POLITECNICO DI MILANO
Facoltà di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

SLD4SM : un ambiente integrato per la gestione di Stream di Social Media tramite Linked Data

Relatore: Prof. Emanuele Della Valle
Correlatore: Ing. Davide Francesco Barbieri

Tesi di Laurea Specialistica di:
Marco Balduini - Matricola 731284
Mirko Bratomi - Matricola 725117

Anno Accademico 2010/2011

Alle nostre famiglie...
e alla nostra amicizia.

Ringraziamenti

Un particolare ringraziamento a tutta la mia famiglia, che mi è sempre stata vicina, mi ha sostenuto e aspettato in questi anni.

A Bea, la mia anima gemella, che mi sta sempre accanto anche quando sono insopportabile e che mi ha accompagnato per mano fino a questo traguardo.

A Mirko, mio compagno di lavoro, ma soprattutto amico che ha condiviso con me questo sforzo. Grazie di tutto.

A tutti gli amici che mi sono stati vicini, ringrazio anche quelli che ho perso un po di vista per mille ragioni.

Un particolare ringraziamento e un ricordo ai familiari che sono venuti ultimamente a mancare, questo lavoro e questo traguardo raggiunto è dedicato a voi.

Grazie a Emanuele e a Davide che mi hanno aiutato, con la loro competenza e con i loro insegnamenti, in questo ultimo lavoro, che è il coronamento di tutti i miei anni di studio. Quello che mi avete insegnato va oltre il semplice lavoro universitario.

Grazie a tutti.

Marco

Dopo quarantanove esami e tanto sudore mi ritrovo, finalmente, a scrivere questi ringraziamenti che sanciscono la fine del mio percorso di studi.

Il mio primo ringraziamento non può che andare alla mia famiglia, in particolare ai miei genitori. Grazie papà e grazie mamma per avermi sempre supportato (e sopportato) in questi anni. Grazie per non avermi mai caricato di eccessive responsabilità e per esserci sempre stati. Grazie per avermi trasmesso la voglia di imparare sempre qualcosa di nuovo. Grazie per la vostra passione nel seguirmi. Vi adoro.

Grazie a Roberta, la mia musa, che è sempre accanto a me e mi regala tanto amore. Grazie anche ai suoi genitori per averla messa al mondo.

Grazie agli amici di sempre, in particolare ad Andrea. Sei più che un grande amico, sei un fratello.

Grazie a tutti gli amici conosciuti al Politecnico, mi spiace non poter riportare tutti i vostri nomi ma voi sapete di essere importanti per me. Tra di voi, in particolare, vorrei ringraziare Stefano. Il primo giorno di lezione ti sei seduto accanto a me e da allora non ci siamo più persi. Sei stato un grande compagno di avventura e sarai per sempre un grande amico.

Grazie a Marco. Sei un grande amico e una persona stupenda. Grazie di tutto.

L'ultimo, ma non meno importante, ringraziamento va a Emanuele e Davide che, con tanta pazienza e tanto coinvolgimento, mi hanno aiutato a raggiungere quest'ultimo traguardo. Siete stati importantissimi con la vostra competenza e la vostra simpatia. Mi auguro di conoscere molte persone come voi. Grazie.

Grazie a tutti.

Mirko

Prefazione

Recentemente nel mondo della rete si sta assistendo ad un'esplosione dei Social Media, ossia di quei servizi che permettono a qualsiasi utente di diventare, oltre che un fruitore, un creatore di informazione, rendendo il Web sempre più aperto verso i propri utilizzatori.

L'analisi dei Social Media offre la possibilità di percepire quali siano gli argomenti che più attirano l'attenzione e quale sia l'opinione su di essi da parte degli utenti: analizzando le informazioni è possibile dedurre i personaggi più in voga, i cantanti più seguiti, i libri più letti, i luoghi più visitati dai turisti, etc. Il tipo di dato che meglio rappresenta questo flusso continuo di dati è lo stream.

La necessità di utilizzo degli stream ha portato, agli fine degli anni novanta, allo sviluppo dei *Data Stream Management System* (DSMS) da parte della comunità delle basi di dati, ovvero di quei sistemi software in grado di gestire e interrogare stream di dati. Parte del nostro lavoro ha riguardato la sperimentazione di questi sistemi.

Un'altra categoria di tecnologie che si stanno rivelando di crescente interesse per l'analisi dei Social Media sono quelle che appartengono alla sfera del Web Semantico. Con la nascita del Web Semantico, infatti, si è passati ad una diversa modalità di pubblicazione e fruizione dei contenuti informativi presenti sul Web che sta ulteriormente evolvendo con le potenzialità offerte dagli stream.

La barriera principale che rallenta lo sviluppo di tecnologie legate all'utilizzo degli stream in associazione con il Web Semantico è l'estrema disomogeneità dei formati in cui l'informazione viene offerta all'utilizzatore finale.

Lo scopo di questa tesi è stato quello di creare un metodo per analizzare e ripubblicare i dati provenienti dai Social Media in un modello semantico basato sul framework RDF e sul linguaggio ontologico RDF-S. Ciò ha dato vita a *Streaming Linked Data for Social Media* (in breve SLD4SM), un sistema in grado di recuperare i dati dai Social Media e di salvarli, ripubblicarli o interrogarli tramite

query continue (grazie al linguaggio C-SPARQL e allo C-SPARQL Engine sviluppati dal team LarKC del Politecnico di Milano).

L'analisi delle performance di efficienza ed efficacia svolta sul sistema, ci permette di affermare che SLD4SM è un framework performante, scalabile, flessibile ma soprattutto estendibile.

Indice

1 Introduzione **12**
- 1.1 Web semantico e stream reasoning 12
- 1.2 Linked Data e Social Media 14
- 1.3 Stream Reasoning nel progetto LarKC 16
- 1.4 Descrizione del problema affrontato 16
- 1.5 Struttura dell'elaborato 17

2 Stato dell'Arte **18**
- 2.1 Data Stream e DSMS 18
- 2.2 SemanticWeb 21
  - 2.2.1 RDF e RDF-S 23
  - 2.2.2 SPARQL 26
  - 2.2.3 Linked Data 35
  - 2.2.4 Jena 39
- 2.3 C-SPARQL 43
  - 2.3.1 RDFStream 44
  - 2.3.2 Finestre 44
  - 2.3.3 Query Registration 46
  - 2.3.4 Stream Registration 46
  - 2.3.5 C-SPARQL Engine 47
- 2.4 Restlet 48

3 Streaming Linked Data for Social Media: Progetto della Soluzione **50**
- 3.1 Problematiche Affrontate 50
- 3.2 Architettura Generale 54
  - 3.2.1 DataGetter 56
  - 3.2.2 Adapter 56
  - 3.2.3 C-SPARQL Adapter 58
  - 3.2.4 SLD4SM Core 59
- 3.3 SLD4SM e i Design Pattern 63
  - 3.3.1 Ereditarietà e overloading 64

3.3.2 Il design pattern Factory 65
3.3.3 Il design pattern Observer 66
3.3.4 Il design pattern Facade 67
3.4 Conoscenza Stream e Conoscenza Statica 70
3.4.1 Pubblicazione di uno Stream 70
3.4.2 Conoscenza Statica 72
3.5 Rappresentazione grafica dei gestori di Named Graph 74

4 Implementazione Prototipo 77
4.1 Architettura Generale 77
4.1.1 Interfacce e Strutture Dati Principali 77
4.1.2 Adapter 80
4.1.3 C-SPARQL Adapter 86
4.1.4 SLD4SM Core 88
4.1.5 SLD4SM Server 97
4.1.6 Implementazione Factory Method Pattern . . 99
4.2 Conoscenza Statica 101
4.2.1 Conoscenza statica relativa agli utenti 101
4.2.2 Conoscenza statica relativa agli oggetti 102
4.3 Implementazione servizio REST 103

5 Test e Validazione prototipo 108
5.1 Test di efficienza 108
5.1.1 Replayer 108
5.1.2 Recorder 109
5.1.3 Windower 112
5.1.4 C-SPARQL Engine 114
5.2 Test di efficacia sullo C-SPARQL Engine 116

6 Conclusioni e Sviluppi Futuri 120
6.1 Conclusioni 120
6.2 Sviluppi futuri 121
6.2.1 Content Negotiation 122
6.2.2 Aggiunta di nuovi Social Network come fonti di dati 123
6.2.3 Ulteriori funzionalità aggiuntive 124

A Manuale Utente 126
A.1 Installazione dell'applicazione 126
A.2 URI del Servizio Rest 127

B Esempi e casi d'uso significativi 129

Elenco delle figure

2.1 Differenze tra DBMS e DSMS 20
2.2 Esempio di windowing 20
2.3 Struttura del Semantic Web 21
2.4 Schema Tripla RDF 24
2.5 LOD Cloud nel Settembre 2010 37
2.6 Architettura dello C-SPARQL engine 48
2.7 Struttura del Restlet Framework 48

3.1 Schema comune di rappresentazione dell'informazione nei Social Network 51
3.2 Architettura generale del sistema 55
3.3 Particolare dei componenti che permettono il funzionamento dello C-SPARQL Engine 59
3.4 Struttura di un componente Recorder 60
3.5 Struttura di un componente DataGetter 61
3.6 Struttura di un componente Replayer 62
3.7 Struttura di un componente Publisher 63
3.8 Design pattern Factory: schema UML 66
3.9 Design pattern Observer/Observable: schema UML . 67
3.10 Esempio di utilizzo Design Pattern Facade 68
3.11 Rappresentazione della gerarchia che lega i vari tipi di grafo 71
3.12 Utilizzo dei vari tipi di grafo durante la pubblicazione di uno stream di dati 72
3.13 Schema generale di collegamento per la conoscenza statica 73
3.14 Rappresentazione grafica dei componenti dell'applicazione 74
3.15 Esempio di rappresentazione grafica dei componenti . 75
3.16 Esempio di rappresentazione grafica dei componenti . 76

4.1 Class Diagram relativo alla struttura dati TimestampedModel 78
4.2 Class Diagram relativo all'interfaccia DataGetter . . 79

4.3 Class Diagram relativo all'interfaccia Adapter 80
4.4 Class Diagram relativo all'interfaccia per la comunicazione con GetGlue 84
4.5 Class Diagram relativo all'interfaccia per la comunicazione con Twitter 85
4.6 Class Diagram relativo all'oggetto CSPARQLAdapter 86
4.7 Class Diagram relativo all'interfaccia Recorder 88
4.8 Class Diagram relativo all'interfaccia Publisher 91
4.9 Class Diagram relativo all'interfaccia Replayer 92
4.10 Class Diagram relativo all'interfaccia Windower . . . 95
4.11 Class Diagram relativo a SLDServer 97
4.12 Class Diagram relativo alle strutture dati utilizzate per tenere traccia dello stato dei vari componenti . . 98
4.13 Class Diagram relativo alla classe SubClassFinder . . 100
4.14 Esempio di implementazione del pattern factory . . . 100
4.15 Class Diagram relativo agli oggetti a cui è delegata la creazione della conoscenza statica relativa agli utenti 101
4.16 Class Diagram relativo agli oggetti a cui è delegata la creazione della conoscenza statica relativa agli oggetti 102
4.17 Class Diagram relativo alla classe RestServer 104

5.1 Capacità di SLD4SM di soddisfare le richieste di rigenerazione dello stream da parte dell'utente. Il sistema permette di rigenerare lo stream a una velocità n volte maggiore, fino ad un massimo di 800 triple/sec 109
5.2 Le prestazioni di SLD4SM per la registrazione su disco si deteriorano con il passare dei minuti in quanto cresce la dimensione dell'rGraph e, quindi, il relativo tempo di salvataggio 110
5.3 Le prestazioni di SLD4SM per la registrazione su database sono poco variabili nel tempo e si attestano su una media di registrazione di 40 triple/sec 111
5.4 Le prestazioni di SLD4SM per la registrazione su modello persistente sono poco variabili nel tempo e si attestano su una media di registrazione di circa 28 triple/sec. Il setting sperimentale non permette di oltrepassare i quattro minuti di registrazione 112
5.5 Prestazioni di un Windower logico con dimensione finestra 20s, step 13s e expire time 30s. Per valori di input inferiori alle 80 triple/sec il Windower è in grado di servire al client circa 650 mila grafi al secondo, per valori superiori satura ed è in grado di fornire circa 260 mila grafi al secondo 113

5.6 Prestazioni per quattro esperimenti di un Windower logico con dimensione finestra 20s, step 13s e expire time 30s. Tutti mostrano che ad un valore di input di 80 triple/sec il numero di grafi serviti al client cala di colpo e si assesta a circa 260 mila grafi al secondo . 114
5.7 Prestazioni dello C-SPARQL Engine con sorgente sintetica 115
5.8 Prestazioni dello C-SPARQL Engine con sorgente rigenerata 115
5.9 Schema complesso utile per misurare le prestazioni . 116

6.1 Esempio di URI associata a risorse diverse 122

B.1 Schema Esempio 1 129
B.2 Schema Esempio 2 132
B.3 Schema Esempio 3 134
B.4 Schema Esempio 4 135
B.5 Schema Esempio 5 138

Capitolo 1

Introduzione

1.1 Web semantico e stream reasoning

Il World Wide Web rende disponibile una raccolta molto vasta di contenuti (file, testi, ipertesti, suoni, immagini, animazioni, filmati) che rappresentano ormai una buona percentuale dell'intera conoscenza umana. La sua nascita risale ai primi anni novanta ed è dovuta ad un ricercatore inglese del CERN, Tim Berners-Lee, che si pose il problema di consentire a tutti i ricercatori l'accesso e la fruizione delle informazioni create nei vari laboratori per mezzo di computer dislocati in strutture sparse per il territorio. Data tale esigenza ideò un sistema di archiviazione e di ricerca delle informazioni in cui i documenti venivano organizzati in modo ipertestuale, cioè in modo che ognuno di essi fosse collegabile ad altri tramite link. Il miglioramento dell'idea originale e lo sviluppo di nuove idee, ha portato a quello che è il web attuale: tutto è basato su un linguaggio (l'HTML e le sue evoluzioni) che definisce in che modo debbano essere rappresentate le informazioni, ad esempio, come debbano essere visualizzate sullo schermo di un pc, piuttosto che su quello di un dispositivo mobile. L'HTML permette di definire stile e posizione dei vari elementi all'interno della pagina, permettendo così al browser di rappresentarli seguendo le indicazioni dell'utente senza comprenderne il contenuto informativo. In poche parole, il Web è un sistema che permette la rappresentazione delle informazioni in un formato leggibile e comprensibile all'essere umano per favorire lo scambio di informazioni fra persone, ma non quello fra applicazioni. Si tralasciano, cioè, la struttura e il significato del contenuto; questo pone notevoli difficoltà nel reperimento e riutilizzo delle informazioni. Per rendersi conto di quanto sia problematico il riuso delle informazioni si pensi ai motori di ricerca che per ogni query eseguita restituiscono dei risultati che solo in piccola percentuale sono di interesse per la

ricerca effettuata. Questo perché ogni interrogazione è a rischio di ambiguità. Ad esempio, cercando la parola *jaguar* si possono trovare contenuti che si riferiscono a una grande quantità di risorse diverse, dalla chitarra Fender Jaguar fino alla console Atari Jaguar [1].

Una delle prime proposte per la risoluzione di tale problema arrivò dal World Wide Web Consortium (W3C), un'associazione il cui scopo è sviluppare tecnologie che garantiscano l'interoperabilità (specifiche, guidelines, software e applicazioni) per portare il Web al massimo del suo potenziale. Nel 1998 definì lo standard eXtensible Markup Language (XML), i cui obiettivi si focalizzavano principalmente sulla possibilità di aggiungere informazioni semantiche ai dati attraverso la definizione di opportuni tag, in modo che la manipolazione, la visualizzazione e le ricerche sui dati potessero essere eseguite in maniera più precisa e soprattutto contestuale. Le specifiche XML hanno però una lacuna molto importante: non definiscono alcun meccanismo univoco e condiviso per specificare relazioni tra informazioni espresse sul Web per una loro elaborazione automatica (ad es. più documenti che parlano dello stesso argomento, persona, organizzazione, oggetto), rendendo comunque molto difficile la condivisione delle informazioni.

Un ulteriore passo avanti fu fatto ancora grazie al W3C, che propose la formalizzazione del web semantico. L'idea è di generare documenti che possano non solo essere letti e capiti da esseri umani, ma anche accessibili e interpretabili da agenti automatici per la ricerca di contenuti. A tale scopo sono stati definiti alcuni linguaggi, tra i quali troviamo il Resource Description Framework Schema (RDF-S) o OWL, che consentono di esprimere le relazioni tra le informazioni rifacendosi alla logica dei predicati mutuata dall'intelligenza artificiale e sono stati creati modelli di dati, il più importante dei quali è rappresentato da RDF. Sono nati quindi sistemi ormai abbastanza diffusi di interrogazione e reasoning su dati RDF, modellati in RDF-S e OWL, tramite il linguaggio SPARQL, proposto dal W3C come standard per questo settore. Questi sistemi sono molto efficienti se i dati su cui operano sono di tipo statico e fortemente orientati al web semantico.

Parallelamente, si sono affermati i sistemi Data Stream Management System (DSMS) che eseguono delle query preventivamente registrate su stream di dati. Il flusso informativo è continuo e per questo non è archiviabile nel suo complesso. Tipologie di dati simili sono utili in applicazioni che sorvegliano entità finanziarie, ma anche il comportamento di una rete, del traffico automobilistico o, non

[1] Si veda la lista di significati su `http://en.wikipedia.org/wiki/Jaguar_%28disambiguation%29`

meno importanti, applicazioni legate al Real Time Web [2]: tutte situazioni in cui è importante aggiornare costantemente la situazione dei dati, ed effettuare interrogazioni molto frequenti per rendere più attendibili i risultati dei processi.

Agli albori e fino alla metà degli anni novanta, il Web era composto prevalentemente da siti web statici in cui l'utente non aveva alcuna possibilità di interazione eccetto la normale navigazione tra le pagine, l'utilizzo delle posta elettronica e dei motori di ricerca. Con l'evoluzione di internet si è passati dal cosidetto Web 1.0 al Web 2.0, cioè l'insieme di tutte quelle applicazioni online che permettono un elevato livello di interazione tra il sito e l'utente. Il Web 2.0 costituisce un approccio filosofico alla rete che ne connota la dimensione sociale, della condivisione e dell'autorialità rispetto alla sola fruizione. Nasce in questo ambito il concetto di Social Media, ovvero quelle forme di comunicazione partecipativa in cui sono i lettori (utenti) a condividere e creare contenuti tramite la pubblicazione spontanea di conoscenze ed esperienze. I servizi di Social Media possono essere considerati i pilastri fondamentali per lo sviluppo di grafici sociali, per la promozione della condivisione di informazioni e per l'innovazione del Web. Anche se molti di essi hanno creato strumenti incredibili, la loro più grande innovazione potrebbe in realtà essere l'apertura dei loro prodotti a sviluppatori esterni attraverso l'utilizzo di API. Le API (Application Programming Interface) sono un modo per offrire al mondo esterno le proprie funzionalità per creare nuovi prodotti o applicazioni e possono essere di tipo REST o di tipo STREAM. Le API REST sono le più comuni e vengono offerte dalla maggior parte dei Social Network, seppur con limitazioni, e permettono di ottenere solo in seguito ad invocazioni da parte del client le informazioni richieste. Le API di tipo STREAM possono essere considerate una evoluzione delle API REST e la loro caratteristica principale è che il compito di notificare al client la presenza di nuove informazioni spetta al server.

1.2 Linked Data e Social Media

Nel nostro lavoro di tesi abbiamo progettato un ambiente integrato per la gestione di stream di dati provenienti dai Social Media basato su tecnologie del Semantic Web.

I Social Media possono assumere differenti forme, inclusi forum, message board, blog, wiki, podcast e così via e risultano essere una fonte inesauribile di dati in continuo rinnovamento che possono esse-

[2] Si veda `http://en.wikipedia.org/wiki/Real-time_web`

re manipolati ed utilizzati per altri scopi. Proprio per questi motivi, i Social Media presentano una eterogeneità di interfacce che non sono facilmente uniformabili e rendono difficoltoso realizzare applicazioni che consumano stream di dati provenienti da fonti diverse.

Tra i Social Media più in voga in questo periodo è possibile trovarne alcuni che adottano dei pattern comuni come, per esempio, la pubblicazione di informazioni secondo il paradigma "l'utente esprime una opinione riguardo ad un oggetto" e che, inoltre, presentano caratteristiche simili per quanto riguarda le informazioni sugli utenti (informazioni del profilo, friends, followers, etc) e la descrizione degli oggetti. La maggior parte dei Social Media mette a disposizione di utenti e sviluppatori una serie di API che permettono l'interazione con la piattaforma per via programmatica (non si limitano, cioè, all'accesso tramite browser).

La piattaforma Streaming Linked Data for Social Media è uno strumento che permette di processare i dati ottenuti dai Social Media che adottano il pattern "l'utente esprime una opinione riguardo ad un oggetto". In particolare, SLD4SM offre la possibilità di:

- registrare uno stream (su disco, database o modello persistente);
- generare uno stream partendo da dati precedentemente registrati;
- pubblicare uno stream (tramite l'utilizzo di finestre applicate al flusso di dati);
- registrare le informazioni sugli utenti e la descrizione degli oggetti[3]);
- eseguire interrogazioni sullo stream e sulla conoscenza statica (tramite lo C-SPARQL Engine);
- invocare qualsiasi operazione da remoto tramite l'esposizione di interfacce REST;

Come indicato dal nome, SLD4SM trasforma i dati ottenuti dai Social Media in Linked Data, cioè nel formato standard del Web Semantico (vedi paragrafo 2.1), in particolare utilizza ed amplia una proposta di pubblicazione di Data Stream formita dal gruppo LarKC tramite l'utilizzo di grafi RDF [5]. E' importante sottolineare che SLD4SM permette di ampliare tutti i macro-servizi offerti in modo completamente trasparente all'utilizzatore finale, ma di questo parleremo approfonditamente nei capitoli 2 e 3.

[3]Chiameremo questi dati conoscenza statica ad indicare che varia molto lentamente rispetto ai dati provenienti dallo stream.

1.3 Stream Reasoning nel progetto LarKC

Un gruppo di ricerca del Politecnico di Milano, nell'ambito di un progetto della Comunità Europea chiamato LarKC (The Large Knowledge Collider[4]), ha trovato un punto di incontro tra la tecnologia dei DSMS e i dati in forma RDF. L'unione di queste due tecnologie ha dato vita al concetto di stream reasoning [13], ovvero un ambito in cui le query vengono eseguite su uno stream RDF tramite un nuovo linguaggio di interrogazione chiamato C-SPARQL (Continuous SPARQL [4]), ottenuto come estensione del già conosciuto SPARQL. Lo sviluppo del progetto ha, inoltre, portato alla progettazione di un C-SPARQL engine, ovvero un'architettura in grado di eseguire query C-SPARQL, interrogazioni registrate ed eseguite continuamente su repository RDF e su stream RDF. Uno dei problemi che il team ha dovuto affrontare, è stata la mancanza di stream RDF reali per il testing e la messa in opera del sistema. Il testing è stato fatto generando dei dati casuali, ma il problema non è stato risolto.

1.4 Descrizione del problema affrontato

La tesi si colloca nell'ambito del progetto LarKC, in particolare riguarda lo sviluppo degli strumenti necessari alla generazione e alla standardizzazione degli stream RDF necessari al funzionamento e al testing dello C-SPARQL engine. Per prima cosa abbiamo sviluppato due generatori in grado di prelevare i dati da due dei social network più in voga di questi tempi, Twitter[5] e GetGlue[6], e generare delle triple RDF. Il secondo passo è stato sviluppare uno strumento in grado di prelevare dai generatori le triple RDF ed effettuare in base alle richieste sia locali che remote (tramite interfaccia REST):

- la registrazione dei dati in file RDF, su database o su modelli persistenti;
- la generazione dello stream a partire dai dati precedentemente registrati;
- la pubblicazione dello stream RDF;
- la registrazione della conoscenza statica;
- interrogazioni incrociate su conoscenza statica e stream RDF tramite lo C-SPARQL Engine.

[4] Si veda `http://www.larkc.eu`
[5] `http://twitter.com`
[6] `http://getglue.com`

1.5 Struttura dell'elaborato

L'elaborato è organizzato come segue:

- Sezione 2: presenta una rassegna del background tecnologico utile per capire il significato e l'utilità dell'applicazione, in particolare presenta lo stato dell'arte riguardante il web semantico, gli stream di dati e la loro implementazione tramite il framework RDF, oltre ai linguaggi utili per l'estrazione di informazioni tramite query;
- Sezione 3: presenta il progetto concettuale di tutta l'infrastruttura software, presentando e giustificando le scelte architetturali che hanno guidato tutta la creazione di SLD4SM;
- Sezione 4: presenta il livello implementativo dell'applicazione, in particolare le scelte che sono state effettuate a livello di programmazione, con relative giustificazioni;
- Sezione 5: presenta i test svolti per la validazione del funzionamento dell'infrastruttura software, riguardanti in particolare performance e robustezza nei vari ambiti di utilizzo;
- Sezione 6: presenta le conclusioni relative all'intero progetto, insieme a uno sguardo sui possibili sviluppi futuri;
- Sezione 7: nell'appendice vengono presentati un piccolo manuale utente che guiderà l'utente nell'installazione e nell'utilizzo dell'applicazione e alcuni esempi di funzionamento nei casi d'uso più significativi.

Capitolo 2

Stato dell'Arte

2.1 Data Stream e DSMS

In molte applicazioni reali, i dati devono obbligatoriamente avere la forma di uno stream (flusso) continuo di informazioni piuttosto che di un insieme statico di dati immagazzinati in un repository. E' questo il caso di applicazioni che riguardano il monitoraggio del traffico di reti di computer, la gestione di reti cellulari, il monitoraggio e la gestione del traffico stradale, il monitoraggio di titoli di borsa e molti altri ambienti. In tali applicazioni la distinzione tra acquisizione ed elaborazione non è possibile per vari motivi: la velocità di generazione dei dati è maggiore del tempo necessario per memorizzarli su supporti a lunga durata, le informazioni ricavate da un'analisi futura non sono più utili, il costo economico di memorizzazione di tutti i dati supera i benefici dovuti alla loro analisi, etc. Quindi, le classiche query one-shot, tipiche dei database relazionali, devono essere sostituite da query continue, le quali restituiscono nuovi risultati non appena arrivano nuovi dati dallo stream.

In modo formale è possibile definire un data stream come una sequenza illimitata di dati variabili nel tempo, mentre in modo meno formale, come un flusso informativo continuo di dati in cui l'informazione più recente diventa maggiormente rilevante per la descrizione di un dato sistema dinamico.

I dati, ricevuti continuamente e in tempo reale, possono essere ordinati implicitamente, in base al momento di arrivo, o esplicitamente, tramite l'utilizzo di timestamp. Non solo è tipicamente impossibile controllare l'ordine in cui gli oggetti arrivano ma, cosa più importante, non è possibile memorizzare localmente lo stream nella sua totalità. A causa di tutte queste peculiarità, i tradizionali *Database Management System* (brevemente DBMS) e i classici algoritmi di manipolazione dei dati non sono utilizzabili per maneggiare

numerose e complesse query continue su data stream. Nascono così – verso la fine degli anni novanta – sistemi di gestione dei dati ad hoc, chiamati *Data Stream Management System* (brevemente DSMS).

Uno dei primi modelli proposti per i data stream è stato il Chronicle Data Model[10], in cui veniva introdotto il concetto di cronache come sequenze ordinate di tuple. La questione centrale di tale modello era il mantenimento incrementale di viste materializzate indipendenti dalla dimensione dello stream e lo studio delle query che potevano essere risolte utilizzando tali viste. OpenCQ[11] e NiagaraCQ[8] introdussero l'uso di query continue per il monitoraggio di insiemi di dati persistenti diffusi all'interno di reti WAN. La collaborazione tra il MIT, la Brandeis University e la Brown University diede vita al progetto Aurora[2], il cui fine era unire tre tipi di applicazione in un unico framework: applicazioni di monitoraggio real-time, applicazioni di archiviazione e applicazioni di spanning. In seguito Aurora si è evoluto nel progetto Borealis[1] che affronta il problema della distribuzione dei dati, della revisione dinamica dei risultati delle query e della modifica dinamica di quest'ultime. Babu e Widom[3] affrontano il problema delle query continue, soffermandosi sulla specifica semantica di un linguaggio per l'interrogazione e sull'efficienza mostrata da questo nuovo tipo di query; propongono inoltre un'architettura per un DSMS general-purpose. Si potrebbero portare molti altri esempi di progetti di ricerca riguardanti lo sviluppo dei DSMS, lasciamo al lettore più interessato tale compito[9].

Tornando alle questioni tecniche, la differenza fondamentale tra un classico DBMS e un DSMS risiede nel data stream model: mentre in un DBMS le query vengono elaborate su un insieme di dati persistenti, precedentemente memorizzati su disco, in un DSMS le query vengono eseguite continuamente su uno stream di dati, in modo tale da valutare costantemente le informazioni in arrivo.

In un data stream, le informazioni arrivano e restano in memoria solo per un limitato periodo di tempo, di conseguenza il DSMS ha la necessità di gestire i dati prima che ne arrivino di nuovi che determinino la cancellazione di quelli vecchi. L'ordine in cui le informazioni arrivano non può essere controllato dal sistema e una volta che una informazione è stata processata non si può risalire ad essa a meno che non venga esplicitamente memorizzata.

Gli operatori standard che sono supportati dalla maggior parte dei DSMS esistenti sono il filtraggio, il mapping, gli aggregati e il join. Dal momento che un data stream potrebbe non terminare mai, i risultati intermedi delle query continue sono spesso generati su finestre di tempo predefinite (*Window*) e poi memorizzati, ag-

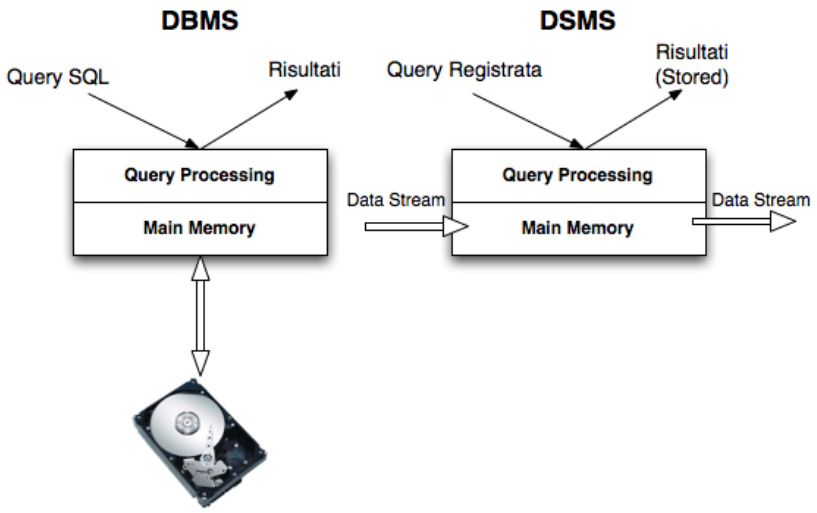

Figura 2.1: Differenze tra DBMS e DSMS

giornati o utilizzati per generare un nuovo flusso continuo di dati dei risultati intermedi. Per definire una finestra occorre specificare essenzialmente due parametri: la dimensione della finestra (*window size*) e quanto spesso si deve rigenerare la finestra (*hop*).

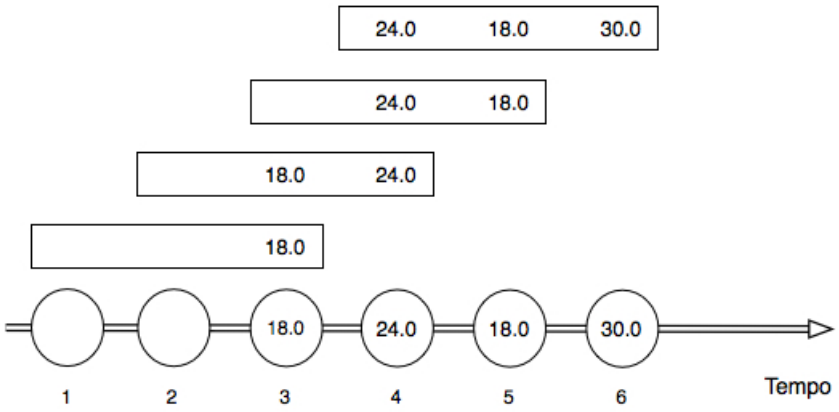

Figura 2.2: Esempio di windowing

Una finestra estrae gli ultimi elementi di uno stream, che sono gli unici che vengono presi in considerazione durante l'esecuzione di una query. L'estrazione può essere fisica, se eseguita su un preciso numero di triple, o logica, se si prendono in considerazione tutte le triple che intercorrono in un dato intervallo di tempo. Le finestre logiche vengono dette sliding, se scorrono in avanti di un dato inter-

vallo detto step e sono tumbling se questo intervallo di scorrimento equivale alla grandezza della finestra.

I DSMS sono progettati per processare query parallele real-time, ma non possono eseguire compiti di ragionamento complessi. Inoltre, l'interpretazione di enormi quantità di dati provenienti da data stream richiede una ricca base di conoscenze statiche, oltre all'integrazione di metodi di ragionamento induttivi e deduttivi, che favoriscono il reasoning in real-time su data stream "rumorosi", in modo da supportare l'uso concorrente di vari processi decisionali.

2.2 SemanticWeb

> Il Web semantico darà struttura al contenuto significativo delle pagine Web, creando un ambiente dove gli agenti software possano svolgere velocemente compiti complessi per i loro utenti. [...] Il Web semantico è un'estensione di quello attuale, in cui alle informazioni viene attribuito un significato definito, che permette a computer e utenti umani di lavorare meglio insieme. [6]

Il Web è sempre più ricco di conoscenza ma, molto spesso, le informazioni che vi sono presenti non sono riutilizzabili da parte delle applicazioni. L'esempio più calzante di questo problema è Wikipedia [1] che, come una classica enciclopedia cartacea, offre una grande quantità di informazioni comprensibili dall'uomo ma che risultano incomprensibili ed inutilizzabili dalle macchine. La nuova dimensione semantica del Web non presuppone la capacità di capire qualunque tipo di informazione da parte delle applicazioni ma, più semplicemente, che la conoscenza potrà essere manipolata meccanicamente per ottenere nuova conoscenza utile alle finalità umane.

