con l'utilizzo di una vista tale operazione si riduce a recuperare i risultati della vista stessa, fatta con una funzione Python richiederebbe:

- lo scorrimento di tutti i documenti salvati sul database
- la lettura dell'identificativo del documento per capire se è una tripla o altro (ad es. un namespace)
- accedere al campo "context" dei documenti relativi alle triple
- salvare in una lista i documenti il cui campo "context" corrispondo al grafo cercato

Il risparmio computazionale raggiunto è dunque evidente.

4.6 Sviluppo codice

Il prototipo vero e proprio di Insieme si sviluppa in un'unica classe denominata *Insieme*, dotata dei metodi necessari per implementare le funzionalità previste dai blocchi 2, 3 e 4 della struttura di Insieme (illustrazione 14). Si riporta una breve lista in cui i metodi vengono suddivisi nei relativi blocchi funzionali:

- *Istanziazione della classe*: metodi *__new__()* e *__init__()*
- *Inserimento contenuti*: metodi *statement()*, *create_content()* e *bind()*
- *Logica di controllo*: metodi *node_subnodeof()*, *indomain()* e *inrange()*

Il blocco di *ricerca dei contenuti* non compare nella lista in quanto si sfrutta una classe costruita appositamente per interrogare CouchDB, *CouchdbQuery*, senza basarsi su RDFLib e che si trova nello stesso modulo in cui è presente lo store sviluppato per CouchDB.

4.6.1 Istanziazione della classe

L'istanziazione della classe Insieme avviene attraverso due funzioni: *__new__()* ed *__init__()*.

Nel linguaggio *Python* queste due funzioni sono considerate speciali: definendole all'interno di una classe, è possibile controllare il processo di istanziazione ed inizializzazione delle istanze di tali classi (i cosiddetti *oggetti* del *paradigma ad oggetti*). Più specificatamente il metodo *__new__* viene eseguito prima che l'istanza della classe venga creata, mentre il metodo *__init__* viene richiamato subito dopo la creazione dell'istanza della classe.

Durante l'utilizzo di *Insieme* su un calcolatore, è prevista la seguente regola:
ogni utente può avere una ed una sola istanza attiva di Insieme
Se un utente avesse attive più istanze (alle quali sarebbero collegate database diversi), i contenuti da lui inseriti sarebbero sparpagliati in più punti e senza la possibilità di legarli tra loro: ciò causerebbe inoltre duplicazione di contenuti (l'utente potrebbe dimenticare dove li ha inseriti) o, peggio ancora, perdita accidentale degli stessi. Per evitare ciò, si controlla la creazione delle istanze attraverso i metodi *__new__* ed *__init__* seguendo lo schema riportato nell'illustrazione 25:

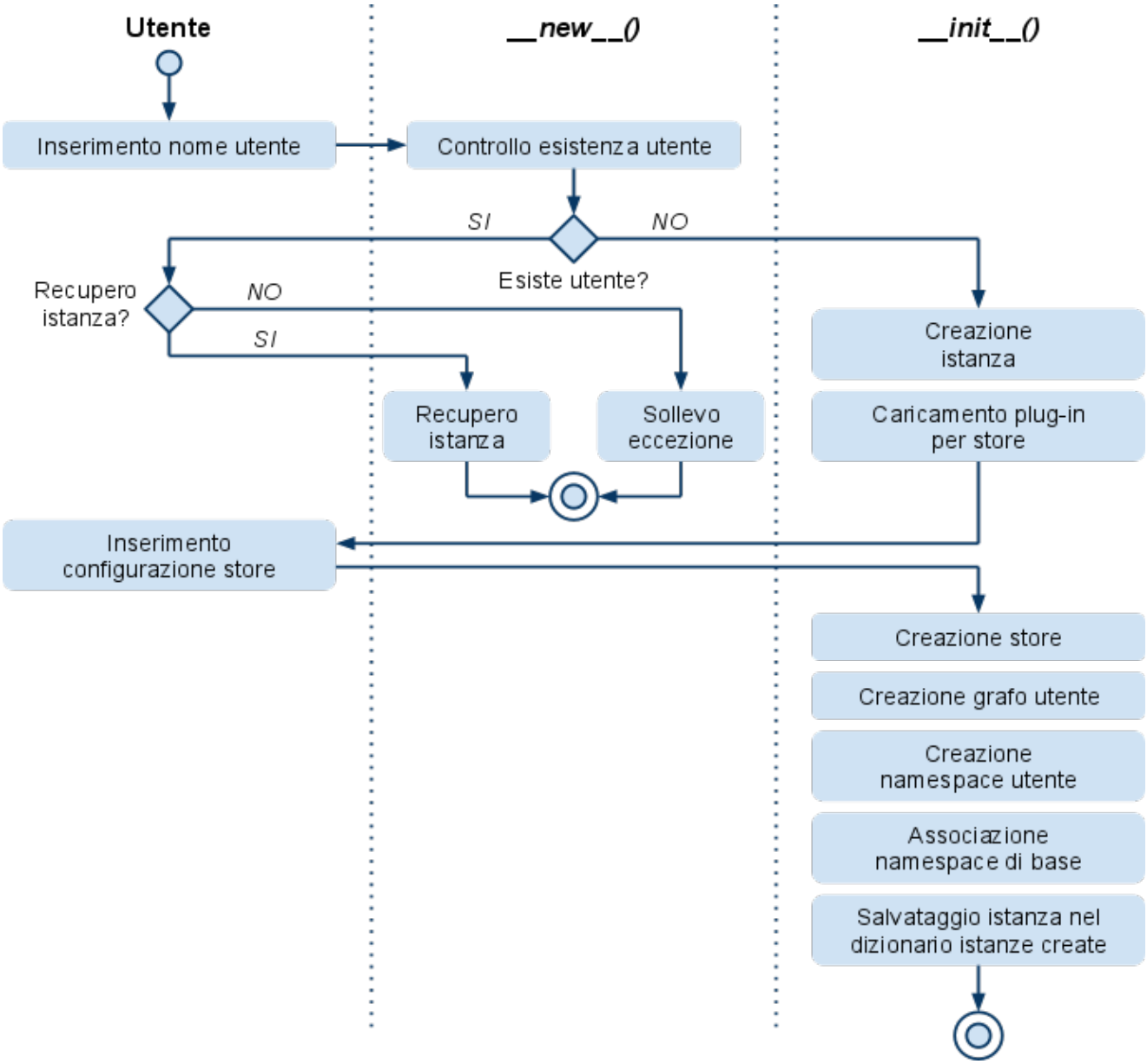

Illustrazione 25: Schema funzione di istanziazione

Il primo passo avviene con la specifica, da parte dell'utente, del nome da utilizzare come identificativo dell'istanza: la funzione *__new__* controlla innanzitutto che il parametro passato sia una stringa valida, poi controlla se nel dizionario delle istanze esistenti di Insieme ce ne è una che corrisponde al nome scelto. Tale dizionario, chiamato *USERS_INSIEME* e accessibile come parametro della classe *Insieme*, salva i riferimenti alle istanze già create usando come chiave per la loro indicizzazione il nome passato dall'utente all'atto dell'inizializzazione.
Se esiste una chiave corrispondente al nome scelto, allora vi sono due possibilità: o l'utente recupera l'istanza già salvata, oppure si solleva un'eccezione. Se invece non esiste una chiave corrispondente, il controllo passa alla funzione *__init__*.

Il metodo __init__ si preoccupa di inizializzare le variabili e strutture interne dell'istanza, partendo anzitutto dallo *store*: viene importato il plug-in per lo store basato su CouchDB analizzato in precedenza attraverso il codice 20 (Appendice A), e, attraverso la configurazione data dall'utente (che sono valori specifici per lo store, quali indirizzo del database e il nome del database), viene associato lo *store* all'istanza salvandone il riferimento nel parametro *_store* (codice 21 - Appendice A).

Le operazioni che seguono riguardano:

- creazione del grafo personale dell'utente (codice 22 – Appendie A): si crea un grafo utilizzando la classe *Graph* di RDFLib e gli si assegna un identificatore costruito con il nome dell'utente (*'http://localhost/' + nome utente*)
- creazione del namespace personale dell'utente (codice 23 - Appendice A): si crea l'URL di base del vocabolario dell'utente, costruito con il nome dell'utente (*'http://localhost/' + nome utente*)
- associazione dei namespace di base e salvataggio dei loro vocabolari (codice 24 - Appendice A): si associano all'istanza di Insieme i vocabolari ritenuti necessari per la definizione di tutti i contenuti che verranno creati od importati dall'utente.

L'ultimo punto fa riferimento a vocabolari ritenuti basilari per il corretto funzionamento di Insieme. Tali vocabolari sono:

- RDF
- RDFS
- OWL
- XSD
- il vocabolario base di Insieme

Per ognuno di essi viene inoltre salvato nel database il grafo delle triple che li compongono, assegnando come contesto delle triple l'indirizzo base del vocabolario a cui appartengono.

L'ultima operazione, prima di ritornare l'istanza creata, è l'aggiornamento del dizionario delle istanze, salvando quella appena creata.

4.6.2 Inserimento contenuti

Il blocco logico dell'inserimento dei contenuti è coperto da due metodi, *statement()* e *create_resource()*: mentre il primo è pensato per l'annotazione di contenuti esistenti, il secondo è invece dedicato alla creazione di nuovi.
In entrambi i casi, gli statement creati vengono inseriti nel grafo personale dell'utente, evitando di "inquinare" eventuali grafi importati di cui l'utente non è il proprietario e che quindi potrebbero cambiare senza preavviso.

Da un punto di vista tecnico, il metodo *create_resource()* è basato sul metodo *statement()*, in quanto lo richiama al suo interno all'atto di crearezione degli statement, pertanto si analizza brevemente il primo metodo, per poi passare al secondo.

