POLITECNICO DI MILANO
Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Traduzione automatica di interrogazioni Continuous-SPARQL su stream RDF in EPL e SPARQL

Relatore: Ing. Emanuele Della Valle

Correlatore: Ing. Davide Francesco Barbieri
Correlatore: Ing. Daniele Braga

Laureando:
Marco Regaldo Matr. 680108

Anno Accademico 2009-2010

guardo la luna:
nuvole se alzo gli occhi, se li abbasso
il sereno

mireba kumori
mineba hareyuku
tsukimi kana

Miura Chora

Sommario

Alla fine degli anni '90, la comunità delle basi di dati pose la sua attenzione sugli stream di dati e pose le basi della ricerca sui Data Stream Managment Systems (DSMS). Oggi, dopo un decennio di ricerca accademica e numerose startup HiTech,i DSMS si possono dire una realtà affermata. Al contrario la comunità del Semantic Web sembra aver quasi ignorato la dinamicità del Web e, soprattutto, il crescente uso di stream di dati. Continuous-SPARQL (o semplicemente C-SPARQL) è un primo tentativo di applicare tecniche tipiche dei DSMS al Semantic Web. C-SPARQL estende SPARQL con un nuovo tipo di dato, lo stream RDF, e con la possibilità di registrare query continue in un apposito motore di esecuzione di interrogazioni C-SPARQL. Tale motore, sviluppato al Politecnico di Milano, è composto da un DSMS e da un motore SPARQL.

Oggetto di questa tesi è stato quello di progettare, implementare e validare un traduttore automatico che consentisse verificare la correttezza sintattica e semantica di un'interrogazione C-SPARQL e, poi, di generare due due interrogazioni: una per il DSMS e una per il motore SPARQL. Per verificare la correttezza sintattica e semantica di un'interrogazione C-SPARQL si è definita la grammatica C-SPARQL e si è proceduto a generarne il parser utilizzando ANTLR. Per quanto riguarda, poi, la traduzione, sono stati progettati e implementati due traduttori che recuperano le informazioni necessarie dall'albero sintattico prodotto dal parser a partire dall'interrogazione C-SPARQL.

L'approccio scelto è caratterizzato da un'alta flessibilità: consente di trattare i due aspetti di C-SPARQL separatamente, permette di gestire l'evoluzione dei due sottocomponenti di un motore C-SPARQL in modo indipendente, abilita la sostituzione di Esper con un altro DSMS e l'utilizzo di motori SPARQL non standard. Non da ultimo, permetterà di sviluppare ottimizzatori di interrogazioni C-SPARQL che sfruttino la diversa selettività degli operatori C-SPARQL e l'intrinseco parallelismo degli stream di dati.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Il contesto: stream reasoning e web semantico

Negli ultimi anni si è assistito all'affermarsi di concetti di collaborazione e condivisione, soprattutto sul web grazie a tecnologie come XML, feeds e tag, quest'ultima con lo scopo di permettere la catalogazione di una crescente mole di informazioni che vede nell'essere ricercata correttamente uno dei suoi principali problemi. Infatti i tag sono delle semplici parole non contestualizzate che vengono associate ai documenti. Il problema sta proprio nel fatto che essendo prive di contesto, possono generare rumore nella selezione dei risultati.

Ed è proprio per superare i limiti intrinsechi dei tag che è nata un'altra tecnologia, Resource Description Framework (RDF), che permetteo di dare un preciso significato ai tag facendoli diventare delle proprietà (es. nome di un autore o data di documento). Sono nati quindi sistemi ormai abbastanza diffusi di interrogazione e reasoning su dati RDF statici 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 hanno dato un notevole impulso al web semantico.

Parallelamente, si sono affermati i sistemi DSMS (Data Stream Management Systems) che eseguono su stream di dati delle query preventivamente registrate. Il flusso di dati è continuo e per questo non è archiviabile nel suo complesso. Tipologie di dati simili possono essere quelli provenienti da reti di sensori, "Urban Computing, quotazioni borsa ecc. Questi sistemi sono intrinsecamente molto efficienti nell'elaborazione di dati temporanei in continuo mutamento.