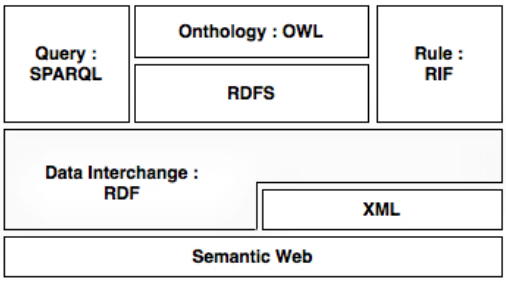

Figura 2.3: Struttura del Semantic Web

[1] http://wikipedia.org

Una possibile applicazione di Web semantico potrebbe essere la seguente: una organizzazione pubblica un database riguardante la sua linea di produzione, contenente prodotti e relative descrizioni, una seconda sorgente pubblica un database con gli stessi prodotti, contenente le opinioni degli utenti che li hanno provati, mentre una terza risorsa, dedicata ai rivenditori, pubblica la quantità di materiale in magazzino. Come è possibile uniformare i dati in modo che siano condivisibili e utilizzabili per l'estrazione di nuova conoscenza? In un sistema decentralizzato, come quello appena descritto, è necessario uno standard che permetta di scambiarsi i dati e di uno che permetta di capire a fondo le informazioni contenute nella base di conoscenza, manipolarla, fonderla e proporla all'utente o all'applicazione. Tali standard ci aiutano a comunicare e scambiare informazioni in modo sicuro e veloce in un ambiente distribuito. La Figura 2.3 mostra i componenti di maggior interesse del Semantic Web:

- XML: fornisce una sintassi elementare per la struttura relativa al contenuto dei documenti, non associandovi nessuna informazione semantica;
- RDF: rappresenta un semplice linguaggio per esprimere modelli di dati, utile per mappare le relazioni che esistono tra vari oggetti (Risorse). Un modello RDF può essere rappresentato tramite sintassi XML;
- RDF-S (*RDF Schema*): rappresenta un vocabolario per la descrizione di proprietà e classi legate alle risorse RDF;
- OWL: rappresenta una famiglia di linguaggi utilizzati per creare ontologie, estende le potenzialità di RDF tramite l'aggiunta di vocabolari per la descrizione delle relazioni tra le classi, la cardinalità e l'uguaglianza;
- SPARQL: rappresenta un protocollo e un linguaggio di query per risorse legate al Semantic Web;
- RIF (*Rule Interchange Format*): la presenza di questo componente è fortemente raccomandato da parte del W3C e la sua creazione è legata all'osservazione approfondita dei rule languages che esistono e alla necessità di farli comunicare tra loro.

2.2.1 RDF e RDF-S

Il *Resource Description Framework* (brevemente RDF) rappresenta lo strumento principale proposto dal W3C per la codifica, lo scambio e il riutilizzo di informazioni e consente l'interoperabilità tra applicazioni che si scambiano informazioni sul Web ed è stato progettato per raggiungere i seguenti obiettivi:

- offrire un modello di dati semplice;
- offrire una semantica formale e permettere l'inferenza;
- possibilità di utilizzare un vocabolario URI-based estendibile;
- utilizzare una sintassi basata su XML;
- supporto ai datatype contenuti nella sintassi degli XML-Schema.

Il componente principale del Resource Description Framework è rappresentato dallo *RDF Data Model*, che espone la struttura del modello RDF e ne descrive una possibile sintassi.

Scrivendo enunciati RDF, si può dunque concretizzare la rappresentazione delle risorse, il framework fornisce il predicato *rdf:type* che permette un certo grado di tipizzazione, ma ci serve un meccanismo per definire in modo chiaro i tipi delle risorse, cioè gli oggetti degli enunciati che contengono il predicato *rdf:type* e per codificare relazioni complesse tra le risorse. Esistono diversi linguaggi utili per i nostri scopi, il W3C raccomanda l'uso di RDF Schema (RDF-S), che è un'estensione di RDF che fornisce gli strumenti per descrivere vocabolari, ossia permette di descrivere gruppi di risorse collegate e le proprietà che le legano tra loro.

Un modello RDF è essenzialmente formato da risorse (*Resource*), proprietà (*Property*), valori (*Literal*) e asserzioni (*Statement*). Ogni entità descritta in RDF prende il nome di risorsa ed è identificata in modo univoco da un URI e, nella maggior parte dei casi, può trovarsi sul Web. Le risorse sono legate tra loro da proprietà, che non sono altro che relazioni identificate anch'esse in modo univoco da URI. Una proprietà può, inoltre, legare una risorsa a un valore che rappresenta un tipo di dato primitivo (una stringa, un numero intero, etc).

```
1.  s: http://example.com/myBook
    p: http://example.com/hasAuthor
    o: http://example.com/DanteAlighieri
```

Esempio 1: Rappresentazione di "L'autore del mio libro è Dante Alighieri"

Le triple <*risorsa, predicato, risorsa*> o <*risorsa, predicato, literal*> vengono dette Statement. Ogni tripla può essere vista come formata da un soggetto (s), un predicato (p) e un oggetto (o), e l'insieme di più Statement rappresenta un *RDF Data Model* che ci permette di descrivere la realtà tramite le relazioni che legano le diverse risorse. Una possibile rappresentazione della frase "L'autore del mio libro è Dante Alighieri" è quella mostrata nell' Esempio 1.

Legando tra di loro Statement riguardanti lo stesso soggetto (che come detto prima viene univocamente identificato da una URI) è possibile formare espressioni sempre più complesse. Nell'Esempio 2 sono rappresentate le asserzioni "Il titolo del mio libro è Divina Commedia" e "La Divina Commedia è stata scritta nel 1307".

```
2 - s: http://example.com/myBook
    p: http://example.com/hasTitle
    o: http://example.com/DivinaCommedia

3 - s: http://example.com/DivinaCommedia
    p: http://example.com/hasBeenWritten
    o: 1307
```

Esempio 2: Triple RDF

Il risultato finale tradotto in italiano è il seguente: "Il mio libro ha come autore Dante Alighieri, ha come titolo 'Divina Commedia' ed è stato scritto nel 1307". Un qualunque insieme di triple può essere rappresentato sotto forma di grafo RDF, il quale è formato da nodi ed archi orientati; ogni tripla RDF sarà quindi formata da nodo-arco-nodo.

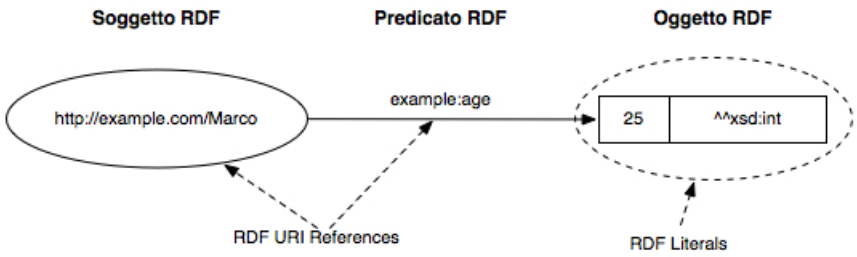

Figura 2.4: Schema Tripla RDF

Grazie a questo tipo di rappresentazione la mancanza di una tripla non pregiudica la creazione o l'utilizzo di un grafo, ma indica soltanto che quel tipo di informazione è sconosciuta. Questa caratteristica, nota come assunzione del mondo aperto, ci dice che tutto ciò che è esplicitamente asserito nella base di conoscenza è vero mentre

nulla si può dire su quello che non è esplicitamente asserito, ed è opposta all'assunzione del mondo chiuso, usata nei modelli relazionali e ad oggetti, che asserisce che tutto ciò che è esplicitamente asserito è vero, mentre ciò che non viene asserito è falso.

Il limite di un documento RDF è che si preoccupa soltanto di descrivere le proprietà di un oggetto, senza però definire alcun livello di gerarchia o di relazione tra le classi, questo compito appartiene agli *RDF Schema* (o RDF-S). Gli RDF-S sono utili in quanto possono essere utilizzati per validare, dal punto di vista semantico, i modelli RDF e in quanto la loro semantica permette semplici inferenze, tipiche dei linguaggi ontologici (es. Se r è di classe R e R è sottoclasse della classe S, allora r è di classe S). Inoltre RDF-S fornisce gli strumenti necessari alla creazione di vocabolari RDF, cioè dei documenti che descrivono le classi e le proprietà utilizzate in una certa applicazione. Gli RDF-S presentano anche alcune limitazioni dovute principalmente alla scarsa espressività: non permettono di definire vincoli di esistenza e di cardinalità e non permettono di definire proprietà transitive, inverse e simmetriche. Inoltre RDF-S presenta difficoltà nel fornire servizi di ragionamento, a causa dell'assenza di reasoner "nativi".

Un modello RDF è completamente indipendente rispetto alle varie sintassi usate per esprimere il suo contenuto, di conseguenza un grafo RDF può essere rappresentato e distribuito mediante un'operazione di serializzazione. Per serializzazione si intende la conversione di una struttura dati o oggetto in un formato che può essere salvato e letto successivamente sulla stessa, o su un'altra, macchina e/o applicazione.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
              xmlns:j.0="http://example.com/" >
<rdf:Description rdf:about="http://example.com/DivinaCommedia">
<j.0:hasAuthor rdf:resource= "http://example.com/DanteAlighieri"/>
</rdf:Description>
</rdf:RDF>
```

Esempio 3: Dante Alighieri è l'autore della divina commedia in formato RDF/XML

Il W3C ha emanato le specifiche e consigliato principalmente quattro formati di serializzazione anch'essi basati su XML:

1. *RDF/XML*: è il formato di default per la distribuzione dei dati RDF, che vengono serializzati in formato XML. Tale formato è stato creato principalmente per il trasferimento dei dati da

macchina a macchina, visto che il risultato della serializzazione è di difficile lettura da parte dell'uomo. Vedi Esempio 3;

2. *N3*: è un formato di serializzazione non XML creato appositamente per rendere i modelli RDF più leggibili all'occhio umano. Questo formato è molto potente e possiede una serie di funzionalità aggiuntive che permettono di andare oltre la semplice serializzazione per modelli RDF. Vedi Esempio 4;

```
<http://example.com/Divina Commedia>
<http://example.com/hasAuthor>"Dante Alighieri"^^
        <http://www.w3.org/2001/XMLSchema#string> .
```

Esempio 4: Dante Alighieri è l'autore della divina commedia in formato N3

3. *Turtle*: rappresenta un sottoset di componenti del linguaggio N3 appositamente studiati per essere utilizzati solamente per RDF;

4. *N-Triples*: rappresenta anch'esso un formato per salvare e trasmettere i dati RDF sotto forma di testo, è un sottoset dei componenti del formato *Turtle*.

L'utilizzo di RDF in un'applicazione isolata non mostra tutte le sue potenzialità, che però diventano chiare quando entra in gioco la condivisione dell'informazione tra diversi processi o applicazioni sparsi per il Web.

2.2.2 SPARQL

SPARQL (acronimo ricorsivo che significa *SPARQL Protocol and RDF Query Language*), rappresenta un termine generale che si riferisce sia a un protocollo che a un linguaggio per svolgere query. Il protocollo fornisce un metodo per l'invocazione da remoto di query in formato SPARQL e specifica una semplice interfaccia che può essere utilizzata attraverso HTTP o SOAP, permettendo a un utente di lanciare una qualunque query verso un qualunque endpoint. Nella sua veste di linguaggio per l'interrogazione di dati la sua sintassi è simile a quella di SQL e viene utilizzato per interrogare grafi RDF usando il pattern-matching, offrendo basic conjuntive patterns, filtri sui valori, optional patterns e pattern disjunction. Per tale motivo risulta utile per effettuare ricerche impiegando le proprietà definite convenzionalmente metatag.

Sintassi

All'interno di un grafo RDF, tutto ruota intorno al contenuto delle triple, il resto, a partire dal tipo di serializzazione passa in secondo piano. Il W3C raccomanda l'uso di RDF/XML, ma da punto di vista pratico, non è la scelta migliore, infatti non rende univoca la rappresentazione di un grafo; per questo motivo il linguaggio SPARQL utilizza la sintassi TURTLE, un'estensione di N-TRIPLES, su cui non vige nessuno standard ma che ha raggiunto una notevole popolarità come un'alternativa human-friendly a RDF/XML (Vedi Paragrafo 2.2.1).

Esecuzione e Forma Query

L'esecuzione di una query si basa sul principio di triple pattern matching, uno speciale meccanismo di matching che rispecchia a fondo la conformazione a triple delle asserzioni in linguaggio RDF. Per una interrogazione in cui vogliamo ottenere tutti i soggetti e gli oggetti messi in relazione da uno specifico predicato possiamo scrivere:

```
?subject example:predicate ?object
```

In questo modo il simbolo ? trasforma soggetto e oggetto in due incognite all'interno della tripla, che possono assumere diversi valori durante l'esecuzione della query.

```
PREFIX book : <http://book.org/>
SELECT  ?author ?title ?pages
FROM <http://book.org/books.ttl>
WHERE

{
       ?author book:write ?title .
       ?title book:has ?pages .
}
```

Query 1

Un'interrogazione SPARQL è formata da:

- *Prefix Declarations*: utile per abbreviare le URI;
- *Result Clause*: identifica quali informazioni deve ritornare la query;
- *Query Pattern*: specifica cosa cercare nel dataset specificato;

- *Query Modifiers*: è opzionale e modifica le informazioni restituite dall'interrogazione.

In Query 1 la parola chiave *PREFIX* indica l'inizio del blocco Prefix Declarations, che in questo caso permette di dichiarare il prefisso e di legarlo al nome book, in modo da non doverlo ripetere più volte all'interno della query. *SELECT* indica il blocco Result Clause, che specifica le variabili utili per la ricerca e per produrre il risultato, mentre FROM specifica il dataset su cui svolgere la query (SPARQL permette di riferirsi a più di un set di dati contemporaneamente utilizzando le clausole *FROM NAMED* e *GRAPH*). La clausola WHERE permette di definire i criteri di estrazione dei dati e rappresenta la parola chiave che apre il blocco Query Pattern. Attraverso l'esecuzione della query estraiamo le triple

```
<autore, titolo, numero di pagine>
```

dal set di dati presente all'indirizzo http://book.org/books.ttl.

In questa query non è presentata la parte di modifica dei risultati, che cercheremo di spiegare meglio nei prossimi paragrafi.

Restrizioni e modifiche dei risultati

SPARQL offre la possibilità di restringere i valori possibili da associare a una variabile attraverso la clausola *FILTER*, utilizzabile su funzioni algebriche (<, >, + ecc) e su stringhe di caratteri (attraverso l'utilizzo della regex).

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
SELECT  ?author
FROM  <http://book.org/books.ttl>
WHERE

{
       ?author dc:write ?title

       FILTER  {regex(?title, "^SPARQL")}
 }
```

Query 2

Nell'interrogazione 2 la parola chiave regex permette la restrizione dei valori assumibili dalla stringa relativa agli autori. L'interrogazione estrae tutti gli autori i cui libri contengono nel titolo la stringa "SPARQL".
Il risultato della query viene presentato nella forma seguente:

Author
author

Le possibilità di modificare i risultati ottenuti da una generica query sono essenzialmente quattro (Vedi query 3):

1. *ORDER BY*: permette di dare un ordinamento, crescente o decrescente dei risultati di una query;
2. *DISTINCT*: permette di eliminare dal risultato i valori ripetuti. Tale parola chiave va utilizzata dopo la clausola *SELECT*;
3. *LIMIT*: restringe il numero di risultati. L'interrogazione proposta mostra come limitare il numero di risultati a 5;
4. *OFFSET*: elimina dall'insieme di risultati un certo numero di valori. L'interrogazione proposta mostra come restituire i risultati eliminando i primi 10.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT  ?name
FROM <http://example.org/name.ttl>
WHERE { ?x foaf:name ?name }
ORDER BY { ?name }
LIMIT 5
OFFSET 10
```

Query 3

SPARQL 1.1

SPARQL 1.1 supera alcune lacune che non permettevano a SPARQL di esser considerato un linguaggio maturo. Cercheremo ora di elencare, con una breve spiegazione, le principali migliorie apportate dalla nuova versione.

Funzioni di aggregazione

Le funzioni di aggregazione eseguono calcoli su set di valori predefiniti e restituiscono un valore singolo. L'opzione predefinita specifica tutte le soluzioni in un unico gruppo, ma diversi raggruppamenti possono essere specificati attraverso le clausole *GROUP BY* e *HAVING*.

Altre funzioni di aggregazione introdotte da SPARQL 1.1 sono rappresentate da*COUNT, SUM, MIN, MAX, AVG* (Vedi specifica del W3C [2]).

La query 4 mostra l'utilizzo di due funzioni di aggregazione: la *GROUP BY* e la *HAVING*, che ha un comportamento analogo alla *FILTER* ma agisce sui gruppi invece che sulle singole soluzioni. Nelle interrogazioni che utilizzano aggregati o nelle sottoquery, solo le espressioni all'interno della *GROUP BY* possono essere proiettate.

Dati:

```
@prefix : <http://books.example/> .
:org1 :affiliates :auth1, :auth2 .
:auth1 :writesBook :book1, :book2 .
:book1 :price 9 .
:book2 :price 5 .
:auth2 :writesBook :book3 .
:book3 :price 7 .
:org2 :affiliates :auth3 .
:auth3 :writesBook :book4 .
:book4 :price 7 .
```

Query:

```
PREFIX <http://books.example/>

SELECT ((SUM(?lprice ) AS ?totalPrice)
WHERE

{
      ?org :affiliates ?auth .
        ?auth :writesBook ?book .
      ?book :price ?lprice .
}

GROUP BY?org
HAVING(?totalPrice > 10)
```

TotalPrice
21

Query 4

Sottoquery

Una sottoquery è un'istruzione *SELECT* nidificata in un'altra sottoquery o in un'altra istruzione. L'opportunità di innestare le sot-

[2] http://www.w3.org/TR/2009/WD-sparql11-update-20091022/

toquery rende il linguaggio più potente ed espressivo (Vedi Query 5).

Dati:

```
@prefix : <http://people.example/> .
:alice :name "Alice", "Alice Foo", "A. Foo" .
:alice :knows :bob, :carol .
:bob :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
PREFIX <http://people.example/>

SELECT ?y ?minName
WHERE

{
        :alice :knows ?y .
{

SELECT ?y (MIN (?name) AS ?minName)
WHERE

{
         ?y :name ?name .
}

GROUP BY ?y


 }
 }
```

y	name
:bob	B. Bar
:carol	C. Baz

Query 5

Presenza o assenza di un pattern

L'introduzione delle nuove clausole *EXIST* e *NOT EXIST* permette di testare se una tripla trova o meno riscontro all'interno di un insieme di dati, non generando legami aggiuntivi riguardanti le variabili. Le due nuove parole chiave possono essere usate all'interno della clausola *FILTER* e, mentre quest'ultima si applica a tutto il gruppo di soluzioni, *EXIST* e *NOT EXIST* si riferiscono solo alle variabili dichiarate precedentemente. Da notare che la clausola *NOT EXIST* è equivalente, e può dunque essere tradotta con *not(EXIST...)*.

Query 6 mostra come restituire la persona che, presente nel set di dati, non rispetta quanto contenuto nella clausola FILTER NOT EXISTS.

Dati:

```
@prefix  :       <http://example/> .
@prefix  rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  foaf:   <http://xmlns.com/foaf/0.1/> .
:alice  rdf:type   foaf:Person .
:alice  foaf:name  "Alice" .
:bob    rdf:type   foaf:Person .
```

Query:

```
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?person
WHERE

{
        :alice :knows ?y .
{

SELECT?y(MIN(?name) AS?minName) WHERE

 {
     ?person rdf:type  foaf:Person .

FILTER NOT EXISTS{ ?person foaf:name ?name }


 }
```

Person
<http://example/bob>

Query 6

Alternative nel pattern matching

SPARQL fornisce un metodo per combinare diversi graph pattern, in modo tale che almeno una delle alternative fornite possa trovare riscontro, se il match riguarda più alternative, vengono tutte prese in considerazione.
Tutte le alternative prese in considerazione, vengono espresse attraverso la keyword UNION.

L'interrogazione 7 trova tutti i titoli dei libri contenuti nell'insieme dei dati, sia che questi siano registrati utilizzando la versione 1.0 della proprietà del Dublin Core, si utilizzando la versione 1.1.

Dati:

```
@prefix dc10:  <http://purl.org/dc/elements/1.0/> .
@prefix dc11:  <http://purl.org/dc/elements/1.1/> .

_:a  dc10:title     "SPARQL Query Language Tutorial" .
_:a  dc10:creator   "Alice" .

_:b  dc11:title     "SPARQL Protocol Tutorial" .
_:b  dc11:creator   "Bob" .

_:c  dc10:title     "SPARQL" .
_:c  dc11:title     "SPARQL (updated)" .
```

Query:

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>

PREFIX dc11:  <http://purl.org/dc/elements/1.1/>
SELECT ?title
WHERE  { { ?book dc10:title  ?title }
UNION
{ ?book dc11:title  ?title }
}
```

title
SPARQL Protocol Tutorial
SPARQL
SPARQL (updated)
SPARQL Query Language Tutorial

Query 7

Valori opzionali

In generale il graph pattern permette alle applicazioni di scrivere query in cui l'intero pattern dei dati deve coincidere completamente con la richiesta per essere incluso nelle soluzioni, ma spesso i dati nel grafo non hanno una forma che corrisponde alla struttura completa contenuta nella query. La keyword *OPTIONAL* permette alle informazioni disponibili di essere aggiunte alle soluzioni e di non essere scartate se una parte del pattern non coincide con la richiesta dell'interrogazione, per la parte non coincidente non viene creato il binding, ma la tupla non viene scartata dall'insieme delle soluzioni.

Dati:

```
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

_:a  rdf:type        foaf:Person .
_:a  foaf:name       "Alice" .
_:a  foaf:mbox       <mailto:alice@example.com> .
_:a  foaf:mbox       <mailto:alice@work.example> .

_:b  rdf:type        foaf:Person .
_:b  foaf:name       "Bob" .
```

Query:

```
PREFIX dc10:  <foaf: <http://xmlns.com/foaf/0.1/>>

SELECT ?name ?mbox
WHERE

     { ?x foaf:name  ?name .


     OPTIONAL { ?x  foaf:mbox  ?mbox}
}
```

name	mbox
Alice	**<mailto:alice@example.com>**
Alice	**<mailto:alice@work.example>**
Bob	

Query 8

La query 8 ritorna tutti i nomi delle persone presenti nei dati con i relativi indirizzi mail, grazie alla *OPTIONAL* anche le persone senza mailbox specificata vengono ritornate e incluse nell'insieme delle soluzioni.

2.2.3 Linked Data

L'obiettivo dei Linked Data è di permettere la condivisione sul Web di dati strutturati in modo molto più semplice di quello attuale. Il principio che sta alla base del loro utilizzo è che il valore e l'utilità dei dati pubblicati sul Web cresce quanto è maggiore la loro interconnessione con altri dati.

I quattro principi fondamentali su cui sono basati i Linked Data sono:

- utilizzare gli URI come nomi per le cose;
- utilizzare gli URI HTTP in modo da consentire l'accesso a descrizioni relative a quei nomi;
- quando qualcuno accede ad un URI bisogna fornire informazioni utili tramite RDF e SPARQL;
- includere link verso altre URI in modo da permettere la scoperta di altri concetti collegati.

I Linked Data si basano su due tecnologie fondamentali per il Web: gli *Uniform Resource Identifiers* (brevemente URI) e il protocollo HTTP. Mentre gli *Uniform Resource Locator* (brevemente URL) sono diventati familiari come indirizzi di documenti e altre entità che possono essere situati sul Web, gli URI forniscono un mezzo più generico per identificare qualsiasi entità che esiste nel mondo. Se le entità sono identificate da URI che usano lo schema *http://*, tali entità possono essere consultate semplicemente dereferenziando lo URI tramite il protocollo HTTP. In tal modo, il protocollo HTTP fornisce un meccanismo semplice ma universale per il recupero delle risorse, le quali possono essere serializzate come un flusso di byte. URI e HTTP vengono affiancate da un'altra tecnologia fondamentale per il Web semantico, di cui abbiamo parlato nei capitoli precedenti, l'RDF: l'utilizzo dello *RDF Data Model* e dei link RDF, il primo per la pubblicazione dei dati in forma strutturata e i secondi per la realizzazione delle interconnessioni dei dati provenienti da diverse sorgenti, porta alla creazione di una comunità per la condivisione dei dati, in cui le persone e le organizzazioni possono pubblicare ed utilizzare i dati, il Web semantico.

I link RDF hanno la forma di una normale tripla RDF, dove il soggetto della tripla è un URI referenziato al namespace di una sorgente di dati, mentre l'oggetto della tripla è un URI referenziato ad un'altra sorgente. L'Esempio 5 afferma che la descrizione del film

Subject: `http://data.linkedmdb.org/page/film/3`
Property: `http://www.w3.org/2002/07/owl#sameAs`
Object: `http://dbpedia.org/page/Batman`

Esempio 5: Tripla RDF formata da URI referenziate

Batman presente nel *Linked Movie Database* è connessa alla descrizione del film fornita da Dbpedia. I benefici derivanti dall'utilizzo dello RDF Data Model sono essenzialmente i seguenti:

- le applicazioni possono dereferenziare ogni URI presente in un grafo RDF per avere informazioni aggiuntive;
- il merge delle informazioni provenienti da diverse fonti avviene in modo naturale;
- il Data Model favorisce la creazione di link RDF tra dati provenienti da fonti differenti;
- il Data Model permette la rappresentazione delle informazioni espresse secondo regole differenti in un unico modello;
- combinato con differenti schema language, come per esempio OWL o RDF-S, il Data Model permette di utilizzare tante strutture quante sono necessarie, ciò significa che è possibile rappresentare sia dati ben strutturati che dati semi-strutturati.

Le ultime rilevazione ci dicono che più di 25 miliardi di triple e più di 800 milioni di link RDF sono stati pubblicati nella LOD [3] cloud (vedi Figura 2.5).

E' bene sottolineare che i Linked Data dovrebbero sempre essere pubblicati in associazione a diversi tipi di metadata, al fine di incrementare la loro utilità per chi li consumerà. Per consentire ai client di valutare la qualità dei dati pubblicati e di capire quanto siano affidabili, i dati dovrebbero essere associati a delle meta-informazioni relativi al creatore, la data di pubblicazione e al metodo di creazione.

Nel Web semantico è possibile accedere ai dati in modo semplice: tramite l'utilizzo di un comune browser è possibile navigare fra le informazioni presenti nelle varie sorgenti di dati tramite l'utilizzo dei link RDF. Questo permette all'utente (o all'applicazione) di iniziare la consultazione delle informazioni da una certa sorgente dati e poi muoversi attraverso una serie, potenzialmente infinita, di altre sorgenti di dati connesse tramite link RDF.

Dal punto di vista dell'applicazione, il Web semantico ha le seguenti caratteristiche:

[3] Linking Open Data

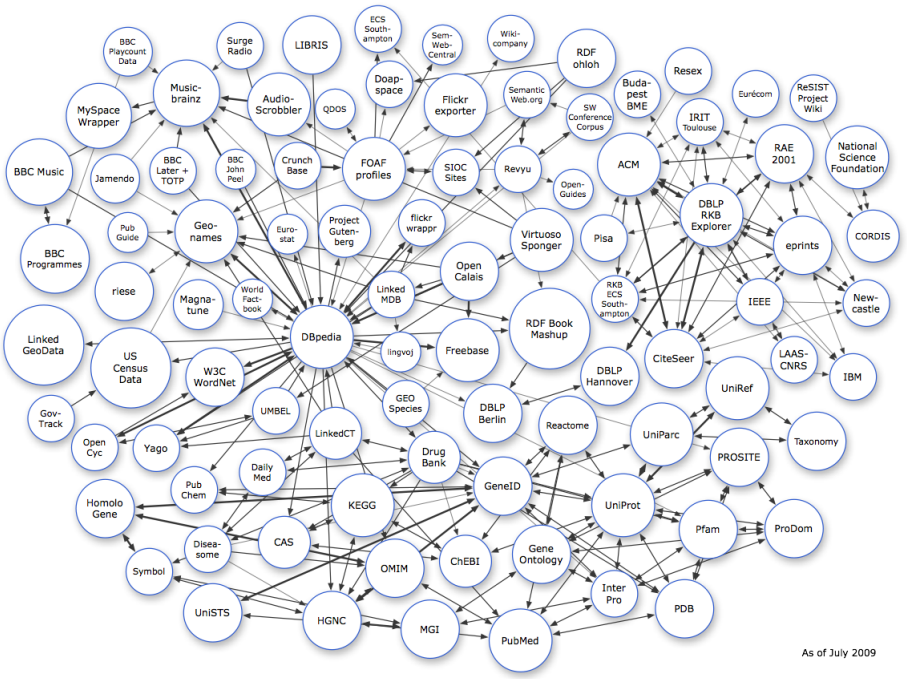

Figura 2.5: LOD Cloud nel Settembre 2010

- il contenuto informativo è completamente separato dalla sua formattazione;
- i dati sono auto-descrittivi. Se un'applicazione utilizza Linked Data descritti tramite un vocabolario sconosciuto, essa può dereferenziare lo URI che identifica il vocabolario al fine di tradurne il contenuto in un formato compatibile;
- l'uso di HTTP come meccanismo standard di accesso ai dati e di RDF come Data Model standard semplifica l'accesso ai dati rispetto alle classiche API offerte dai servizi Web, che solitamente si basano su Data Model ed interfacce d'accesso eterogenee;
- il Web semantico è aperto, ciò significa che le applicazioni non devono essere implementate utilizzando un set predefinito di risorse, ma possono trovarne di nuove a run-time seguendo i link RDF.

Tim Berners-Lee ha proposto uno schema a 5 stelle per valutare il livello raggiunto mediante la pubblicazione dei Linked Data. Di seguito lo presenteremo fornendo una lista di costi e benefici, suddivisi in base ai diversi punti di vista.

1. Rendere il vostro materiale disponibile su Web in qualsiasi formato con licenza aperta

   Come utilizzatore dei dati:

   - è possibile accedere ai dati;
   - è possibile stampare i dati;
   - è possibile salvare i dati localmente;
   - è possibile inserire i dati che leggi in un altro sistema in modo manuale.

   Come publisher dei dati:

   - è semplice pubblicare i dati.

2. Renderlo disponibile in formati strutturati (es. rendere disponibile una tabella in Excel o come immagine)

   Come utilizzatore dei dati, è possibile svolgere tutte le azioni presenti al punto precedente più:

   - è possibile processare direttamente i dati con il software proprietario con cui sono stati creati;
   - è possibile esportare i dati in un altro formato (meglio se strutturato).

   Come publisher dei dati:

   - è ancora più semplice pubblicare i dati.

3. Usare formati non proprietari (es. usare CSV piuttosto che Excel)

   Come utilizzatore dei dati, è possibile svolgere tutte le azioni presenti al punto precedente più:

   - è possibile manipolare i dati in qualsiasi modo, senza essere limitato dalle capacità di un software proprietario.

   Come publisher dei dati:

   - potrebbe nascere il bisogno di plug-in o convertitori per esportare i dati da un formato proprietario ad uno aperto;
   - è ancora piuttosto semplice pubblicare i dati.

4. usare URI per identificare le cose, in modo tale che le persone possano puntare direttamente ai vostri dati

   Come utilizzatore dei dati, è possibile svolgere tutte le azioni presenti al punto precedente più:

   - è possibile avere un collegamento ai propri dati indipendentemente dal luogo in cui ci si trova (locale o Web);
   - è possibile trasformare i dati in segnalibri;
   - è possibile riutilizzare parte dei dati.

   Come publisher dei dati:

   - è necessario assegnare URI ai dati e pensare a come rappresentarli;
   - si ha un maggiore controllo sui dati e si può ottimizzare l'accesso ad essi (load balancing, caching, ecc).

5. Collegare i propri dati ad altri dati per definirne il contesto

   Come utilizzatore dei dati, è possibile svolgere tutte le azioni presenti al punto precedente più:

   - è possibile trovare dati aggiuntivi collegati ai dati che si stanno consumando;
   - è possibile imparare direttamente dallo schema dei dati.

   Come publisher dei dati:

   - è necessario investire risorse per collegare i dati ad altri dati presenti sul Web;
   - rende i dati rilevabili nel Web;
   - aumenta il valore dei dati.

2.2.4 Jena

Esistono molti framework applicativi che possono venire in aiuto agli sviluppatori di applicazioni legate al Web semantico. Tali framework sono disponibili per diversi linguaggi di programmazione e forniscono un ambiente di lavoro completo che comprende librerie e moduli specifici.

Alcuni esempi significativi sono dotNetRDF[4], nato per lo sviluppo in linguaggio C-Sharp su piattaforme Microsoft, o Redland[5] che offre una serie di librerie open source per il supporto a RDF e che è utilizzabile per una grande varietà di linguaggi di programmazione (C-sharp, java, python, ruby, obj-C ecc). Per altri framework rimandiamo al sito del W3C [6].

Il motore C-SPARQL utilizza Sesame[7], che è un framework legato al mondo java e offre un database RDF open source, con supporto per inferenza e query su RDF-S.

Ogni prodotto offre caratteristiche interessanti, ma la nostra scelta si è orientata su Jena[8] per la sua semplicità d'utilizzo e per le sue ottime prestazioni, sia in fase di creazione di modelli che per quanto riguarda l'interrogazione su modelli esterni. Jena è open source (con licenza BSD[9]) ed inizialmente fu sviluppato per la realizzazione di applicazioni che supportassero RDF e RDF-S, ma successivamente è stato esteso verso i linguaggi per le ontologie (OWL).

Jena è in grado di offrire:

- API RDF;
- la possibilità di leggere e scrivere RDF in RDF/XML, N3 e N-Triples;
- API OWL;
- in-memory e persistent storage;
- un potente motore per query SPARQL.

Nel Paragrafo 2.2.1 abbiamo visto cosa si intenda con il termine risorsa e siamo ora in grado di introdurre come il framework operi su tali entità. In Jena un grafo RDF viene chiamato modello e viene rappresentato tramite l'interfaccia `Model`. Le singole risorse vengono istanziate tramite l'interfaccia `Resource`, le proprietà tramite l'interfaccia `Property` e i valori tramite l'interfaccia `Literal`. L'Esempio 6 mostra come creare un grafo contenente le due affermazioni "La Divina Commedia è stata scritta da Dante Alighieri" e "Ivanhoe è stata scritto da Walter Scott" . Si inizia con la definizione di alcune costanti e la creazione di un modello vuoto tramite l'utilizzo del metodo *createDefaultModel()* di `ModelFactory`. In seguito vengono create le risorse e la proprietà che andranno a formare le asserzioni da aggiungere al modello.