Osservando il grafico di funzionamento di *create_resource()* (illustrazione 26), si può vedere come i contenuti creati debbano innanzitutto avere un nome che non sia già presente nel grafo dell'utente, altrimenti viene sollevata una eccezione (si violerebbe il principio di unicità dell'URI). Dopodichè viene richiesto all'utente di che tipo è il contenuto da creare, il quale ha tre opzioni:

- creare una classe, specificando rdfs:Class
- creare una proprietà, specificando rdf:Property
- creare l'istanza di una classe, specificando l'URI della classe desiderata

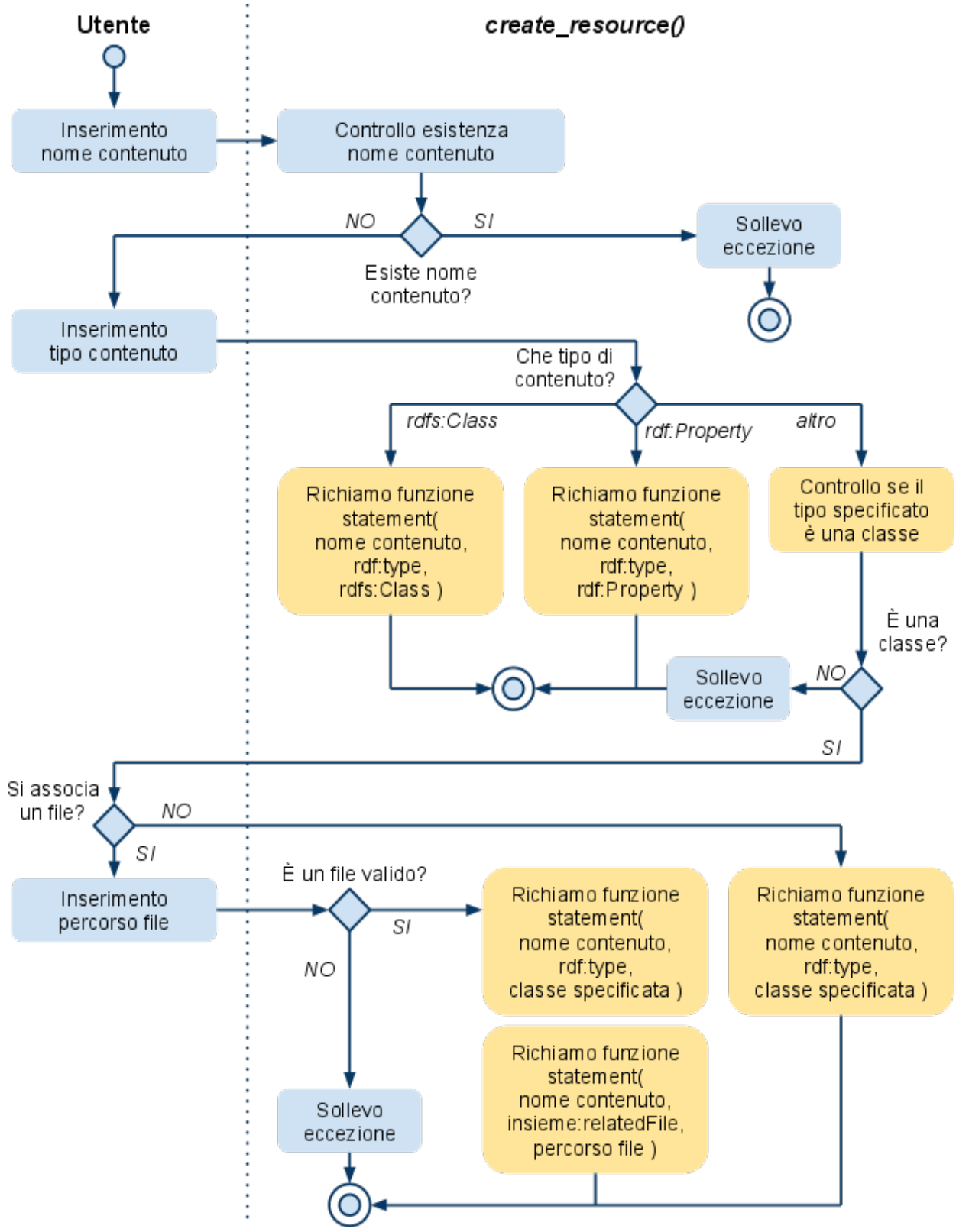

Illustrazione 26: Schema di creazione di una risorsa

Mentre per le prime due opzioni viene subito richiamato il metodo *statement()*, l'ultima opzione richiede alcuni passi aggiuntivi: per prima cosa entra in gioco un metodo appartenente al blocco della logica di controllo, e cioè *node_subnodeof()*, che però viene trattato nel sottocapitolo dedicato alla logica di controllo. Successivamente si chiede all'utente se desidera collegare al contenuto creato il percorso di un file (rendendo così il contenuto la rappresentazione virtuale del file all'interno di *Insieme*): in caso positivo viene controllato se il percorso passato è di un file valido o meno. Come ultimo passo c'è il richiamo alla funzione *statement()*.

Anche la funzione *statement()* ha, di per sè, un codice abbastanza minimale, basato in gran parte sul richiamo di metodi appartenenti al blocco della logica di controllo: questo è abbastanza naturale in quanto non vi sono operazioni necessarie alla preparazione della tripla da inserire, visto che oltretutto *Insieme* sfrutta lo stesso formato dati di *RDFLib*, eliminando perciò la necessità di convertire i dati.

I controlli richiamati sono, in ordine, i seguenti:

- controllo se gli elementi della tripla rientrano in uno dei namespace associati all'istanza dell'utente (se l'oggetto è un letterale il controllo non viene eseguito)
- controllo che il predicato sia una proprietà (cioè di tipo rdf:Property o sottoproprietà di una proprietà)
- controllo sull'appartenenza o meno di soggetto al dominio della proprietà specificata
- controllo sull'appartenenza o meno dell'oggetto al codominio della proprietà specificata

Se i controlli vengono tutti superati, allora viene richiamata la funzione *add()* della classe *Graph* della libreria *RDFLib*, la quale si occupa di aggiungere la tripla al grafo dell'utente e passarla allo *store* per il salvataggio sul database.

4.6.3 Logica di controllo

Il blocco della logica di controllo dei contenuti inseriti è quello con il maggior numero di funzioni. Fanno parte di questo blocco i seguenti metodi:

- *get_ns* : dato un URI, recupera il namespace di cui fa parte ed associato all'istanza di insieme
- *node_subnodeof* : dati due nodi, controlla se il primo è in qualche modo legato al secondo attraverso proprietà quali:
    - rdf:type
    - rdfs:subClassOf
    - rdfs:subPropertyOf

- *indomain* : dato un elemento ed una proprietà, controlla se l'elemento rientra nel dominio della proprietà
- *inrange* : dato un elemento ed una proprietà, controlla se l'elemento rientra nel codominio della proprietà

La prima funzione, *get_ns*, presenta un codice intuitivo. Prende in ingresso un URI e, confrontandolo con i namespaces associati all'istanza di Insieme, cerca quello a cui l'URI fa riferimento: ciò è ottenuto sfruttando una semplice comparazione tra stringhe, cercando il namespace il cui URI corrisponde alla parte iniziale dell'URI passato. Se per caso nessuno dei namespace associati coincide con l'inizio dell'URI, il metodo ritorna il valore nullo.

Il metodo *node_subnodeof* è importante, in quanto può essere considerato il controllo basilare su cui sviluppare gli altri: infatti, anche *indomain* e *inrange* ricorrono, all'interno del loro codice, a questo metodo per svolgere il loro compito. È insito nel concetto di gerarchia il porre in una relazione di subalternità degli elementi gli uni con gli altri, pertanto, considerato che Insieme basa il suo lavoro su una gerarchia di contenuti costruita dall'utente, è naturale avere la necessità di disporre di un metodo che permetta di controllare la relazione gerarchica presente tra due elementi.

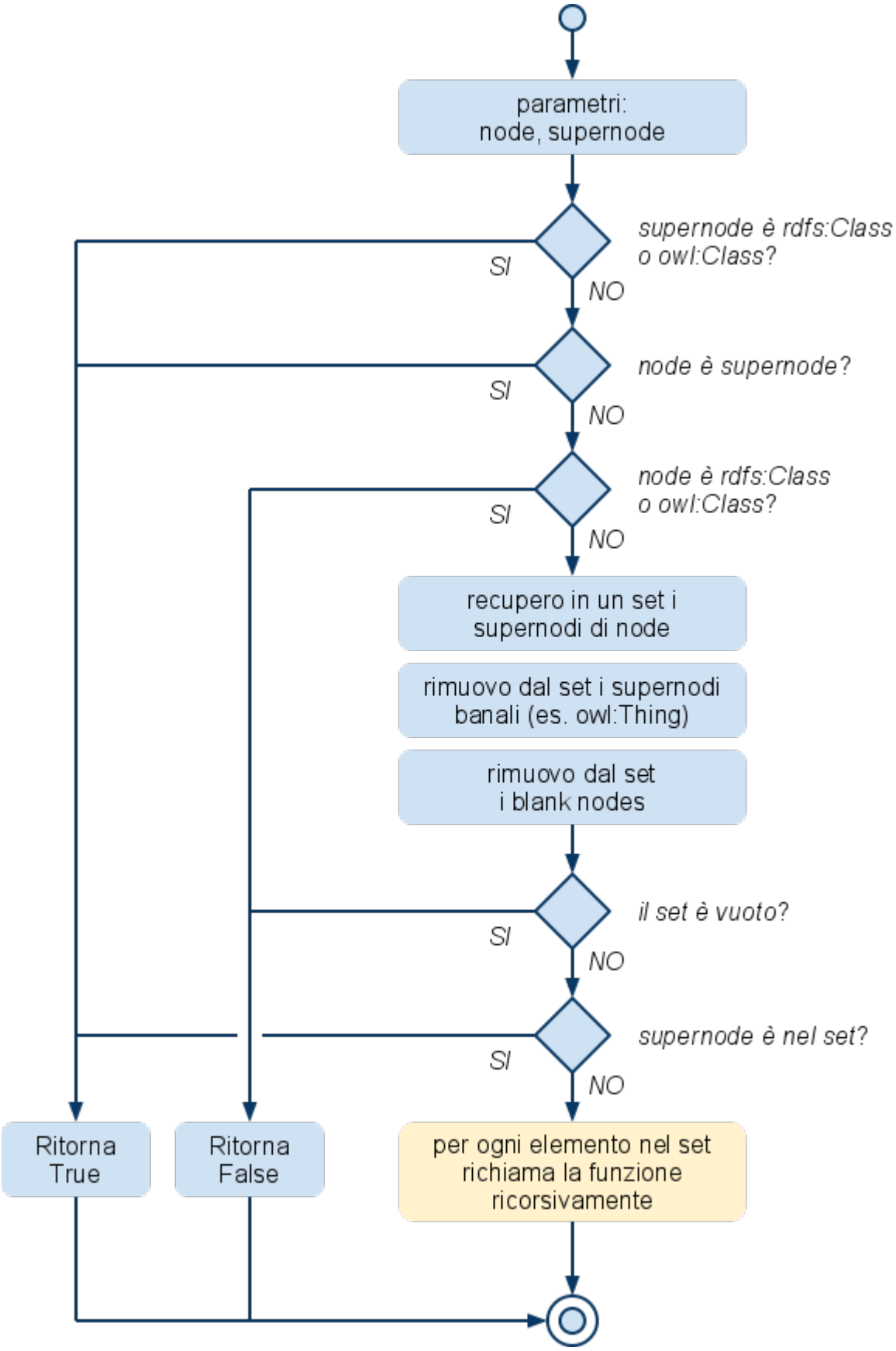

Illustrazione 27: Schema della funzione node_subnodeof()

Il principio di funzionamento di *node_subnodeof*, come illustrato nello schema (illustrazione 27), è la ricorsione (elemento colorato in giallo), attraverso la quale si ripercorre la gerarchia a cui l'elemento, passato come possibile subalterno, appartiene.
Nello schema (illustrazione 27) si parla di supernodi banali, cioè tutti quei vocaboli che rappresentano il concetto del "tutto". Sono considerati tali i vocaboli:

- rdfs:Resource
- rdfs:Class
- owl:Thing

in quanto nelle specifiche dei relativi linguaggi vengono descritti come classi delle quali ogni risorsa fa parte e sono di conseguenza superclassi di ogni elemento.