Da qui l'idea di combinare queste due tecnologie a formare un nuovo concetto, "stream reasoning", che prevede la possibilità per i reasoner di operare su dati in rapido mutamento oltre alle solite sorgenti di dati statiche, funzionalità questa per ora negata alla comunità del web semantico. Si è voluto così colmare le lacune di entrambe le tecnologie unendone i vantag-

gi. Il nuovo elemento su cui si eseguono le interrogazioni è *RDF Stream*. Per eseguire le query su questo nuovo tipo di dati se è sviluppato un linguaggio, C-SPARQL, che è un estensione di SPARQL progettata per esprimere interrogazioni che vengono registrate ed eseguite continuamente su repository RDF e *RDF streams*. C-SPARQL permette l'esecuzione di query del tipo: "Quante auto stanno continuamente entrando nel centro città?".

È stato quindi il linguaggio C-SPARQL il punto di partenza dl lavoro descritto in questo documento.

1.2 Il progetto LarKC e il linguaggio C-SPARQL

A seguito della definizione del nuovo linguaggio C-SPARQL, un gruppo di ricerca del politecnico di milano ha iniziato all'interno del progetto della comunità europea LarKC (The Large Knowledge Collider) a progettare una possibile architettura che potesse fungere da engine di esecuzione per le query C-SPARQL. Come mostrato in figura (1.1) l'architettura ipotizzata prevede un componente di front-end che fornisce le funzionalità di registrazione ed esecuzione continua delle query. Per assolvere a questi compiti si appoggia, a due sottocomponenti: un motore DSMS di esecuzione di query continue e uno per l'esecuzione di query SPARQL. L'idea dietro a questa struttura è stata quella di sfruttare con un approccio plug-in gli engine già affermatisi sul mercato che forniscono le necessarie garanzie di affidabilità ed efficacia.

A fornire gli input corretti a questi due motori di esecuzione ci pensa il *Query Transaltor* che ha il compito il compito da un lato di produrre una query continua che possa essere eseguita dal *Continuous Engine* e dall'altro ricavare la query SPARQL che possa essere eseguita dal rispettivo engine. Utilizzando in sequenza i risultati di queste due esecuzioni separate il motore C-SPARQL è in grado prima di estrarre dagli RDF Sreams delle istantanee di tripe RDF corredate da timestamp e poi di eseguirvi sopra delle normali query SPARQL.

1.3 Descrizione del problema affrontato

La tesi ha preso corpo all'interno del costesto di Larkc e nello specifico ha riguardato la progettazione e lo sviluppo del componente *Query Transaltor*. Il suo ruolo è duplice rispetto all'operazione di traduzione.

Innanzitutto deve verificare che la query C-SPARQL fornita in ingresso sia effettivamente scritta in quel linguaggio, ovvero sia sintatticamente e semanticamente corretta. Verrà di seguito descritto (cap 4) il modo con cui sia stato affrontato questo problema che è stato infine risolto con la stesura di una grammatica formale per C-SPARQL a partire da quella di SPARQL e la generazione da essa di un parser. Grazie al parser si è in grado di ottenere

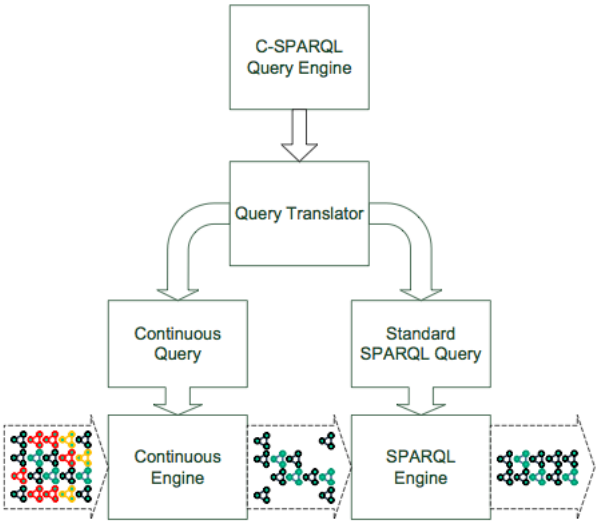

Figura 1.1: Architettura C-SPARQL engine

un albero sintattico dell'interrogazione se si rivela aderente alle specifiche del linguaggio.