[4] `http://www.dotnetrdf.org`
[5] `http://librdf.org/`
[6] `http://www.w3.org/2001/sw/wiki/Category:Programming_Environment`
[7] `http://www.openrdf.org/`
[8] `http://jena.sourceforge.net/`
[9] `http://it.wikipedia.org/wiki/Licenze_BSD`

```
String bookURI = "http://library/DivinaCommedia";
String authorURI = "http://library/DanteAlighieri";
String ivanhoeURI = "http://library/Ivanhoe";
String walterScottURI = "http://library/WalterScott";

Model library = ModelFactory.createDefaultModel();

Resource divinaCommedia =
 library.createResource(bookURI);
Resource danteAlighieri =
library.createResource(authorURI);
Resource ivanhoe =
library.createResource(ivanhoeURI);
Resource walterScott =
library.createResource(walterScottURI);

Property hasAuthor =
library.createProperty("http://library/hasAuthor");

divinaCommedia.addProperty(hasAuthor, danteAlighieri);
ivanhoe.addProperty(hasAuthor, walterScott);
```

Esempio 6: Listato che permette di creare un oggetto di tipo Model

In Esempio 6 ogni chiamata di *addProperty(Property, Resource)* aggiunge una nuova tripla al modello. Si noti che, siccome un `Model` è un insieme di `Statement`, l'aggiunta di duplicati non ha effetti sul modello, cioè essi non vengono aggiunti. In Esempio 7 possiamo vedere come l'interfaccia `Model` definisce un metodo *listStatements()* che restituisce un `StmtIterator` contenente tutti gli `Statement` presenti nel modello. L'interaccia `Statement` offre dei metodi che permettono l'accesso a soggetto, predicato e oggetto della tripla. Un `Object` può essere o una `Resource` o un `Literal`, il metodo getObject() restituisce dunque un oggetto di tipo RDFNode, che è super-classe sia di `Resource` che di `Literal`.

Jena attualmente offre un unico componente interno per il salvataggio dei modelli in modo persistente. Tale componente, `RDBMaker` offre metodi automatici per la creazione e il caricamento di modelli da database. Mentre per la creazione di un modello in memoria è necessaria una singola chiamata a Jena, per la creazione di un modello partendo da un database sono richieste diverse chiamate. I passi che occorre eseguire sono i seguenti:

1. caricare i driver JDBC, questo passo consente al framework di comunicare con l'istanza del database;

```
StmtIterator i = library.listStatements();

while(i.hasNext()){
Statement stmt = i.next();
Resource subject = stmt.getSubject();
Property property = stmt.getPredicate();
RDFNode object = stmt.getObject();
System.out.println("Subject: " + subject.toString());
System.out.println("Property: " + property.toString());
System.out.println("Object: " + object.toString());
System.out.println("**********");
}
```

Esempio 7: Listato che stampa il contenuto di un oggetto di tipo Model

2. creare la connessione vera e propria al database, questo passo permette la creazione di un oggetto java relativo alla connessione;
3. creare un ModelMaker per il database;
4. creare un modello per dati nuovi o preesistenti (su database).

Oltre `RDB-Maker`, Jena offre un componente aggiuntivo (un plugin) per la memorizzazione e l'interrogazione di grafi RDF: `TDB`. Esso può essere utilizzato per il salvataggio dei modelli su una singola macchina in modo non transazionale e ad alte performance.

```
String path = "./TDBmodel/";
String divinaCommediaURI =
 "http://library/DivinaCommedia";
String danteAlighieriURI =
"http://library/DanteAlighieri";

Model persistent = TDBFactory.createModel(path);

Resource divinaCommedia =
persistent.createResource(divinaCommediaURI);
Resource danteAlighieri =
persistent.createResource(danteAlighieriURI);
Property hasAuthor =
persistent.createProperty("http://library/hasAuthor");

divinaCommedia.addProperty(hasAuthor, danteAlighieri);
```

Esempio 8: Esempio di utilizzo di TDB

Un dataset creato utilizzando `TDB` viene salvato in una singola directory contenuta nel file system ed è formato da tre componenti principali:

1. tabella dei nodi, che contiene una rappresentazione dei termini RDF;
2. tabella dei prefissi, che utilizza la tabella dei nodi e gli indici per risalire alle URI partendo dal grafo e passando per i prefissi;
3. triple e indici Quad.

Nell'Esempio 8 viene mostrato un caso d'uso del plugin `TDB`.

2.3 C-SPARQL

Continuous SPARQL[4] (brevemente C-SPARQL) è un nuovo linguaggio, sviluppato nell'ambito del progetto LarKC, per effettuare interrogazioni continue su stream di dati in formato RDF. L'utilizzo di stream in formato RDF garantisce l'interoperabilità tra sistemi differenti che trattano i medesimi dati, aprendo di fatto le porte alla nascita di applicazioni che trattano la conoscenza che si evolve nel tempo.

I linguaggi di interrogazione per il formato RDF (in particolare SPARQL) agiscono su grandi basi di dati statici. Non essendo possibile salvare l'intero contenuto di uno stream, l'approccio classico dei linguaggi di querying non può più essere utilizzato per dati di tipo stream: le query one-shot perdono la propria utilità a favore delle query continue. Il paradigma del linguaggio di interrogazione viene ribaltato: non saranno più i dati ad essere salvati per poi essere interrogati, ma saranno le query ad essere registrate per essere poi riutilizzate ad intervalli regolari sulle finestre di dati.

La combinazione di dati RDF statici e delle informazioni provenienti dallo stream ha portato alla nascita dello *stream reasoning*[12], che permette di abilitare un reasoning di tipo logico e in tempo reale su una grande quantità di dati. I risultati delle query C-SPARQL possono essere considerati l'input di reasoner specializzati che utilizzano, oltre a tali dati, la conoscenza statica riguardante uno specifico dominio per prendere decisioni in tempo reale; in questi scenari il reasoner lavora su pezzi di conoscenza (dinamica), continuamente rinnovati attraverso i dati ottenuti dalle interrogazioni.

2.3.1 RDFStream

C-SPARQL aggiunge ai datatype disponibili per SPARQL gli RDF stream, definiti come una sequenza ordinata di coppie, ognuna formata da una tripla RDF e da un timestamp τ:

$$<subj_i, pred_i, obj_i, \tau_i>$$
$$<subj_{i+1}, pred_{i+1}, obj_{i+1}, \tau_{i+1}>$$

I vari timestamp possono essere considerati come annotazioni delle triple RDF e sono monotonicamente non decrescenti in tutto lo stream (cioè vale sempre $\tau_i \leq \tau_{i+1}$), più precisamente i timestamp non sono strettamente crescenti poiché non è necessario che siano unici, il fatto che un gruppo di triple di una qualsiasi catena abbiano le stesso timestamp significa che tali dati sono arrivati nello stesso istante di tempo.

Come esempio consideriamo uno scenario di Urban Computing in cui gli stream di dati sono associati ai caselli autostradali. In questi stream ogni tripla corrisponde ad un auto che passa attraverso il casello e ogni auto è identificata dalla sua targa. Il predicato della tripla (t:registers) è fisso, mentre il soggetto (*?tollgate*) e l'oggetto (*?car*) sono variabili. Questa configurazione è coerente con le basi di dati RDF i cui predicati sono presi da piccoli vocabolari che costituiscono una sorta di schema, ma l'interpretazione C-SPARQL non fa specifiche assunzioni o richiede particolari restrizioni sui collegamenti delle variabili relative alle triple degli stream. Nell'Esempio 9 è fornito viene mostrato il caso in cui si prendono in considerazione cinque auto che passano da tre caselli differenti.

Triples	**Timestamp**
c:Distr1 t:registers 156	t100
c:Distr2 t:registers 75	t101
c:Distr1 t:registers 130	t102
c:Distr2 t:registers 95	t103
c:Distr3 t:registers 65	t104

Esempio 9: Esempio di dati appartenenti a uno stream

2.3.2 Finestre

L'introduzione degli stream di dati in C-SPARQL richiede l'abilità di identificare alcune fonti di dati, associando ad ogni stream un IRI[10]

[10] *Internationalized Resource Identifier*, una forma di URI che utilizza solo caratteri ASCII per una maggiore compatibilità in contesti internazionali

```
FromStrClause -> 'FROM' ['NAMED'] 'STREAM' StreamIRI '
[ RANGE' Window ']'
Window -> LogicalWindow | PhysicalWindow
LogicalWindow -> Number TimeUnit WindowOverlap
TimeUnit -> 'ms' | 's' | 'm' | 'h' | 'd'
WindowOverlap -> 'STEP'   Number TimeUnit | 'TUMBLING'
PhysicalWindow -> 'TRIPLES' Number
```

Esempio 10: Query in linguaggio C-SPARQL

differente (che rappresenta la fonte dello stream), e di specificare dei criteri di selezione su di essi tramite l'utilizzo di finestre (per nozioni sulle finestre vedi Paragrafo 2.1). Esempio 10 mostra come sia possibile identificare, tramite la clausola *FROM STREAM*, il flusso di dati su cui effettuare le interrogazioni.

Nel Codice 2.1 consideriamo l'interrogazione che estrae tutti i film visti dagli amici di John negli ultimi 15 minuti. La query considera solo gli ultimi 15 minuti (*dimension*) e la finestra scorrevole viene modificata ogni minuto (*step*), rinnovando, in questo modo, i risultati estratti dall'interrogazione.

```
1  SELECT DISTINCT ?topic
2  FROM STREAM http://streamingsocialdata.org/interact.trdf
3     [RANGE 15m STEP 1m]
4  WHERE {
5     ?user sd:accesses ?document .
6     ?user foaf:knows ?john .
7     ?john foaf:name "John" .
8     ?document :describes ?topic .
9     ?topic skos:subject yago:Movies .
10 }
```

Codice 2.1: Query in linguaggio C-SPARQL

La query interroga sia conoscenza statica che dati provenienti da uno stream. Per prima cosa tutte le triple che hanno sd:accesses come predicato vengono estratte dalla finestra corrente, dopodiché viene effettuato il matching applicando le condizioni di join espresse dal collegamento tra le variabili ?user e ?document per identificare il ?topics osservato. I risultati delle query riguardano solo i dati contenuti all'interno della finestra e le triple che arrivano sullo stream durante l'intervallo scorrimento (*step*) vengono messe in coda, in attesa dello scorrimento successivo e fino a quel momento, non contribuiscono al risultato.

2.3.3 Query Registration

Come accennato all'inizio del Paragrafo 2.3, l'utilizzo di uno stream come fonte di informazione fa nascere la necessità di salvare le query in modo da poterle utilizzare più volte. Tutte le interrogazioni che riguardano dati di tipo RDF vengono dette continuous query, tale denominazione è dovuta al fatto che queste producono continuamente dati, sotto forma di tabelle, variabili o grafi RDF. Tuttavia una query registrata non produce come risultato uno stream di dati.

```
Registration   -> 'REGISTER QUERY' QueryName
        ['COMPUTED EVERY' Number TimeUnit]
        'AS' Query
```

Esempio 11: Porzione di codice che permette di registrare un query

L'Esempio 11 mostra come le query possano essere registrate in memoria tramite la clausola *REGISTER QUERY* per poi essere rieseguite ad intervalli regolari di tempo (clausola *COMPUTED EVERY*).

```
1  REGISTER QUERY GlobalCountOfInteractions
2      COMPUTED EVERY5m AS
3  SELECT ?user
4  COUNT(?document) as ?numberOfInteractions
5  COUNT((DISTINCT ?topic) as ?numDifferentTopics
6  FROM STREAM <http://streamingsocialdata.org/interact.trdf>
7      [RANGE30m STEP5m]
8  WHERE {
9      ?user sd:accesses ?document .
10     ?document sd:describes ?topic .
11 }
12 GROUP BY {
13     ?user
14 }
```

Codice 2.2: Query in linguaggio C-SPARQL

Il Codice 2.2 riporta un piccolo esempio che riguarda il salvataggio delle interrogazioni: l'interrogazione conta il numero di interazioni svolte da ogni utente negli ultimi 30 minuti e il numero di topics distinti a cui si riferisce il documento preso in considerazione dall'utente.

2.3.4 Stream Registration

I risultati di una query C-SPARQL possono anche essere sotto forma di stream RDF, per generare questo flusso continuo di risultati, la query va registrata utilizzando la sintassi mostrata nell'Esempio 12.

```
Registration   -> 'REGISTER STREAM' QueryName
        ['COMPUTED EVERY' Number TimeUnit]
        'AS' Query
```

Esempio 12: Porzione di codice che permette di registrare uno stream

La clausola *COMPUTED EVERY* è opzionale e ha lo stesso significato del caso riguardante la registrazione delle query. Se non è indicata nessuna frequenza di aggiornamento, la query viene eseguita a una frequenza automaticamente determinata dal sistema.

```
1  REGISTER STREAM MoviesJohnsFriendsLike
2      COMPUTED EVERY 5m
3  AS CONSTRUCT{?user sd:likes ?document}
4  FROM STREAM <http://streamingsocialdata.org/interact.trdf>
5      [RANGE30m STEP5m]
6  WHERE {
7      ?user sd:likes ?document .
8      ?user foaf:knows ?john .
9      ?john foat:name "John" .
10     ?document sd:describes ?topic .
11    ?topic skos:subject yago:Movies .
12 }
```

Codice 2.3: Query in linguaggio C-SPARQL

Il Codice 2.3 mostra un piccolo esempio di registrazione di uno stream. Il risultato è uno stream RDF, che può contenere da un minimo di zero triple fino ad un massimo di triple corrispondente al contenuto dell'intero grafo. Il timestamp delle triple in uscita dipende dall'istante di esecuzione della query e non viene estrapolato dalle triple su cui viene effettuata l'interrogazione.

2.3.5 C-SPARQL Engine

La Figura 2.6 mostra il design architetturale del motore C-SPARQL sviluppato nell'ambito del progetto LarKC. Tale motore è in grado di registrare ed eseguire le query in modo continuo. L'architettura è composta essenzialmente di tre componenti.

Al primo modulo, chiamato `C-SPARQL Query Parser`, viene data in pasto una query C-SPARQL e produce in uscita i dati necessari al `Data Stream Manager Layer` e allo `SPARQL Endpoint Layer` per eseguire la query. Il `Data Stream Manager` registra gli stream specificati nell'interrogazione e applica finestre logiche o fisiche, producendo degli snapshot temporizzati in formato RDF (formati da triple RDF, ognuna delle quali associata ad un timestamp). Quando il grafo risultante è stato prodotto, la parte SPARQL della query C-SPARQL viene eseguita dallo `SPARQL Endpoint`. Tale processo

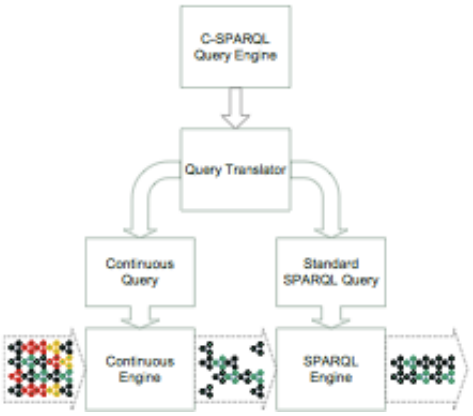

Figura 2.6: Architettura dello C-SPARQL engine

viene eseguito tanto frequentemente quanto specificato nella clausola REGISTER della query C-SPARQL. Infine, al risultato ottenuto viene aggiunto un timestamp e viene restituito in output. Sia il data stream manager che lo SPARQL endpoint vengono considerati plugin, al fine di essere indipendenti dall'attuale implementazione di DSMS utilizzata.

2.4 Restlet

Possiamo definire le Restlet come un framework con supporto alla tecnologia REST, leggero e open source, sviluppato per piattaforme Java, possiamo vederne la struttura in figura 2.7. Con l'acronimo REST (*Representational State Transfer*) ci riferiamo a una stile architetturale di sviluppo del software per ambienti con contenuti ipermediali distribuiti, come il World Wide Web.

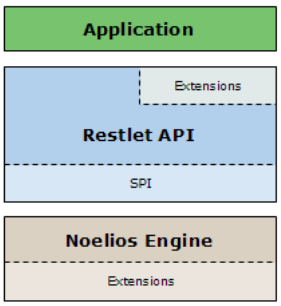

Figura 2.7: Struttura del Restlet Framework

Le Restlet sono caratterizzate da una grande adattabilità, sia lato client che lato server, e grazie al supporto relativo alla maggior parte degli standard web (*HTTP, HTTPS, SMTP, XML, JSON*, etc), sono andate a riempire una grande mancanza nell'ambiente Java, che non disponeva di un framework adatto a trattare con ambienti di tipo REST.

Il supporto agli standard Web è possibile grazie all'utilizzo dei connettori(*Connector*), elementi software che permettono la comunicazione tra componenti implementando una parte del protocollo di rete e che si dividono essenzialmente in due tipi (di cui citiamo le principali implementazioni):

- Server connectors: Servlet, Spring, Jetty, Grizzly, Simple, JAXB, JAX-RS, JiBX, Velocity;
- Client connectors: Internal, Apache HTTP Client, Net (JDK's HttpURLConnection), JavaMail, JDBC.

Nell'ambito Java questo framework si presenta come un sostituto ideale per le Servlet, anche se i due ambienti possono lavorare insieme, all'interno della release 0.19 beta viene anche fornito un connettore che permette l'esecuzione di una Restlet attraverso un web container come *Tomcat*, permettendo una transizione ancora più dolce dalle Servlet verso l'ambiente REST.

Capitolo 3

Streaming Linked Data for Social Media: Progetto della Soluzione

In questo capitolo mostreremo e giustificheremo le scelte architetturali che hanno guidato la creazione di SLD4SM. Partiremo presentando le problematiche che abbiamo dovuto affrontare, discutendo le possibili soluzioni, e proseguiremo andando nel dettaglio della progettazione concettuale dell'architettura.

3.1 Problematiche Affrontate

Nello sviluppo di SLD4SM abbiamo dovuto affrontare problematiche di vario tipo. Uno dei primi problemi ha riguardato il modo di integrare dati di tipo stream con dati di tipo statico (o semi-statico).

I dati stream sono essenzialmente sequenze di informazioni prodotte in modo continuo e ordinate in base al timestamp di arrivo o a quello di generazione dell'informazione. Visto che tali sequenze sono generate in modo continuo, esse potrebbero essere infinite o di dimensione sconosciuta. Data la natura profondamente mutevole delle informazioni nell'ambito di sorgenti di dati di tipo stream, i dati vengono, in molti casi, considerati validi solo instantaneamente, o comunque per un limitato periodo (detto *expire time*). Per SLD4SM abbiamo deciso di modellare le informazioni in stream RDF, ovvero in dati formati da una tupla RDF `<soggetto, predicato, oggetto>` e dal timestamp di generazione della tupla. La scelta è ricaduta su questo formato per la compatibilità con il linguaggio C-SPARQL per l'interrogazione su stream RDF e, conseguentemente, con lo `C-SPARQL Engine`.

Per quanto riguarda i dati statici, vista la natura poco variabile delle informazioni e la semplicità espressiva, abbiamo deciso di modellizzarli tramite classiche triple RDF:

<soggetto, predicato, oggetto>.

Un altro problema era rappresentato dal metodo di estrazione dei dati dai Social Network. Il mondo delle reti sociali è in continua espansione, la sua penetrazione nella società e nella vita di tutti i giorni è sempre maggiore e la quantità di informazioni utili, da poter estrarre da questa enorme mole di dati è in continua crescita. Anche se ogni social network si specializza in campi diversi, tutti hanno un unico scopo, permettere all'utente di condividere informazioni relative a qualche argomento.

Nonostante tutte le differenze relative ai diversi metodi di salvataggio delle informazioni, possiamo notare la presenza di uno schema informativo minimo comune (vedi Figura 3.1) alla base di tutti i siti sociali che abbiamo preso in considerazione.

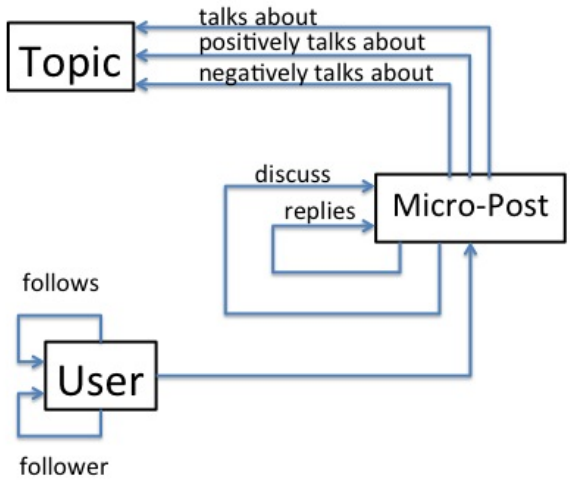

Figura 3.1: Schema comune di rappresentazione dell'informazione nei Social Network

In tale schema possiamo notare i tre frammenti informativi principali su cui si basa tutto il mondo sociale:

- *soggetto*: rappresenta l'utente che è intervenuto;
- *topic*: rappresenta l'oggetto attorno a cui ruota l'intervento dell'utente;
- *micro-post*: rappresenta l'intervento vero e proprio ed è il collante che lega il soggetto all'oggetto della discussione.

Ad ogni singola parte di questo schema sono associate informazioni aggiuntive che ci aiutano a comprenderne l'utilità. Ad ogni utente è solitamente associato un profilo, che comprende una sua descrizione e la sua rete sociale, formata da amici e follower. I post possono essere seguiti, citati, geolocalizzati, etc, ma cosa più importante possono parlare bene o male di un dato oggetto (*topic*). Un *topic* avrà associata una descrizione, una categoria, la lista degli utenti a cui è piaciuto o che non è piaciuto e altri attributi che lo rappresentano nella realtà.

Il principale problema legato all'estrazione dei dati dalla rete, nasce dalla estrema disomogeneità dei formati presenti su Web e dalle differenti proposte per accedervi. La maggior parte dei Social Network mette a disposizione delle API per interagirvi senza la necessità di collegarsi tramite browser. Le Web API più comuni sono di tipo *pull*, cioè è il client a dover effettuare continue richieste al server per ottenere nuovi dati. D'altra parte, meccanismi più evoluti di accesso permettono di ricevere nuovi dati tramite una vera e propria interfaccia stream (API STREAM), che in modo nativo fornisce un flusso continuo di informazione tramite notifiche push dal server al client. Generalmente le Web API sono soggette a rate limiting, il che ha influenzato la scelta dei Social Network ai quali appoggiarci, mentre le API STREAM non hanno questo tipo di limitazione ma sono comunque ancora rare nel panorama del Web.

Una volta trovato il metodo per ottenere i dati dai social network è stato necessario pensare ad una struttura dati comune da utilizzare in tutto il sistema. In questo ci è stata molto d'aiuto la proposta del team LarKC per la pubblicazione di data stream[5]. La proposta (per maggiori dettagli vedi Paragrafo 3.4.1) prevede l'utilizzo dei *Named Graph*[5], in particolare ne prevede di due tipi: gli *Instantaneous Graph* (in breve *iGraph*) e gli *Stream Graph* (in breve *sGraph*). Gli *iGraph*, organizzati in base al timestamp, contengono tutti i dati ricevuti con lo stesso timestamp; un *sGraph* contiene i riferimenti a tutti gli *iGraph* facenti parte di un certo intervallo di tempo. Tale intervallo rappresenta una finestra (Window). Questo tipo di soluzione è risultata utile e ha ispirato la soluzione da noi proposta per la pubblicazione dei dati, per il salvataggio della conoscenza statica e dello stream. Per il salvataggio dello stream abbiamo mantenuto il concetto di *iGraph* e introdotto i *Recording Graph* (in breve *rGraph*, per dettagli vedi Paragrafo 3.4.1) per indicizzare gli *iGraph*. Gli *rGraph* sono una estensione degli *sGraph*, e a loro differenza tengono i riferimenti a tutti gli *iGraph* dal momento in cui è partita la registrazione fino al suo termine. Per il salvataggio dei dati statici abbiamo introdotto due tipi di *Named Graph*:

gli *Active Facts Graphs* (in breve *afGraph*) sono modelli RDF simili agli *iGraph* che contengono le informazioni vere e proprie, gli *Active Knowledge Graph* (in breve *akGraph*) sono dei modelli RDF simili agli rGraph utili per indicizzare tutti gli *afGraph* (per dettagli vedi Paragrafo 3.4.1).

Per modellare e gestire i *Named Graph* avevamo bisogno di un application framework che supportasse RDF. Vista la sua estrema compatibilità con Java e i suoi punti di forza legati alle prestazioni e alla facilità di utilizzo nella rappresentazione e manipolazione di modelli RDF abbiamo scelto di utilizzare il framework Jena (per dettagli vedi Paragrafo 2.2.4). Jena possiede una struttura dati (`Model`) che si sposa perfettamente con le nostre necessità e che ci ha permesso di creare la struttura dati per l'interscambio dei dati tra le componenti di SLD4M, il `TimestampedModel`, la cui forma è `<Model, timestamp>`, che sarà trattato in modo approfondito nel Paragrafo 4.1.1.

Trovati gli schemi comuni di accesso, interazione e salvataggio dei dati dai Social Network è stato necessario progettare gli oggetto in grado di svolgere tali operazioni. Per rendere SLD4SM il più interattivo possibile abbiamo deciso di aggiungere dei metodi per attivare, mettere in pausa, riattivare e spegnere tali oggetti (per dettagli vedi Paragrafo 4.1).

Una volta giunti a questo punto abbiamo fatto una riflessione: il mondo sempre più connesso costringe ogni applicazione ad avere un'interfaccia che renda fruibili i suoi servizi via Web, come è possibile offrire un controllo remoto completo mantenendo, allo stesso tempo, un elevato livello di sicurezza e di integrità del sistema? Questa necessità di controllo sul sistema, che comprende creazione, registrazione e pubblicazione di uno stream di dati oltre alla possibilità di effettuare interrogazioni tramite lo `C-SPARQL Engine` sugli stream creati, ci ha portato alla creazione di Web API. Attraverso esse ed il protocollo HTTP è possibile avere un controllo completo del sistema mantenendo il funzionamento interno trasparente all'utilizzatore (maggiori dettagli nel Paragrafo 4.3).

Passiamo ora a una rassegna approfondita di tutte le soluzioni adottate, partendo da una rappresentazione ad alto livello dell'architettura generale, utile per avere una visione completa dell'intera infrastruttura.

3.2 Architettura Generale

Tutta la nostra infrastruttura software nasce dalla necessità di estrarre e manipolare dati grezzi in modo da ricavarne, in modo efficiente, informazione utile, mettondola poi a disposizione del sistema in un formato standard.

Riprendendo in modo più approfondito il contenuto del paragrafo 1.2 e ampliandolo, possiamo anticipare in modo superciale i compiti dei vari componenti, per poi riprenderli in modo più specifico; `SLD4SM`, come mostrato in figura 3.2, dovrà quindi essere in grado di:

- estrarre dati in tempo reale dal Web, o da un supporto di memorizzazione in cui sono stati precendentemente salvati i dati (compito svolto rispettivamente dagli `Adapter` e dai `Replayer`);
- manipolare l'informazione grezza trasformandola in un formato standard, attraverso il salvataggio di uno stream, la sua pubblicazione su un supporto persistente o una sua pubblicazione in memoria (compito riservato al blocco `SLD4SM Core` e ai suoi componenti interni);
- permettere la registrazione di uno stream precedentemente creato e di query ad esso collegate all'interno dello `C-SPARQL Engine`;
- formattare l'output dello `C-SPARQL Engine` in modo da renderlo compatibile con il resto dell'architettura (compito riservato al `C-SPARQL Adapter`) e viceversa;
- invocare qualsiasi operazione da remoto tramite l'esposizione di interfacce REST.

Proviamo ad andare un po più in profondità nella presentazione di `SLD4SM` che è divisa in cinque macrocomponenti ben distinti:

1. `Adapter`: sono degli oggetti il cui compito è connettersi a servizi presenti sul web per recuperare i dati da cui partire per generare uno stream RDF; come si può notare dalla figura ogni `Adapter` avrà come input un insieme di dati formattati in modo dipendente dalla piattaforma da cui vengono estrapolati, ma dovranno dare in output informazione in un formato predefinito;
2. `SLD4SM Core`: tale componente può essere considerato il cuore del sistema, grazie ad esso è possibile generare uno stream RDF partendo dai dati provenienti dagli `Adapter`, registrare tale stream e rigenerarlo;

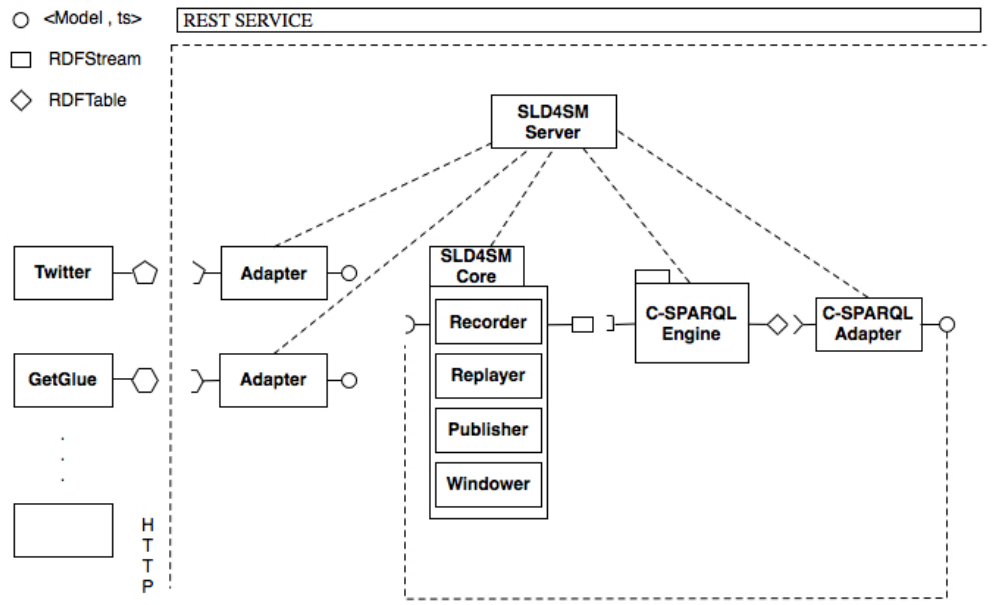

Figura 3.2: Architettura generale del sistema

3. `C-SPARQL Engine`: è il componente sviluppato dal team LarKC che permette di registrare ed eseguire query continue su stream RDF

4. `C-SPARQL Adapter`: è essenzialmente un Adapter il cui input non è nient'altro che l'output del motore C-SPARQL;

5. `SLD4SM Server`: questo oggetto potrebbe essere definito come il telecomando di tutto il sistema in quanto rappresenta il front-end nei confronti dei vari client, offrendo i metodi necessari per controllarne tutte i componenti.

Da un punto di vista concettuale possiamo dire che i fornitori di dati per il sistema sono essenzialmente gli `Adapter`, i `Replayer` e i `C-SPARQL Adapter`. La suddivisione in tre oggetti distinti, nasce in modo naturale prendendo in considerazione le fonti da cui vengono estratti i dati:

- `Adapter`: le informazioni utili vengono prese direttamente dalla rete;
- `Replayer`: le informazioni vengono prese da supporti di memorizzazione su cui precendentemente sono stati salvati i dati;
- `C-SPARQL Adapter`: le informazioni vengono messe a disposizione dallo `C-SPARQL Engine`.

Nonostante le differenze evidenti, possiamo trovare dei punti comuni tra i tre componenti, tra cui la possibilità di essere comandati da `SLD4SM Server` (creati, messi in pausa o distrutti) e la formattazione finale dei dati; queste somiglianze permettono di classificare tutti i tipi di fornitori di dati come `DataGetter` (non riportato in Figura 3.2 e discusso in dettaglio nel paragrafo 4.1.1).

L'obiettivo finale è la completa trasparenza all'utente di tutte le operazioni di acquisizione dati e la loro trasformazione in un formato standard, fruibile da diverse applicazioni in modo completamente indipendente dal contenuto dei dati e dal servizio a cui i dati vengono richiesti; questo sforzo di omogeneizzazione di tutti gli output dei vari nodi del sistema è necessario per affrontare un ambiente sempre più connesso e con sempre più sete di informazioni, che è necessario aggregare e prendere da diversi servizi online.

Passiamo ora ad una visione più dettagliata dei vari componenti.

3.2.1 DataGetter

Il componente `DataGetter` rappresenta il generico oggetto che fornisce i dati al sistema, rappresenterà quindi il prototipo per gli `Adapter`, i `Replayer` e lo `C-SPARQL Adapter`. Al suo interno sono presenti tutte le caratteristiche e le funzionalità che accomunano i componenti che la estendono e che ne modificano i metodi per raggiungere scopi più specifici.

3.2.2 Adapter

Ogni `Adapter` ha il compito di collegarsi ad uno specifico servizio presente in rete e recuperare i dati utili per essere manipolati dal resto del sistema. La maggior parte dei Social Media presenti sul Web mette a disposizione di utenti e sviluppatori API che consentono una interazione diretta con tali sistemi, senza dover passare obbligatoriamente per i browser. Analizzando un buon numero di Social Media, abbiamo deciso di focalizzarci su quelli che potevano offrirci un buon compromesso tra qualità e quantità dei dati. Tra questi sono subito risaltate le profonde differenze nei paradigmi di offerta dei dati e ci siamo resi conto che le API per interagire con tali servizi, possono essere essenzialmente di due tipi: API REST e API Stream. Sebbene queste ultime siano le più utili ai nostri fini fornendo un flusso continuo di dati, esse sono ancora ben poco utilizzate a favore delle API REST, le quali, invocate periodicamente, generano degli snapshot di dati.

La difficoltà maggiore che abbiamo dovuto affrontare durante la progettazione di questi componenti è stata l'estrema eterogeneità dei servizi offerti dai Social Media; infatti, l'offerta molto diversificata delle modalità di autenticazione e l'estrema disomogeneità per quanto riguarda i formati con cui vengono forniti i dati sono state un grosso ostacolo al tentativo di standardizzazione di tali componenti. Nonostante tali differenze siamo stati in grado di identificare i tre compiti essenziali che deve svolgere un Adapter:

1. autenticarsi presso la sorgente dati;
2. ricevere informazioni dalla sorgente i dati;
3. generare in output uno stream di dati.

Una volta identificate le caratteristiche che accomunano tutti gli `Adapter`, abbiamo deciso di appoggiarci su due Social Media completamente differenti: Twitter e GetGlue. Entrambi mantengono le caratteristiche generali di cui abbiamo parlato, ma si diversificano, e quindi danno maggiori possibilità di validazione della scelta progettuale, essenzialmente per i due motivi che ora andremo ad illustrare.

Il primo motivo è che pur utilizzando entrambi come metodo di autenticazione OAuth, richiedono metodi diversi per il passaggio delle chiavi di autenticazione. Il secondo motivo è che forniscono due tipologie differenti di API: GetGlue offre unicamente API di tipo REST, mentre Twitter, oltre queste, è in grado di offrire API di tipo Stream, le quali risultano molto utili per via delle limitazioni (rate limit) imposte da Twitter sulle API REST.