Le funzioni *indomain* ed *inrange* basano il loro funzionamento, come anticipato in precedenza, sull'uso della funzione *node_subnodeof* per capire se un elemento rientra nel dominio o codominio di una data proprietà e ciò significa proprio capire se l'elemento è un sottonodo dell'elemento specificato come dominio o codominio. Di seguito si riporta lo schema del metodo *indomain*; si tralascia invece *inrange* in quanto, nonostante concettualmente sia simile, richiede controlli in più e renderebbe la trattazione più difficoltosa. Ciò si deve al fatto che, mentre il dominio di una proprietà può solamente essere una classe, il range di una proprietà può essere:

- una classe
- una lista di nodi classe, quindi un blank node visto che le liste in rdf vengono create sfruttando i blank node
- un datatype
- una lista di nodi datatype, quindi un blank node
- un letterale
- una lista di letterali, quindi un blank node

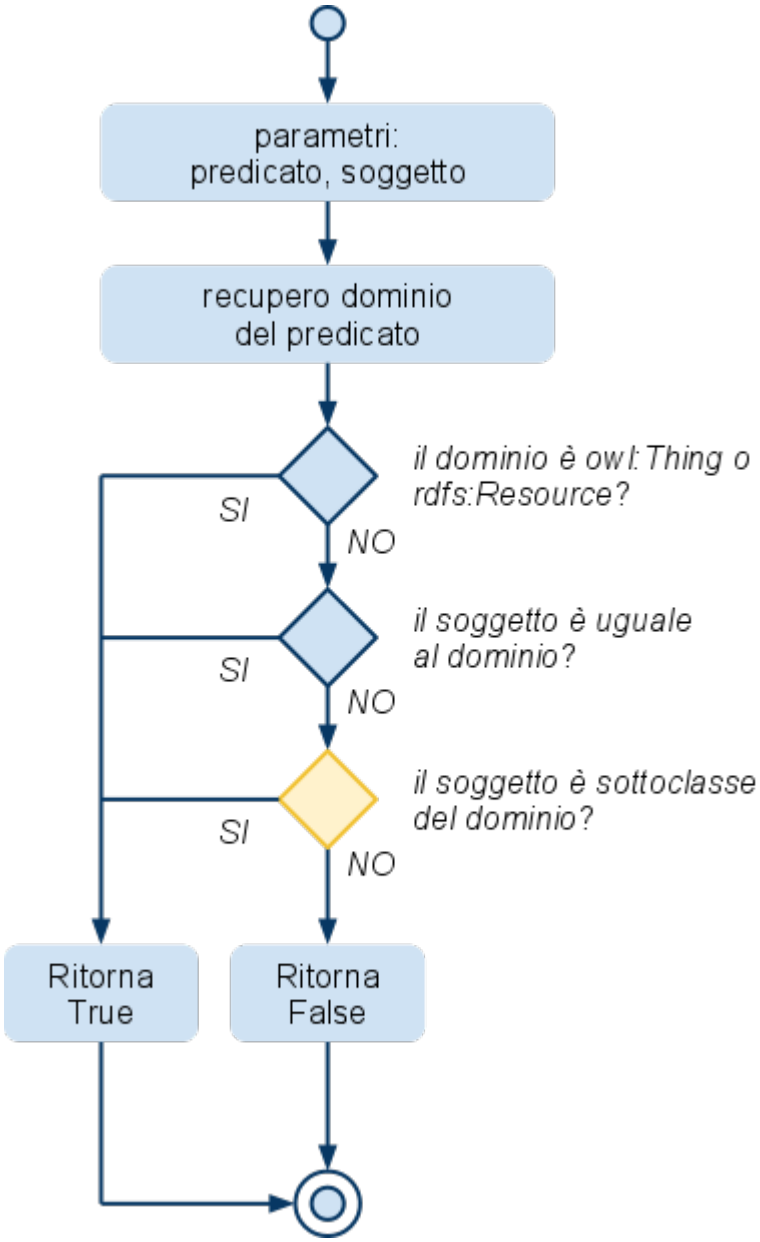

Illustrazione 28: Schema della funzione indomain()

Come si può vedere dallo schema in figura 28, i controlli eseguiti all'interno del metodo *indomain* non sono complessi (sono semplici operazioni di confronto tra stringhe per verificare l'uguaglianza tra URI), lasciando la parte più onerosa alla funzione *node_subnodeof* (elemento colorato in giallo).

I metodi elencati non implementano ovviamente tutti i controlli che sarebbero necessari per avere un'ontologia vera e propria, ma ne rappresentano comunque un primo abbozzo, andando a coprire gli aspetti basilari e dando un'idea di come sviluppare i restanti: servono inoltre come metro di misura delle prestazioni, aiutando a capire come migliorare la struttura di Insieme e dei dati stessi.

4.6.4 Ricerca contenuti

Per eseguire una ricerca dei contenuti (chiamate anche *query*), si utilizza una classe presente all'interno del modulo "couchdb_store.py" ed estranea alla libreria RDFLib: in tale modulo si trova inoltre il codice che rende utilizzabile il database *CouchDB* come magazzino di triple. La classe in questione è *CouchdbQuery*, che si preoccupa di:

- costruire la query
- eseguirla
- ritornare i risultati ripulendoli dagli elementi non richiesti

La query deve essere scritta in *Javascript*[52] ed è in tutto e per tutto uguale ad una vista, tanto che nella documentazione di CouchDB [22] viene definita "temporary view" (cioè "vista temporanea"). L'interrogazione viene definita attraverso i seguenti metodi:

- *__init__* : inizializza la query
- *where* : specifica la prima condizione di ricerca
- *_and* : specifica ulteriori condizioni di ricerca
- *end_query* : chiude la query specificando i parametri desiderati in uscita
- *execute* : esegue la query sul database e ritorna i risultati

I metodi elencati possono essere concatenati tra loro al fine di definire la query desiderata, avendo cura di rispettare lo schema di composizione riportato in figura 29.

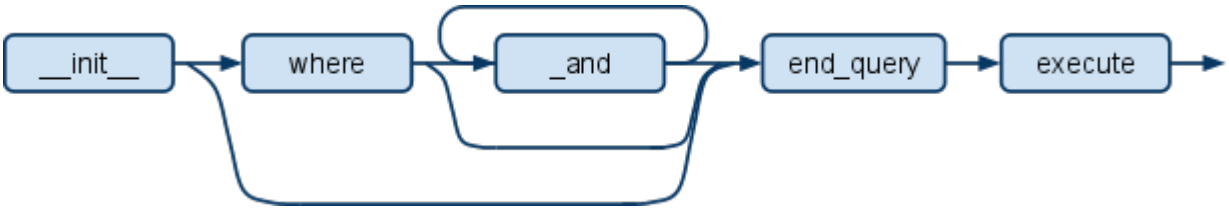

Illustrazione 29: Schema di composizione di una query

__init__ è la funzione speciale di Python per l'inizializzazione di un'istanza ed in questo caso si inizializza una stringa in cui salvare la query. La query viene inizializzata a:

"function(doc) {\n\tif (doc._id.match(/^triple_'/) && doc.context.id==\"+ user_namespace + '\'){}}"

che corrisponde alla ricerca di tutte le triple appartenenti al grafo dell'utente (definito dalla variabile *user_namespace*).

52 http://www.ecma-international.org/publications/standards/Ecma-262.htm

La query può essere raffinata tramite la clausola *where* (utilizzabile una sola volta) e *_and* (utilizzabile un numero illimitato di volte). Tutte e due prevedono gli stessi parametri, lasciando all'utente la possibilità di specificare:

- una proprietà specifica da ricercare o parte del suo nome (nel caso non ricordasse il nome intero)
- un oggetto della tripla specifico, parte di esso (valido solo per valori testuali) o un range di valori

Una volta inserite tutte le clausole di ricerca, si chiude la query con *end_query*, i cui due parametri (*key* e *value*) rappresentano i valori che l'utente desidera ricevere in caso di esecuzione della query. *key* può assumere uno dei seguenti valori:

- "_id": l'id del documento della tripla salvato sul database
- "s": il soggetto della tripla
- "p": il predicato della tripla
- "o": l'oggetto della tripla

mentre *value* può assumere i seguenti valori:

- "s": il soggetto della tripla
- "p": il predicato della tripla
- "o": l'oggetto della tripla
- "doc": tutto il documento
- "null": il valore null

5. Esempio di utilizzo

In questo capitolo si dimostrano le funzionalità implementate (descritte nel Capitolo 5) attraverso un esempio strutturato su vari punti che ricalchi un possibile flusso di lavoro di un utente.
All'inizio vengono presentati i passi dell'esempio, per poi osservare ognuno di essi più in dettaglio, riportando anche spezzoni di codice, l'output testuale e qualche schermata del database.