In un secondo momento, a partire da questo albero sintattico, comincia il lavoro di traduzione vero e proprio che genera le rispettive query SPARQL ed EPL[1]. Questa operazione di traduzione duplice è stata progettata e implementata, per motivazioni di flessibilità illustrati nel corso del documento, tramite due traduttori separati.

1.4 Struttura della tesi

Il documento è strutturato secondo i seguenti capitoli, di cui i primi tre sono di carattere introduttivo mentre i seguenti sono relativi alla progettazione e all'implementazione del componente *Query Transaltor*, oggetto della tesi (si noti che una descrizione introduttiva sullo stato dell'arte di ogni tecnologia è incluso nella parte iniziale del rispettivo capitolo):

- il capitolo 1 descrive il contesto delle tecnologie che ruotano intorno a questo progetto, in particolare SPARQL quale linguaggio standard per l'esecuzione di interrogazioni su dati di tipo RDF e la sua evoluzione C-SPARQL che permette l'esecuzione di query su RDF Streams, inserendo di fatto nel mondo del web semantico l'idea di sream di dati già presente nei sistemi DSMS;

[1]Come descritto ampiamente nei capitoli che seguono (3.3) è questo il linguaggio scelto per implementare le query continue sugli stream RDF

- il capitolo 2 presenta un'approfondita analisi dei due linguaggi SPARQL e C-SPARQL, analizzandone le caratteristiche, la sintassi e le motivazioni che hanno portato alla creazione di questo nuovo linguaggio;
- il capitolo 3 parla invece dell'altra faccia della medaglia, ovvero dei *Data Stream Management Systems*, quali strumenti di esecuzione di query continue su stream di dati;
- il capitolo 4 inizia a parlare dell'oggetto di questa tesi, con la progettazione del parser e la preliminare stesura della grammatica del linguaggio C-SPARQL;
- il capitolo 5 invece si occupa della progettazione delle classi del traduttore, che utilizza il parser per ottenere l'input necessario a generare le interrogazioni SPARQL ed EPL a partire da una query C-SPARQL;
- il capitolo 6 descrive alcune caratteristiche salienti dell'implementazione del traduttore (tra cui citiamo il tool che ha reso possibile la generazione automatica del parser a partire dalla sua grammatica, ANTLR, e la scelta di utilizzare EPL come linguaggio per le query continue), il tutto corredato da alcuni diagrammi UML che ne descrivono la sua struttura a livello di classi;
- il capitolo 7 mostra una serie dettagliata di esempi che vanno a sollecitare tutte le caratteristiche e i componenti del traduttore sviluppato in questo lavoro di tesi;
- il capitolo 8 trae le conclusioni del lavoro svolto, descrivendo quanto è stato ottenuto e quali sono i suoi possibili sviluppi futuri;
- nell'appendice, infine, viene riportata la grammatica completa C-SPARQL prodotta aggiornata alla versione 1.1 del linguaggio SPARQL.

Capitolo 2

Sparql / C-Sparql

2.1 SPARQL

È un linguaggio di query su dati RDF e un protocollo di invio sul web di interrogazioni e risultati. La sigla è acronimo di Simple Protocol And Rdf Query Language.