Twitter API

Twitter può essere considerata una piattaforma aperta e, grazie all'offerta di apposite API, permette di allargare gli orizzonti del social network includendo un intero ecosistema di partner esterni che contribuiscono alla crescita esponenziale del fiume di informazioni che passa attraverso Twitter. Tali partner a volte sono solo semplici sperimentatori, in altri casi sono vere e proprie aziende che basano il loro business sulla raccolta di informazioni di carattere sociale ed economico, da riutilizzare o rivendere ad altri. Per tale motivo Twitter fissa una serie di regole da seguire per la creazione di prodotti e servizi che interagiscono con il social network, che tutelano sia gli interessi commerciali dei partner che quelli del singolo utente; tali regole mostrano cosa i partner possono fare con i contenuti e le

informazioni condivisi tramite Twitter. Le API vengono suddivise in tre categorie:

- REST API: permettono agli sviluppatori l'accesso ai dati core di Twitter, quali per esempio le timeline (ovvero l'elenco dei Tweet delle persone seguite dal singolo utente), i dati relativi allo status e le informazioni degli utenti;
- Search API: permettono agli sviluppatori di interagire con la ricerca integrata di Twitter e con i trends data (ovvero le persone e gli argomenti più seguiti dalla comunità)
- Streaming API: permettono agli sviluppatori di accedere ad un elevato volume di tweet in tempo reale (o quasi), dando la possibilità di filtrarli in vari modi.

La limitazione principale delle REST API e delle Search API è rappresentata dal rate-limiting, quindi il loro utilizzo per l'interazione con grosse moli di dati è problematico, d'altro canto, le Streaming API permettono una singola connessione a lungo termine che non presenta tali limitazioni.

GetGlue API

Le API di GetGlue permettono di avere accesso all'intera libreria di risorse del sito (film, canzoni, serie televisive ecc), che è formata da circa 3 milioni di oggetti. Le API sono di tipo REST e il loro utilizzo permette agli sviluppatori di leggere/scrivere ogni oggetto di GetGlue.

Le possibilità offerte da queste librerie permettono di rendere il social network una piattaforma aperta, fonte di conoscenza che potrà essere aggregata con altra proveniente dalle fonti più diverse. Le API di GetGlue possono essere suddivise in diverse categorie, di cui parleremo in modo approfondito in seguito.

Ogni singola categoria di API permette all'utente di leggere/-scrivere tutte le informazioni riguardanti oggetti o utenti, inoltre GetGlue permette l'utilizzo del motore semantico AdaptiveBlue che dà accesso ai metadati riguardanti milioni di oggetti sparsi per la rete.

3.2.3 C-SPARQL Adapter

SLD4SM permette la registrazione di query sullo stream generato dai `DataGetter` attraverso lo `C-SPARQL Engine`, il quale è in grado di restituire in output i risultati di tali interrogazioni in una

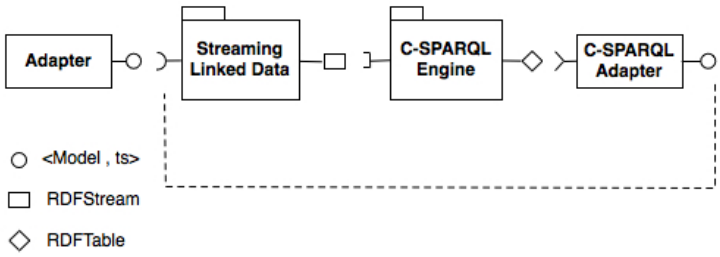

Figura 3.3: Particolare dei componenti che permettono il funzionamento dello C-SPARQL Engine

formattazione propria del motore. Un `C-SPARQL Adapter` è un componente il cui compito consiste nel registrare un RDFstream e una query relativa allo stream stesso, prendere i dati in uscita dallo `C-SPARQL Engine` e formattarli secondo le specifiche dell'intero sistema.

In Figura 3.3 mostriamo un particolare dell'architettura generale che comprende i componenti utili al funzionamento dello `C-SPARQL Engine` e per la formattazione dei dati in uscita da quest'ultimo.

I dati provengono da un `Adapter` (o indifferentemente da un `Replayer`), passano attraverso `SLD4SM Core` che ne permette la pubblicazione su RDF stream, un formato compatibile con il motore C-SPARQL; quest'ultimo produce in output una rappresentazione tabulare dei risultati di una query C-SPARQL chiamata RDFTable, che viene data in ingresso al C-SPARQL Adapter che manipola i dati in modo da presentare in uscita uno stream di dati compatibile con tutti gli altri componenti, dandoci la possibilità di offrire nuovamente queste informazioni in input al `SLD4SM Core`.

3.2.4 SLD4SM Core

`SLD4SM Core` è un pacchetto i cui componenti hanno la capacità di manipolare i dati forniti dai `DataGetter`.

La principale caratteristica di tutti i componenti di `SLD4SM Core` è la loro capacità di manipolare i dati indipendentemente dal loro contenuto, ciò dà la possibilità di applicare qualsiasi funzione offerta da `SLD4SM Core` ai dati provenienti da una qualunque fonte, sia essa già stata sviluppata da noi oppure una qualsiasi fonte sviluppata nel futuro.

Le funzionalità offerte da `SLD4SM Core` sono quelle base dell'intero sistema e permettono di manipolare uno stream di dati, salvandolo, pubblicandolo o rigenerandolo, garantendoci una gestione più mirata dei carichi di lavoro dovuti all'estrazione di informazione

utile da grandi moli di dati. I componenti di `SLD4SM Core` possono essere divisi in quattro tipologie: i `Recorder`, i `Replayer`, i `Publisher` e i `Windower`.

Andiamo ad esaminare le caratteristiche e le funzionalità offerte da ognuna di esse.

Recorder

Un `Recorder` ha il compito di salvare i dati provenienti da un `Adapter`, o indifferentemente da un `Replayer`, in modo da consentirne un riutilizzo differito nel tempo.

Un oggetto di questo tipo può salvare lo stream sotto forma di snapshot di dati utilizzando i `Named Graph`, secondo la convenzione accennata nella sezione 3.1 e che approfondiremo in seguito nella sezione 3.4.1. In particolare il salvataggio degli snapshot sotto forma di iGraph può avvenire secondo tre diverse modalità, le quali hanno reso necessaria la creazione di tre tipi differenti di `Recorder`:

1. `RecorderDisk`, permette il salvataggio degli iGraph e del relativo rGraph su disco sottoforma di file RDF;
2. `RecorderDB`, permette il salvataggio degli iGraph e del relativo rGraph sotto forma di tabelle in un database MySQL;
3. `RecorderTDB`, permette il salvataggio degli iGraph e del relativo rGraph su disco secondo le specifiche del componente TDB di Jena.

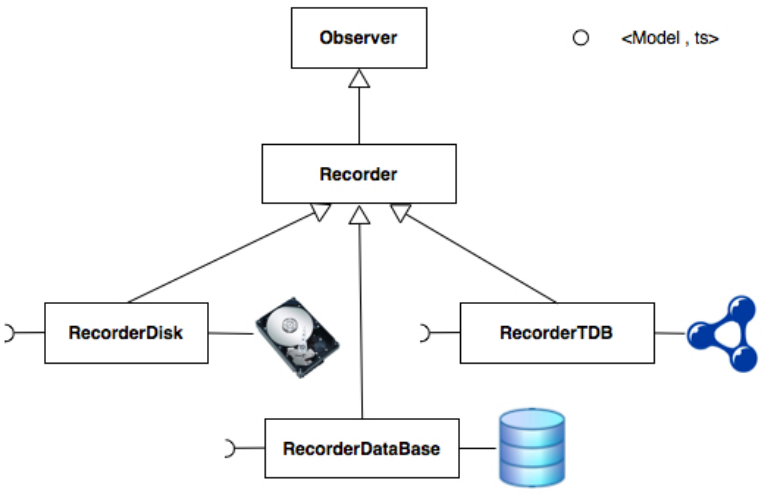

Figura 3.4: Struttura di un componente Recorder

La scelta del metodo di salvataggio dei dati e dell'ubicazione di tale registrazione viene lasciata all'utente in modo da rendere più flessibile tale meccanismo.

Replayer

`Replayer` presenta una particolarità rispetto a tutti gli altri componenti di `SLD4SM Core`: esso non è un manipolatore di informazioni ma piuttosto un produttore di dati e per tale caratteristica, unitamente al suo funzionamento e alle caratteristiche del suo output, può essere equiparato ad un `Adapter` (In Figura 3.5 possiamo vedere come entrambi condividano la super classe `DataGetter`). La capacità dell'oggetto `Replayer` di far rivivere uno stream salvato in locale sotto forma di snapshot di dati, consente un utilizzo più equilibrato di tutte le componenti del sistema, permettendo la riduzione del carico di lavoro e dando la possibilità di frammentare i compiti in istanti di tempo indipendenti tra loro. Il suo utilizzo, chiaramente legato ai compiti svolti dagli oggetti `Recorder`, permette di elaborare le informazioni contenute in un flusso di dati in un tempo successivo rispetto alla sua creazione.

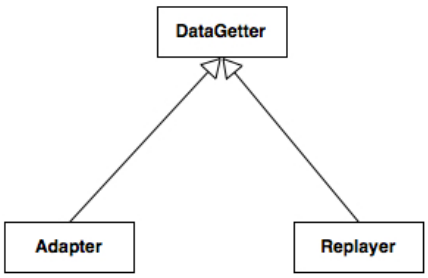

Figura 3.5: Struttura di un componente DataGetter

Gli intervalli di tempo che intercorrono nella realtà tra i vari snapshot di dati, vengono rispettati durante le operazioni di replay dello stream e l'utente può controllare la velocità di esecuzione, in modo da rendere l'esperienza di replay il più reale possibile.

Dato che un oggetto `Recorder` può registrare lo stream di dati in tre differenti modalità, è necessario che `Replayer` sia in grado di rigenerare lo stream partendo da queste. Avremo quindi tre tipi di `Replayer`:

1. `ReplayerFromDisk`, lo stream viene rigenerato partendo dai dati salvati sottoforma di file RDF;

2. `ReplayerFromDB`, lo stream viene rigenerato partendo dai dati salvati in un database MySQL;
3. `ReplayerFromTDB`, lo stream viene rigenerato partendo dai dati salvati sotto forma di TDB.

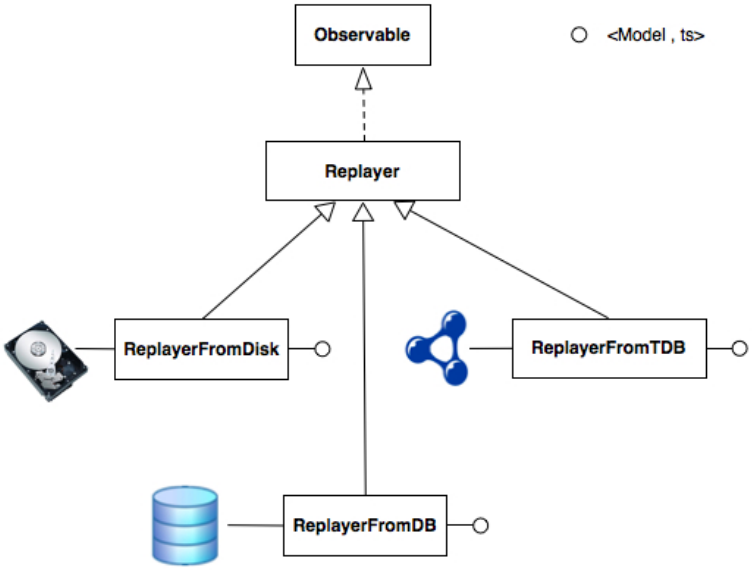

Figura 3.6: Struttura di un componente Replayer

Publisher

La pubblicazione vera e propria di uno stream di dati viene delegata all'oggetto `Publisher`. Il suo compito principale è quello di presentare l'insieme di elementi atomici che compongono il flusso di dati su una piattaforma scelta dall'utente. L'azione di publishing più importante è sicuramente quella legata all'inserimento delle informazioni in uno stream RDF, in modo da rendere disponibili i dati a tutta una serie di applicazioni che agiscono nell'ambito del Web Semantico e che richiedono flussi informativi strutturati secondo il framework RDF (nel nostro caso `C-SPARQL Engine`).

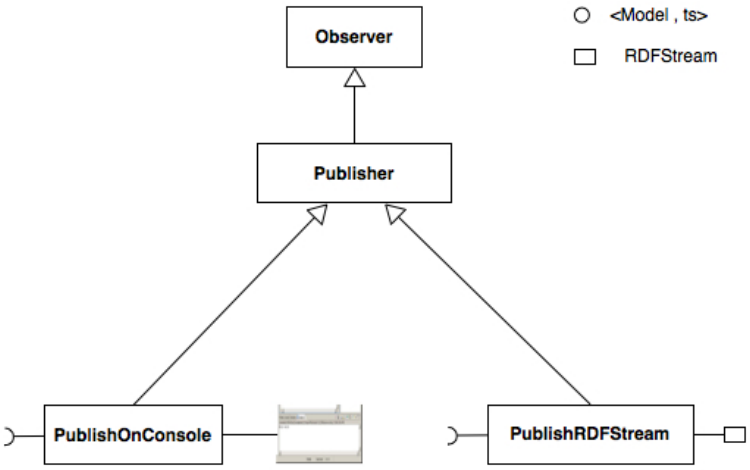

Figura 3.7: Struttura di un componente Publisher

Windower

Un `Windower` è un componente che concettualmente dovrebbe rientrare nella categoria `Publisher` ma il cui funzionamento può essere equiparato a quello di un `Recorder`. Il compito di un `Windower` è quello di tenere in memoria delle finestre di dati di grandezza e durata prestabilita (Vedi sezione 2.1). E' compito dell'utente scegliere il tipo di finestra (logica o fisica), la sua dimensione e, nel caso in cui si tratti di una finestra logica, quanto spesso debba essere rigenerata la finestra. Come avviene per i `Recorder` i dati vengono organizzati in `iGraph` e il grafo contenente le referenze a tali grafi non si chiamerà più `rGraph` ma prenderà il nome di `sGraph`. E' importante sottolineare che i grafi non vengono salvati su alcun tipo di supporto ma restano in memoria per essere forniti con minor latenza ai possibili richiedenti.

3.3 SLD4SM e i Design Pattern

Nella progettazione di qualsiasi tipo di applicativo ci si ritrova ad affrontare dei problemi di carattere generale molto comuni. Un design pattern può essere definito come una soluzione progettuale generale ad un problema ricorrente. Esso non è una libreria o un componente software riusabile, quanto piuttosto una descrizione o un modello da applicare per risolvere un certo tipo di problema che può presentarsi in diverse situazioni durante la progettazione e lo sviluppo del software. E' bene sottolineare che vi è una grossa differenza fra

algoritmo e design pattern: il primo risolve i problemi di carattere computazionale, mentre il secondo è legato agli aspetti progettuali del software.

Le prossime sezioni saranno dedicate a una descrizione concettuale di tutti i design pattern utilizzati nell'architettura, in modo da rendere più chiare le loro caratteristiche e l'utilizzo che ne è stato fatto all'interno di `SLD4SM`.

3.3.1 Ereditarietà e overloading

Alcune componenti di `SLD4SM`, sebbene in prima battuta possano sembrare differenti fra loro, presentano delle caratteristiche simili che permettono di schematizzarne il funzionamento in una serie di azioni standard. E' il caso degli `Adapter` e delle componenti di `SLD4SM Core`, ovvero dei `Recorder`, dei `Publisher`, dei `Replayer` e dei `Windower`. In particolare gli `Adapter`, risultano differenti tra loro nel metodo di autenticazione e nel metodo di reperimento dei dati - questo perchè si collegano a sorgenti dati eterogenee - ma presentano delle caratteristiche comuni che li rendono omogenei all'occhio dell'utente: tutti, infatti, devono essere in grado di autenticarsi presso la sorgente dati a cui sono associati, ottenere in input i dati e restituire in output i dati così ottenuti secondo una determinata formattazione sotto forma di stream RDF.

Per quanto riguarda `SLD4SM Core` è molto più semplice trovare delle similitudini tra gli oggetti; se prendiamo come esempio le componenti che effettuano la registrazione dello stream - `RecorderDisk`, `RecorderDB` e `RecorderTDB` - esse devono tutte essere in grado di ottenere lo stream restituito dai `DataGetter` e di registrarlo in un formato prestabilito. Il supporto di memorizzazione, sia esso disco, database o modello persistente, passa in secondo piano rispetto alla caratteristiche che accomunano tali oggetti, le quali, una volta identificate, permettono una macro-progettazione forte e coerente. Tale discorso può certamente essere ripetuto sia per gli oggetti `Replayer` che per gli oggetti `Publisher`: tutti i `Replayer` hanno in comune la capacità di rigenerare lo stream - in modo coerente a quello originale - partendo dai dati precedentemente salvati dai `Recorder`; i `Publisher`, a loro volta, devono essere in grado di pubblicare lo stream generato degli `Adapter` (o rigenerato dai `Replayer`) su una specifica piattaforma.

3.3.2 Il design pattern Factory

Riprendendo quanto detto nella sezione precedente, possiamo dire che una metodica progettazione concettuale di classi e sottoclassi, unita all'utilizzo del design pattern Factory Method, permette di instanziare le classi in modo molto semplice, in quanto tale pattern cerca di risolvere il problema della creazione di oggetti di cui non è fornita a priori una esatta specifica della classe di appartenenza. Questa peculiarità facilita lo sviluppo di nuove funzionalità simili a quelle già esistenti non comportando la modifica di altre parti dell'architettura.

Ma in che modo il pattern Factory semplifica il modo di instanziare una classe? In breve, un oggetto di tipo Factory restituisce al chiamante una istanza di una delle possibili classi istanziabili, in modo dipendente dai parametri che gli sono stati forniti. Solitamente, proprio come nel nostro caso, tutte le classi restituite dal Factory hanno una sopraclasse e metodi comuni, ma ognuna di esse esegue diverse funzionalità ed è ottimizzata per lavorare su diversi tipi di dati. Non utilizzando il pattern Factory, la creazione vera e propria di istanze di oggetti conduce, a volte, ad una duplicazione di codice e, inoltre, potrebbe necessitare di informazioni non accessibili agli oggetti e non sempre garantire un adeguato livello di astrazione. Lo scopo del pattern è quello di definire una interfaccia per la creazione di un oggetto (questo avviene nella sopraclasse), ma delegare ad una classe derivata il tipo di oggetto da istanziare in modo che le modalità di creazione degli oggetti e il loro utilizzo sia completamente disaccoppiato e che non ci sia la necessità da parte dei client di specificare i nomi delle classi concrete da instanziare all'interno del proprio codice. In questo modo si permette che un sistema sia indipendente dall'implementazione degli oggetti concreti.

L' essenza vera e propria di questo design pattern può essere riassunta con la seguente definizione:

"Il Factory Method è utile per definire un'interfaccia per la creazione di oggetti, lasciando alle sottoclassi il compito di decidere quale classe istanziare. Questo pattern permette che le classi deleghino la creazione delle istanze vere e proprie alle sottoclassi".

In poche parole, in un ambiente orientato agli oggetti, un Factory è un oggetto utile per creare altri oggetti proprio come se fosse un costruttore, o almeno una sua astrazione.

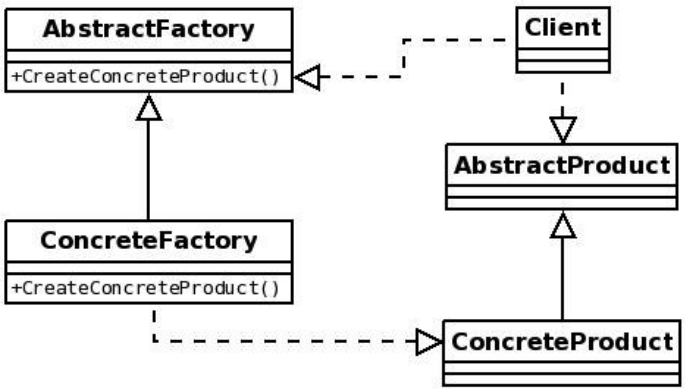

Figura 3.8: Design pattern Factory: schema UML

3.3.3 Il design pattern Observer

La maggior parte dei componenti del sistema sono basati sugli eventi. In particolare, quando un `DataGetter`, sia esso un `Adapter` o un `Replayer`, recupera dei nuovi dati da inserire nello stream ha la necessità di avvisare tutti quei componenti che gli sono stati associati, siano essi `Recorder`, `Publisher` o `Windower`.

Per tale scopo è risultato utile l'utilizzo del Design Pattern Observer, che fa parte della famiglia dei Publish/Subscrive Pattern. E' un pattern utilizzato come base architetturale di molti sistemi di gestione degli eventi ed il suo funzionamento risulta molto semplice ed intuitivo. Sostanzialmente il pattern si basa su uno o più oggetti, chiamati osservatori o listener (`Observer`), che vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto osservato (`Observable`).

Per descrivere il funzionamento generale del pattern ci viene in aiuto la Figura 3.9. `Subject` è una classe che fornisce le interfacce per registrare o rimuovere gli *observer* e che implementa le funzioni di *attach()*, *detach()* e *notify()*. `ConcreteSubject` è una classe che fornisce lo stato dell'oggetto agli *observer* e che si occupa di effettuare le notifiche chiamando la funzione *notify()* specificata nella classe `Subject`. `Observer` è una classe che definisce una interfaccia per tutti gli *observer* per ricevere le notifiche dal `Subject` ed è utilizzata come classe astratta per implementare gli `Observer` veri e propri, ossia i `ConcreteObserver`. `ConcreteObserver` è una classe

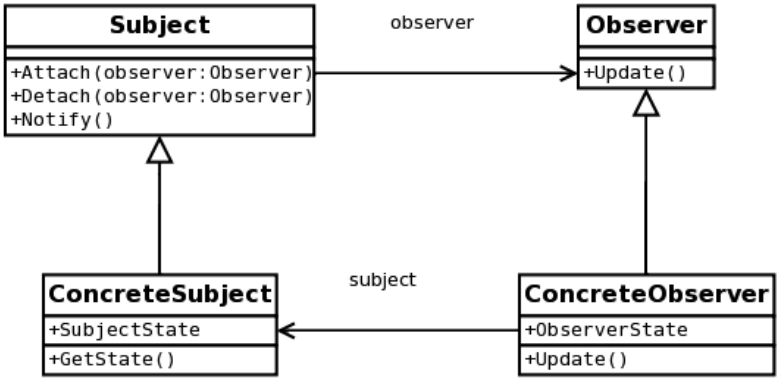

Figura 3.9: Design pattern Observer/Observable: schema UML

che mantiene un riferimento al `ConcreteSubject` per riceverne lo stato quando avviene una notifica. Tale notifica viene ricevuta ed elaborata dalla funzione *update()* di `ConcreteSubject`.

Nel caso del nostro sistema, un `DataGetter` è un oggetto di tipo `Observable` e i vari `Recorder`, `Publisher` o `Windower` che gli sono associati sono oggetti di tipo `Observer`. Per effettuare l'associazione di un `Recorder`, `Publisher` o `Windower` ad un `DataGetter` è necessario richiamare la funzione *attach()*. Quando un `DataGetter` ha nuovi dati da introdurre nello stream, richiama la funzione *notify()* e gli oggetti `Observer` associati a tale `DataGetter` mandano in esecuzione la funzione *update()*.

Questo semplice funzionamento ci permette di ottimizzare l'utilizzo delle risorse di sistema, infatti non sono i manipolatori di dati che chiedono continuamente ai fornitori se hanno novità, ma sono i fornitori stessi che li avvisano non appena hanno dei dati da introdurre nello stream.

3.3.4 Il design pattern Facade

La moltitudine di componenti che formano l'architettura e la difficoltà nella gestione delle interazioni fra questi, ha reso necessario pensare ad un metodo per renderne la gestione combinata il più semplice possibile; inoltre, la presenza di diversi componenti che possono essere attivi nello stesso istante fa si che sia molto importante tenerne traccia dello stato. Per entrambe le motivazioni risulta molto

utile il design pattern Facade. Si tratta di un pattern strutturale che viene solitamente utilizzato per nascondere la complessità di un sistema e ridurre la comunicazione e la dipendenza dai client che utilizzano l'applicazione. L'utilizzo di tale pattern prevede l'esposizione di una interfaccia che offre le totalità delle funzionalità, in modo da semplificarne l'invocazione da parte di un client. In uno scenario architetturale come quello che abbiamo presentato, tutti i client che intendono utilizzare le funzionalità del sistema, devono conoscerne le relazioni interne, per quanto complesse siano. Nel momento in cui tali relazioni vengano modificate, è necessario notificare a tutti i client interessati tali cambiamenti per apportare le modifiche del caso. In una situazione del genere l'accoppiamento tra client e sistema è elevato, il che - solitamente - è altamente sconsigliato.

Per disaccoppiare abbiamo deciso di introdurre il pattern Facade in modo che questi mascheri la complessità del sistema e rappresenti l'anello di collegamento tra i client e il sistema. In tal modo le eventuali modifiche verranno apportate in modo centralizzato al Facade (nel nostro caso `SLD4SM Server`) senza impattare sui singoli client, tagliando il legame forte che si era instaurato. In Figura 3.10 viene presentato il nostro sistema con `SLD4SM Server`. Abbiamo già sottolineato che la gestione dello stato dei componenti che formano il sistema trae vantaggio dall'utilizzo del pattern Facade. Prima di addentrarci sulle motivazioni che rendono possibile tale affermazione, è d'obbligo porsi una domanda: per quale motivo è necessario mantenere lo stato di ogni singolo componente del sistema?

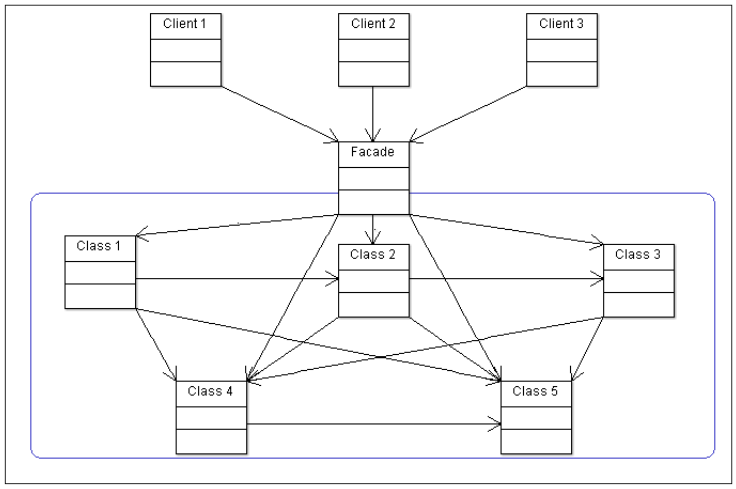

Figura 3.10: Esempio di utilizzo Design Pattern Facade

Partiamo da una considerazione fatta nella parte iniziale di questa sezione: al sistema possono teoricamente connettersi più client, ognuno dei quali ha la possibilità di agire come meglio crede e secondo le funzionalità che gli sono necessarie. Avendo un'unica l'istanza del sistema, se un client cerca di attivare un componente che svolge un'attività svolta da un altro componente già attivo (attivato da un altro client), ciò non porterebbe ad uno spreco di risorse? In tal caso, come potrebbe fare un client a sapere se un dato componente è già attivo o meno? In generale ogni componente detiene le informazioni sui compiti che sta eseguendo e, come abbiamo visto nella sezione precedente, tutti i componenti che svolgono un ruolo attivo nel sistema, ovvero i generatori di stream di modelli (`Adapter` e `Replayer`) e gli strumenti che ne rendono possibile il salvataggio (`Recorder`) e la pubblicazione (`Publisher` e `Windower`), possono essere istanziati, messi in pausa, fatti ripartire o fermati. Quindi, ognuno di questi componenti, in un qualsiasi istante può trovarsi in uno solo dei seguenti stati:

- *running* : se in quel dato momento il componente è attivo;
- *sleeping*: se in quel dato momento il componente è stato messo in pausa;

In base allo stato dei componenti, un client può quindi agire in modo opportuno decidendo se utilizzare o meno un componente già istanziato o se attivarne uno nuovo. Il pattern Facade è molto utile per la gestione dello stato in quanto consente di tenere traccia dei componenti e dei loro stati in modo centralizzato, in quanto la loro l'attivazione e le relazioni fra di essi, dipendono esclusivamente dall'interfaccia centralizzata. Un buon metodo per tenere traccia degli stati è utilizzare una struttura astratta simile alla tabella qui sotto, in cui per ogni generatore di stream di modelli ne viene indicato lo stato di attività e tutti i componenti che vi sono associati.

DataGetter	**Observer**	**Status**
Adapter 1	-	Running
Adapter 1	Recorder (disk)	Running
Adapter 1	Recorder (db)	Paused
...	...	...
Replayer 1	-	Running
Replayer 1	Publisher (RDFStream)	Running
...	...	...

Esempio 13: Possibile schema di salvataggio dello stato dei componenti

Riassumendo, l'utilizzo Facade offre i seguenti vantaggi:

- espone una singola interfaccia a grana grossa unificata e omogenea (e quindi semplificata) dell'intero sistema;
- riduce il numero di funzionalità di basso livello esposte al client;
- riduce l'accoppiamento tra il client e il sistema;
- migliora le performance diminuendo le interazioni tra il sistema e i client per tenere allineati questi ultimi;
- nasconde la complessità delle interazioni e delle dipendenze tra le funzionalità di basso livello;
- non esclude l'uso diretto del sistema, infatti il client può comunque utilizzarlo direttamente se lo ritiene necessario e se sa come farlo;
- permette un controllo centralizzato sullo stato dei componenti attivati nel sistema.

3.4 Conoscenza Stream e Conoscenza Statica

3.4.1 Pubblicazione di uno Stream

Uno stream RDF è definito come una sequenza ordinata di coppie, ognuna formata da un tripla (`<soggetto, predicato, oggetto>`) e da un timestamp. La pubblicazione del flusso continuo di dati sul web o il suo salvataggio in locale è da sempre un problema studiato in profondità, alla ricerca di un metodo per poter rigenerare, manipolare o trasportare un intero flusso di dati. Nell'ambito del progetto LarKC è stato proposto un metodo legato all'utilizzo dei linked data, utile per mantenere in memoria non solo i dati, che sono il cuore dello stream, ma anche informazioni ad essi legate, come ad esempio il timestamp relativo a ogni singolo snapshot RDF.

La proposta dal team LarKC fa uso dei *Named Graph*[5], dei grafi RDF che possono essere sostanzialmente di due tipi differenti: gli *stream Graph* (abbreviati in *sGraph*) e gli *instantaneus Graph* (abbreviati in *iGraph*). Gli *sGraph* contengono dei metadati informativi e descrivono il contenuto corrente della finestra sullo stream RDF introducendo una serie di riferimenti agli iGraph, i quali, a loro volta, contengono i dati veri e propri provenienti dallo stream e salvati sottoforma di triple RDF. Durante la progettazione di SLD4SM, abbiamo deciso di aggiungere tre nuovi tipi di *Named Graph*: i *registered*

Graph (abbreviati in *rGraph*), gli *active knowledge Graph* (abbreviati in *akGraph*) e gli *active facts Graph* (abbreviati in *afGraph*). (Vedi Figura 3.11)

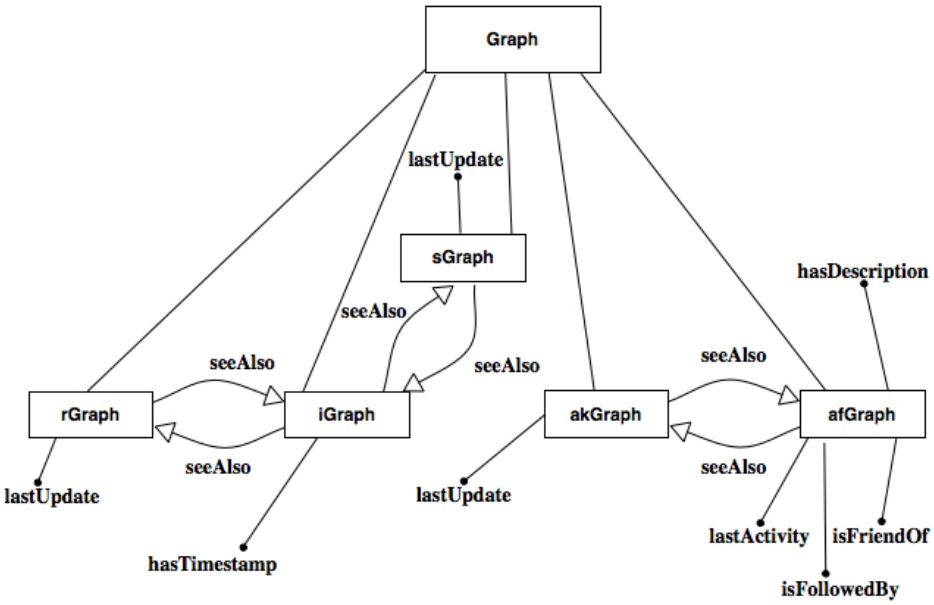

Figura 3.11: Rappresentazione della gerarchia che lega i vari tipi di grafo

Gli *rGraph* contengono le stesse informazioni degli *sGraph*, la differenza principale è rappresentata dal fatto che ciò che è presente in uno *sGraph* viene manipolato, cancellandone delle parti, ogni volta che la finestra posta sullo stream scorre in avanti, mentre un *rGraph* contiene tutti i riferimenti ai dati che sono passati attraverso lo stream dal momento in cui è iniziata l'osservazione.

Gli *akGraph* e gli *afGraph* ma rappresentano per la conoscenza statica ciò che gli *rGraph* e gli *iGraph* rappresentano per lo stream: un *akGraph* tiene traccia di tutti gli oggetti della conoscenza statica di cui si possiede una descrizione e i riferimenti agli *afGraph* relativi che contengono la descrizione vera e propria sotto forma di triple RDF. Come già detto, le differenze fra *sGraph*, *rGraph* e *akGraph* non riguardano il modo di esprimere i contenuti ma semplicemente il tipo di contenuto. La specifica per la stesura di un *sGraph* contenuta in [?] mette in risalto la presenza di tre fondamentali proprietà:

- *seeAlso*: le triple riguardanti tale proprietà indicano in quali *iGraph* o *afGraph* trovare le informazioni che si cercano;
- *receivedAt*: tale proprietà indica il timestamp di ricezione di un dato *iGraph* o *afGraph*;

- *lastUpdate*: tale proprietà indica il timestamp dell'ultimo aggiornamento effettuato sul grafo che si sta leggendo.