La dimostrazione che si intende mostrare prende spunto da una situazione in cui un utente può ritrovarsi durante l'utilizzo del computer, descritta di seguito:

> *"Marco è appena tornato dalle vacanze che ha trascorso con i suoi amici Alex, Chiara, Gianluca, Giovanni e Federica. Ha fatto delle foto e vuole annotare quali persone ci sono in ogni foto. Gli sono poi arrivati dei messaggi e il sistema annota da chi provengono.*

Ora Marco vorrebbe poter eseguire alcune ricerche sui contenuti che ha salvato:

1. *vorrebbe innanzitutto vedere le foto*
2. *poi solo le foto in cui c'è l'amico Alex*
3. *vorrebbe allargare la ricerca e vedere qualsiasi contenuto in cui Alex è stato annotato, non importa per quale motivo*
4. *vorrebbe vedere un messaggio in particolare, ma si ricorda solamente che gli è stato spedito da qualcuno che inizia per 'Gi'*

"

Scomposto in singole operazioni diventa:

- Si crea l'istanza di Insieme per l'utente Marco
- Si creano le classi "Foto_Vacanze" e "Messaggi"
- Si creano le istanze di 10 foto e 4 messaggi, collegandovi altrettanti file
- Si carica all'interno di Insieme il vocabolario FOAF
- Si creano le istanze dei 5 amici: Alex, Chiara, Gianluca, Giovanni, Federica
- Si creano le proprietà "include_persona", per dire che una foto ha dentro una certa persona, e la proprietà "mandato_da", per dire che un messaggio è stato mandato da una certa persona
- Si annotano le istanze delle foto e dei messaggi con le proprietà appena create ed assegnandovi valori diversi
- Esecuzione delle query
  - Query "tutte le foto"
  - Query "tutte le foto con Alex"
  - Query "tutti i contenuti con Alex come oggetto"
  - Query "tutti i messaggi mandati da una persona il cui nome inizia per 'Gi' "

Tali passi coprono tutte le funzionalità sviluppate nel prototipo e mostrano i potenziali vantaggi di un file system semantico rispetto ad uno tradizionale, ma anche alcuni punti critici che vengono esposti durante l'analisi.

La spiegazione si avvale di alcune schermate di *Futon*[53], un'interfaccia grafica per gestire il database CouchDB: questo strumento torna molto utile per controllare il buon esito delle operazioni eseguite nel codice Python e vedere se i dati sono stati salvati correttamente.

53 http://answers.oreilly.com/topic/1395-introduction-to-couchdbs-futon-administration-interface/

5.1.1 Creazione dell'istanza di Insieme

La creazione di un'istanza di Insieme richiede solamente due parametri: il nome dell'utente e se si vuole recuperare un database esistente oppure crearlo da zero. Il codice per eseguire tale operazione è:

```
utente = 'marco'
retrieve = False
insieme = Insieme(utente, retrieve=retrieve)
```

[Codice 1]

Specificando *retrieve=False*, si crea da zero l'istanza per l'utente, eliminando possibili database precedentemente creati con il nome dell'utente; durante la creazione vengono analizzati e salvati nel database anche i vocabolari di RDF, RDFS, OWL, XSD e il vocabolario proprio di *Insieme*. Altro elemento caricato sono i frammenti di codice javascript che definiscono le viste specifiche per CouchDB.
Il successo dell'operazione di creazione è osservabile tramite Futon, dove si vedono i documenti caricati, ed in particolar modo i namespace e le triple appartenenti agli schemi dei vocabolari.

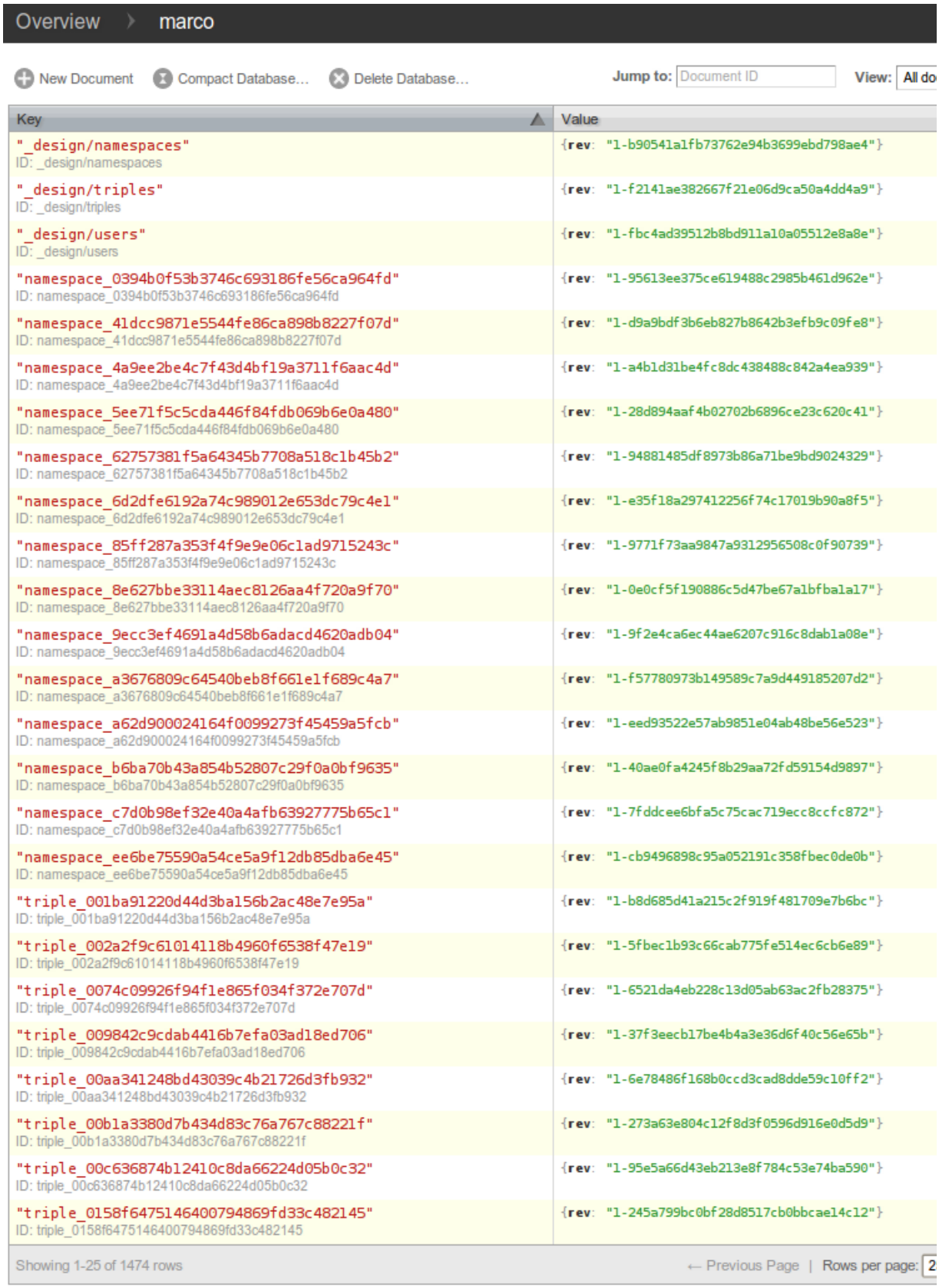

Illustrazione 30: Schermata di Futon dei documenti del database

Accedendo ad una delle triple è possibile verificare la correttezza dei dati inseriti. Di seguito è riportata la tripla in cui viene definita la classe *rdfs:Class*:

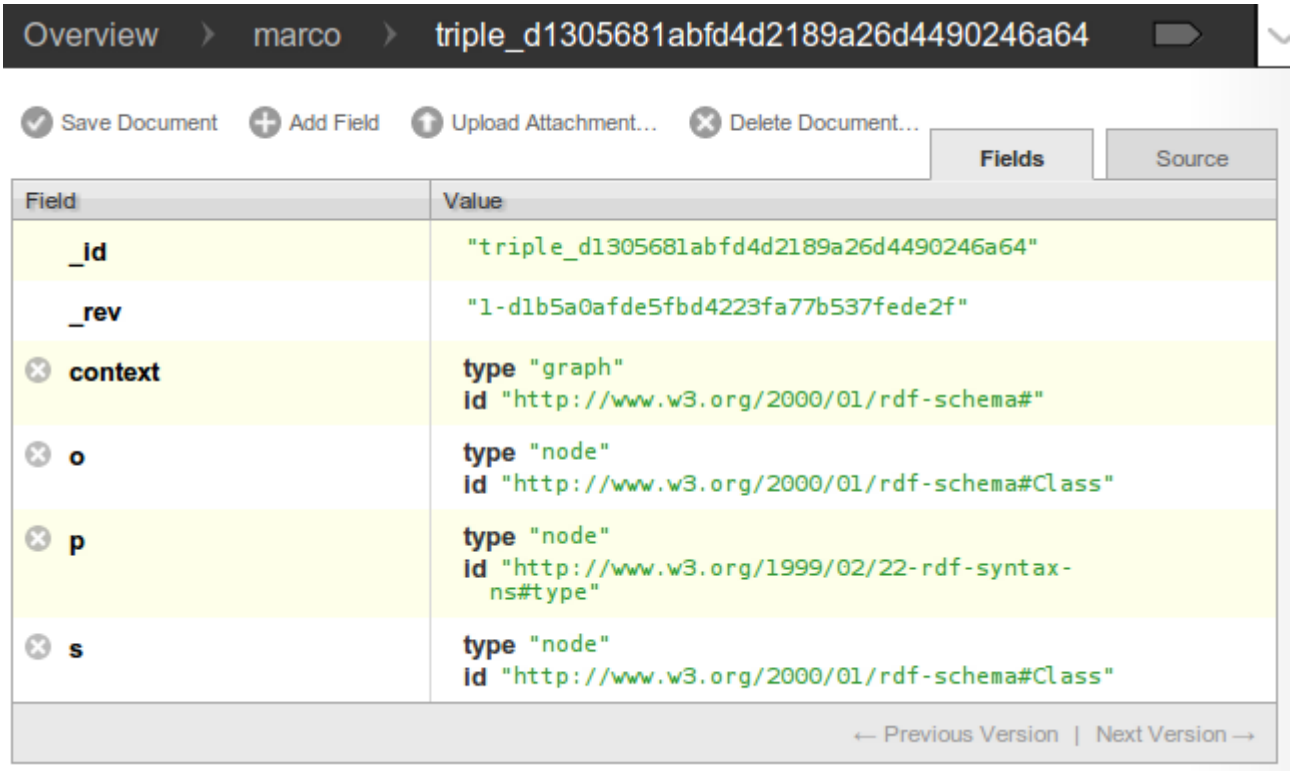

Illustrazione 31: Schermata di Futon di una tripla

Nel caso l'utente avesse già creato un database e volesse recuperarlo, bisogna settare l'opzione *retrieve=True*, evitando perciò di dover scrivere tutte le triple dei namespace caricati in precedenza e, non meno importante, perdere tutti i contenuti precedentemente creati.