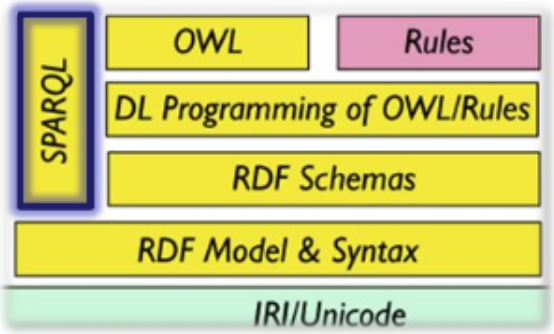

Figura 2.1: Architettura semantic web

2.1.1 Origini e motivazioni

L'invasione dei concetti di collaborazione e condivisione, che si sono affermati grazie anche a tecnologie chiave come l' XML, rendono il web 2.0 una realtà consolidata. Myspace, Facebook, Youtube, Wikipedia, i blog e gli aggregatori, sono solo alcuni esempi di successo.

Ora che tutti possono contribuire con estrema rapidità e facilità alla conoscenza, si è evidenziato il problema di tenere organizzate queste informazioni, e lo si sta già facendo: servizi come del.icio.us, ovvero di social bookmarking, servono esattamente a quello e allo scopo sono appositamente stati inventati i tag, che permettono un' indicizzazione più elastica dei post o degli articoli.

L' XML ha permesso di creare strumenti come i feeds, che permettono di separare il contenuto (la conoscenza) dal contenitore (un sito web, che sia un blog o qualsiasi altra cosa). I lettori di feeds, permettono di aggregare tutti gli articoli e le notizie pubblicate dalle fonti che più interessano e delle quali ci si fida, mentre i tag filtrano i documenti, alla ricerca di quello che interessa. Tutto questo è fantastico ma ha dei limiti, che sono stati risolti tramite le tecnologie di seguito discusse.

RDF è un acronimo che sta per Resource Description Framework, vale a dire un metodo per descrivere efficacemente i contenuti, del web e non. L' evoluzione della catalogazione delle risorse ha portato all'abbandondono delle categorie (che concettualmente sono uguali alle cartelle di Windows) in favore dei tag, che sono più elastici dato che permettono di raggruppare un' informazione sotto diversi nomi quando è il caso. Qual' è però il limite dei tag?

I tag sono semplici parole non contestualizzate che se usate per reperire documenti da più fonti possono generare molti risultati inutili (rumore) poiché ogni documento può intendere la stessa parola in un contesto diverso. Inoltre attribuendo numerosi tag ad ogni documento si tende ad aumentare il volume dei risultati per ogni tag richiamato.

L' RDF aggiunge un significato preciso ai tag e amplia le possibilità di organizzazione della conoscenza ben oltre le capacità di questi. I tag si trasformano in proprietà, che a loro volta possono assumere un valore. Una proprietà potrebbe essere il nome dell' autore, la data in cui il documento è stato scritto, la lingua, la materia trattata e infinite altre cose. Non c' è un certo numero di proprietà predefinite da rispettare, ma ognuno è libero di creare un struttura in base alle proprie esigenze. C' è da aspettarsi però che in futuro verranno identificati delle regole standard per rendere le realtà del web interoperabili tra loro e per fare in modo che possano essere indicizzate da appositi servizi, al pari dei lettori di feed che usiamo oggi.

Qui entra in gioco SPARQL, che di recente è stato approvato dal W3C e quindi può ora essere definito uno standard per il web. L' RDF serve solo alla definizione della semantica, ma da solo non può far molto. Si deve poter effettuare delle ricerche all' interno tra le proprietà di tutti i documenti indicizzati, e per fare questo (e fare in modo che in futuro questo compito potrà essere svolto da un calcolatore) serve un linguaggio creato appositamente allo scopo.

Grazie a SPARQL si potranno creare ad esempio software e servizi web, che come gli attuali lettori di feed, aggregano le notizie che vogliamo, ma in modo molto più evoluto, mirato ed efficace. Ci permetteranno, continuando sulla stessa linea di esempi, di cercare gli articoli scritti nei siti web in lingua italiana, scritti da una determinata persona, in un determinato periodo, che parlano di una determinato argomento.