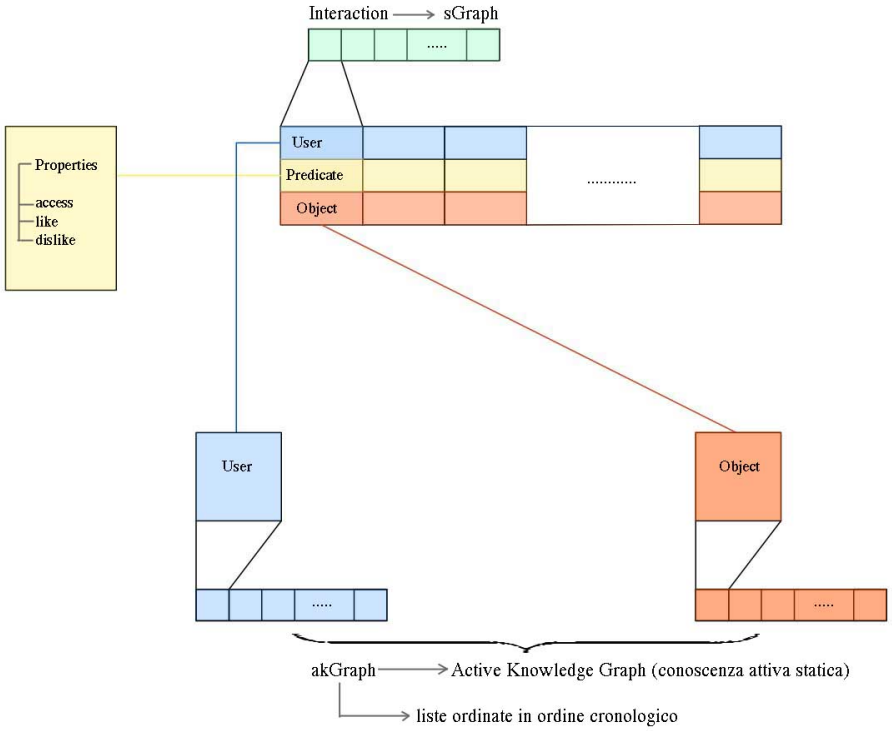

Figura 3.12: Utilizzo dei vari tipi di grafo durante la pubblicazione di uno stream di dati

In figura 3.12 possiamo vedere come l'architettura sfrutti questo tipo di pubblicazione: gruppi di interazioni provenienti dai social network vengono salvate all'interno degli *iGraph*, che contengono il vero contenuto informativo dello stream di dati; da ognuno di essi, tramite la proprietà *seeAlso*, è possibile risalire all'*sGraph* corrispondente. Ogni flusso di dati è legato ad un singolo *sGraph*, che indicizza tutti gli *iGraph* che lo rappresentano. Utilizzando la stessa logica, le informazioni legate a ogni soggetto e a ogni oggetto contenuti nello stream, vengono salvate negli *afGraph*, i quali sono indicizzati dall'*akGraph*.

3.4.2 Conoscenza Statica

Come abbiamo già specificato all'interno del paragrafo 3.1 per estrapolare informazioni utili dai dati online, è necessario incrociare lo

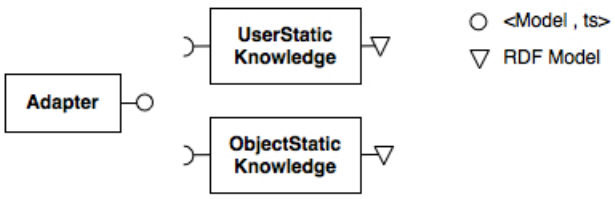

Figura 3.13: Schema generale di collegamento per la conoscenza statica

stream di informazioni con una base di dati che ha come caratteristica un grado di cambiamento molto basso nel tempo, che possiamo chiamare conoscenza statica.

La conoscenza statica contiene informazioni specifiche di utenti e oggetti che stanno ai due estremi delle triple che compongono il nostro stream (vedi immagine 3.1) ed è necessaria per avere una visione più reale dei dati digitali, infatti i dati di utenti e oggetti ci permettono di identificare nella realtà i soggetti a cui si riferiscono i nostri dati.

Per ogni utente vengono richiesti al servizio web la sua angrafica, una descrizione e il suo universo sociale, formato da amici e follower (che a loro volta sono risorse di tipo utente). Per ogni oggetto è possibile ottenere un tipo, un nome, una descrizione e i suoi follower.

Come per ogni altro componente della nostra architettura ogni specifica struttura che richiede la conoscenza statica deve essere modellata specicamente sul servizio web che fa da provider dei dati, ma si possono facilmente notare dei punti in comuni tra le varie implementazioni. Tutti i fornitori di conoscenza statica devono infatti:

- connettersi alla fonte dei dati secondo le modalità richieste;
- richiedere i dati, di solito tramite apposite URI;
- elaborare i dati e formattarli in uno specifico formato;
- salvare i dati ottenuti in modo da renderli riutilizzabili.

L'ultimo punto dell'elenco è fondamentale per rendere disponibili le informazioni per una rielaborazione successiva (tramite query).

Abbiamo deciso di utilizzare il metodo di pubblicazione di uno stream (vedi paragrafo 3.4.1) per salvare la conoscenza statica, che infatti può essere vista anch'essa, in modo superficiale, come un flusso continuo di informazioni il cui contenuto cambia molto raramente. Abbiamo creato quindi gli *afGraph* che corrispondono agli

igraph e contengono le informazioni specifiche della risorsa e gli *akGraph* che indicizzano gli afGraph. La nostra scelta di utilizzare questo stile di salvataggio è motivata dal fatto che l'accesso ai dati relativi alle singole risorse deve essere veloce, era dunque necessaria un indicizzazione che rendesse inoltre la ricerca il meno dispendiosa possibil in termini di consumo.

In Figura 3.13 possiamo vedere come l'infrastruttura per il recupero dei dati che formano la conoscenza statica sia collegata agli `Adapter` (o indifferentemente a `Replayer`), in questo modo può usufruire dei dati contenuti all'interno dello stream formattato ed utilizzarli per richiedere informazioni più approfondite che riguardano utenti o oggetti.

3.5 Rappresentazione grafica dei gestori di Named Graph

Presentiamo ora una possibile rappresentazione grafica dei nodi appartenenti all'applicazione.

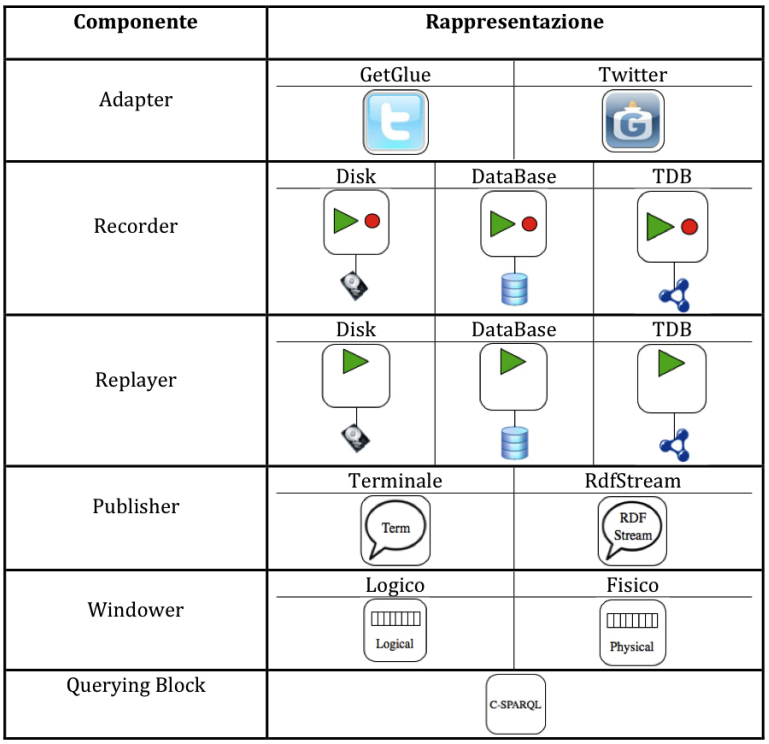

Figura 3.14: Rappresentazione grafica dei componenti dell'applicazione

Grazie alla modularità dell'applicazione e utilizzando i simboli contenuti nella tabella 3.14 è possibile comporre varie infrastrutture software, in base alle esigenze dell'utilizzatore. Ogni componente, presentato individualmente nel corso del capitolo 3, si suddivide in sottocomponenti di diverso tipo, rappresentati attraverso simboli che ne fanno intuire l'utilizzo e la piattaforma su cui è specializzato.

Presentiamo ora qualche caso più o meno complesso, per dimostrare la validità del nostro sistema di rappresentazione.

In figura 3.15 viene mostrato un esempio di composizione complessa, rappresentata tramite i simboli descritti in precedenza. I dati proveniente da un `Adapter` di GetGlue vengono inviati, contemporaneamente, a due `Recorder` che li salvano su piattaforme differenti, nel nostro caso disco e database, e a un `Publisher` su stream RDF. Il flusso di dati formattato viene utilizzato come fonte di dati da interrogare da parte del blocco `C-SPARQL`, i cui risultati vengono presentati all'utente finale tramite un `Windower` di tipo logico.

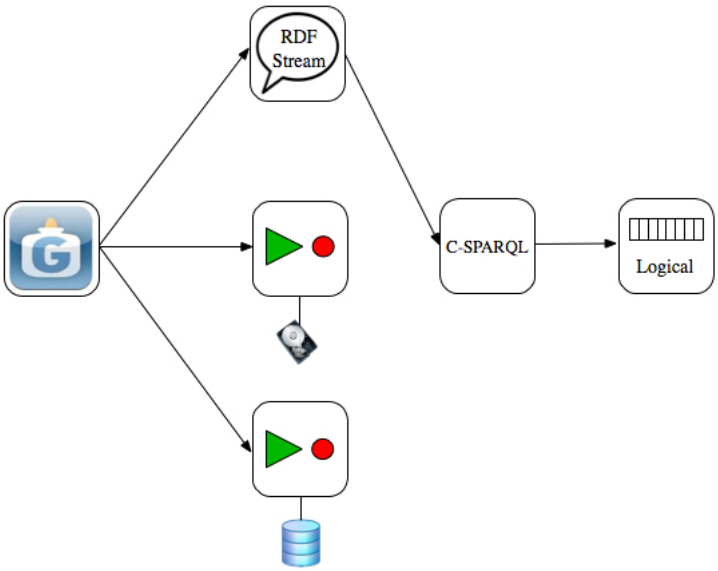

Figura 3.15: Esempio di rappresentazione grafica dei componenti

In figura 3.16 viene presentato un altro esempio di utilizzo e rappresentazione dei componenti che formano l'infrastruttura software. In questo caso, i dati provenienti da un `Replayer` da disco vengono registrati su TDB tramite l'apposito `Recorder` e, contemporaneamente, messi in ingresso ad un `Publisher` che crea uno stream RDF che verrà utilizzato dal blocco `C-SPARQL`, i cui risultati vengono presentati all'utente finale tramite un `Publisher` su console di controllo.

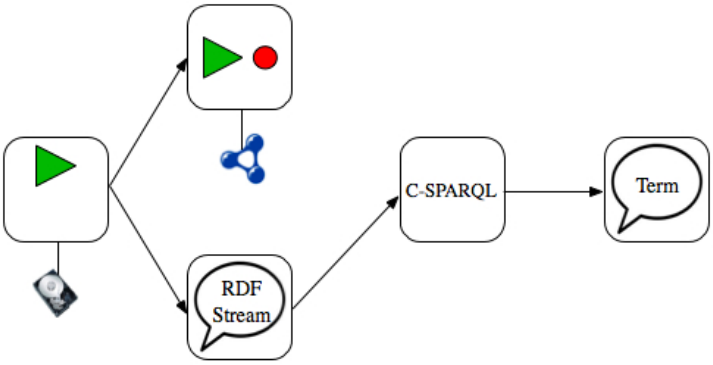

Figura 3.16: Esempio di rappresentazione grafica dei componenti

Capitolo 4

Implementazione Prototipo

In questo capitolo discuteremo le scelte implementative adottate nella progettazione e nella creazione di `SLD4SM`. Seguendo lo schema impostato nel capitolo precedente, giustificheremo ogni scelta fatta, partendo da una visione generale dell'applicazione e prendendo in considerazione ogni classe che compone l'architettura, fino ad arrivare all'implementazione dei design pattern utilizzati durante la progettazione del prototipo.

4.1 Architettura Generale

In questa sezione illustreremo lo sviluppo del sistema: inizieremo mostrando la interfacce e le strutture dati comuni di cui abbiamo fatto uso, per poi spostarci sull'implementazione vera e propria di tutte le componenti del sistema.

4.1.1 Interfacce e Strutture Dati Principali

Nell'ingegneria del software si dice classe astratta un'oggetto che definisce un'interfaccia senza implementarla completamente, ciò serve come base di partenza per generare una o più classi specializzate aventi tutte la stessa interfaccia di base. Una classe astratta da sola non può essere istanziata, ma viene progettata solo per svolgere la funzione di classe base, di super-classe, da cui le classi derivate possono ereditare i metodi e specializzarli. Il comportamento definito da queste classi è generico e la maggior parte dei metodi della classe sono indefiniti e non implementati.

Questo tipo di soluzione ci ha aiutato molto nello sviluppo del sistema. Grazie ad essa è stato possibile identificare, tra le funzionalità che il sistema deve essere in grado di offrire, quelle che hanno più caratteristiche in comune, ovvero quelle che presentano

uno schema di funzionamento condiviso ma che differiscono per delle particolarità. In questa sezione presenteremo l'unica interfaccia comune a più componenti del sistema (`DataGetter`), lasciando per le prossime sezioni le interfacce particolari del singolo componente. Presenteremo inoltre il tipo di dato che è stato progettato per lo scambio di informazioni cross-componente (`TimestampedModel`).

TimeStampedModel

I dati possono essere considerati il fulcro del sistema, infatti tutte le operazioni eseguite dai sui componenti, vengono svolte sui dati. E' stato quindi necessario progettare una struttura dati ad hoc in grado di contenere tutti le informazioni necessarie e che fosse compatibile per tutte le componenti di `SLD4SM`. La crezione della struttura dati `TimestampedModel` e il suo utilizzo vanno inseriti nell'ottica di standardizzazione dell'output che ha guidato la progettazione e la creazione dell'intera infrastruttura software.

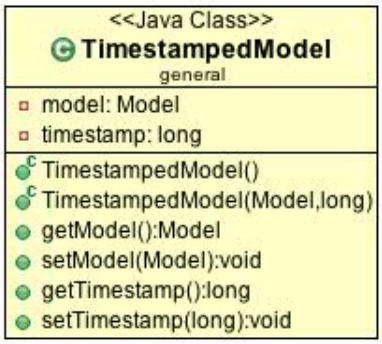

Figura 4.1: Class Diagram relativo alla struttura dati TimestampedModel

Un `TimestampedModel` (vedi Figura 4.1) è composto da un `Model` di Jena, che rappresenta il contenuto informativo vero e proprio della struttura dati, e da un dato di tipo `long`, che rappresenta il timestamp di arrivo, espresso in millisecondi, del frammento informativo. La creazione di un `TimestampedModel` avviene nel modo seguente: quando in un istante `t`, ad un generico `DataGetter`, arriva un frammento informativo, esso provvede a riempire un `Model` di Jena con tale frammento e a impostare il timestamp del `TimestampedModel` a `t`. E' molto importante che sia il `DataGetter` ad impostare il timestamp del `TimestampedModel` all'istante di arrivo effettivo del frammento informativo, questo perchè se il timestamp venisse aggiunto dai componenti a valle del `DataGetter`, si potrebbe avere lo stesso frammento informativo identificato con timestamp diversi, ponendo il sistema in uno stato inconsistente.

DataGetter

Come abbiamo avuto modo di spiegare nel capitolo precedente, vi sono essenzialmente tre tipi di fornitori di dati:

- gli `Adapter`, che recuperano i dati dai social network e generano uno stream di `TimestampedModel`;
- i `Replayer`, che sono in grado di generare uno flusso continuo di `TimestapedModel` partendo dai dati precedemente salvati sotto forma di grafi RDF;
- lo `C-SPARQLAdapter`, che prende i dati di output del motore C-SPARQL e genera uno stream di `TimestampedModel`.

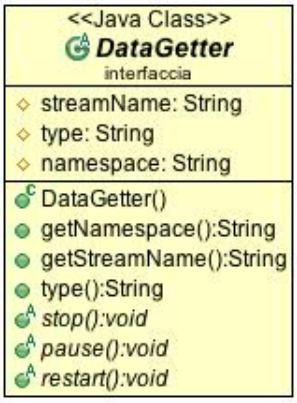

Figura 4.2: Class Diagram relativo all'interfaccia DataGetter

Questi tre oggetti implementano la classe astratta `DataGetter`, di cui possiamo vedere il Class Diagram in figura 4.2. La classe `DataGetter`, per una coerente gestione degli eventi attraverso l'uso del design pattern Observable/Observer, estende `Observable`, che la rende osservabile da oggetti che implementano `Observer`.

L'interfaccia `DataGetter` possiede dei metodi di controllo che permettono di gestire il funzionamento dell'oggetto: *pause()*, *restart()* e *stop()*. La funzionalità del metodo *pause()* è quella di mettere in pausa il `DataGetter`, cioè di bloccare momentaneamente la generazione dello stream di `TimestampedModel`. Nel caso dell'`Adapter` lo stato di pausa implica che, fintanto che l'oggetto si trova in questo stato, i dati provenienti dal Social Network associato vengano persi; nel caso di `Replayer` la pausa non implica il mancato inserimento di una parte di dati nello stream di `TimestampedModel`, ma semplicemente che nel momento in cui viene invocata la pausa

la generazione dello stream si ferma per poi riprendendere, partendo dal punto di interruzione, nel momento in cui viene invocato il metodo *restart()*; nel caso dello `C-SPARQLAdapter` lo stato di pausa implica che il componente rimane attivo ma non associato allo `C-SPARQL Engine`, quindi i dati di output dello `C-SPARQL Engine` non verranno gestiti dallo `C-SPARQLAdapter`.

Quando un `DataGetter` si trova nello stato *paused*, l'invocazione del metodo *restart()* permette il passaggio allo stato *running* e il `DataGetter` riprende tutte le sue funzionalità. Il metodo *stop()* serve a terminare l'utilizzo del `DataGetter` in modo *safe*, così che non rimangano attive istanze di `DataGetter` e in modo che il sistema si trovi sempre in uno stato consistente.

Vi sono poi metodi utili per ottenere informazioni sullo stream:

- *getStreamName()* permette di ottenere il nome dello stream;
- *getNamespace()* permette di ottenere il namespace dello stream;
- *type()* è un metodo necessario per gestire il pattern Factory e restituisce il tipo di `DataGetter` relativo all'istanza. (Es. una invocazione di questo metodo sull'oggetto `GlueAdapter` restituirà la stringa "glue").

Da questa classe astratta derivano le classi astratte `Adapter` e `Replayer` di cui forniremo, nella prossima sezione, una descrizione più dettagliata.

4.1.2 Adapter

La classe astratta `Adapter` è sottoclasse di `DataGetter`, quindi ne estende i metodi già implementanti e ne implementa di propri.

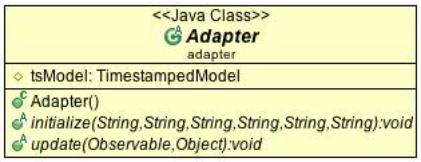

Figura 4.3: Class Diagram relativo all'interfaccia Adapter

Rappresenta la super classe di tutti i fornitori di dati che prendono le informazioni dalla rete. Implementa l'interfaccia `Observer`, in modo da poter ricevere i dati dalle classi delegate alla connessione a basso livello con le interfacce web dei social network ed estende la classe `Observable` (tramite la superclasse `DataGetter`), così da

poter fornire i dati ricevuti e manipolati alle classi poste a valle in osservazione.

In Figura 4.3 è mostrato il class diagram relativo alla classe astratta `Adapter`. Gli unici due metodi che una sottoclasse di `Adapter` deve avere sono il metodo *Initialize()* e il metodo *update()*. Il metodo *Initialize()* è sicuramente il più importante contenuto nella classe astratta, vista la sua utilità nell'utilizzo del Factory Method Pattern, e permette di inizializzare una specifica sottoclasse di `Adapter` utilizzando gli attributi presenti nella signature del metodo. Il metodo *update()* è tipico del pattern Observer/Observable e serve a recuperare i dati dall'oggetto osservato e a manipolarli per darli in pasto al sistema.

La presenza della classe astratta `Adapter`, che permette di tipizzare un certo tipo di fornitori di dati, dà la possibilità agli sviluppatori di aggiungere estensioni in modo completamente trasparente, per allargare il numero e la tipologia delle fonti a disposizione del sitema. Per ora sono stati sviluppati due tipi di `Adapter`, che recuperano, rispettivamente, i dati da Twitter e da GetGlue.

Nel Paragrafo 3.2.2 abbiamo introdotto brevemente i tipi di API messe a disposizione dai due Social Network in questione, nei prossimi paragrafi entreremo un pò più nel dettaglio presentando il metodo di autenticazione, le API e le librerie di terze parti che abbiamo utilizzato.

GlueAPI

Le API di GetGlue possono essere suddivise in tre diverse categorie:

- `/user`: permettono di avere accesso a uno specifico utente e a tutte le informazioni che lo riguardano, in particolare al profilo, agli amici, ai follower e ai suoi interessi divisi per categorie;
- `/object`: espongono le attività dei vari utenti riguardanti l'oggetto in questione, permettono di conoscere gli oggetti simili e garantiscono l'accesso al motore semantico utile per le ricerche riguardanti i collegamenti dell'oggetto preso in considerazione su tutto il web;
- `/glue`: permettono all'utente di esplorare le categorie di oggetti, ricercare oggetti o user specifici su tutta la libreria di `GetGlue` e di avere accesso all'attività sullo stream principale del social network.

Come per Twitter anche in questo caso, i meccanismi di autenticazione, necessari per accedere ai dati, sono molto diversi tra loro

e dipendono dagli ambiti di utilizzo dei dati, per GetGlue possiamo dividerli in 2 gruppi:

- Token Based : l'utente deve autorizzare l'applicazione attraverso una connessione di tipo HTTPS inserendo le proprie credenziali di login, utilizzando per le chiamate successive il token ritornato dalla prima richiesta;
- oAuth Based: viene utilizzato l'open protocol oAuth ed è necessario richiedere Consumer Key e Consumer Secret via email autorizzandosi come sviluppatore. Attraverso oAuth si potranno poi richiedere il Token e il Token Secret da utilizzare per la chiamata ad ogni singola API.

Anche in questo ambito l'autenticazione Token Based sta rapidamente scomparendo in favore della più sicura autenticazione di tipo oAuth.

Per le chiamate alle librerie non abbiamo utilizzato nessuna specifica libreria, ma, essendo semplici api REST, abbiamo utilizzato il protocollo HTTP attraverso la classe java `HTTPUrlConnection`.

Per l'autenticazione attraverso il protocollo oAuth ci siamo dovuti appoggiare a una libreria prodotta da terzi e la nostra scelta è caduta su SignPost perchè rappresenta il metodo più semplice per utilizzare connessioni HTTP in conformità con lo standard oAUTH. Utilizzando un design modulare e flessibile, permette di utilizzare le più svariate tipologie di messaggio HTTP.

TwitterAPI

L'utilizzo delle API offerte da Twitter è subordinato a all'uso, da parte dell'utente, di un meccanismo di autenticazione, tali meccanismi si possono raggruppare in tre gruppi distinti, che differiscono soprattutto per la tecnologia usata e hanno ambiti di utilizzo diversi tra loro:

- Basic Authentication: bisogna fornire all'applicazione username e password dell'account Twitter. Per le REST API tale metodo non è più utilizzabile, mentre per le Streaming API Twitter non ha ancora preso una decisione in proposito;
- Oauth Authentication: la prima cosa da fare è registrare l'applicazione su un account di Twitter, a tale applicazione verranno associati una Consumer Key e un Consumer Secret che possono essere visti come una chiave pubblica ed una chiave privata in un algoritmo crittografico. La Consumer Key e il

Consumer Secret devono essere utilizzati con una libreria di autenticazione OAuth, per autenticarsi e avere la possibilità di fare richieste tramite le API. Tale tipo di autenticazione viene utilizzata soprattutto per le Web Application;

- xAuth Authentication: è l'equivalente di OAuth ma viene utilizzata principalmente per Desktop e Mobile Application. A differenza di OAuth non basta registrare l'applicazione sul proprio account di Twitter ma occorre richiedere l'autorizzazione a Twitter facendo presente l'utilizzo che si vorrà fare dell'applicazione.

Con il passare del tempo, l'uso dell'autenticazione di tipo basic, in cui è necessario fornire username e password dell'utente, sarà sempre più limitato, visti i suoi problemi di sicurezza, questo porterà a un passaggio definitivo a Oauth per quanto riguarda tutti i tipi di librerie.

Per facilitarci il compito di implementazione riguardante l'interfaccia da utilizzare con Twitter, abbiamo deciso di utilizzare una libreria creata appositamente a questo scopo, Twitter4J, che è una libreria non ufficiale ma riconosciuta da Twitter, creata da Yusuke Yamamoto[1]. I suoi principale punto di forza è la sua ampia diffusione, che garantisce un suo continuo aggiornamento che segue ogni cambiamento delle API di Twitter e la presenza un nutrito gruppo di sviluppatori con cui scambiare opinioni e risolvere i problemi.

Implementazione Interfaccia per Glue

La classe `GlueAdapter` sfrutta le classi contenute nel progetto `GlueAdapter` per connettersi a GetGlue. La Figura 4.4 ci mostra la struttura delle classi che permettono la connessione al social network GetGlue tramite le API offerte agli sviluppatori:

- `GetInteractions`: è il cuore di `GlueAdapter`, permette l'autenticazione tramite oAuth e fa le richieste tramite HTTP al social network. Il risultato delle richieste è un file XML di cui viene fatto il parsing;
- `Generator`: tramite i dati che arrivano da `GetInteractions`, contribuisce alla creazione degli oggetti di tipo `Interactions` e `ActionObject`, che rappresentano il contenuto informativo vero e proprio dello stream di dati;

[1] `http://twitter4j.org`

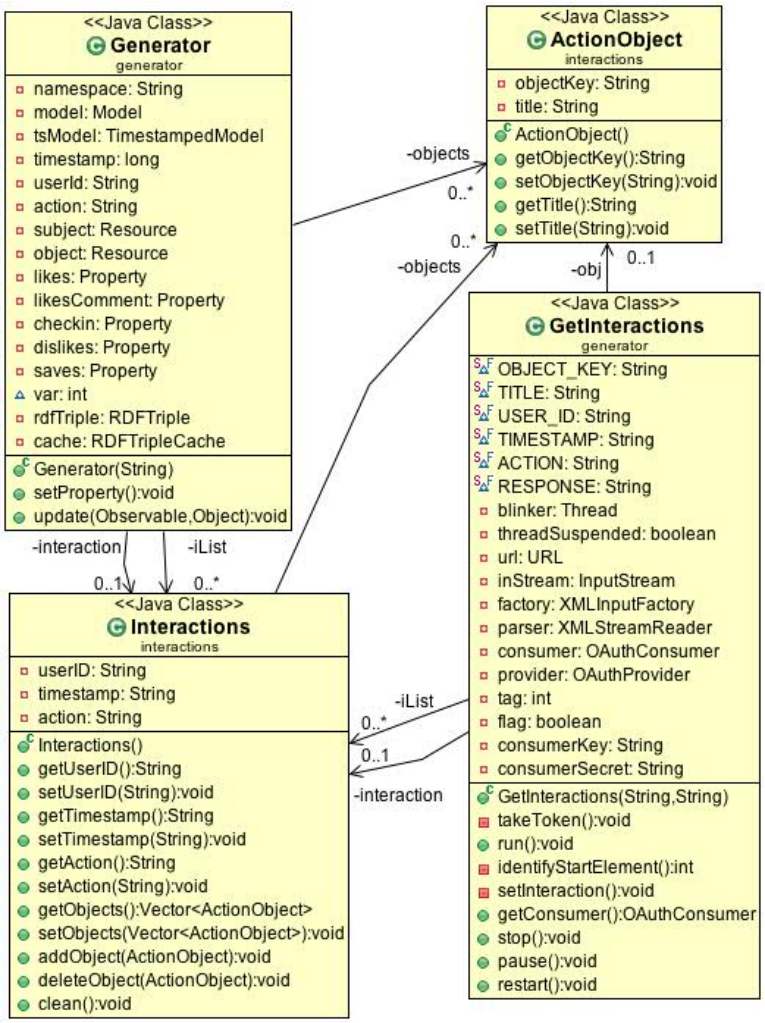

Figura 4.4: Class Diagram relativo all'interfaccia per la comunicazione con GetGlue

- `Interactions`: contiene informazioni relative all'utente (UserID univoco) e all'azione compiuta, oltre che un puntatore ad un `ActionObject` e il timestamp relativo al momento di arrivo nel sistema dell'informazione;
- `ActionObject`: contiene le informazioni relative ad un oggetto (titolo e ID univoco), è collegato alle `Interaction`, in modo da formare una tripla soggetto, predicato e oggetto da inserire in un modello di dati.

Le informazioni contenute e manipolate da questa infrastruttura software rappresentano il grado minimo per rendere univoca

una tripla che rappresenta un azione compiuta da un utente su un oggetto.

Implementazione Interfaccia per Twitter

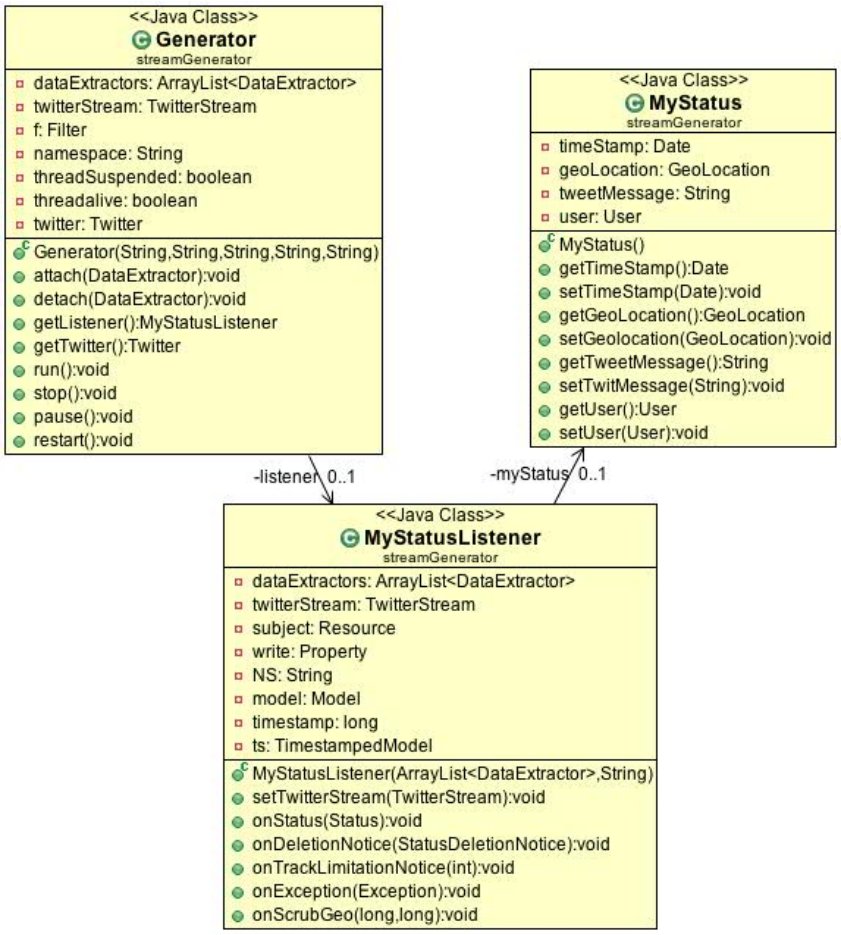

Figura 4.5: Class Diagram relativo all'interfaccia per la comunicazione con Twitter

La classe `TwitterAdapter` sfrutta le classi contenute nel progetto `TwitterAdapter` per connettersi a Twitter.

La figura 4.5 illustra una descrizione più dettagliata del codice implementato per estrarre e manipolare i dati tramite le API offerte da Twitter. Il package `streamGenerator` contiene tutte le classi utili per collegarsi a Twitter, tramite autenticazione oAuth, scaricare i dati e inserirli, dopo averli manipolati e trasformati nella giusta forma, in una struttura dati di tipo `TimestampedModel`. Le classi contenute nel package sono le seguenti:

- `MyStatus`: rappresenta un tweet che è presente sullo stream, contiene tutte le informazioni sull'utente che l'ha scritto e pubblicato e sul messaggio stesso, comprese le eventuali coordinate legate alla geolocalizzazione;
- `Generator`: permette la vera e propria connessione a Twitter, fornendo i dati dell'account per la connessione allo stream di dati. Offre anche i metodi necessari per l'aggiunta o la rimozione dei `DataExtractor`, classi che controllano le parole chiave da inserire nel fitlro;
- `MyStatusListener`: ascolta lo stream e ad ogni cambiamento di stato permette di inviare i dati contenuti nel nuovo tweet alle classi a cui è delegata la loro manipolazione.

Questa infrastruttura permette di salvare i dati dopo averli estratti dallo stream proveniente dal servizio web, dando modo ai metodi di manipolare il contenuto dei dati, salvandoli nel modo corretto e uniformando la tipologia di informazioni in uscita.

4.1.3 C-SPARQL Adapter

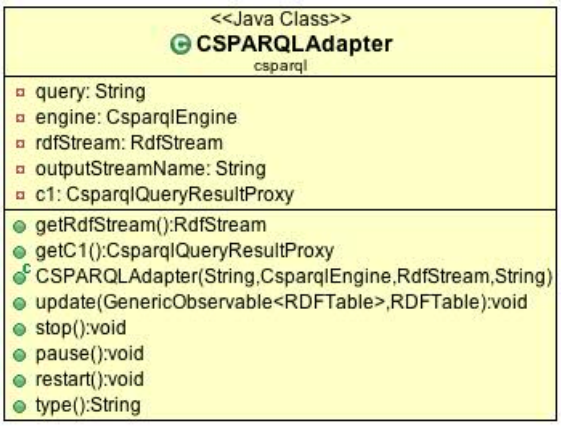

Figura 4.6: Class Diagram relativo all'oggetto CSPARQLAdapter

Come abbiamo specificato nel paragrafo 3.2.3 possiamo definire anche lo `CSPARQL Adapter`, di cui possiamo vedere il class diagram in figura 4.6, come un fornitore di dati per l'intero sistema, è sottoclasse di `DataGetter` e ne eredita e specializza i metodi.

Il suo compito principale è la trasformazione dei dati da `RDFTable` in un flusso continuo di `TimestampedModel`.

I tipi di oggetti utilizzati in questa classe, `RDFStream`, `C-SPARQL Engine` e `RDFTuple`, sono tutti specificati all'interno del pacchetto `C-SPARQL Engine` che è esterno al nostro progetto, questo `Data Getter` è stato appositamente creato per poter comunicare con lo `C-SPARQL Engine`. L'intera `RDFTable` che utilizziamo per riempire il nostro modello viene inviata come parametro del metodo update, infatti come specificato nella classe `DataGetter`, la classe `C-SPARQL Adapter` è un `Observer` che attende le notifiche e i relativi dati provenienti dall'engine tramite il pattern Observer/Observable.