5.1.2 Creazione di classi

La creazione di classi da parte dell'utente avviene eseguendo il metodo *create_resource()* dell'istanza di *Insieme*: nell'esempio l'utente vuole creare due classi in particolare, "*Foto_Vacanze*", in cui mettere tutte le foto scattate durante le vacanze, e la classe "*Messaggi*", per i messaggi ricevuti.

```
insieme.create_resource('Foto_Vacanze', RDFS.Class)
insieme.create_resource('Messaggi', RDFS.Class)
```

[Codice 2]

Tale funzione si rende necessaria per la creazione di risorse che ancora non esistono nel grafo dell'utente: se si cercasse infatti di utilizzare il metodo *statement()* dell'istanza di *Insieme*, questa solleverebbe un'eccezione, in quanto,

controllando la presenza degli elementi specificati per formare la tripla all'interno dei vari grafi salvati sul database, non li troverebbe.

Proseguendo con l'esempio, bisogna specificare che la classe "*Foto_Vacanze*" è una sottoclasse della classe "*Insieme:Image*", mentre la classe "*Messaggi*" è una sottoclasse della classe "*Insieme:Text*":

```
insieme.statement(insieme.namespace['Foto_Vacanze'],
RDFS.subClassOf, INSIEME.Image)
insieme.statement(insieme.namespace['Messaggi'],
RDFS.subClassOf, INSIEME.Text)
```

[Codice 3]

Con il metodo *statement()* si attivano i blocchi logici, delineati nel Capitolo 5, di inserimento del contenuto e, soprattutto, di logica di controllo: quando viene creato uno statement, si controlla che il soggetto e l'oggetto rientrino nel dominio e codominio del predicato, sollevando un'eccezione in caso negativo. Che le operazioni (codice 2 e codice 3) siano andate a buon fine lo attestano le schermate di Futon relative ai documenti della classe "*Foto_Vacanze*" (discorso identico per la classe "*Messaggi*"):

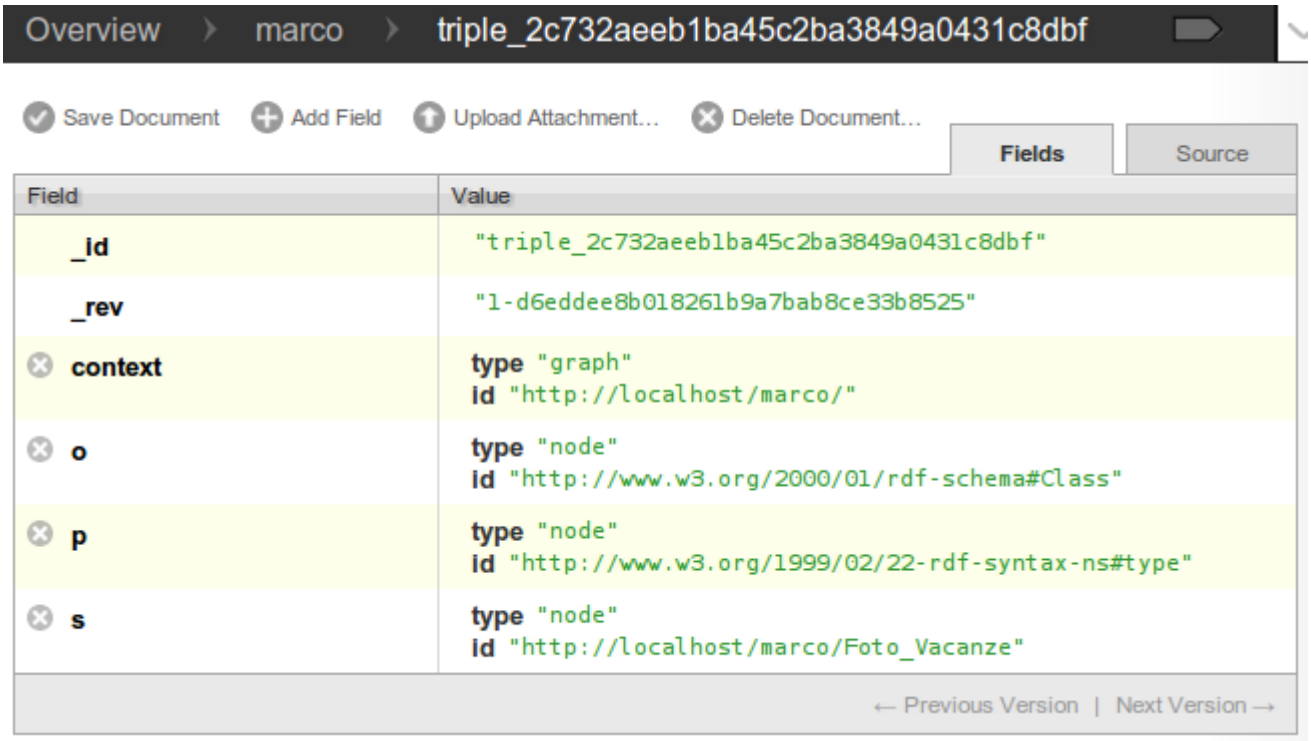

Illustrazione 32: Il documento della classe "Foto_Vacanze"

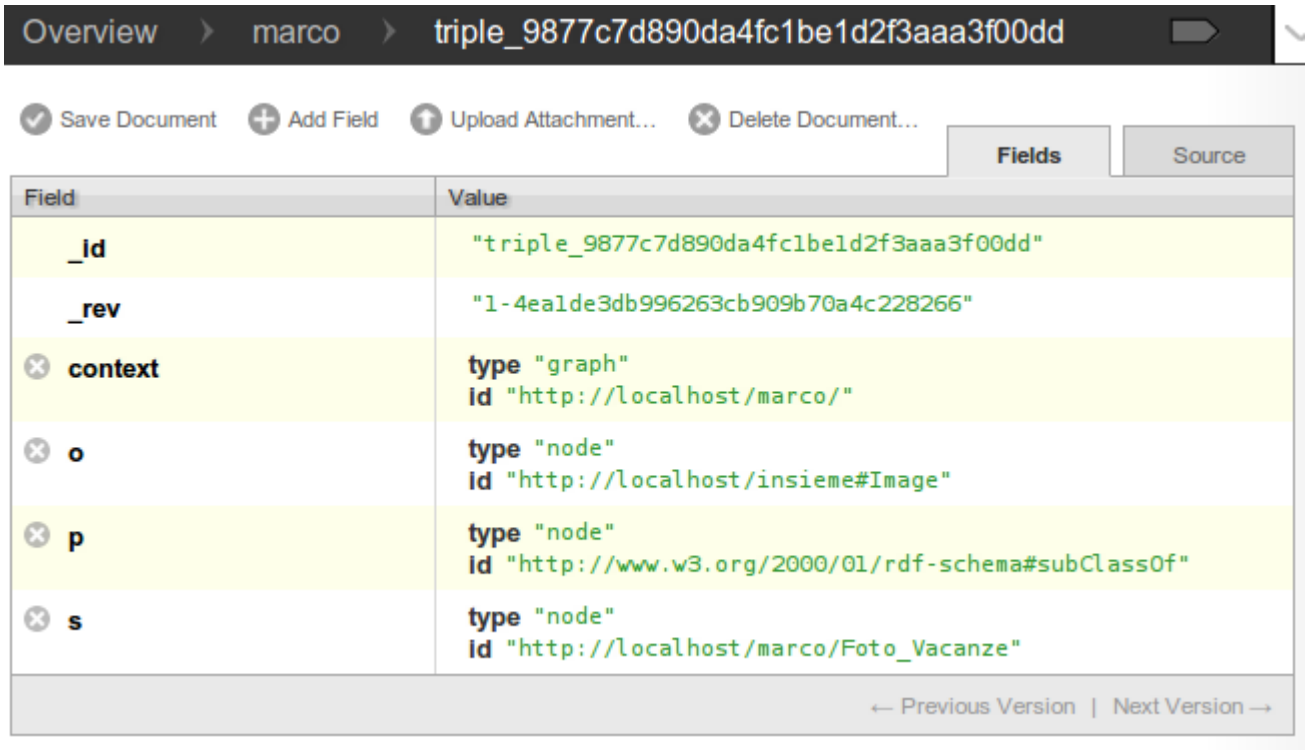

Illustrazione 33: Il documento della proprietà "rdfs:subClassOf"

Il codice 2 ed il codice 3 vedono l'uso delle funzioni *create_resource()* e *statement()*, funzioni che rappresentano il blocco logico "Inserimento contenuti"; i metodi *in_domain()* e *in_range()* appartengono invece al blocco "Logica di controllo", senza però renderlo completo, in quanto rimangono da implementare altre funzioni di controllo.

5.1.3 Creazione delle istanze di classe

Si passa ora alla creazione delle istanze delle classi appena costruite: come esplicitato nell'esempio, l'utente Marco vuole agganciare al sistema dei file per poterli annotare. Tale operazione viene svolta sempre attraverso il metodo *create_resource()* visto in precedenza.
Si riporta di seguito, per motivi di sintesi, solamente il codice per la creazione della prima foto e del primo messaggio:

```
insieme.create_resource('Foto_10',
insieme.namespace['Foto_Vacanze'], 'esempio/foto10.jpg')
insieme.create_resource('Messaggio_1',
insieme.namespace['Messaggi'], 'esempio/messaggio01.txt')
```

[Codice 4]

Si nota come, a differenza del codice 2, il metodo *create_resource()* (codice 4) presenta un parametro in più, che è il percorso relativo del file che si vuole agganciare ad *Insieme*; il metodo si preoccupa di controllare che il percorso

passato faccia riferimento ad un file valido, altrimenti viene sollevata un'eccezione.
In caso il file passato sia valido, il sistema si occupa di creare due triple:

- una tripla che definisce l'appartenenza alla classe
- una tripla in cui si annota la risorsa con la proprietà Insieme:related_file, in cui il valore della proprietà è l'indirizzo del file

Sempre guardando Futon, si può osservare il buon esito dell'inserimento di quest'ultima operazione relativa alla prima foto:

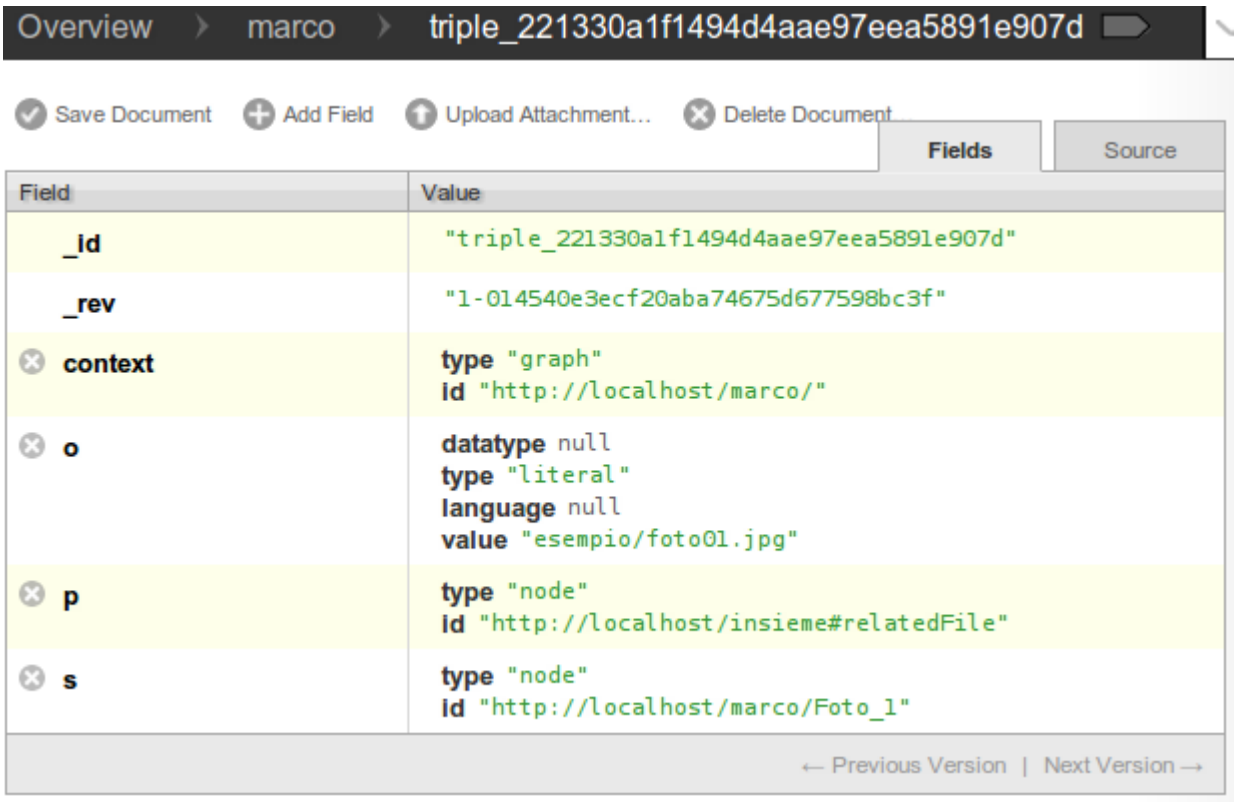

Illustrazione 34: Il documento della proprietà "Insieme:relatedFile"

5.1.4 Aggiunta di un vocabolario

Il passo successivo dell'esempio vede l'utente Marco voler annotare i file appena caricati con la rappresentazione virtuale dei suoi amici; perché ciò sia possibile sono necessari alcune operazioni preliminari, prima di tutte è l'importazione del vocabolario FOAF per istanziare le rappresentazioni virtuali degli amici, sfruttando la classe Foaf:Person.

Si osserva innanzitutto che il tentare di utilizzare un vocabolario non ancora inserito dentro Insieme ritorna errore. Cercando di creare la persona "Alex" con il seguente codice:

```
insieme.statement(insieme._prefixes['foaf'].Person,
insieme._prefixes['foaf'].name, "Alex")
```
[Codice 5]

viene sollevata un'eccezione, bloccando di fatto il comando eseguito. Ciò è dovuto all'assenza del prefisso "*foaf*" nella lista dei vocabolari disponibili, denominata "*_prefixes*" (come si può osservare nel codice 5)

Per inserire un nuovo vocabolario, in questo caso quello FOAF, si utilizza il metodo bind() dell'istanza di Insieme (codice 6). Tale metodo necessita di tre parametri:

- il prefisso da dare al vocabolario
- l'URI base del vocabolario
- il file testuale contenente il vocabolario con tutti i vocaboli

Il comando completo è:

```
insieme.bind('foaf', 'http://xmlns.com/foaf/0.1/',
'namespaces/foaf.n3')
```
[Codice 6]

La schermata di Futon in cui si vede il namespace FOAF salvato:

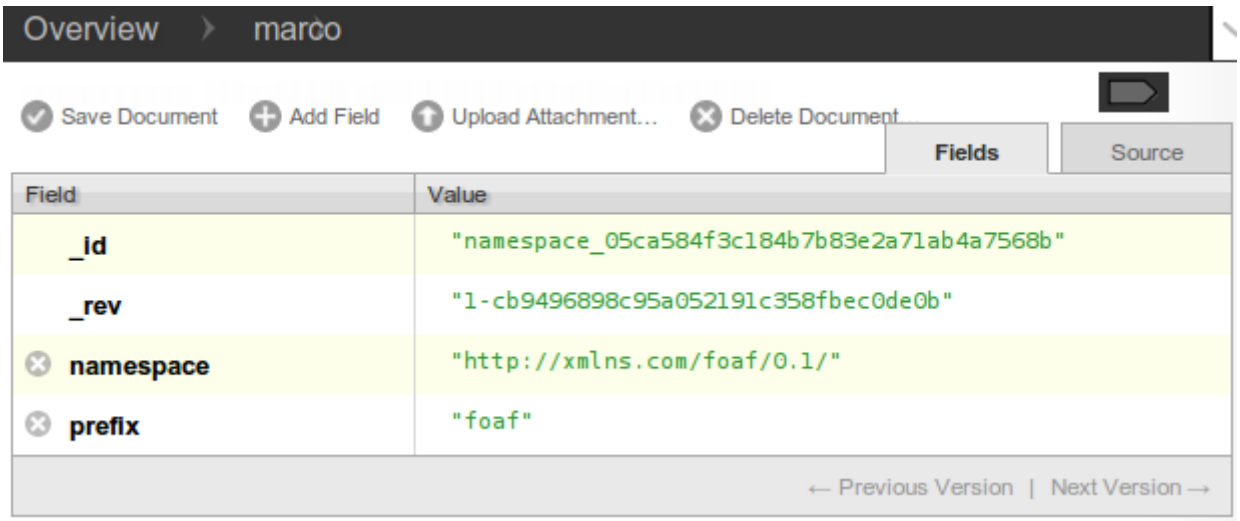

Illustrazione 35: Il documento del namespace "Foaf"

Rieseguendo il codice 5, l'operazione viene eseguita correttamente, confermata anche dalla schermata di Futon:

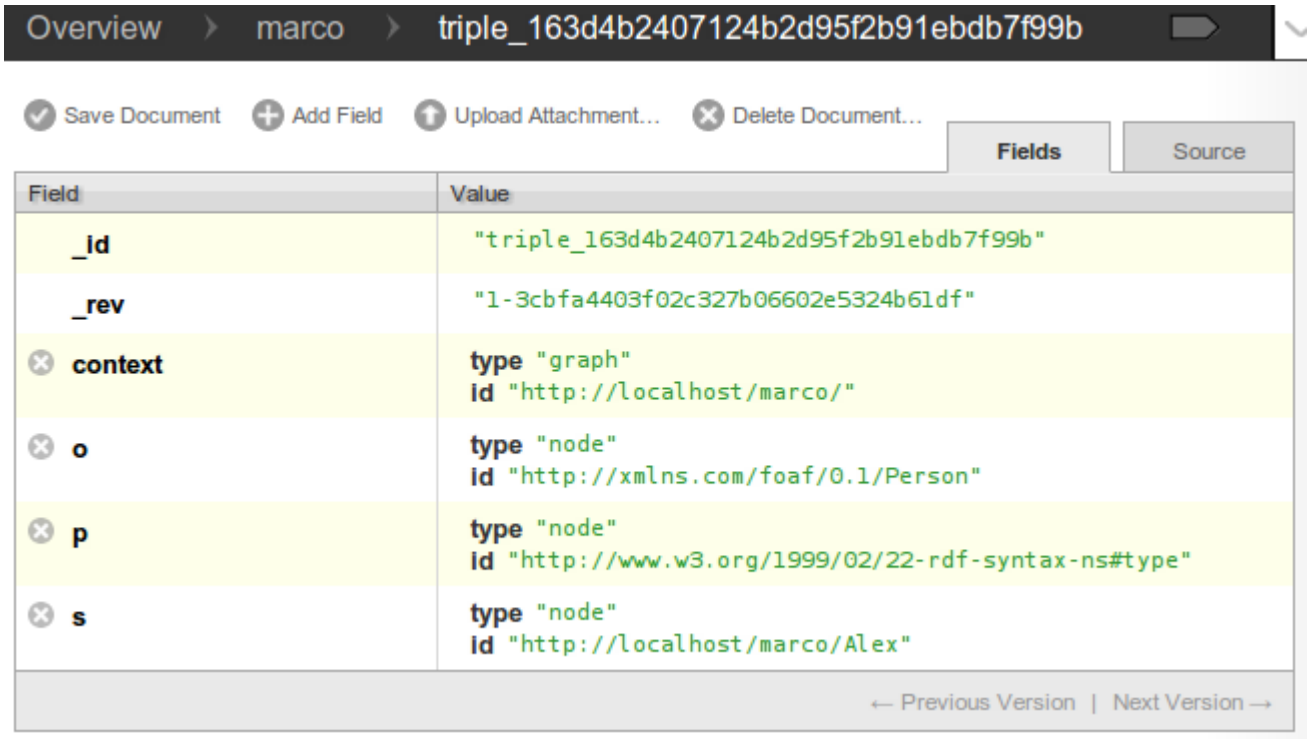

Illustrazione 36: Il documento della persona "Alex"

Seguendo la stessa metodologia si inseriscono anche i rimanenti amici.