2.1.2 Ontologia

Una definizione largamente citata di ontologia è : *"Un'ontologia è una specificazione di una concettualizzazione"(Gruber, T. 1993)*

Più semplicemente un'ontologia permette di specificare, in modo aperto e significativo, i concetti e le relazioni che caratterizzano un certo dominio di conoscenza. Ci sono differenti modi di scrivere una ontologia e molte opinioni su quali tipi di definizione vadano inserite dentro di essa. Tuttavia forma e contenuto di un'ontologia sono fortemente guidati dal tipo di applicazione per la quale sarà utilizzata. Poiché siamo interessati alle ontologie costruite per lo sviluppo del Web semantico, verranno illustrati i formalismi costruiti sul linguaggio RDF/XML. Una ontologia è rappresentata quindi da un documento che descriva una collezione di oggetti rdf e sia pensato per poter essere riusato e ampliato. Ad esempio la dichiarazione rdf `s:Focus p:manufacturedBy o:Ford` può essere stata generata da un ontologia riguardante un produttore di automobili. In generale, le ontologie tendono ad essere o molto specifiche nel loro dominio o molto generali (come nel caso di Word-net ad esempio). Il primo linguaggio basato su RDF/XML per specificare le ontologie per il web semantico è stato RDF Schema, evolutosi poi in OWL (Web Ontology Language).

2.1.3 RDF

Il cuore del semantic web come si è detto nella sezione precedente è il modello RDF. ll Resource Description Framework (RDF) è lo strumento base proposto da W3C per la codifica, lo scambio e il riutilizzo di metadati strutturati e consente l'interoperabilità tra applicazioni che si scambiano informazioni sul Web. È costituito da due componenti:

- RDF Model and Syntax: espone la struttura del modello RDF, e descrive una possibile sintassi.
- RDF Schema: espone la sintassi per definire schemi e vocabolari per i metadati.

Qualunque cosa descritta da RDF è detta risorsa. Principalmente una risorsa è reperibile sul web, ma RDF può descrivere anche risorse che non si trovano direttamente sul web. Ogni risorsa è identificata da un URI, un identificatore univoco di risorse. Il modello di dati RDF è formato da risorse, proprietà e valori. Le proprietà sono delle relazioni che legano tra loro risorse e valori, e sono anch'esse identificate da URI. Un valore, invece, è un tipo di dato primitivo, che può essere anche una stringa contenente l'URI di una risorsa. L'unità base per rappresentare un'informazione in RDF è lo *statement*. Uno statement è una tripla del tipo: *Soggetto - Predicato - Oggetto* dove il soggetto è una risorsa, il predicato è una proprietà e l'oggetto è un valore (es. `s:myCar p:hasColour o:Blue`). È possibile poi collegare

statements riguardanti la stessa URI per costruire statements più complesse come: `s:myCar p:is o:Focus, s:Focus p:manufacturedBy o:Ford`, che può essere tradotta con una frase del tipo: "La mia auto è una Focus, la Focus viene prodotta dalla Ford".

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

La rappresentazione fisica del modello è quella di un grafo orientato in cui i nodi rappresentano le risorse o i tipi primitivi mentre gli archi rappresentano le proprietà. Una delle peculiarità di questo modello è che una tripla mancante non è significativa, in quanto indica soltanto che l'informazione non è conosciuta. Seguendo gli esempi sopra riportati, alla domanda: "quante auto si possiede"la risposta è "almeno 1". Questo è nota come assunzione di un open-world, opposta a quella closed world usata nei modelli relazionali e ad oggetti, in cui la risposta sarebbe stata: "esattamente 1".