Attraverso il ciclo presentato nel listato di codice 4.1 da ogni tupla contenuta nell'`RDFTable` viene preso il frammento informativo di nostro interesse e viene inserito all'interno di un modello RDF che poi verrà a sua volta inserito in un `TimestampedModel` con il relativo timestamp, riuscendo così a trasformare i dati nel formato che abbiamo scelto per far comunicare tra loro tutti gli oggetti della nostra infradtruttura.

```
1  Model model = ModelFactory.createDefaultModel();
2
3  for (final RDFTuple t : q) {
4   String subject = t.get(0);
5   String property  = t.get(1);
6   String object  = t.get(2);
7   String[] objectParts = object.split("\\^\\^");
8
9   Resource rSubject = model.createResource(subject);
10  Property pProperty = model.createProperty(property);
11
12    if (objectParts.length>1) {
13
14          Literal lObject =
15                  model.createTypedLiteral(
16                          objectParts[0].replaceAll("\"", ""),
17                           new XSDDatatype(objectParts[1]));
18
19          Subject.addLiteral(pProperty, lObject);
20
21    } else {
22           Resource rObject = model.createResource(object);
23           rSubject.addProperty(pProperty, rObject);
24    }
25
26 }
27  TimestampedModel ts = new TimestampedModel(model,
28                    new GregorianCalendar().getTimeInMillis());
```

Codice 4.1: trasformazione dei dati da parte del CSPARQLAdapter.

4.1.4 SLD4SM Core

Recorder

Come visto nel capitolo precedente, un oggetto `Recorder` permette di salvare uno stream di `TimestampedModel` sotto forma di grafi RDF in tre modi differenti: su disco (`RecorderDisk`), su modello persistente (`RecorderTDB`) o su database (`RecorderDB`).

La classe astratta `Recorder`, per una corretta gestione degli eventi attraverso l'uso del design pattern Observable/Observer, implementa l'interfaccia `Observer`, questo permette ad un `Recorder` di essere associato ad un `DataGetter` per ottenere notifiche sui dati di output di quest'ultimo. Il funzionamento di un `Recorder` è molto semplice: una volta instanziato un `DataGetter`, gli si associa come observer un oggetto di tipo `Recorder` e ogni volta che il `DataGetter` ha nuovi dati da introdurre nello stream di `TimestampedModel`, lo notificherà al `Recorder` che, a sua volta, provvederà a manipolarli per poterli poi salvare sottoforma di grafi RDF.

Figura 4.7: Class Diagram relativo all'interfaccia Recorder

Come gli oggetti di tipo `Adapter`, anche i `Recorder` possiedono

dei metodi per la gestione dello stato dell'oggetto. Il metodo *pause()* è invocabile quando lo stato dell'oggetto è *running* e permette di bloccare temporaneamente la registrazione dello stream; fintanto che il `Recorder` si trova nello stato *paused* tutti i dati ricevuti, provenienti dai `DataGetter`, vengono persi. Il metodo *restart()*, invocato quando lo stato del `Recorder` è *paused*, permette di riprendere la registrazione dello stream (lo stato dell'oggetto passa in questo caso a *running*). Non è stato implementanto il metodo *stop()* in quanto si occupa di questo `SLD4SM Server` mentre è stato implementato il metodo *checkStatus()* che permette di controllare se lo stato dell'oggetto è *paused* o *running*. In figura 4.7 è possibile vedere il class diagram della classe astratta `Recorder`.

Vi sono poi metodi per la gestione dell'oggetto e dello stream. Il metodo *getStreamName()* permette di ottenere il nome dello stream di cui si sta effettuando la registrazione, il metodo *setProperty()* serve ad impostare le proprietà base dei `Model` di Jena utilizzati per la creazione dei grafi RDF e il metodo *type()* è un metodo necessario per gestire il pattern Factory e restituisce il tipo di `Recorder` relativo all'istanza. (Es. una invocazione di questo metodo sull'oggetto `RecorderDisk` restituirà la stringa "disk"). Come per gli `Adapter`, è fondamentale per l'utilizzo del Factory Method Pattern il metodo *initialize()*, che permette di inizializzare una specifica sottoclasse di `Recorder` utilizzando unicamente gli attributi presenti nella signature del metodo. Il metodo *update()* è necessario e tipico per l'utilizzo del pattern Observer/Observable ed è simile per tutti le sottoclassi di `Recorder`: la sua funzionalità principale è quella di creare l'rGraph e i relativi iGraph partendo dal `TimestampedModel` ricevuto. Il metodo *graphWriter()* è necessario e differente per ogni tipo di `Recorder` in quanto i essi si differenziano per il tipo di salvataggio dei grafi.

```
1  public void graphWriter(String name, long timestamp, Model model){
2     OutputStream out;
3     File f = new File(path);
4     if(!f.isDirectory()){
5        f.mkdirs();
6     }
7
8     if (index == 0){
9       out = new FileOutputStream(path + name + ".rdf");
10    } else{
11      out = new FileOutputStream(path + name + index + ".rdf");
12    }
13    model.write(out);
14    out.close();
15 }
```

Codice 4.2: Il metodo graphWriter() per la classe RecorderDB

```
1  public void graphWriter(String name, long timestamp, Model model){
2     Resource rGraph;
3     File f = new File(path);
4
5     if(!f.isDirectory()){
6       f.mkdirs();
7     }
8
9     Model persistent;
10    Property lastUpdate;
11
12    if (timestamp != 0){
13      persistent = TDBFactory.createModel(
14                          path + name + timestamp);
15      rGraph = persistent.createResource(
16                          namespace+ "rGraph" + streamName);
17      lastUpdate = persistent.createProperty(namespace+"lastUpdate");
18      rGraph.removeAll(lastUpdate);
19      persistent.add(model);
20      persistent.close();
21    }else{
22      persistent = TDBFactory.createModel(path + name);
23      lastUpdate = persistent.createProperty(namespace+"lastUpdate");
24      rGraph = persistent.createResource(
25                          namespace + "rGraph" + streamName);
26      rGraph.removeAll(lastUpdate);
27      persistent.add(model);
28      persistent.close();
29    }
30 }
```

Codice 4.3: Il metodo graphWriter() per la classe RecorderTDB

Codice 4.2 e Codice 4.3 mostrano, rispettivamente, come avviene il salvataggio nelle classi `RecorderDisk` e `RecorderTDB`. Per prima cosa viene controllato se il percorso indicato per il salvataggio dei grafi esiste e nel caso in cui non esistesse viene creato, in seguito si passa al salvataggio dei grafi vero e proprio: quando il timestamp è uguale a 0 significa che si sta salvando l'rGraph, altrimenti un iGraph.

Nel Codice 4.4 è presentanto il metodo di salvataggio dei grafi su database MySQL: all'atto dell'inizializzazione di un oggetto di questo tipo viene effettuata la connessione ad un database MySQL che rimane attiva fintanto che è attivo l'oggetto. Il metodo *graphWriter()* in questo caso si occupa del salvataggio dei soli iGraph, in quanto il fatto che la connessione al DB rimanga sempre attiva permette che la modifica e l'aggiornamento (che avvengono nel metodo *update()*) dell'rGraph siano contemporanee.

Publisher

Come la classe astratta `Recorder`, anche la classe astratta `Publisher` implementa l'interfaccia `Observer`, che le permette di essere asso-

```
1 public void graphWriter(String name,long timestamp,Model model){
2     Model DBmodel = maker.createModel(name + timestamp);
3     DBmodel.add(model);
4 }
```

Codice 4.4: Il metodo graphWriter() per la classe RecorderDB

ciata ad un `DataGetter` per ottenere notifiche sui dati di output di quest'ultimo.

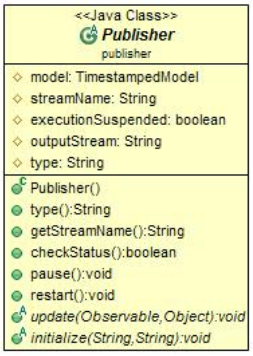

Figura 4.8: Class Diagram relativo all'interfaccia Publisher

Sono state sviluppate due sottoclassi di `Publisher`: `PublishOn Console` permette la visualizzazione dello stream su console (o terminale) ed è utile puramente a scopo di debugging; `PublishRDF Stream` permette la pubblicazione dei dati sotto forma di stream RDF, ovvero di uno stream formato da quadrupla <Soggetto, Predicato, Oggetto, Timestamp>. Quest'ultima classe risulta particolarmente utile per l'utilizzo dello `C-SPARQL Engine`, che accetta come input solo stream RDF.

La classe astratta `Publisher` è molto simile alla classe astratta `Recorder`: oltre a possedere gli stessi metodi per la gestione dello stato, hanno in comune il metodo *type()* per la gestione del Factory Method Patter e il metodo *getStreamName()* per ottenere il nome dello stream. Sono invece differenti per i parametri passati al metodo *initialize()*. In figura 4.8 è possibile vedere il class diagram della classe `Publisher`.

Replayer

La classe astratta `Replayer` è sottoclasse di `DataGetter`, quindi ne estende i metodi già implementati e ne implementa di propri.

Rappresenta la super classe di tutti i generatori di stream che prendono le informazioni da dati precedentemente salvati da un og-

```
public void update(Observable arg0, Object arg1) {
    String sub;
    String pre;
    String obj;

    if(!checkStatus()){
      model = (TimestampedModel) arg1;
      if(model.getModel() != null){
        Model publishModel = model.getModel();
        StmtIterator si = publishModel.listStatements();
        while (si.hasNext()){
            Statement s = si.next();
            sub = s.getSubject().toString();
            pre = s.getPredicate().toString();
            obj = s.getObject().toString();
            defaultRDFInputStream.put(new RdfQuadruple(
                  sub, pre, obj, System.nanoTime()));
        }
      }
     model = new TimestampedModel();
    }

}
```

Codice 4.5: Il metodo update() per la classe PublishRDFStream

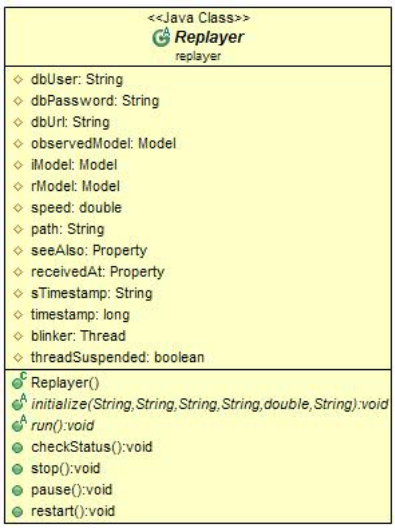

Figura 4.9: Class Diagram relativo all'interfaccia Replayer

getto di tipo `Recorder`. Anche in questo caso, per implementare il Factory Method Pattern, oltre al metodo *type()* che restituisce il tipo di `Replayer` dell'oggetto instanziato, abbiamo bisogno del metodo *initialize()* che fornisce un metodo standard di inizializzazione delle specifiche sottoclassi di `Replayer`. A differenza della classe astratta `Adapter` non implementa l'interfaccia `Observer` in quanto genera lo stream partendo dai dati precedentemente salvati dagli

oggetti di tipo `Recorder`, ma è caratterizzata dall'avere gli stessi metodi di gestione dello stato dell'oggetto (in quanto, come detto, sia `Adapter` che `Replayer` sono sottoclassi di `DataGetter`).

Le classi `ReplayerFromDisk`, `ReplayerFromTDB` e `ReplayerFromDB` sono sottoclassi di `Replayer` e generano un flusso continuo di `TimestampedModel` partendo, rispettivamente, da dati salvati su disco, su modello persistente e su database MySQL. Le tre implementazioni, come è facile intuire, sono molto simili fra di loro e presentano molto codice in comune, in quanto devono svolgere essenzialmente le stesse operazioni:

1. collegarsi alla sorgente dati;
2. leggere l'rGraph e scorrerlo identificando tutti gli iGraph appartenenti allo stream registrato;
3. prendere ogni iGraph e trasformarlo in un `TimestampedModel`;
4. introdurre sullo stream il `TimestampedModel` così ottenuto.

```
String path = "...";
String iGraphName = "...";
Model model = ModelFactory.createDefaultModel();
...

File file = new File(path);
if (file.isDirectory()){
    File[] files = file.listFiles();
      if(files.length > 0){
        for (int i = 0 ; i < files.length; i++){
          File f = files[i];
          if(f != null && f.getName().equals(iGraphName)){
            InputStream ins;
            ins = FileManager.get().open(f.getAbsolutePath());
            model.read(ins, null);
            ...
          }
          }
       }
}
```

Codice 4.6: Esempio di recupero di un iGraph da parte della ReplayerFromDisk

```
String path = "...";
String iGraphName = "...";

Model model = TDBFactory.createModel(path + iGraphName);
```

Codice 4.7: Esempio di recupero di un iGraph da parte della ReplayerFromTDB

Ciò che differenzia una `Replayer` dall'altra è la connessione alla sorgente e la lettura dei dati. Per `ReplayerFromDisk` e `Replayer`

FromTDB, dato il nome dello stream da rigenerare, il sistema è in grado di capire la path in cui sono posizionati i dati; da tale path saranno in grado di recuperare l'rGraph e tutti i relativi iGraph (vedi Codice 4.6 e Codice 4.7). Anche per quanto riguarda ReplayerFromDB è dal nome dello stream da rigenerare che si ricava la posizione dei relativi dati: una volta identificata l'ubicazione dei dati, il sistema si connette al database recupera l'rGraph e tutti i relativi iGraph generando lo stream di TimestampedModel; la connessione al database MySQL viene effettuata all'atto di istanza del ReplayerFromDB e viene mantenuta attiva fintanto chè non terminata la rigenerazione dello stream (vedi Codice 4.8).

```
1  String dbURL = "...";
2  String dbUser = "...";
3  String dbPassword = "...";
4  String graphName = "...";
5  IDBConnection conn;
6  ModelMaker maker;
7
8  conn = new DBConnection(dbUrl, dbUser, dbPassword, "MySQL");
9  maker = ModelFactory.createModelRDBMaker(conn);
10
11 Model model = maker.openModel(graphName, false);
12 ...
13 conn.close();
```

Codice 4.8: Esempio di recupero di un iGraph da parte della ReplayerFromDB

E' importante sottolineare che per rendere possibile l'utilizzo dei Windower (vedi Paragrafo 4.1.4) su dati rigenerati, è necessario che, nella ricreazione di un TimestampedModel partendo da un iGraph, i timestamp vengano aggiornati con l'istante di rigenerazione del TimestampedModel; questo perchè, come vedremo, la generazione delle finestre logiche si basa sul tempo.

Windower

Un oggetto di tipo Windower ha il compito di tenere in memoria finestre sui dati di dimensione (*size*) e durata (*expire time*) stabilite a priori dall'utente. Le finestre possono essere di due tipi: fisica, se il numero di elementi presenti rimane fisso e per ogni nuovo elemento arrivato nello stream essa viene rinnovata; logica, se è composta da tutti gli elementi arrivati sullo stream in un certo lasso di tempo (*size*).

WindowerPhysical, che permette la creazione di finestre fisiche, e WindowLogical, che permette la creazione di finestre logiche sui dati, sono entrambe sotto-classi di Windower. Come per tutti gli elementi di SLD4SM Core, anche in questo caso, per implementare

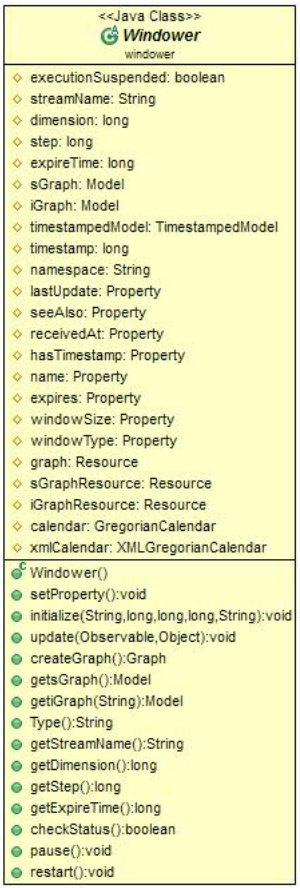

Figura 4.10: Class Diagram relativo all'interfaccia Windower

il Factory Method Pattern, oltre al metodo *type()* che restituisce il tipo di `Windower` istanziato, è presente il metodo *initialize()*, un metodo standard per instanziare tutte le sotto classi di `Windower`. Anche la classe astratta `Windower`, come `Recorder` e `Publisher`, implementa l'interfaccia `Observer` per una corretta gestione degli eventi.

Il funzionamento di `Windower` è il seguente: una volta instanziato un `DataGetter`, gli si associa come observer un oggetto di tipo `Windower`; ogni qualvolta sarà disponibile un nuovo `TimestampedModel` sullo stream, il `DataGetter` lo notificherà al `Windower` che provvederà alla sua gestione per la creazione della finestra, sia essa logica o fisica.

`Windower`, oltre ai metodi per la gestione dello stato dell'oggetto (*restart()*, *pause()* e *checkStatus()*), possiede dei metodi per la notifica delle informazioni riguardanti l'oggetto:

- *getDimension()*, restituisce la dimensione della finestra;
- *getStep()*, restituisce lo *step* tra una finestra e la successiva;
- getExpireTime(), restituisce l'*expire time*, ovvero il massimo intervallo di tempo per cui è garantita la possibilità di servire un iGraph dal momento del suo arrivo nel sistema.

I metodi caratterizzanti la classe astratta `Windower` sono i metodi *createGraph()*, *getsGraph()* e *getiGraph()*: mentre gli ultimi due sono uguali per entrambi i `Windower`, il primo è implementato in modi differenti per le due sottoclassi. Il metodo *getsGraph()* restituisce l'*sGraph* relativo alla finestra attiva nel momento dell'invocazione; il metodo *getiGraph()* richiede il passaggio di un parametro di tipo `String`, che non è null'altro che il nome dell'*iGraph* che si vuole ottenere in risposta. Il metodo *createGraph()* genera gli *iGraph* e il relativo *sGraph* a partire da una cache di `TimestampedModel`, tenuti in memoria secondo l'ordine di arrivo.

Se `WindowerPhysical` è stato instanziato per avere una finestra di **N** grafi, di conseguenza metterà a disposizione del sistema gli ultimi **N** *iGraph* e il relativo *sGraph*. In questo tipo di modello la finestra è in continua variazione, infatti lo stream di `TimestampedModel` è sempre attivo e solo i più recenti **N** `TimestampedModel` vengono trasformati in *iGraph* ed indicizzati in un *sGraph*. Ogni qualvolta arriva un nuovo `TimestampedModel` sullo stream viene rinnovata la finestra eliminando da essa l'*iGraph* più vecchio. Ciò non comporta l'eliminazione instantanea dell'iGraph dalla memoria, bensì questi rimane nel sistema fintanto che non è passato un determinato intervallo di tempo (*expire time*) dal momento del suo arrivo.

Riguardo il funzionamento del `WindowerLogical`, all'instante t_{0} viene instanziato ed inizia il popolamento della prima finestra, che risulta vuota fintanto che non è passato un intervallo di tempo pari a durata, solo da questo instante (*t_{0} + durata*) la finestra risulta disponibile ed invocabile e conterrà al suo interno tutti gli iGraph relativi ai `TimestampedModel` arrivati fra l'istante *t_{0}* e l'istante *t_{0}+durata*. Il popolamento della seconda finestra inizia nell'istante *t_{0}+step* e termina nell'istante *t_{0}+step+durata* e conterrà tutti gli iGraph relativi ai `TimestampedModel` arrivati in questo intervallo di tempo. Il sistema tiene in memoria e rende invocabili gli iGraph relativi a finestre non attive per un tempo pari all'*expire time*, che viene calcolato dal momento di creazione della finestra.

Figura 4.11: Class Diagram relativo a SLDServer

4.1.5 SLD4SM Server

In figura 4.11 viene presentata la struttura di `SLD4SM Server` che, facendo riferimento al paragrafo 3.3.4, possiamo considerare il telecomando con cui è possibile comandare tutta l'infrastruttura software. Al suo interno viene tenuta traccia di tutti i `DataGetter` connessi al sistema, attraverso l'utilizzo di una `Hashmap`, che inidicizza gli oggetti al suo interno attraverso il nome dello stream che l'utente ha associato al fornitore di dati. Gli oggetti in questione

sono rappresentati da `Container` (Vedi figura 4.12).

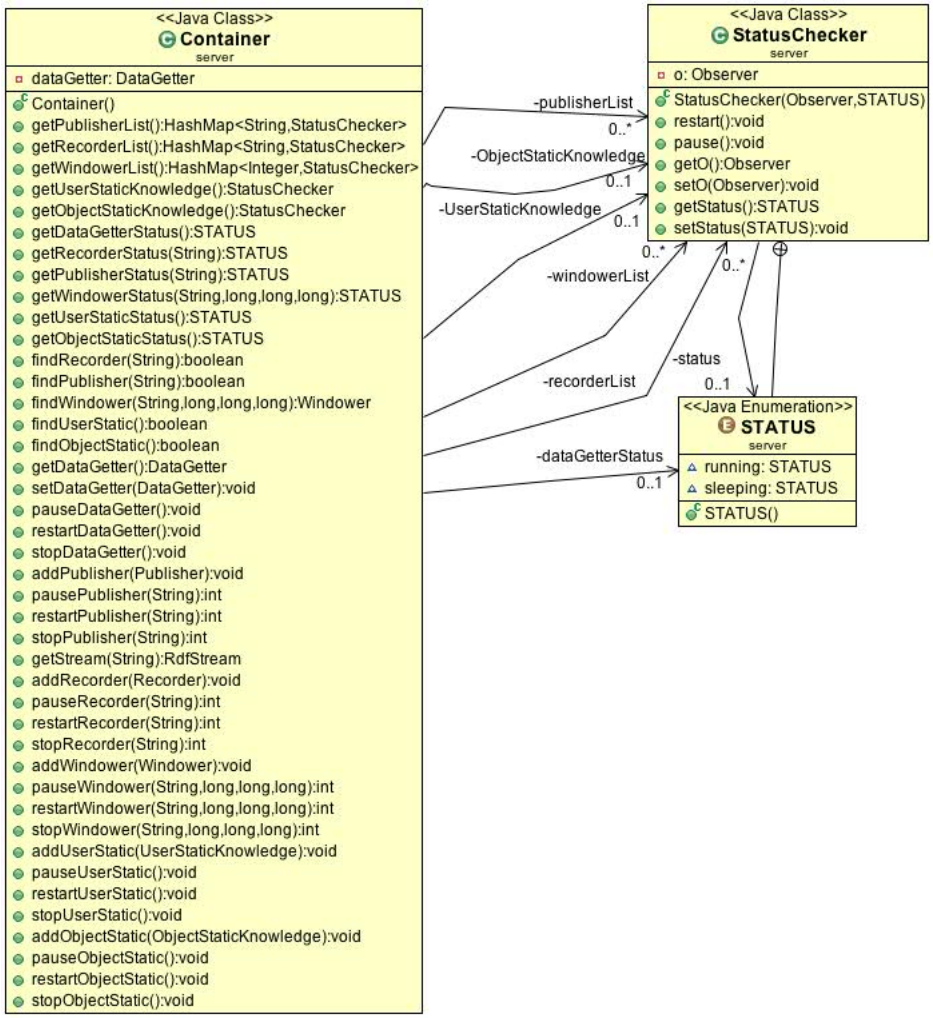

Figura 4.12: Class Diagram relativo alle strutture dati utilizzate per tenere traccia dello stato dei vari componenti

All'interno di ogni `Container`, sono presenti un puntatore a un `DataGetter` e una serie di `Hashmap` che contengono a loro volta tutti gli oggetti collegati al `DataGetter` stesso e il loro stato, che è rappresentato dall'oggetto `StatusChecker`, che contiene:

- *status*: variabile che rappresenta lo stato vero e proprio del componente;
- *STATUS*, una variabile di tipo enumerativo che contiene tutti i posibili valori da assegnare alla variabile *status*;

- i metodi che consentono di cambiare il valore di status: *pause()* che consente di portare il valore di *status* a *sleeping*, e *restart()* che riporta il valore di *status* a *running*.

La variabile di tipo `ENUM` è stata usata per vincolare a una serie finita di valori la variabile *status*.

Durante la progettazione dell'intera applicazione abbiamo fatto largo uso delle `Hashmap`, che sono un'implementazione della classe `Map`, sviluppata con l'utilizzo delle funzioni hash. Grazie alla loro struttura permettono performance costant-time per le operazioni di base (get e put), mentre il consumo di risorse legato all'iterazione su tutti gli elementi contenuti nella `Map` è legato al numero stesso degli elementi e alla grandezza del vettore delle chiavi. Il punto di forza di questa struttura dati è dunque rappresentato dalla velocità e dal consumo ridotto di risorse per svolgere le attività di base, per contro si ha un consumo di memoria maggiore per lo storage, ma le considerazioni appena presentate hanno fatto cadere la nostra scelta su questa struttura dati.

Attraverso i metodi contenuti in `SLD4SM Server` è possibile cambiare stato, aggiungere o togliere uno qualunque di questi oggetti, che non sono direttamente visibili all'utente ma possono essere comandati solo passando attraverso il telecomando del sistema.

La scelta di utilizzare un meccanismo estremamente centralizzato, come quello proposto, è dettata dai vincoli di sicurezza e integrità da rispettare nella creazione di un sistema complesso e utilizzabile anche da remoto.

4.1.6 Implementazione Factory Method Pattern

In questo paragrafo viene presentata la soluzione implementativa adottata per l'utilizzo del Factory Pattern, presentato concettualmente nel paragrafo 3.3.2. L'importanza del Factory Pattern è dovuta alla possibilità offerta allo sviluppatore di istanziare l'oggetto, specificato dall'utente attraverso un parametro, che specializza una classe nota.

Il primo passo da effettuare, per poter offrire questa funzionalità, è rappresentato dal mapping di tutte le sottoclassi di un oggetto specificato. Attraverso la Reflection API offerte da java non è possibile avere l'elenco delle sottoclassi, l'unica alternativa è rappresentata dalla scansione manuale del classpath, a questo scopo abbiamo utilizzato la classe `SubClassFinder` presentata in figura 4.13, che contiene, in particolare, il metodo `FindSubClass`, che riceve in ingresso, come parametro, il nome della superclasse di cui ricercare gli

oggetti che ne ereditano le funzionalità consentendo una ricerca approfondita anche all'interno delle librerie (.jar) presenti all'interno del classpath.

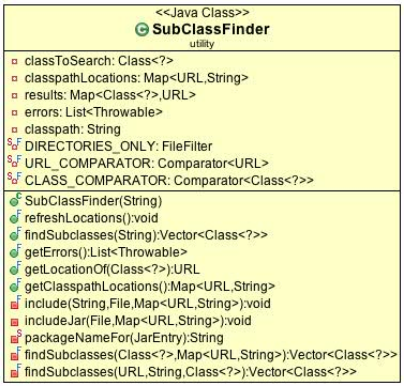

Figura 4.13: Class Diagram relativo alla classe SubClassFinder

Ad ogni tipologia di componente è dunque legata una classe di tipo Factory che, sfruttando le potenzialità di `SubClassFinder`, riempie una hashmap con un'istanza di ogni sottoclasse trovata, indicizzandole tramite una stringa, che proviene direttamente dal metodo type della sottoclasse appena istanziata e ne rappresenta il tipo vero e proprio. Questa hashmap di oggetti indicizzati, permette all'utente di scegliere l'istanza di suo interesse tramite un parametro (che deve corrispondere a un tipo conosciuto dall'applicazione), che fungerà da discriminante nella ricerca della sottoclasse all'interno della mappa.

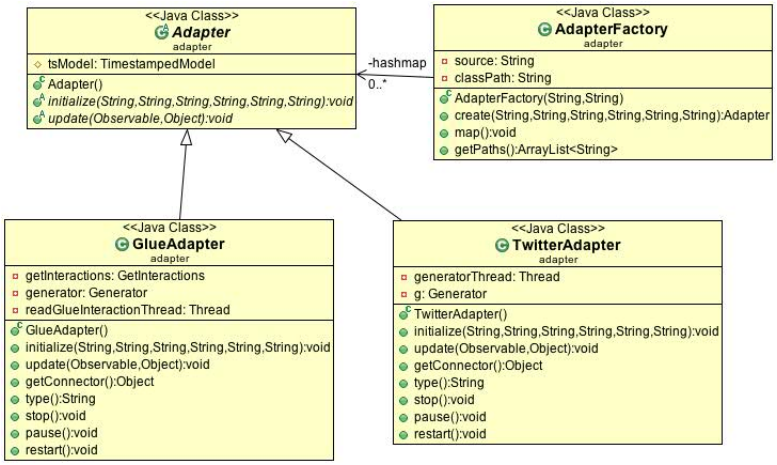

Figura 4.14: Esempio di implementazione del pattern factory

In figura 4.14 presentiamo un esempio concreto di implementazione del design pattern Factory, riguardante gli Adapter.

4.2 Conoscenza Statica

Come accennato nel paragrafo 3.4.2 la conoscenza statica si può suddividere in due sottoclassi specializzate, una si preoccupa dell'estrazione e della manipolazione dell'informazione relativa agli utenti, l'altra svolge le stesse funzioni per gli oggetti. La creazione dell'istanza più consona avviene sempre tramite l'utilizzo del design pattern Factory, che grazie a un parametro passato dall'utente all'atto della creazione istanzia l'oggetto desiderato.

Passiamo ora a una visione più dettagliata delle classi considerate.

4.2.1 Conoscenza statica relativa agli utenti

In figura 4.15 possiamo vedere lo schema delle classi relative alla User Static Knowledge, le signature dei metodi in comune tra le due implementazioni, che riguardano GetGlue e Twitter, sono specificate all'interno della classe astratta `UserStaticKnowledge`.

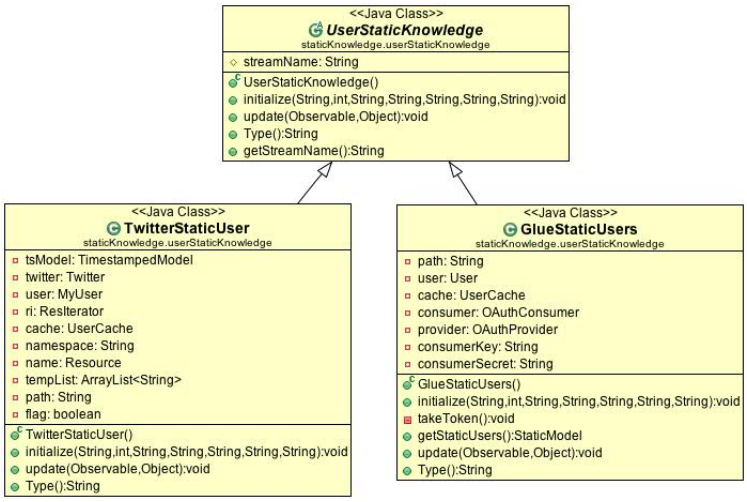

Figura 4.15: Class Diagram relativo agli oggetti a cui è delegata la creazione della conoscenza statica relativa agli utenti

Il metodo *initialize()* è legato all'utilizzo del design pattern Factory e permette di generalizzare l'inizializzazione dei parametri delle classi coinvolte, così come il metodo *type()*, che permette la scelta dell'oggetto da implementare da parte del pattern sopracitato; come

possiamo notare dalla presenza del metodo *update()* la superclasse implementa `Observer`, che grazie alla modularità che contraddistingue tutto il sistema, permette una comunicazione con il fornitore di dati a cui è associata.

Le due sottoclassi permettono di richiamare, all'interno del metodo *update()* metodi presenti all'interno di due librerie esterne (nel nostro caso rappresentate da `GlueAdapter` e `TwitterAdapter`), che consentono l'autenticazione, il recupero e la formattazione vera e propria dei dati relativi a ogni singolo utente presente nello stream.

La principale problematica incontrata durante lo sviluppo di questo frammento dell'applicazione è legata all'utilizzo, da parte dei servizi Web, di API REST, che come abbiamo già accennato, sono soggette a rate limiting da parte del provider, dunque la quantità di dati che è possibile estrarre risulta esigua rispetto alla grande mole di dati che fa parte dello stream.

4.2.2 Conoscenza statica relativa agli oggetti

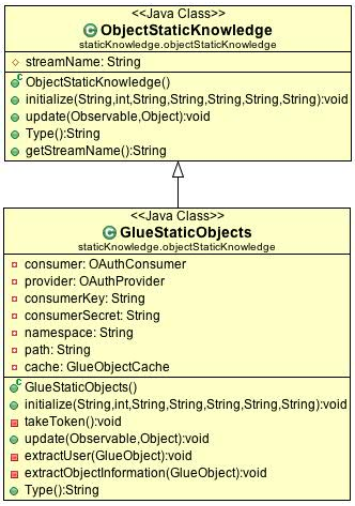

Figura 4.16: Class Diagram relativo agli oggetti a cui è delegata la creazione della conoscenza statica relativa agli oggetti

Come mostrato in Figura 4.16, le considerazioni fatte per conoscenza statica relativa agli utenti possono essere riportate anche in questo caso, con una sola grande differenza, la classe che eredita dalla classe astratta `ObjectStaticKnowledge` è una sola, che si riferisce al social network GetGlue. Per quanto riguarda Twitter, non è stata implementata nessuna classe relativa alla conoscenza statica degli oggetti, in attesa del relativo componente che risulta in via di

sviluppo da parte della società Coreana Saltlux nel contesto del progetto LarKC, anche se la nostra applicazione risulta già predisposta per restituire i dati sull'oggetto.

Andiamo con ordine, nella classe `ObjectStaticKnowledge` possiamo notare i metodi *intialize()* e *type()* che sono necessari per il funzionamento del pattern Factory, oltre al metodo *update()* che consente alle sottoclassi, che ereditano dalla superclasse l'implementazione di `Observer`, di ricevere le informazioni dal fornitore di dati. L'estrazione, la manipolazione e il salvataggio dei dati vengono delegati a metodi contenuti in una libreria esterna (`GlueAdapter`).

Utilizzo dei Thread per l'estrazione della conoscenza Statica

L'estrazione di tutte le informazioni relative a utenti e oggetti, in relazione a tutti i social network che abbiamo considerato, viene effettuata attraverso continue richieste a specifiche API REST, fatte tramite chiamate HTTP alle URI del REST Server del provider. La possibilità di parallelizzare le varie fasi di estrazione, permette una gestione più efficiente di tutta la fase di recupero dell'informazione, specialmente se ci si trova ad utilizzare l'applicazione su macchine dotate di più processori.

Nella porzione di codice 4.9 riportiamo il particolare della funzione run() del thread relativo all'estrazione dei friends di un utente di GetGlue, mentre in 4.10 riportiamo il particolare dell' invocazione dei thread sull'oggetto.

L'azione di più thread sullo stesso oggetto potrebbe portare a problemi di consistenza dell'informazione, infatti bisogna attendere, prima di continuare con l'elaborazione dei dati, che tutte le funzioni, che agiscono contemporaneamente, su un qualsiasi oggetto, abbiano concluso il loro lavoro. L'inserimento di una *join()* per ogni thread assolve a questa funzione e permette di ottenere informazioni consistenti per l'elaborazione successiva all'estrazione (vedi Codice 4.10 righe 14-16).