Una delle funzioni di controllo eseguite durante l'inserimento è la rilevazione di introduzioni doppie della stessa tripla: viene infatti proibito, per evitare problemi di consistenza e logici, l'inserimento di una stessa tripla più volte sollevando un'eccezione a livello di codice. Ecco l'output da riga di comando quando si tenta di inserire nuovamente la persona "*Chiara*":

```
Il nome specificato è già stato usato.
----------------------------------------
```

Illustrazione 37: Errore per la doppia creazione della persone "Chiara"

5.1.5 Creazione di proprietà

Altra caratteristica importante di Insieme è la possibilità di definire, all'interno del grafo dell'utente, nuove proprietà da poter utilizzare per l'annotazione, definendone inoltre dominio e codominio.
Questa funzionalità viene usata dall'utente Marco per definire due proprietà:

- la proprietà "*include_persona*", per dire che una foto ha dentro una certa persona
- la proprietà "*mandato_da*", per dire che un messaggio è stato mandato da una certa persona

Innanzitutto si creano le proprietà definendole come istanze di *rdf:Property* attraverso il metodo *create_resource()*:

```
insieme.create_resource('include_persona', RDF.Property)
insieme.create_resource('mandato_da', RDF.Property)
```

[Codice 7]

Poi si definiscono dominio e codominio attraverso la creazione di uno statement, come si vede di seguito per la proprietà "*include_persona*":

```
insieme.statement(insieme.namespace['include_persona'],
RDFS.domain, INSIEME.Image)
insieme.statement(insieme.namespace['include_persona'],
RDFS.range, insieme._prefixes['foaf'].Person)
```

[Codice 8]

in cui si specifica che il dominio sono solo le istanze di Insieme:Image (o istanze di sue sottoclassi) ed il codominio sono solo le istanze di foaf:Person (o istanze di sue sottoclassi).
Procedimento analogo per la proprietà "*mandato_da*".

5.1.6 Annotazione con nuove proprietà

L'utente Marco, dopo aver creato le due proprietà "*include_persona*" e "*mandato_da*", procede con l'annotare i contenuti inseriti inizialmente: tale operazione si svolge, come negli esempi precedenti, con il metodo *statement()*.

Si riportano di seguito due tentativi di inserimento di triple non valide, una perché il soggetto specificato non rientra nel dominio ed una perché l'oggetto non rientra nel codominio:

```
insieme.statement(insieme.namespace['Foto_1'],
insieme.namespace['mandato_da'],
insieme.namespace['Federica'])
insieme.statement(insieme.namespace['Foto_8'],
insieme.namespace['include_persona'], 'Federica')
```

[Codice 9]

La logica di controllo impedisce tali inserimenti, in quanto nella prima tripla si cerca di applicare la proprietà "*mandato_da*" ad una foto (il dominio della proprietà permette solo istanze di *Insieme:Text*), mentre nella seconda si cerca di ingannare il sistema mettendo il nome "*Federica*" sotto forma di stringa (è invece richiesto l'URI dell'istanza di *foaf:Person*). Ecco gli output a riga di comando degli errori ritornati:

Illustrazione 38: Errore per il dominio

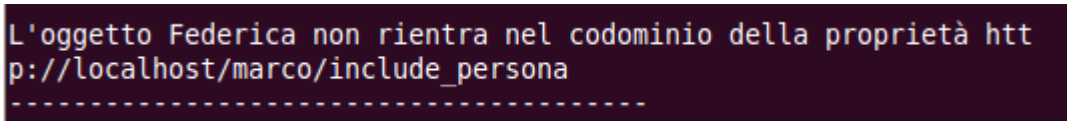

Illustrazione 39: Errore per il codominio

5.1.7 Query

L'ultima operazione rimasta e che interessa il blocco "Ricerca contenuti" è quello della query. Si mostra tale funzionalità riprendendo le query che l'utente Marco vuole eseguire:

- query "tutte le foto"
- query "tutte le foto con Alex"
- query "tutti i contenuti con Alex come oggetto"
- query "tutti i messaggi mandati da una persona il cui nome contiene le lettere 'Gi' "

Per eseguire le query, ci si avvale di una classe all'interno del modulo "*couchdb_store.py*", modulo in cui si trova inoltre il codice che rende utilizzabile il database *CouchDB* come magazzino di triple; la classe in questione è *CouchdbQuery*, che si preoccupa di:

- costruire la query
- eseguirla
- ritornare i risultati ripulendoli dagli elementi non richiesti

Ecco l'esempio di codice che serve per definire la prima query:

```
query =
couchdb_store.CouchdbQuery(insieme.namespace._namespace).where
(property=RDF.type,
value=insieme.namespace['Foto_Vacanze']).end_query('s','null')
```

[Codice 10]

Si vede che l'oggetto *query*:

- viene creato come istanza di *couchdb_store.CouchdbQuery* specificando il namespace dell'utente (parametro *insieme.namespace._namespace*)
- viene definito attraverso la clausola *where()* in cui si richiedono tutte le triple in cui compare la proprietà *rdf:type* con valore *insieme.namespace['Foto_Vacanze']*

- viene chiusa con la clausola *end_query()* specificando che i valori da ritornare sono "*s*" (cioè il soggetto della tripla) e "*null*" (valore nullo)

Il codice 10 serve per la costruzione della query, che viene salvata nell'attributo interno *query* dell'istanza di *CouchdbQuery* sotto forma di stringa:

```
function(doc) {
    if (doc._id.match(/^triple_/) &&
doc.context.id=='http://localhost/marco/'){
       if(doc.p.id == 'http://www.w3.org/1999/02/22-rdf-
syntax-ns#type' &&
     doc.o.id == 'http://localhost/marco/Foto_Vacanze' ){
          emit(doc.s,null);
       }
    }
}
```

[Codice 11]

La sua esecuzione avviene richiamando il metodo *execute()*, che, passando come parametro il grafo su cui eseguire la query, ritorna una lista dei risultati che rispondono alle condizioni imposte:

```
for row in query.execute(insieme._graph):
     print (row)
```

[Codice 12]

L'output a riga di comando del codice 12 è:

```
RISULTATI:
({'type': 'node', 'id': 'http://localhost/marco/Foto_1'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_10'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_2'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_3'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_4'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_5'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_6'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_7'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_8'}, None)
({'type': 'node', 'id': 'http://localhost/marco/Foto_9'}, None)
------------
```

Illustrazione 40: Risultati della prima query

La seconda query (codice 13) mostra l'utilizzo della clausola *_and()*, che permette di definire ulteriori restrizioni sui contenuti cercati. Il codice della seconda query è perciò:

```
query =
couchdb_store.CouchdbQuery(insieme.namespace._namespace).
where(property=RDF.type,
value=insieme.namespace['Foto_Vacanze']).
_and(property=insieme.namespace['include_persona'],
value=insieme.namespace['Alex']).end_query('s','null')
```

[Codice 13]

Con tale query si uniscono perciò le triple che presentano lo stesso soggetto (che è il contenuto che si sta ricercando) ed imponendo condizioni su proprietà diverse (cioè appartenenti a triple distinte). Il risultato della query è:

```
RISULTATI:
{'type': 'node', 'id': 'http://localhost/marco/Foto_1'}
{'type': 'node', 'id': 'http://localhost/marco/Foto_4'}
{'type': 'node', 'id': 'http://localhost/marco/Foto_8'}
-------------
```

Illustrazione 41: Risultati della seconda query

e, andando a controllare su Futon, si può verificare la veridicità dei risultati:

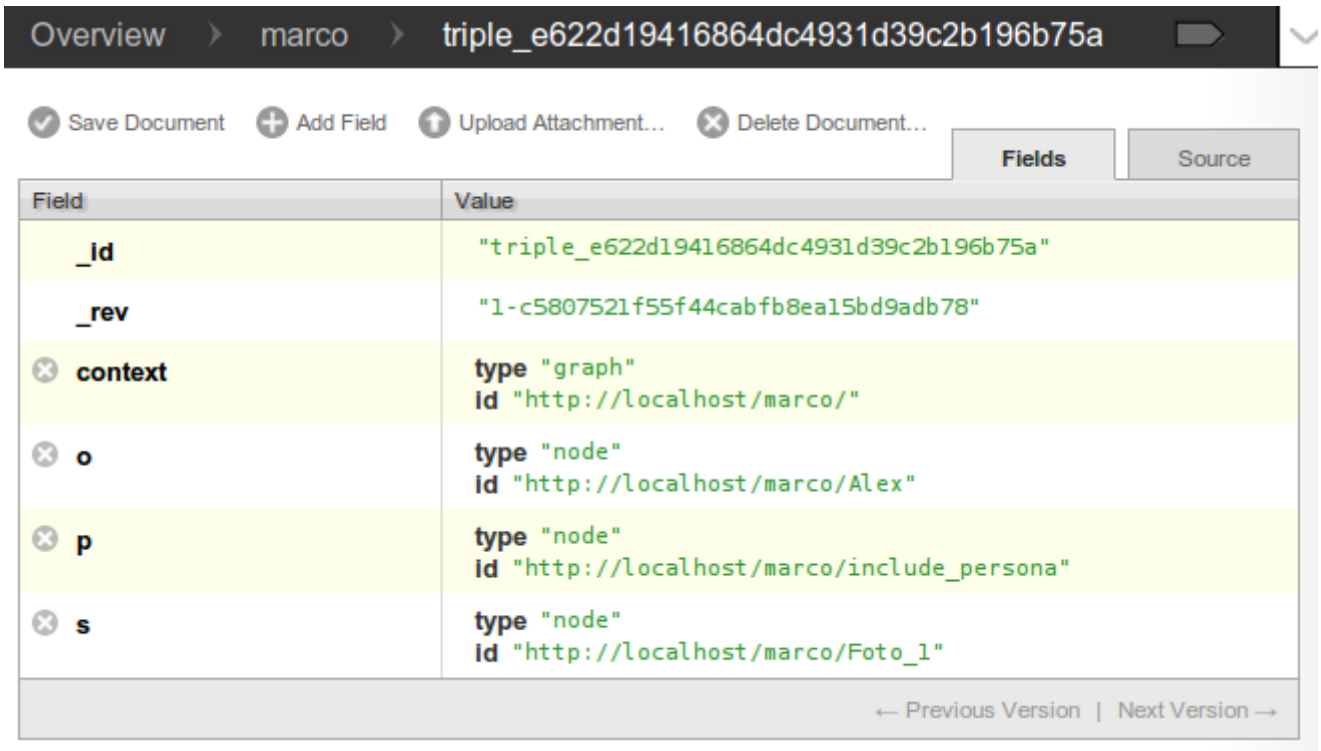

Illustrazione 42: Tripla appartenente ai risultati della seconda query

6. Conclusioni

Il futuro della gestione dati, ed in generale l'interazione con i calcolatori, è legata sempre di più agli sviluppi nel campo della *Rappresentazione della Conoscenza*. Lo testimonia il numero costantemente crescente di soluzioni che cercano di porre rimedio alla staticità imposta dai normali file system, soluzioni quali database, sistemi di gestione integrata dei documenti, riproduttori musicali in grado di organizzare la propria libreria musicale, siti web che permettono di taggare i contenuti trovati in rete, etc.