Sintassi RDF

Il modello RDF è indipendente dalla particolare sintassi usata per esprimerlo ed esistono molti tools e librerie per convertire semplicemente da una sintassi ad un altra. Un grafo RDF è rappresentato fisicamente mediante una serializzazione. Le principali serializzazioni adottabili per un grafo RDF sono:

- XML: l'RDF è serializzato in un file XML. È stato definito principalmente per il trasferimento macchina-macchina in quanto per l'uomo è di difficile lettura, come di seguito illustrato.

```
Es. "Mario_Rossi" è"_autore_di" "Rosso_di_sera_bel_tempo_si_spera"
     si serializza in RDF/XML :

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:au="http://description.org/schema/">
<rdf:Description about="http://www.book.it/
    Rosso_di_sera_bel_tempo_si_spera/">
  <au:author>Mario_Rossi</au:author>
</rdf:Description>
</rdf:RDF>
```

- N-TRIPLE: si serializza il grafo come un insieme di triple soggetto - predicato - oggetto

- N3: si serializza il grafo descrivendo, una per volta, una risorsa e tutte le sue proprietà

2.1.4 Scrittura di una semplice query SPARQL

Dopo aver introdotto a cosa serve SPARQL e su che tipologia di dati deve eseguire le interrogazioni, si illustrerà con un breve esempio applicativo del linguaggio per introdurne la sintassi. L'esempio seguente mostra una interrogazione SPARQL per cercare il titolo di un libro all'interno del grafo dei dati disponibili. La query è composta da due parti: la clausola SELECT che identifica le variabili che compariranno nei risultati, e la clausola WHERE che consente di definire il graph pattern di base da usare per il match dei dati. Il graph pattern di questo esempio consiste in singola tripla con una singola variabile (?title) nella posizione di oggetto dello statement RDF.

Dati:

```
<http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title>
     "SPARQL Tutorial" .
```

Query:

```
SELECT ?title
WHERE
{
  <http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title
      > ?title .
}
```

Risultato query:

title
SPARQL Tutorial

2.1.5 Sintassi

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

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

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

Le asserzioni sono espresse in concise sequenze soggetto-predicato-oggetto e delimitate da un punto fermo. @prefix introduce prefissi e namespace; i due punti senza prefisso (seconda riga) definiscono il namespace di default. Gli URI sono inclusi tra parentesi uncitate. I letterali di tipo stringa sono contrassegnati da virgolette.

2.1.5.1 Il triple pattern matching

Le query SPARQL si basano sul meccanismo del pattern matching e in particolare su un costrutto, il triple pattern, che ricalca la configurazione a triple delle asserzioni RDF fornendo un modello flessibile per la ricerca di corrispondenze. Infatti soggetto, predicato e oggetto possono essere delle variabili come nel seguente caso:

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

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

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

Per chiarire, ecco una semplice query di selezione SPARQL:

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

L'analogia con SQL è lampante.

PREFIX dichiara prefissi e namespace.

SELECT definisce le variabili di ricerca da prendere in considerazione nel risultato (nell'esempio: titolo, autore e anno).

FROM specifica il set di dati su cui dovrà operare la query (si suppone che le triple siano immagazzinate presso l'indirizzo fittizio http://cd.com/listacd.ttl).

È inoltre possibile ricorrere a clausole FROM NAMED e alla parola chiave GRAPH per specificare più set di dati: per dettagli circa questa utilissima funzionalità, si vedano le sezioni 7. RDF Dataset e 8. Querying Dataset della specifica SPARQL Query Language for RDF. [link esterno]

La clausola WHERE , infine, definisce il criterio di selezione specificando tra parentesi graffe uno o più triple patterns separati da punto fermo.

Applicando la query al set di triple del capitolo precedente, si ottiene il seguente risultato:

titolo	autore	anno
Amber	Autechre	1994

Il binding tra variabili e termini reperiti corrispondenti (in questo caso, un termine per ciascuna variabile) è reso in forma di tabella come un rapporto campo-valore: le righe rappresentano i singoli risultati, le intestazioni di cella rappresentano le variabili definite nella clausola SELECT , le celle i termini associati alle variabili.

2.1.5.2 Restrizioni sui valori

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

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

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

Per l'elenco completo degli operatori supportati da SPARQL si veda la sezione 11.3 Operator Mapping della specifica SPARQL Query Language for RDF [link esterno].