4.3 Implementazione servizio REST

Come abbiamo spiegato nel Paragrafo 2.4, REST è l'acronimo di Representational Transfer State, ed è un paradigma per la realizzazione di applicazioni Web che permette la manipolazione delle risorse per mezzo dei metodi GET, POST, PUT e DELETE del protocollo HTTP. Il concetto che sta alla base del paradigma REST è quello di risorsa, ovvero di una entità identificata univocamente da un URI e

```
URL url;
String friend;

url = new URL
  ("http://api.getglue.com/v2/user/friends?userId=" + user.getID());
HttpURLConnection request =
         (HttpURLConnection) url.openConnection();
consumer.sign(request);
request.connect();

InputStream ins = null;
ins = request.getInputStream();

XMLInputFactory factory =
        XMLInputFactory.newInstance();
XMLStreamReader parser =
        factory.createXMLStreamReader(ins);

int event = parser.next();

while(!(event == XMLStreamConstants.END_ELEMENT &&
parser.getLocalName().equals("response"))){
     event = parser.next();

      if (event == XMLStreamConstants.START_ELEMENT &&
       parser.getLocalName().equals("userId")){
                event = parser.next();
                friend = parser.getText();
                if (!friend.equals(user.getID())){
                        user.addFriend(friend);
                }

       }
}
```

Codice 4.9: Thread per l'estrazione dei friends relativi a un utente di Glue, porzione di codice

a cui è possibile accedere solo tramite invocazione di tale URI. Attori del paradigma REST sono le classiche componenti di una rete client/server che comunicano attraverso il protocollo HTTP e i suoi metodi.

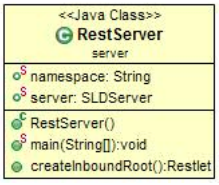

Figura 4.17: Class Diagram relativo alla classe RestServer

In generale i diversi metodi di HTTP vengono utilizzati per operazioni ben precise:

- GET: serve per richiedere al server le informazioni (sotto forma di entità) a cui si riferisce l'URI;

```
1  GetFriends getFriends = new GetFriends(user, consumer);
2  Thread friends = new Thread(getFriends);
3  friends.start();
4
5  GetFollowers getFollowers = new GetFollowers(user, consumer);
6  Thread followers = new Thread(getFollowers);
7  followers.start();
8
9  GetProfile getProfile = new GetProfile(user, consumer);
10 Thread profile = new Thread(getProfile);
11 profile.start();
12
13 try {
14         friends.join();
15         followers.join();
16         profile.join();
17 } catch (InterruptedException e) {
18         e.printStackTrace();
19 }
```

Codice 4.10: Porzione di codice relativa all'invocazione dei thread per l'estrazione di amici, followers e delle informazioni relative a un utente di GetGlue

- POST: serve per richiedere al server di accettare l'entità inviata tramite la richiesta (nel nostro caso, più semplicemente, di creare un'entità);
- DELETE: serve per richiedere al server di eliminare la risorsa identificata dall'URI;
- PUT: serve per richiedere al server di modificare la risorsa (attiva) identificata dall'URI.

La classe `RestServer` (vedi Figura 4.17) rappresenta la componente server della nostra architettura. In `RestServer` viene inizializzato un componente `SLD4SM Server` (Codice 4.11 - Riga 5), cioè il telecomando di tutta l'architettura attraverso il quale vengono svolte tutte le operazioni offerte dal servizio, e vengono definiti:

- il namespace del servizio (Codice 4.11 - Righe 6-7);
- la porta di comunicazione del server (Codice 4.11 - Riga 10: porta 8182 del protocollo HTTP);
- tutti gli URI invocabili sul servizio e la relativa istanza di classe responsabile della gestione della richiesta (Codice 4.11 - Dalla riga 21 in poi).

Tutte le classi che gestiscono i servizi utilizzano l'unica istanza di `SLD4SM Server`, inizializzata in `RestServer`, per l'interazione con il sistema e sono in grado di gestire le invocazioni sugli URI soltanto

```
1  public static String namespace;
2  public static SLDServer server;
3
4  public static void main(String[] args) throws Exception{
5      server = SLDServer.getIstance();
6      namespace = "http://www.example.com/";
7      server.setNamespace(namespace);
8
9      Component component = new Component();
10     component.getServers().add(Protocol.HTTP, 8182);
11
12     RestServer restServer = new RestServer();
13     component.getDefaultHost().attach("", restServer);
14     component.start();
15 }
16
17 public Restlet createInboundRoot(){
18     Router router = new Router(getContext());
19     getContext().getAttributes().put("server", server);
20
21     router.attach("/adapter/{source}/{streamname}",
22                             InitializeAdapter.class);
23     router.attach("/replayer/{streamname}/{platform}/{speed}",
24                              InitializeReplayer.class);
25     ...
26 }
```

Codice 4.11: RestServer, porzione di codice

se il metodo HTTP di invio della richiesta corrisponde con il metodo accettato dal servizio.

Il Codice 4.12 mostra una porzione di codice relativa alla classe che gestisce le richieste riguardanti l'aggiunta di un `Windower` ad uno stream di dati, la sua cancellazione e l'ottenimento dell'sGraph relativo alle finestre generate. Per semplicità è stata omesso il corpo dei metodi che gestiscono la cancellazione del `Windower` e le richieste degli sGraph.Il metodo `attachWindower()` ci mostra due cose essenziali: la prima è che ogni metodo della classe deve essere preceduto da una `Annotation`, ovvero una annotazione per il compilatore che indica l'HTTP Method richiesto perchè sia quel metodo a rispondere; la seconda è che per ogni metodo è possibile seguire uno schema di sviluppo standard diviso in tre parti:

- una prima parte (righe 6-11) in cui si ottiene l'istanza di `SLD4SM Server` attiva e in cui si ottengono i dati inviati tramite la richiesta HTTP;
- una seconda parte (riga 13) in cui si utilizza `SLD4SM Server` per effetturare l'operazione richiesta;
- una terza parte (righe 15-26) in cui, in base al codice restituito dall'invocazione dei metodi su `SLD4SM Server`, si crea la risposta da inviare al client.

```
public class AttachWindower extends ServerResource{
   private SLDServer sldServer;

   @Put
   public void attachWindower(){
     sldServer = (SLDServer) getContext()
                 .getAttributes().get("server");
     String streamName = (String) this.getRequest()
                 .getAttributes().get("streamname");
     String type = (String) this.getRequest()
                 .getAttributes().get("type");
     long dimension = Long.parseLong(
                 (String) this.getRequest().getAttributes()
                 .get("dimension"));
     long step = Long.parseLong((String) this.getRequest()
                 .getAttributes().get("step"));
     long expireTime = Long.parseLong(
                 (String) this.getRequest().getAttributes()
                 .get("expiretime"));

     int res = sldServer.addWindower(streamName,
                         type, dimension, step, expireTime);

     String sStatus;
     Status status;

     if(res == 0){
       sStatus = "Creato con successo";
       status = Status.SUCCESS_CREATED;
      } else if (res == 1){
        sStatus = "Windower esistente per questo stream";
        status = Status.SUCCESS_OK;
      } else {
        sStatus = "Lo stream indicato non esiste.";
        status = Status.CLIENT_ERROR_CONFLICT;
      }

      this.getResponse().setStatus(status, sStatus);
      this.getResponse().setEntity(sStatus, MediaType.TEXT_PLAIN);
   }

   @Delete
    public void deleteWindower(){
    ....
     }

     @Get
     public void getsGraph(){
     ....
     }
}
```

Codice 4.12: AttachWindower, porzione di codice

Capitolo 5

Test e Validazione prototipo

In questa sezione andremo a mostrare le caratteristiche in termini di efficienza ed efficacia del sistema `SLD4SM`.

5.1 Test di efficienza

Per i test di efficienza abbiamo utilizzato un personal computer con le seguenti caratteristiche: processore Intel Core 2 Duo T7500 a 2.2GHz, 4GB di RAM DDR2 a 667 MHz, Hard Disk a 5400 rpm.

5.1.1 Replayer

I test di rigenerazione di uno stream precedentemente registrato sono stati effettuati partendo dai dati ottenuti dalla sorgente GetGlue nei giorni 26 e 27 Aprile 2011.

Analizzando il campione di dati a disposizione, siamo giunti alla conclusione che l'`Adapter` collegato a GetGlue è in grado di fornire al sistema dei `TimestampedModel` per una media di uno ogni 7/8 secondi e che i `Model` ivi presenti contengono una media di 30 triple.

In Figura 5.1 la linea di colore blu mostra l'andamento dell'output del `Replayer` da disco a diverse velocità di replay. Nel caso preso in esame, il sistema è in grado di rigenerare lo stream fino ad una velocità massima di circa 2000x, per velocità superiori il sistema va a regime ed è in grado di processare al massimo tra le 790 e le 800 triple/sec. In rosso sono mostrate le prestazioni del `Replayer` da database MySQL. L'andamento del suo output è molto simile a quello della `Replayer` da disco con un output massimo compreso tra le 615 e le 630 triple/sec.

Trattandosi di velocità di replay di tre ordini di grandezza maggiore rispetto alla velocità di registrazione, confidiamo che il siste-

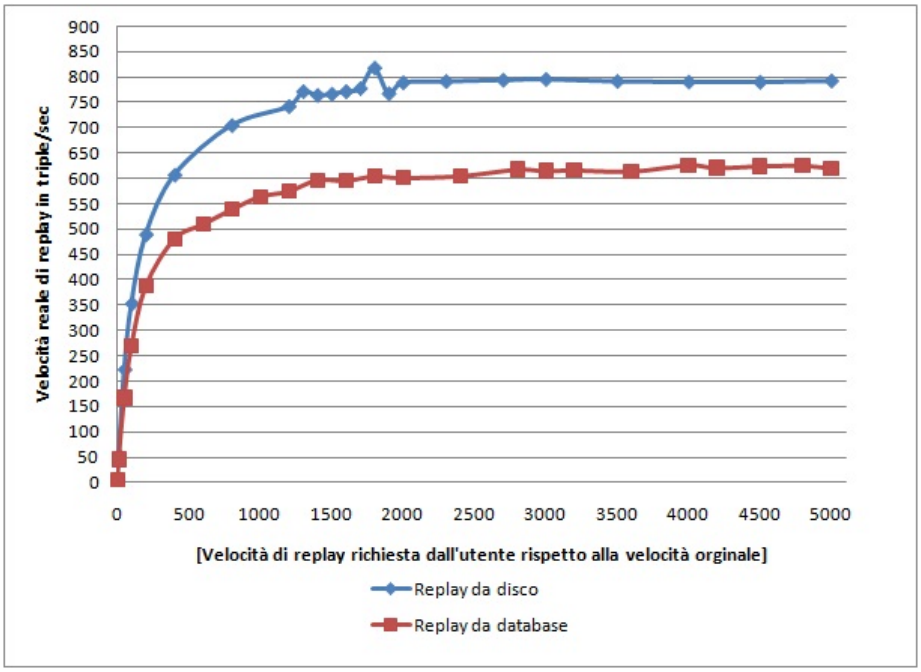

Figura 5.1: Capacità di SLD4SM di soddisfare le richieste di rigenerazione dello stream da parte dell'utente. Il sistema permette di rigenerare lo stream a una velocità n volte maggiore, fino ad un massimo di 800 triple/sec

ma sia in grado di raggiungere effettivamente le velocità di output massime trovate.

Non è stato possibile ricavare dati di efficienza dalla `Replayer` da modello persistente perchè, come vedremo nel prossimo paragrafo, il setting sperimentale non ha reso possibile registrare su modello persistente un campione significativo di dati, nè da sorgente reale, nè da generatore sintetico di dati. Quello che possiamo ipotizzare è che, con una macchina in grado di gestire la tecnologia TDB, l'andamento sarebbe molto simile a quello degli altri due `Replayer` ma con un output massimo minore.

5.1.2 Recorder

Per effettuare i test sulle prestazioni di registrazione di `SLD4SM` abbiamo utilizzato un generatore sintetico di triple RDF. Questo si è reso necessario in primo luogo per fornire, in modo controllato, un flusso costante di dati e in secondo luogo per evitare che ritardi dovuti alle invocazioni HTTP (in caso di utilizzo di `Adapter`) o al recupero di dati precedentemente registrati (in caso di utilizzo di `Replayer`) rallentassero il sistema. Per rendere i test più affidabili

abbiamo ripetuto le prove cinque volte e i dati di seguito presentati rappresentano una media di tutte le prove.

Come detto nel paragrafo precedente, l'`Adapter` collegato a GetGlue è in grado di fornire al sistema dei `TimestampedModel` i cui `Model` contengono una media di trenta triple, quindi abbiamo impostato il generatore sintetico di dati a questo valore e abbiamo ottenuto i risultati che ora andiamo a presentare.

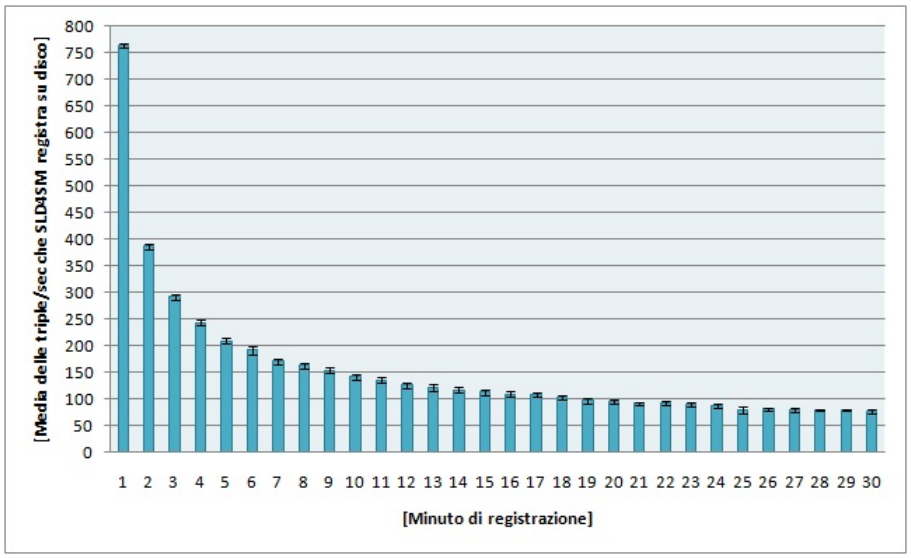

Figura 5.2: Le prestazioni di SLD4SM per la registrazione su disco si deteriorano con il passare dei minuti in quanto cresce la dimensione dell'rGraph e, quindi, il relativo tempo di salvataggio

In Figura 5.2 è mostrato l'andamento nel tempo delle triple al secondo che il sistema di registrazione su disco è in grado di processare. Come si può notare, inizialmente il sistema processa un elevato numero di triple (circa 760 triple/sec) che decresce in modo esponenziale fino al ventesimo minuto e da qui in poi rimane stabile sulle 80/85 triple/sec ma tende a decrescere anche se molto lentamente. Questo andamento è dovuto al metodo di registrazione dei *Named Graph* su disco. Per quanto riguarda la registrazione degli *iGraph* non vi sono problemi di sorta, in quanto avendo impostato la dimensione costante dei `Model` a trenta triple anche il tempo di salvataggio su disco rimane costante. Il crollo delle prestazioni è dovuto al salvataggio su disco dell'*rGraph*, questo perchè dovendo tener traccia di tutti gli *iGraph* registrati, la sua dimensione cresce molto velocemente e la nostra implementazione di `SLD4SM Core` prevede il salvataggio dell'*rGraph* ogniqualvolta arrivi un nuovo `TimestampedModel` dal `DataGetter`.

Una possibile soluzione per migliorare le prestazioni potrebbe essere quella di non registrare su disco l'*rGraph* ad ogni nuovo arrivo di `TimestampedModel`, ma di effettuare tale registrazione ad intervalli regolari o nel momento di fine della registrazione; ciò, d'altro canto, potrebbe portare, nel caso di problemi al sistema ospitante l'applicazione, alla perdita di una parte importante dei dati registrati.

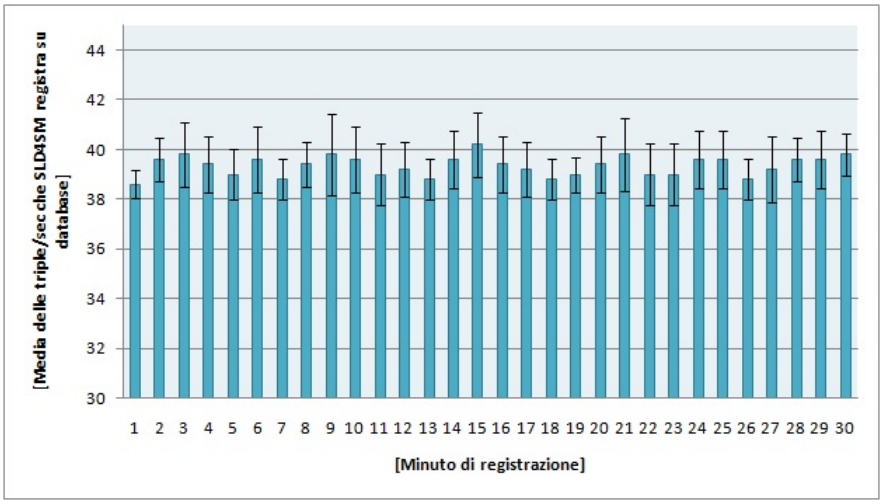

Figura 5.3: Le prestazioni di SLD4SM per la registrazione su database sono poco variabili nel tempo e si attestano su una media di registrazione di 40 triple/sec

In Figura 5.3 sono mostrate le prestazioni della registrazione su database MySQL. Come si può notare qui le prestazioni sono poco variabili e il sistema è in grado di registrare una media di circa 40 triple/sec. Questo andamento non stupisce in quanto il framework Jena utilizza un metodo incrementale per la registrazione dei `Model` su database che risulta utile per il salvataggio dell'*rGraph*: ad ogni nuovo arrivo di `TimestampedModel` corrisponde una *update* sulla relativa tabella del database, con l'inserzione delle poche triple necessarie a tener traccia del nuovo *iGraph* creato. Anche in questo caso il tempo di salvataggio di un *iGraph* rimane costante poichè è costante il numero di triple contenuto nel `TimestampedModel`.

In Figura 5.4 sono mostrate le prestazioni della registrazione su modello persistente (TDB). E' stato possibile registrare per soli quattro minuti in quanto il setting sperimentale non era in grado di supportare le richieste di risorse del framework Jena per la registrazione su modello persistente: superati i quattro minuti l'*rGraph* raggiunge una dimensione tale da provocare un'eccezione del tipo `java.lang.OutOfMemoryError: Java heap space` che, anche au-

mentando la dimensione dell'*heap* di Java fino al massimo disponibile per il nostro setting sperimentale, non è possibile risolvere. I dati ottenuti mostrano che l'output massimo per questo tipo di registrazione è compreso tra le 27 e le 30 triple/sec e che in fase di lancio della registrazione si ottiene il valore minimo di output (una media di circa 27 triple/sec).

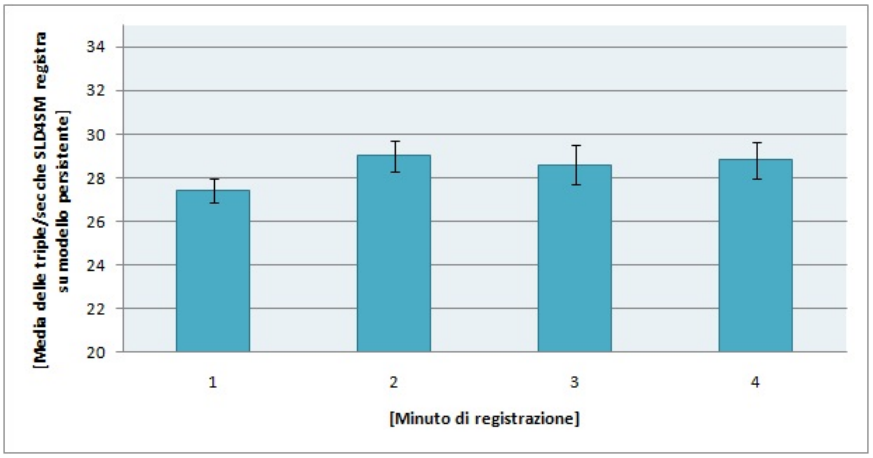

Figura 5.4: Le prestazioni di SLD4SM per la registrazione su modello persistente sono poco variabili nel tempo e si attestano su una media di registrazione di circa 28 triple/sec. Il setting sperimentale non permette di oltrepassare i quattro minuti di registrazione

Osservando i dati prestazionali ottenuti dai `Replayer` e quelli ottenuti dai `Recorder` con il generatore sintetico di dati abbiamo ritenuto non necessario effettuare il testing dei `Recorder` partendo da dati rigenerati. Questa scelta è dovuta al fatto che il throughput del generatore di dati sintetici è maggiore di quello dei `Replayer` e quindi sarebbero questi ultimi a limitare la registrazione dei dati e non la registrazione stessa.

Dall'analisi delle performance è possibile dedurre che il miglior metodo di salvataggio risulta essere quello su database, infatti, sebbene sia in grado di processare un numero di triple al secondo minore rispetto alla registrazione su disco, le sue performance rimangono costanti, non si deteriorano col passare del tempo e sono superiori a quelle relative alla registrazione su modello persistente.

5.1.3 Windower

In questa sezione mostreremo le prestazioni di un `Windower` logico con la seguente configurazione: *dimensione* 20 secondi, *step* 13 secondi ed *expire time* 30 secondi.

Per poter effettuare i test abbiamo implementato un software di prova in grado di richiedere in modo continuo al `Windower` la finestra attuale, cioè l'*sGraph* attuale e i relativi *iGraph*.

In Figura 5.5 sono mostrate le prestazioni relative ad un singolo esperimento: in ascissa è indicata la velocità d'ingresso al Windower delle triple e in ordinata sono indicati i grafi che il software di prova è in grado di consumare in un secondo.

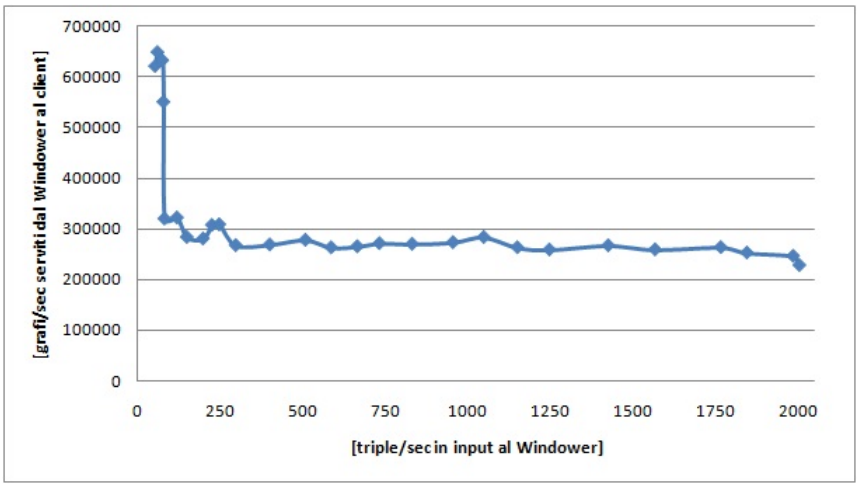

Figura 5.5: Prestazioni di un Windower logico con dimensione finestra 20s, step 13s e expire time 30s. Per valori di input inferiori alle 80 triple/sec il Windower è in grado di servire al client circa 650 mila grafi al secondo, per valori superiori satura ed è in grado di fornire circa 260 mila grafi al secondo

Il grafico mostra che per velocità di input basse (fino a circa 80 triple/sec), il programma di prova è in grado di consumare un numero elevato di grafi pari a poco più del doppio di quelli consumati per velocità di input maggiori di 80 triple/sec. Il motivo per cui accade questo fenomeno è imputabile presumibilmente al fatto che le instanze di `SLD4SM` e del programma di prova risiedono sulla stessa macchina condividendone le risorse, questo comporta che fintanto che il `Windower` non raggiunge limiti che portano vicino alla saturazione delle risorse, il programma di prova riesce ad ottenere il massimo numero possibile di *Named Graph*.

Dal grafico è possibile dedurre che il `Windower` logico è in grado di processare al massimo tra le 75 e le 85 triple/sec, per valori superiori le triple vengono messe in coda in attesa di essere processate. Inoltre, quando la velocità di input supera le 2000 triple/sec il sistema genera un'eccezione del tipo `java.lang.OutOfMemoryError: Java heap space` che, anche in questo caso come nel caso della registra-

zione su modello persistente, non è possibile risolvere aumentando l'*heap* a disposizione di Java.

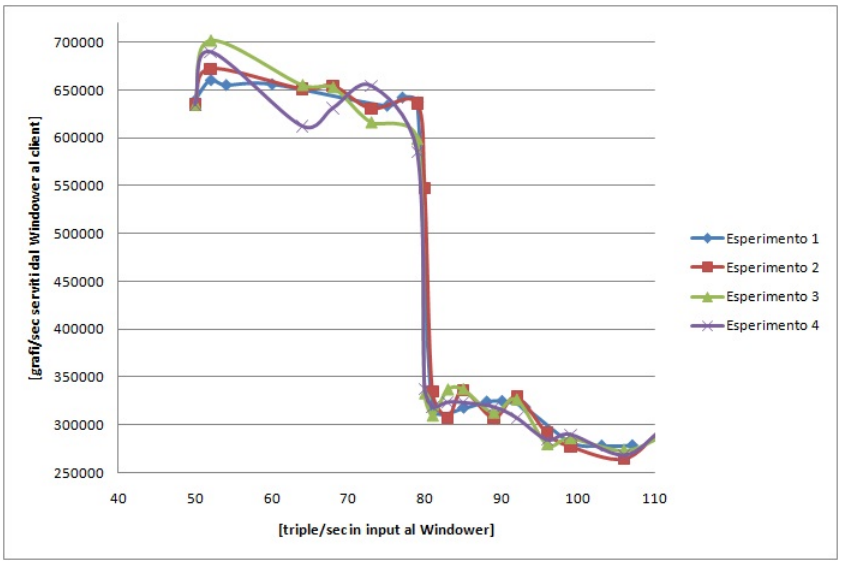

Figura 5.6: Prestazioni per quattro esperimenti di un Windower logico con dimensione finestra 20s, step 13s e expire time 30s. Tutti mostrano che ad un valore di input di 80 triple/sec il numero di grafi serviti al client cala di colpo e si assesta a circa 260 mila grafi al secondo

In Figura 5.6 sono mostrati i dati ricavati da quattro esperimenti per velocità di input che vanno dalle 50 alle 110 triple/sec. Come si può vedere risulta essere proprio l'input maggiore di 80 triple/sec che porta al massimo i dati processabili dal `Windower` e limita il numero di *Named Graph* richiedibili dal programma di prova in un secondo.

5.1.4 C-SPARQL Engine

Per il testing dello `C-SPARQL Engine` abbiamo inizialmente utilizzato il generatore sintetico di dati per poi passare al `Replayer` da disco dello stream salvato dalla sorgente GetGlue nei giorni 26 e 27 Aprile 2011.

```
1 REGISTER QUERY QueryTest AS
2 SELECT ?s ?p ?o
3 FROM STREAM <http://ex.org/glue/>
4  [RANGE TRIPLES 10]
5 WHERE ?s ?p ?o
```

Codice 5.1: Estrazione delle ultime dieci triple dallo stream

Anche in questo caso sono stati effettuati cinque prove sperimentali e i dati che andremo a presentare ne rappresentano la media.

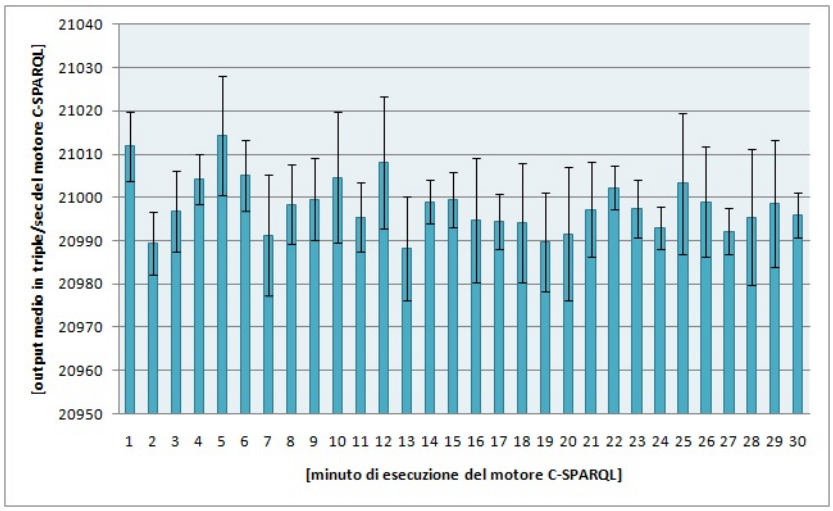

Figura 5.7: Prestazioni dello C-SPARQL Engine con sorgente sintetica

Per evitare che fosse la complessità della interrogazione a limitare la prestazioni dello `C-SPARQL Engine`, abbiamo utilizzato una semplicissima query (mostrata in Codice 5.1) che seleziona le ultime dieci triple `<soggetto, predicato, oggetto>` dallo stream che abbiamo generato (`http://ex.org/glue/`).

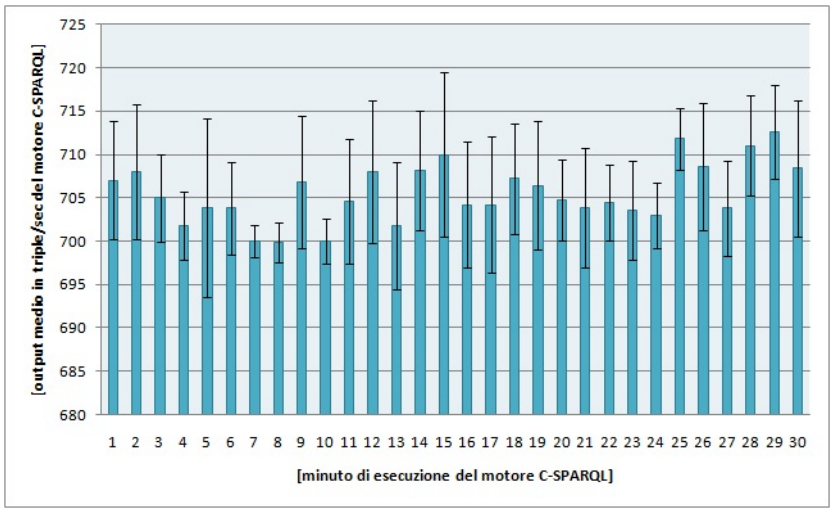

Figura 5.8: Prestazioni dello C-SPARQL Engine con sorgente rigenerata

In Figura 5.7 è mostrato l'andamento dell'input che lo `C-SPARQL Engine` è in grado di processare quando la sorgente è il generatore

sintetico di dati. Come si può notare l'andamento è abbastanza stabile e si attesta tra le 20995 e le 21015 triple/sec, per una media di circa 21000 triple/sec.

In Figura 5.8 è mostrato l'andamento dell'input che lo `C-SPARQL Engine` è in grado di processare quando invece la sorgente è un `Replayer` da disco.

In questo caso lo `C-SPARQL Engine` è in grado di processare una media di circa 705 triple/sec, ovvero circa 85 triple/sec in meno rispetto all'output massimo del `Replayer` da disco (vedi Paragrafo 5.1.1). Questo avviene perchè `C-SPARQL Engine` e `Replayer` risiedono sulla stessa macchina e ne condividono le risorse.

5.2 Test di efficacia sullo C-SPARQL Engine

In questa sezione presenteremo un test di efficacia, partendo dallo schema di componenti presentato in figura 5.9.

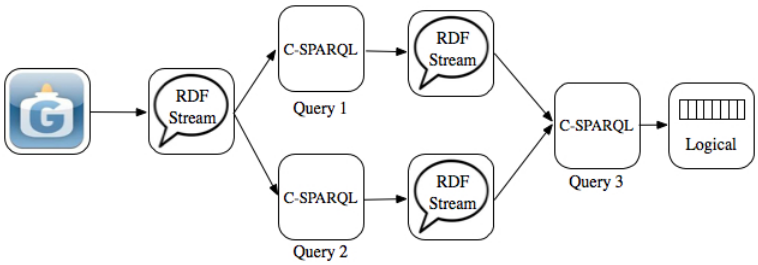

Figura 5.9: Schema complesso utile per misurare le prestazioni

Nello schema proposto, viene creato uno stream RDF, utilizzando i dati provenienti da un `Adapter` per il social network GetGlue, e fornito in ingresso a due blocchi C-SPARQL, i cui risultati, anch'essi presentati come stream RDF, verranno utilizzati come fonte di dati per un altro componente di tipo C-SPARQL. I risultati verranno presentati all'utente finale tramite un `Windower` di tipo logico.

Passiamo ora in rassegna le caratteristiche dei tre blocchi che hanno il compito di interrogare i vari stream, presentando le query che li caratterizzano:

- Prima Query: attraverso l'interrogazione CioChePiaceAgliAmiciDiAlice, presentata in Codice 5.2, vengono estratti tutti i topic che piacciono agli amici di Alice, dallo stream proveniente da GetGlue.

  Tramite la clausola REGISTER STREAM viene imposto al sistema di produrre in output uno stream di triple formate da

```
1  REGISTER STREAM CioChePiaceAgliAmiciDiAlice
2  COMPUTED EVERY 5m AS
3  PREFIX sr: <http://www.streamreasoning.org/sr4ld2011/onto/>
4  PREFIX srd: <http://www.streamreasoning.org/sr4ld2011/data/>
5  PREFIX skos: <http://www.w3.org/2004/02/skos/core/>
6  PREFIX yago: <http://dbpedia.org/class/yago/>
7  CONSTRUCT { ?friend sr: talksAboutPositively ?topic}
8  FROM STREAM <http://ex.org/glue/>
9   [RANGE 30m STEP 5m]
10 WHERE {
11     ?topic a sr:Topic .
12     ?friend sr:follows srd:Alice .
13     sr:posts ?interaction .
14     ?interaction sr:talksAboutPositively ?topic .
15 }
```

Codice 5.2: Interrogazione che estrae le cose che piacciono agli amici di Alice

`<soggetto, talksAboutPositively, topic>`. Partendo dallo stream `http://ex.org/glue/`, attraverso il codice contenuto nella clausola WHERE, presentato dalla linea 11 alla linea 14, vengono estratti gli amici di Alice insieme a tutti i topic di cui hanno parlato bene.