In questo lavoro si è studiato pertanto un aspetto ritenuto basilare per una transizione verso tale futuro, i *file system semantici*, analizzandone i concetti teorici e lo stato dell'arte attuale, per poi sviluppare una proposta basata sulla conoscenza acquisita.
Il sistema creato, denominato *Insieme*, risulta essere allo stato attuale solamente un prototipo (quindi ancora inutilizzabile da un utente comune) e passibile, in sviluppi futuri, di profondi cambiamenti; ma è proprio grazie a tale prototipo che è possibile individuare i punti critici in cui è necessario intervenire, non solo a

livello di codice software, ma anche a livello teorico. Le criticità riscontrate sono le seguenti:

- la scelta di basare la rappresentazione dei contenuti attraverso RDF presenta alcuni svantaggi e limitazioni, dovuti soprattutto al fatto che RDF è pensato principalmente per lo scambio di dati tra calcolatori; si ritiene perciò che lo studio di una metodologia diversa di salvataggio dei dati possa migliorare sia la facilità di gestione degli stessi, che le prestazioni a livello computazionale
- strutturare i contenuti come se appartenessero ad un grafo è reputato corretto, ma migliorabile utilizzando concetti teorici derivanti dalla *Teoria dei Grafi*[54] e, per quanto riguarda il software, librerie pensate specificatamente per eseguire operazioni su grafi
- è considerata altamente critica la necessità di uno studio approfondito sulle interfacce con le quali il sistema interagisce con l'utente, essendo un sistema radicalmente diverso da quelli tradizionalmente usati: l'obiettivo, da giudicarsi primario, è quello di realizzare un'interfaccia intuitiva ed in grado di aiutare l'utente ad eseguire le stesse operazioni nello stesso tempo (se non più velocemente) delle interfacce classiche
- per gli sviluppi futuri è consigliabile studiare ed utilizzare il *Virtual File System* di Linux, potendo in questo modo avvicinarsi di più al problema della creazione di un file system vero e proprio

Oltre ai punti sopra riportati, ve ne sono altri più attinenti al codice del prototipo creato:

- migliorare il motore delle query, in quanto, oltre alla possibilità di imporre solo condizioni su predicato ed oggetto di una tripla, presenta comportamenti errati nell'esecuzione di alcuni tipi di interrogazioni: questo fatto è dovuto ad una limitazione di CouchDB, che non prevede il concetto di "join" dei risultati, costringendo lo sviluppo di una soluzione che si presenta imperfetta
- se si riterrà valido l'uso di CouchDB per l'archiviazione dei dati, si può studiare un metodo diverso e più efficiente di salvataggio dei dati, non più come triple ma sfruttando il concetto di documento insito in CouchDB:
  - creare un documento per ogni risorsa che compaia come soggetto di una tripla e salvare tutte le triple relative ad esso dentro il documento
  - creare delle viste per indicizzare le triple rispetto ai predicati e gli oggetti, evitando quindi di doverle estrarre manualmente dai vari documenti

In conclusione, le questioni da affrontare sono sicuramente molte, altre ne emergeranno man mano che si procede, e probabilmente è necessario un interessamento maggiore da parte delle grandi software house affinché si possa

54 http://it.wikipedia.org/wiki/Teoria_dei_grafi

trovare una soluzione valida ed utilizzabile nella vita di tutti i giorni.
Ciò nonostante, le potenzialità che un *file system semantico* ha da offrire, soprattutto se considerate in una visione di condivisione della conoscenza attraverso una rete personale, sono indubbiamente molte, ed i segnali che un modo diverso di organizzare i dati è sentita, anche in ambito lavorativo, sempre più come una necessità, fanno presagire che l'argomento dei *file system semantici* sarà la "next big thing" nel campo dell'informatica.

Appendice A: codice Python

Codice 14:

```
1. graph.bind(‘RDF', ‘http://www.w3.org/1999/02/22-rdf-
   syntax-ns#’)
```

Codice 15:

```
1. RDF.type
```

Codice 16:

```
1. if self.configuration['db_name'] in self.server:
2.   self.db = self.server[self.configuration['db_name']]
3. else:
4.   if create:
```

```
5.      self.db=self.server.create(self.configuration['db_na
   me'])
6.    else:
7.      raise CouchdbException()
```

Codice 17:

```
1. def element_to_db(element):
2.   if isinstance(element, Graph):
3.     return {'type':'graph', 'id':
   str(element.identifier)}
4.
5.   if isinstance(element, URIRef):
6.     return {'type':'node', 'id':str(element)}
7.
8.   if isinstance(element, BNode):
9.     return {'type':'bnode', 'id':str(element)}
10.
11.   if isinstance(element, Literal):
12.     return {'type':'literal',
   'value':element.toPython(), 'datatype': element.datatype,
   'language':element.language}
13.
14.   return None
```

Codice 18:

```
1. def element_from_db(element):
2.   if element['type'] == 'graph':
3.     return element['id']
4.
5.   if element['type'] == 'node':
6.     return URIRef(element['id'])
7.
8.   if element['type'] == 'bnode':
9.     return BNode(element['id'])
10.
11.   if element['type'] == 'literal':
12.     return Literal(element['value'],datatype =
   element['datatype'], lang = element['language'])
```

Codice 19:

```
1. function(doc) {
2.   if (doc._id.match(/^triple_/) && doc.s){
3.     emit(doc.s, {s:doc.s, p:doc.p, o:doc.o, context:
   doc.context});
4.   }
5. }
```

Codice 20:

```
1. from rdflib import plugin
2. plugin.register('couchdb', Store, 'couchdb_store',
   'CouchdbStore')
```

Codice 21:

```
1. self._store = couchdb_store.CouchdbStore(configuration,
   create=True)
```

Codice 22:

```
1. self._graph = Graph(store=self._store,
   identifier='http://localhost/%s/' % user)
```

Codice 23:

```
1. self.namespace = Insieme_namespace('http://localhost/
   %s/' % user, 'local', self._graph)
2. self._namespaces[self.namespace._namespace] =
   self.namespace
3. self._graph.bind('local', self.namespace._namespace)
```

Codice 24:

```
1. self.bind('rdf', 'http://www.w3.org/1999/02/22-rdf-
   syntax-ns#', 'namespaces/rdf.rdf', override= not
   retrieve)
2. self.bind('rdfs', 'http://www.w3.org/2000/01/rdf-
   schema#', 'namespaces/rdfs.rdf', override= not retrieve)
3. self.bind('owl', 'http://www.w3.org/2002/07/owl#',
   'namespaces/owl.rdf', override= not
   retrieve)self.bind('xsd',
   'http://www.w3.org/2001/XMLSchema#', 'namespaces/xsd.n3',
   override= not retrieve)
4. self.bind('insieme', 'http://localhost/insieme#',
   'namespaces/ontologia_base.ttl', override= not retrieve)
```

Bibliografia

[*1*]: Autori vari, "File system" (url: http://it.wikipedia.org/wiki/File_system)

[*2*]: Autori vari, "*Microsoft Extensible Firmware Initiative FAT32 File System Specification*" (2000), Microsoft Corporation (url: http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx)

[*3*]: Autori vari, "*Inode Definition*" (2006), The Linux Information Project (url: http://www.linfo.org/inode.html)

[*4*]: M. Tim Jones, " *Anatomy of the Linux file system*" (2007), IBM developerWorks, (url: http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/)

[5]: Miro Dogliotti e Luigi Rosiello, "*Lo Zingarelli 1999 : vocabolario della lingua italiana di Nicola Zingarelli*" (1999), Bologna: Zanichelli

[*6*]: Randall Davis, Howard Shrobe and Peter Szolovits, " *What Is a Knowledge Representation?*" (1993), AI Magazine, 17 (url: http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1029)

[*7*]: Autori vari, "Knowledge representation and reasoning" (url: http://en.wikipedia.org/wiki/Knowledge_representation)

[*8*]: Tim Berners-Lee, James Hendler and Ora Lassila, " *The Semantic Web*" (2001), Scientific American Magazine, (url: http://www.scientificamerican.com/article.cfm?id=the-semantic-web)

[*9*]: Autori vari, "*RDF Primer*" (2004), W3C (url: http://www.w3.org/TR/2004/REC-rdf-primer-20040210/)

[*10*]: Autori vari, "*OWL 2 Web Ontology Language Primer*" (2009), W3C (url: http://www.w3.org/TR/2009/REC-owl2-primer-20091027/)

[*11*]: Autori vari, "*OWL 2 Web Ontology Language Profiles*" (2009), W3C (url: http://www.w3.org/TR/owl2-profiles/)

[12]: Stephan Bloehdorn, Olaf Görlitz, Simon Schenk, Max Völkel, "*TagFS - Tag Semantics for Hierarchical File Systems*" (2006), Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) (url: http://www.aifb.kit.edu/web/Inproceedings1140/en)

[*13*]: Seth Nickell, "*A Cognitive Defense of Associative Interfaces forObject Reference*" (2004), (url: http://people.gnome.org/~seth/storage/associative-interfaces.pdf)

[*14*]: O. Gorter, "*Database File System — An Alternative to Hierarchy Based File Systems*" (2004), University of Twente, Computer Sciences (url: http://tech.inhelsinki.nl/dbfs/)

[*15*]: Jörg Richter, Jurij Poelchau, "*DeepaMehta — Another Computer is Possible*" (2007), (url: http://www.deepamehta.de/)

[*16*]: Robert Freund, "*File Systems and Usability - the Missing Link*" (2007), University of Osnabruck (url: http://rffr.de/nhfs)

[*17*]: David Ingram, "*Insight: A Semantic File System*" (2008), Imperial College London (url: http://www.dmi.me.uk/code/insight/)

[*18*]: Mallik Mahalingam, Chunqiang Tang, Zhichen Xu, "*Towards a Semantic, Deep Archival File System*" (2002), HP Laboratories Palo Alto (url: http://www.hpl.hp.com/techreports/2002/HPL-2002-199.html)

[*19*]: Autori vari, "WinFS" (url: http://en.wikipedia.org/wiki/WinFS)

[*20*]: Autori vari, "*NEPOMUK Final Activity Report*" (2009), German Research Center for Artificial Intelligence (DFKI) GmbH (url: http://nepomuk.semanticdesktop.org)

[*21*]: Daniel Bricklin, " *Friend-to-Friend Networks*" (2000), , (url: http://www.bricklin.com/f2f.htm)

[*22*]: Autori vari, "*Introduction to CouchDB Views*" (2010), Apache (url: http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views)