2.1.5.3 Manipolazione del risultato

Come in SQL, è possibile escludere dal risultato i valori duplicati mediante la parola chiave DISTINCT, ad esempio:

```
SELECT DISTINCT ?titolo ?autore
```

Altri costrutti supportati da SPARQL per la manipolazione del risultato sono:

```
ORDER BY DESC(?autore)
LIMIT 10
OFFSET 10
```

L'espressione ORDER BY imposta l'ordine dei risultati della query: stando all'esempio, i risultati verranno presentati in ordine decrescente (DESC) in base alla variabile ?autore.

LIMIT pone restrizioni al numero dei risultati, limitandoli, secondo quanto indicato nell'esempio, ai soli primi 10.

OFFSET permette di saltare un certo numero di risultati, escludendo, stando all'esempio, i primi 10.

2.1.6 Descrizione SPARQL 1.1

Con l'ultima estensione del linguaggio sono state introdotte delle novità fortemente richieste dalla comunità scientifica con l'intento di rendere maturo il linguaggio da un punto di vistra espressivo. Di seguito verranno brevemente illustrate queste nuove funzionalità, introdotte durante l'elaborazione della tesi, e che sono poi state aggiunte in corso d'opera al lavoro in modo che fosse aggiornato con le ultime specifiche di SPARQL.

2.1.6.1 Funzioni aggregate

Gli aggregati sono espressioni che si applicano a un gruppo di soluzioni. Come impostazione predefinita, l'insieme delle soluzioni consiste in un singolo gruppo contenente tutte le soluzioni. I raggruppamenti possono però essere specificati attraverso la sintassi della clausola GROUP BY. Le funzioni aggregate introdotte nella versione 1.1 del linguaggio sono: COUNT, SUM, MIN, MAX, AVG.

Esempio aggregazione

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 (SUM(?lprice) > 10)
```

Risultati:

?totalPrice
21

Nelle interrogazioni aggregate e nelle sottoquery solo le espressioni che sono state usate come GROUP BY o come espressioni aggregate (ad esempio dove tutte le variabili compaiono all'interno dell'aggregazione) possono essere proiettate.

2.1.6.2 Sottoquery

Viene introdotta la possibilità di innestare delle query all'interno della clausola WHERE per rendere più potente e flessibile l'espressività delle interrogazioni.

Esempio Sottoquery

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" .
```

Restituisce il nome (quello minore in ordine alfabetico) tra tutte le persone che conoscono Alice e hanno un nome.

Query:

```
PREFIX : <http://people.example/>
PREFIX : <http://people.example/>
SELECT ?y ?minName
WHERE {
  :alice :knows ?y .
  {
    SELECT ?y (MIN(?name) AS ?minName)
    WHERE {
      ?y :name ?name .
    } GROUP BY ?y
  }
}
```

Risultati:

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

2.1.6.3 Negazioni

Il linguaggio ha introdotto un metodo, attraverso il pattern NOT EXIST, per testare se un graph pattern non trova riscontri nell'insieme dei dati, dati valori delle variabili in-scope. In questo modo non vengono generati legami aggiuntivi a variabili. NOT EXIST può essere usato all'interno di graph patterns e in espressioni di tipo FILTER. Da notare che il filtro si applica a tutto il gruppo mentre il pattern NOT EXIST se usato al suo interno si applica solo alle variabili dichiarate precedentemente.

Viene anche fornita una forma con EXIST, sia come pattern che come espressione per FILTER. Serve a testare se il pattern esegue il match dei dati o no e anch'esso non genera ulteriori legami con nuove variabili.

Test dell'assenza di un pattern

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
{
    ?person rdf:type  foaf:Person .
    FILTER NOT EXISTS { ?person foaf:name ?name }
}
```

Risultato query:

person
<http://example/bob>

Test della presenza di un pattern

Query:

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

SELECT ?person
WHERE
{
    ?person rdf:type  foaf:Person .
    FILTER EXISTS { ?person foaf:name ?name }
}
```

Risultato query:

person
<http://example/alice>

2.1.6.4 Espressioni nella clausola SELECT

Con la nuova versione del linguaggio viene introdotta la possibilità, oltre a scegliere quali delle variabili dei pattern di confronto includere nei risultati, di introdurre nuove variabili insieme ad espressioni che forniscano il valore da collegare ad esse. Le espressioni combinano variabili già presenti nella soluzione della query o dichiarate nella clausola SELECT per produrre un

nuovo valore. La nuova variabile è introdotta con la parola chiave AS; la variabile ovviamente non deve già essere stata dichiarata precedentemente.