- Seconda Query: attraverso l'interrogazione CioChePiaceAgliOpinionMakers, presentata in Codice 5.3, vengono estratti tutti i topic che piacciono agli OpinionMakers dallo stream proveniente da GetGlue;

```
1  REGISTER STREAM CioChePiaceAgliOpinionMaker
2  COMPUTED EVERY 5m AS
3  PREFIX sr: <http://larkc.eu/csparql/sparql/jena/ext/>
4  PREFIX f: <http://larkc.eu/csparql/sparql/jena/ext/>
5  CONSTRUCT opinionMaker a sr:opinionMaker;
6       sr:talksAboutPositively ?topic
7  FROM STREAM <http://ex.org/glue/>
8      [RANGE 30m STEP 5m]
9  WHERE {
10     ?opinionMaker sr:talksAboutPositively ?topic .
11     ?follower sr:follows ?opinionMaker .
12     ?follower sr:talksAboutPositively ?topic .
13     FILTER (f:timestamp(?follower) > f:timestamp(?opinionMaker) )
14 }
15 HAVING (COUNT(DISTINCTfollower) > 3 )
```

Codice 5.3: Interrogazione che estrae le cose che piacciono agli opinionMaker

Tramite la clausola REGISTER STREAM viene imposto al sistema di produrre in output uno stream di triple formate da `<soggetto, talksAboutPositively, topic>`, in questo caso specifico il soggetto è rappresentato da un opinionMakers. Utilizzando lo stream `http://ex.org/glue/` come punto di partenza, attraverso il codice presentato alla linea 10 vengono

estratti gli opinionMakers che parlano positivamente del topic preso in considerazione, attraverso le linee 11 e 12 vengono estratti i followers degli opinionMakers che esprimono anch'essi un'opinione positiva sul topic precedentemente considerato. La FILTER alla linea 13 controlla che il micro-post dei followers, che riguarda lo stesso argomento contenuto nel post degli opinionMakers considerato, avvenga posteriormente nel tempo, infine la clausola HAVING alla linea 15 estrae solo i veri opinionMakers, ossia quelli con almeno 10 followers.

- Terza Query: attraverso Top10TopicChePotrebberoPiacereAdAlice, presentata in Codice 5.4, vengono estratti 10 topic che potrebbero piacere ad Alice, partendo dallo stream che contiene le preferenze dei suoi amici e degli opinionMakers.

```
1  REGISTER STREAM Top10TopicChePotrebberoPiacereAdAlice
2      COMPUTED EVERY 5m AS
3  PREFIX sr: <http://larkc.eu/csparql/sparql/jena/ext/>
4  PREFIX f: <http://larkc.eu/csparql/sparql/jena/ext/>
5  PREFIX srd: <http://www.streamreasoning.org/sr4ld2011/data>
6
7  SELECT ?newTopic ( COUNT ( DISTINCT ?follower) as ?numOfFollowers)
8  FROM STREAM
9      <http://www.streamreasoning.org/CioChePiaceAgliAmiciDiAlice>
10     [RANGE 30m STEP 5m]
11 FROM STREAM
12    <<http://www.streamreasoning.org/CioChePiaceAgliOpinionMakers>>
13     [RANGE 30m STEP 5m]
14 WHERE {
15     ?friend sr:talksAboutPositively ?newTopic .
16     ?opinionMaker a sr:opinionMaker;
17     sr:talksAboutPositively ?newTopic .
18     srd:Alice sr:talksAboutPositively ?oldTopic .
19     FILTER ((?newTopic = ?oldTopic)!
20 }
21 GROUP BY ?newTopic
22 ORDER BY DESC (?numOfFollowers)
23 LIMIT 10
```

Codice 5.4: Interrogazione che estrae le 10 cose che potrebbero piacere ad Alice

Tramite la clausola REGISTER STREAM viene imposto al sistema di produrre in output uno stream di tuple formate da `<soggetto, numOfFollowers>`, in questo caso specifico il soggetto è rappresentato da un topic e numero di followers è una variabile calcolata attraverso una somma di tutti i follower differenti che gradiscono il topic specificato nel soggetto. Partendo dai due stream creati tramite le Query 5.2 e 5.3 vengono estratti, attraverso il codice presentato dalla linea 15 alla linea 18, i topic che potrebbero piacere ad Alice, effettuando un controllo incrociato sulle preferenze (talksAboutPositively) degli amici e degli opinionMakers. La clausola GROUP BY alla riga

21 raggruppa i risultati per topic, tramite la riga 22 i risultati vengono ordinati in modo decrescente per numero di followers, infine la clausola LIMIT alla linea 23 limita a 10 il numero di risultati.

Capitolo 6

Conclusioni e Sviluppi Futuri

6.1 Conclusioni

L'utilizzo di RDF per la creazione di documenti interpretabili da agenti automatici e l'utilizzo del linguaggio di interrogazione SPARQL ha segnato un profondo passo avanti nella storia del Web, dando vita a quello che oggi chiamiamo Web Semantico.

I recenti lavori nell'ambito dello stream reasoning da parte del team LarKC stanno mostrando di essere un anello di congiuzione tra il mondo dei DSMS e quello del Web Semantico: l'estensione di SPARQL in C-SPARQL e il conseguente sviluppo dello C-SPARQL Engine, infatti, offrono nuove possibilità per l'estrazione di informazione utile da flussi continui e altamente variabili di dati.

SLD4SM si inserisce in questo contesto e permette la generazione di stream di dati standardizzati secondo i canoni del Web Semantico attraverso l'utilizzo del framework RDF. In particolare, il sistema è in grado di svolgere un'opera di traduzione delle informazioni grezze provenienti dai Social Network Twitter e GetGlue in una formattazione interna propria e di utilizzare tale formattazione per effettuarne la registrazione, la ripetizione, la pubblicazione in un formato standard, e per fornire allo C-SPARQL Engine una fonte di dati appropriata.

La registrazione di uno stream di dati e la conseguente ripetizione permettono un utilizzo del suo contenuto informativo in un istante di tempo differito, aprendo un enorme insieme di possibilità relative alla fruzione dell'informazione da parte degli utenti.

Un'attenzione maggiore è stata posta per la pubblicazione dei dati attraverso i `Windower`, che come fa intuire il nome, permettono di avere a disposizione l'intero stream suddiviso in finestre, siano esse logiche o fisiche, rappresentate dagli *iGraph* e indicizzate dagli *sGraph*. La presentazione dell'informazione attraverso questo me-

todo, permette di manipolare l'insieme di dati come se si trattasse di un flusso informativo continuo vero e proprio in cui i componenti più recenti hanno maggiore importanza rispetto a quelli più vecchi.

Lo scopo principale di tutta l'infrastruttura è quello di rendere disponibili i dati per il motore C-SPARQL attraverso la pubblicazione del flusso informativo su uno stream RDF. Questa particolare tipologia di stream può essere registrata nello C-SPARQL Engine insieme a una query scritta in linguaggio C-SPARQL, quest'ultima andrà a operare direttamente sul flusso di dati RDF e produrrà risultati sotto forma di `RDFTable`. Per permettere un'ulteriore manipolazione dei dati da parte dell'applicazione, è stato creato il `C-SPARQL Adapter` che risolve i problemi di eterogeneità delle strutture dati tra l'output del motore e l'infrastruttura software, trasformando le `RDFTable` in `TimestampedModel`.

SLD4SM permette una trasformazione totale dei dati, mantenendo intatto tutto il loro contenuto informativo, rendendoli adatti ad un ambiente semantico e grazie alla modularità che la contraddistingue, permette un alto grado di personalizzazione da parte dell'utente. All'interno di `SLD4SM Core` e di `SLD4SM Server` è possibile aggiungere nuovi componenti dichiarandoli sottoclasse di un oggetto astratto già presente, in questo modo è possibile, ad esempio, aggiungere nuove fonti da cui estrarre i dati attraverso gli `Adapter`, o nuove piattaforme su cui registrarli creando un nuovo `Recorder`.

La possibilità di plasmare l'intera applicazione e di modellarla sulle esigenze dell'utente rende l'intera infrastruttura software altamente configurabile e versatile.

6.2 Sviluppi futuri

I dati presentati nel Capitolo 5 rappresentano una versione di testing preliminare. Per comprendere le vere potenzialità di `SLD4SM` sarebbe necessario svolgere dei test estensivi su macchine di maggior potenza o in ambiente cloud; ciò porterebbe ad ottenere dei dati prestazionali di `SLD4SM` più vicini a quelli reali, impedendo al setting sperimentale di limitare il sistema.

Nei prossimi paragrafi di questa sezione passeremo in rassegna le possibili future estensioni per `SLD4SM` in grado di migliorarne l'usabilità, la flessibilità e la potenza. In particolare, presenteremo eventuali estensioni per quanto riguarda:

- la *Content Negotiation*;
- l'aggiunta di nuove fonti di dati;

- ulteriori funzionalità aggiuntive.

6.2.1 Content Negotiation

I servizi REST di SLD4SM rispondono alle invocazioni HTTP senza essere in grado di distinguere se chi effettua la richiesta è una persona o una macchina, cioè non sono in grado di distinguere l'user agent che effettua la richiesta fornendo in output esclusivamente file RDF. Tale comportamento può rappresentare una limitazione in quanto rende adatto il sistema per un ambiente automatizzato, ma lo rende difficilmente utilizzabile per l'interpretazione dei risultati da parte di attori umani. Ciò è dovuto al fatto che il formato RDF è ideale per la formattazione di dati in un mondo semantico, ma risulta difficilmente leggibile e interpretabile da un essere umano.

Per risolvere questo problema potrebbe essere d'aiuto l'utilizzo della *Content Negotiation*. Per *Content Negotiation* in ambito HTTP si intende la possibilità di offrire come risposta all'invocazione di un URI versioni differenti dello stesso documento in base alla tipologia di user agent che ha inoltrato la richiesta: il client inserisce all'interno dell'header del pacchetto HTTP quale tipo di rappresentazione preferisce e il server, ispezionando il pacchetto, può capire che tipo di documento inviare come risposta.

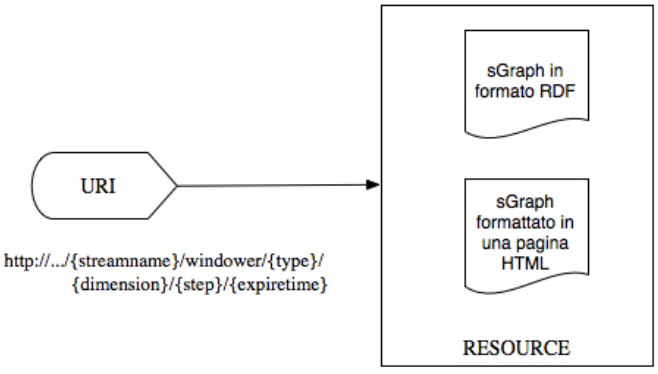

Figura 6.1: Esempio di URI associata a risorse diverse

In Figura 6.1 si può vedere come alla URI di richiesta di un *sGraph* ad un `Windower` (vedi tabella A.1) possono essere associate diverse risorse, da utilizzare all'occorrenza in base alle preferenze specificate dal client nel pacchetto HTTP. In questo piccolo esempio, l'applicazione potrebbe offrire un file di tipo RDF se la richiesta provenisse da un agente automatizzato, mentre potrebbe ritornare

un file formattato in HTML se la richiesta provenisse da un essere umano attraverso un client REST.

Lo sviluppo di tale soluzione per SLD4SM renderebbe il sistema adatto sia per un ambiente completamente automatizzato che per un ambiente in cui l'interfacciamento con l'essere umano è ancora necessario, rendendo human readable dei dati creati prettamente per un ambiente semantico.

6.2.2 Aggiunta di nuovi Social Network come fonti di dati

La modularità di SLD4SM permette l'aggiunta di `Adapter` relativi a nuovi Social Network consentendo, in questo modo, l'ampliamento delle possibili fonti di dati.

Un Social Media che bene si presterebbe ad operare con `SLD4SM` è sicuramente rappresentato da FourSquare[1], un Social Network basato sulla geolocalizzazione e disponibile attraverso browser e applicativi per dispositivi mobile dotati di GPS. Analizzando le caratteristiche di questo Social Media si possono notare vari punti in comune con il paradigma di dati presentato nella Sezione 3.1 e quindi con i due Social Network di cui è stato già implementato l'`Adapter`. In particolare, FourSquare risulta molto simile, per funzionalità, a GetGlue.

L'utente di FourSquare può fare check-in in qualunque luogo si trova, guadagnando *beadge* legati alle varie attività, alla continuità di utilizzo o alle promozioni in atto da parte dei gestori delle attività commerciali registrati sul Social Network. La fedeltà nel check-in in un dato luogo, viene premiata con il titolo di Mayorship.

Dall'analisi effettuata si può intuire che lo schema dei dati che sta dietro all'intera infrastruttura di FourSquare è il seguente:

```
<soggetto , siTrovaIn , oggetto>
```

Il soggetto rappresenta l'utente che sta utilizzando l'applicazione mentre l'oggetto rappresenta un qualsiasi luogo identificato tramite coordinate GPS. Ogni utente può creare la propria rete sociale tramite la possibilità di richiedere l'amicizia, commentando i post di altri utilizzatori o i luoghi da lui visitati.

Le API offerte agli sviluppatori sono particolarmente simili a quelle di GetGlue: esse infatti richiedono lo stesso metodo di autenticazione e di invio delle richieste HTTP.

La risposte alle invocazioni delle Web API sono formattate in file json. Questo standard di rappresentazione delle informazioni rende possibile estenderne l'utilizzo nei più disparati campi.

[1] `https://it.foursquare.com/`

Le invocazioni delle API sono soggette a rate limiting, in particolare ogni utente ha la possibilità di eseguire massimo 500 richieste/ora per ogni tipo di servizio.

6.2.3 Ulteriori funzionalità aggiuntive

MicroPost Analysis

Negli ultimi anni la popolarità e la diffusione di sistemi Location Based (LBS) è aumentata in modo esponenziale, facendo aumentare di conseguenza la quantità di informazioni provenienti dai servizi online, in particolare dai Social Media. Questa mole di dati potrebbe confondere l'utilizzatore che si troverebbe in difficoltà nel gestire questo sovraccarico informativo.

Un sostanziale aiuto per lo user è rappresentato dalla MicroPost Analysis, che utilizza un sistema ibrido di Stream Reasoning, il quale ,fondendo dati provenienti da un'analisi dei Social Network e dati relativi alla geolocalizzazione, si propone di capire in modo più profondo il significato dell'oggetto che compone la tripla del nostro modello di dati. Questa maggiore comprensione dell'informazione, permette di arrivare ad una conoscenza più particolare del topic in questione, estraendone il POI[2] che lo rappresenta nella realtà e le sensazioni, positive o negative, che il creatore del post ha voluto comunicare.

Esistono lavori applicativi in questo senso supportati dal team LarkC e in particolare ci riferiamo all'applicazione mobile BOTTARI [7], sviluppata dalla società coreana Saltlux.

Questo tipo di estensione potrebbe aumentare le potenzialità di `SLD4SM` fornendo all'utente un sistema più semplice e immediato di interfacciarsi con le informazioni provenienti dai Social Media.

Reasoning Induttivo

Per Reasoning Induttivo si intende una particolare tipologia di analisi, che permette di ricavare informazioni tramite una procedura induttiva. Un sistema di questo tipo sarebbe in grado di apprendere dai comportamenti dell'utente e di predire, in modo dinamico, informazioni sui dati relazionali.

Presentiamo ora una query d'esempio (vedi Codice 6.1), che riferendosi allo user Giulia e utilizzando le probabilità, cerca di predire quali utenti dovrebbe seguire Giulia per trovare nuove attrazioni per bambini, prendendo in considerazione anche le persone che non sono tra i suoi amici o followers.

[2] Point Of Interest

```
1  SELECT ?user ?prob
2  FROM STREAM <http://bottari.saltlux.com/OpinionMakers
3   [RANGE 30m STEP 30s]
4  WHERE{
5       ?opinionMaker a twd:opinionMaker ;
6                     twd:discuss [ twd:talksAboutPositively ?poi ] .
7       ?poi skos:subject twd:attractionsForKids .
8       :Giulia twd:following ?opinionMaker. WITH PROB ?prob
9        FILTER(?prob>0.8&&?prob<1)
10 }
11 ORDER BY ?prob
```

Codice 6.1: Interrogazione che estrae, utilizzando le probabilità, gli user più interessanti per Giulia

ll codice presente dalla linea 5 alla linea 7 estrae gli opinionMakers che hanno recentemente espresso parere positivo riguardo alle attrazioni per bambini. Il Triple Pattern alla linea 8 estrae gli utenti seguiti da Giulia e il costrutto WITH PROB estende SPARQL permettendogli di interrogare modelli induttivi. La variabile ?prob assume valore 1 per gli utenti che lei sta già seguendo e assume valori tra 0.8 e 1 per gli utenti che le vengono raccomandati. La clausola ORDER BY permette di ritornare gli utenti ordinati per probabilità decrescente, la query ritorna dunque coppie:

`<Utente, Probabilità>`

Questo tipo di componenti permettono quindi al sistema di creare nuova conoscenza, attraverso lo studio di dati provenienti dai Social Media, in modo da predire, con un certo grado di sicurezza, comportamenti futuri legati ad un utente.

Sviluppo nuove interfacce

La presentazione dei dati all'utente e la sua l'usabilità sono parti importanti di un'applicazione. Lo sviluppo di interfacce di comunicazione tra l'infrastruttura software e lo user permetterebbe di aumentare le possibilità di utilizzo di `SLD4SM`. La grande diffusione dei Social Media va di pari passo con la diffusione di dispositivi mobili che permettono la fruizione dell'informazione in qualsiasi momento e luogo, permettendo all'utilizzatore di essere geolocalizzato con grande precisione. Per questi motivi lo sviluppo di un'interfaccia mobile, per rendere utilizzabile `SLD4SM` su smartphone o tablet, permetterebbe all'applicazione di avere una maggiore diffusione, per via della facilità di utilizzo offerta all'utente.

Appendice A

Manuale Utente

A.1 Installazione dell'applicazione

L'istallazione risulta essere immediata, l'utente deve decomprimere il file zip fornito, al suo interno sono presenti:

- il file *SLD4SM_Restlet.jar*, che rappresenta l'applicazione vera e propria;
- la cartella *SLD4SM_Restlet_lib* che contiene tutte le librerie esterne necessarie al funzionamento dell'applicazione;
- il *config.xml*, che contiene le opzioni di configurazione dell'applicazione, rappresentate dalle credenziali di accesso per i servizi online protetti tramite autenticazione oAuth e dalle credenziali di accesso al database in caso si volessero salvare o estrarre i dati da questo tipo di supporto;

Per iniziare ad utilizzare l'applicazione necessario eseguire il file *SLD4SM_Restlet.jar* ed utilizzare un REST client per inviare le richieste traminte URI HTTP.

Risulta possibile cambiare la configurazione contenuta nel file config.xml, mentre l'applicazione è in stato di running, in modo da poter cambiare le credenziali, potendo così accedere a diversi servizi contemporaneamente.

I componenti rappresentati dal file .jar, dalla cartella delle librerie e dal file config.xml devono sempre essere lasciati nella stessa cartella in modo da non pregiudicare l'accesso ai dati da parte dell'applicazione.

A.2 URI del Servizio Rest

In Tabella A.1 vengono presentate le URI relative al servizio REST offerto da SLD4SM. Tali URI permettono di controllare l'intera infrastruttura software tramite invocazioni HTTP. Le richieste al server devono avvenire attraverso l'utilizzo di client REST in modo da garantire la possibilità di inviare richieste attraverso i metodi del protocollo HTTP. Nel nostro caso specifico abbiamo utilizzato quattro metodi HTTP: GET, POST, PUT, DELETE.

Informazioni	URI	Metodo HTTP
RS: informazioni	http://.../	GET
RS: stop	http://.../	DELETE
A: inizializzazione	http://.../adapter/source/streamname	PUT
RP: inizializzazione	http://.../replayer/streamname/platform/speed	PUT
DG: delete	http://.../streamname	DELETE
DG: informazioni	http://.../streamname	GET
DG: pause	http://.../streamname/pause	POST
DG: restart	http://.../streamname/restart	POST
R: add	http://.../streamname/recorder/platform	PUT
R: delete	http://.../streamname/recorder/platform	DELETE
R: pause	http://.../streamname/recorder/pause/platform	POST
R: restart	http://.../streamname/recorder/restart/platform	POST
P: add	http://.../streamname/publisher/attach/platform/output	PUT
P: delete	http://.../streamname/publisher/platform	DELETE
P: pause	http://.../streamname/publisher/pause/platform	POST
P: restart	http://.../streamname/publisher/restart/platform	POST
W: add	http://.../streamname/windower/type/dimension/step/expiretime	PUT
W: delete	http://.../streamname/windower/type/dimension/step/expiretime	DELETE
W: request sGraph	http://.../streamname/windower/type/dimension/step/expiretime	GET
W: request iGraph	http://.../streamname/windower/type/dimension/step/expiretime/igraph/graphname	GET
W: pause	http://.../streamname/windower/type/dimension/step/expiretime	POST
W: restart	http://.../streamname/windower/type/dimension/step/expiretime	POST
Q: register	http://.../streamname/registerquery	GET

Tabella A.1: URI e relativi metodi d'invocazione a cui risponde il servizio
Legenda: RS: RestServer, A: Adapter, RP: Replayer, DG: DataGetter, R: Recorder, P: Publisher, W: Windower

Appendice B

Esempi e casi d'uso significativi

Iniziamo la presentazione di esempi significativi di utilizzo dell'applicazione con alcune precisazioni, il server viene fatto girare in locale sulla porta 8182 e per utilizzare il servizio REST è necessario utilizzare un REST Client, disponibile ormai per ogni tipo di browser, che consente di inviare specifiche richieste al server stesso, rispettando i comandi del protocollo HTTP.

Esempio 1

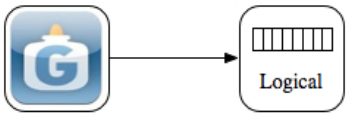

Figura B.1: Schema Esempio 1

1. Inizializzazione di un Adapter:

   `PUT`

   `http://localhost:8182/adapter/glue/glueAdapter`

   In questo caso viene inizializzato un `Adapter` che estrae i dati dal social network GetGlue e viene chiamato glueAdapter, in modo da renderlo identificabile in seguito.

2. Collegamento di un `Windower` logico a glueAdapter

   `PUT`

   `http://localhost:8182/glueAdapter/windower/logical/20000/13000/30000`

   Attraverso questa chiamata viene collegato un windower logico a glueAdapter specificando grandezza delle finestre, step ed expire time che lo caratterizzano (specificati in millisecondi). Nel nostro caso avremo dimensione finestra 20 sec, step 13 sec e expire time di 30 sec.

3. Recupero sGraph

   `GET`

   `http://localhost:8182/glueAdapter/windower/logical/20000/13000/30000`

4. Recupero di uno o più iGraph

   `GET`

   `http://localhost:8182/glueAdapter/windower/logical/20000/13000/30000/igraph/nomeiGraph`

5. Mettere in pausa glueAdapter

   `POST`

   `http://localhost:8182/glueAdapter/pause`

6. Recupero sGraph

   `GET`

   `http://localhost:8182/glueAdapter/windower/logical/20000/13000/30000`

Attraverso questa chiamata è possibile notare che la finestra logica si svuota, visto che il fornitore di dati è in pausa.

7. Rimettere in funzione glueAdapter

   `POST`

   `http://localhost:8182/glueAdapter/pause`

8. Recupero sGraph

   `GET`

   `http://localhost:8182/glueAdapter/windower/logical/20000/13000/30000`

   Attraverso questa chiamata è possibile notare che la finestra logica ricomincia a riempirsi con l'arrivo di nuovi dati.

9. Eliminare il `Windower` logico

   `DELETE`

   `http://localhost:8182/glueAdapter/windower/logical/20000/13000/30000`

10. Eliminare glueAdapter

    `DELETE`

    `http://localhost:8182/glueAdapter`

Esempio 2

Nell'esempio 2, che ruota intorno all'inizializzazione di un oggetto di tipo `ReplayerFromDisk`, è necessario avere dei dati precedentemente salvati, e posizionati all'interno della cartella Salvataggi.

1. Inizializzazione di un `Replayer`.

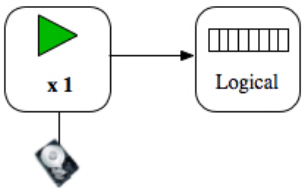

Figura B.2: Schema Esempio 2

`PUT`

`http://localhost:8182/replayer/Glue26Aprile/disk/1`

Attraverso questa chiamata viene inizializzato un `Replayer` di dati glue salvati precedentemente su disco e viene chiamato Glue26Aprile, è necessario inoltre specificare la velocità di ripetizione dello stream, nel nostro caso con 1 viene indicata la velocità originale.

2. Collegamento di un `Windower` logico a Glue26Aprile.

   `PUT`

   `http://localhost:8182/Glue26Aprile/windower/logical/20000/13000/30000`

   Attraverso questa chiamata viene collegato un `Windower` logico a Glue26Aprile specificando grandezza delle finestre, step ed expire time che lo caratterizzano (specificati in millisecondi). Nel nostro caso avremo dimensione finestra 20 sec, step 13 sec e expire time di 30 sec.

3. Recupero sGraph

   `GET`

   `http://localhost:8182/Glue26Aprile/windower/logical/20000/13000/30000`

4. Recupero di uno o più iGraph

`GET`

`http://localhost:8182/Glue26Aprile/windower/logical/20000/13000/30000/igraph/nomeiGraph`

5. Mettere in pausa Glue26Aprile

   `POST`

   `http://localhost:8182/Glue26Aprile/pause`

6. Recupero sGraph

   `GET`

   `http://localhost:8182/Glue26Aprile/windower/logical/20000/13000/30000`

   Attraverso questa chiamata è possibile notare che la finestra logica si svuota, visto che il fornitore di dati è in pausa.

7. Rimettere in funzione il `Replayer`.

   `POST`

   `http://localhost:8182/Glue26Aprile/pause`

8. Recupero sGraph

   `GET`

   `http://localhost:8182/Glue26Aprile/windower/logical/20000/13000/30000`

   Attraverso questa chiamata è possibile notare che la finestra logica ricomincia a riempirsi con l'arrivo di nuovi dati.

9. Eliminare il `Windower` logico

```
DELETE

http://localhost:8182/Glue26Aprile/windower/logical/
20000/13000/30000
```

10. Eliminare Glue26Aprile

```
DELETE

http://localhost:8182/Glue26Aprile
```

Esempio 3

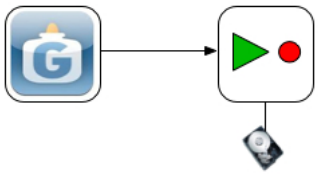

Figura B.3: Schema Esempio 3

In questo esempio viene presentata la funzione di registrazione su disco di dati provenienti da un `Adapter` di GetGlue.

1. Inizializzazione di un `Adapter`:

   ```
   PUT

   http://localhost:8182/adapter/glue/glueAdapter
   ```

   In questo caso viene inizializzato un `Adapter` che estrae i dati dal social network GetGlue e viene chiamato glueAdapter, in modo da renderlo identificabile in seguito.

2. Collegamento di un `Recorder` su disco a glueAdapter

   ```
   PUT

   http://localhost:8182/glueAdapter/recorder/disk
   ```

Attraverso questa chiamata viene collegato un `Recorder` a glueAdapter specificando la piattaforma su cui salvare i dati.

3. Mettere in pausa il `Recorder`.

   `POST`

   `http://localhost:8182/glueAdapter/recorder/pause/disk`

4. Rimettere in funzione il `Recorder`

   `POST`

   `http://localhost:8182/glueAdapter/recorder/restart/disk`

5. Eliminazione del `Recorder`

   `DELETE`

   `http://localhost:8182/glueAdapter/recorder/disk`

6. Eliminazione glueAdapter

   `DELETE`

   `http://localhost:8182/glueAdapter`

Esempio 4

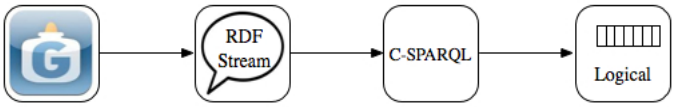

Figura B.4: Schema Esempio 4

In questo esempio viene mostrato l'utilizzo del `C-SPARQL Adapter`, utilizzato per manipolare i dati estratti tramite una query su uno stream RDF di dati e presentati tramite `Windower` logico all'utente.

1. Inizializzazione di un `Adapter`:

   `PUT`

   `http://localhost:8182/adapter/glue/glueAdapter`

   In questo caso viene inizializzato un `Adapter` che estrae i dati dal social network GetGlue e viene chiamato glueAdapter, in modo da renderlo identificabile in seguito

2. Collegamento di un `Publisher` su RDF stream a glueAdapter

   `PUT`
   `http://localhost:8182/glueAdapter/publisher/`
   `attach/stream/glue`

   Attraverso questa chiamata viene collegato un `Publisher` su RDF stream a glueAdapter che, nel nostro caso, pubblicherà i dati sullo stream chiamato http://ex.org/glue/

3. Registrazione della query

   `POST`

   `http://localhost:8182/glueAdapter/registerquery`

   I parametri da passare nel body della post sono due:

   - outputstream = prova
   - query = REGISTER+QUERY + PIPPO+AS+SELECT+ %3FS+%3FP+%3FO+FROM+STREAM+%3Chttp%3A %2F%2Fex.org%2Fglue%2F%3E + %5BRANGE + TRIPLES + 10%5D + WHERE + %7B + %3FS + %3FP+ %3FO+%7D

   Nell'esempio il `C-SPARQL Adapter` che manipolerà i dati in uscita si chiamerà prova e verrà registrata sul motore C-SPARQL la query REGISTER QUERY PIPPO AS SELECT ?S ?P ?O FROM STREAM <http://ex.org/glue/> [RANGE TRIPLES 10] WHERE ?S ?P ?O , che verrà passata nella form dopo averla sottoposta a un URL encoding.

4. Associazione di un `Windower` al nuovo datagetter prova

   `PUT`

   `http://localhost:8182/prova/windower/logical/20000/13000/30000`

5. Richiesta sGraph

   `GET`

   `http://localhost:8182/prova/windower/logical/20000/13000/30000`

6. Richiesta iGraph

   `GET`

   `http://localhost:8182/prova/windower/logical/20000/13000/30000/igraph/nome iGraph`

7. Eliminare il `Windower` da prova

   `DELETE`

   `http://localhost:8182/prova/windower/logical/20000/13000/30000`

8. Eliminare il `C-SPARQL Adapter` prova

   `DELETE`

   `http://localhost:8182/prova`

9. Eliminare il `Publisher` su stream legato a glueAdapter

`DELETE`

`http://localhost:8182/glueMaggio/publisher/stream`

10. Eliminare glueAdapter

    `DELETE`

    `http://localhost:8182/glueAdapter`

Esempio 5

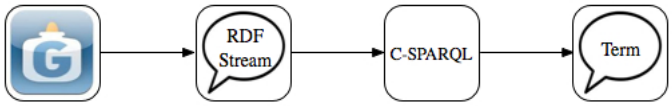

Figura B.5: Schema Esempio 5

In questo esempio viene mostrato l'utilizzo del `C-SPARQL Adapter` utilizzato per manipolare i dati estratti tramite una query su uno stream RDF di dati e presentati tramite publisher su console di controllo.

1. Inizializzazione di un `Adapter`:

   `PUT`

   `http://localhost:8182/adapter/glue/glueAdapter`

   In questo caso viene inizializzato un `Adapter` che estrae i dati dal social network GetGlue e viene chiamato glueAdapter, in modo da renderlo identificabile in seguito

2. Collegamento di un `Publisher` su RDF stream a glueAdapter

   `PUT`

   `http://localhost:8182/glueAdapter/publisher/`
   `attach/stream/glue`

Attraverso questa chiamata viene collegato un `Publisher` su RDF stream a glueAdapter che, nel nostro caso, pubblicherà i dati sullo stream chiamato http://ex.org/glue/

3. Registrazione della query

   `POST`

   `http://localhost:8182/glueAdapter/registerquery`

   I parametri da passare nel body della post sono due:

   - outputstream = prova
   - query = REGISTER+QUERY + PIPPO+AS+SELECT+ %3FS+%3FP+%3FO+FROM+STREAM+%3Chttp%3A %2F%2Fex.org%2Fglue%2F%3E + %5BRANGE + TRIPLES + 10%5D + WHERE + %7B + %3FS + %3FP+ %3FO+%7D

   Nell'esempio il `C-SPARQL Adapter` che manipolerà i dati in uscita si chiamerà prova e verrà registrata sul motore C-SPARQL la query REGISTER QUERY PIPPO AS SELECT ?S ?P ?O FROM STREAM <http://ex.org/glue/> [RANGE TRIPLES 10] WHERE ?S ?P ?O , che verrà passata nella form dopo averla sottoposta a un URL encoding.

4. Attaccare un `Publisher` su console al nuovo datagetter prova

   `PUT`

   `http://localhost:8182/prova/publisher/attach /console/abc`

5. Eliminare `Publisher` su console da prova

   `DELETE`

   `http://localhost:8182/prova/publisher/console`

6. Eliminare il `C-SPARQL Adapter` prova

`DELETE`

`http://localhost:8182/prova`

7. Eliminare il `Publisher` su stream legato a glueAdapter

   `DELETE`

   `http://localhost:8182/glueMaggio/publisher/stream`

8. Eliminare glueAdapter

   `DELETE`

   `http://localhost:8182/glueAdapter`

Bibliografia

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, January 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik. Aurora: a data stream management system. *SIGMOD Conference*, 2003.

[3] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec. 30(3)*, September 2001.

[4] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. C-sparql: a continuous query language for rdf data streams. *Int. J. Semantic Computing 4(1): 3-25*, 2010.

[5] D. F. Barbieri and E. D. Valle. A proposal for publishing data streams as linked data - a position paper. 2010.

[6] T. Barners-Lee, J. Hendler, and O. Lassila. Web semantico. quando internet diventa intelligente: agenti software e rappresentazioni condivise per l'automazione dei servizi in rete... e a casa vostra. *Le Scienze: 76-84*, May 2001.

[7] I. Celino, D. Dell'Aglio, E. D. Valle, Y. Huang, T. Lee, S. Park, and V. Tresp. Making sense of location based micro-posts using stream reasoning. pages 13–18, 2011.

[8] J. Chen, D. DeWitt, F. Tian, and Y. Wang. Nigaracq: A scalable continuous query system for internet databases. *ACM SIGMOD*, 2000.

[9] L. Golab and O. Tamer. *Data Stream Management*. Morgan Claypool Publishers, 2010.

[10] H. V. Jagadish, I. Sigh, and A. Silberschatz. View maintenance issues for the chronicle data model. *14th PODS: 113-124*, 1995.

[11] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering archive, Volume 11 - Issue 4 : 610-628*, July 1999.

[12] E. D. Valle, S. Ceri, D. F. Barbieri, D. Braga, and A. Campi. A first step towards stream reasoning. *FIS : 72-81*, 2008.

[13] E. D. Valle, S. Ceri, F. V. Harmelen, and D. Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems 24(6): 83-89*, November/December 2009.