Example

Dati:

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/book/> .
@prefix ns:   <http://example.org/ns#> .

:book1  dc:title  "SPARQL Tutorial" .
:book1  ns:price  42 .
:book1  ns:discount 0.1 .

:book2  dc:title  "The Semantic Web" .
:book2  ns:price  23 .
:book2  ns:discount 0 .
```

Query:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title (?p*(1-?discount) AS ?price)
   { ?x ns:price ?p .
     ?x dc:title ?title .
     ?x ns:discount ?discount
   }
```

Risultati:

title	price
The Semantic Web	23
SPARQL Tutorial	37.8

Come già descritto, possono essere usate anche variabili precedentemente dichiarate nella stessa clausola SELECT, come nell'esempio seguente:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title (?p AS ?fullPrice) (?fullPrice*(1-?discount) AS ?
    customerPrice)
   { ?x ns:price ?p .
     ?x dc:title ?title .
     ?x ns:discount ?discount
   }
```

Risultato:

title	fullPrice	customerPrice
The Semantic Web	23	23
SPARQL Tutorial	42	37.8

2.2 C-SPARQL

Data Stream Management Systems (DSMS) [11] processano interrogazioni su sorgenti di dati di tipo stream, quali sensori, click streams, stock quotations ecc. Caratteristica dei dati è che arrivano continuamente in tempo reale, ordinati implicitamente in base al tempo di arrivo o esplicitamente grazie al timestamp associato. Per la natura stessa di questi dati sotto forma di flusso continuo risulta non sostenibile una loro memorizzazione integrale, per cui si rende necessario che le queries vengano eseguite continuamente ogni volta che arrivano nuovi dati [12]. Contemporaneamente a questo scenario, i processi di reasoning su grandi collezioni di dati RDF si stanno diffondendo, e SPARQL ha assunto un ruolo di linguaggio standard per le query su questo tipo di dati. Sistemi basati su SPARQL sono ora in grado di interrogare repository integrati e recuperare informazioni da sorgenti multiple. Va considerato però che la maggior parte le basi di conoscenza interrogate da questi sistemi è di tipo statico (come ad esempio Linked Life Data[1]) e l'evoluzione degli stessi non è supportata adeguatamente.

La combinazione di dati statici RDF con informazioni di tipo streaming porta ad un nuovo concetto, **stream reasoning**, che prevede la possibilità per i reasoner di operare su dati in rapido mutamento oltre alle solite sorgenti di dati statiche, funzionalità questa per ora negata alla cominuta del web semantico. C-SPARQL è un estensione di SPARQL progettata per esprimere interrogazioni che vengono registrate ed eseguite continuamente su repository RDF e *RDF streams*. C-SPARQL è in grado di eseguire interrogazioni del tipo "Quante auto stanno continuamente entrando nel centro città?". Queste interrogazioni possono essere poi considerati gli input di sistemi reasoner specializzati per applicazioni di "Urban Computing, in grado dai dati di affluenza di decidere appropriate politiche di gestione della viabilità. Questi reasoner di fatto lavorano su delle istantanee dei dati che devono continuamente essere aggioranti da interrogazioni continue. I reasoner in questo caso sono completamente all'oscuro del trascorrere del tempo e del fatto che le sorgenti dei loro dati sono di tipo stream.

Visto che i DSMS e i sistemi basati su SPARQL si sono affermati saldamente una decina di anni di ricerca e sviluppo, si è deciso di riusare queste tecnologie per supportare C-SPARQL. Tuttavia, l'integrazione di sistemi SPARQL e DSMS non è semplice in quanto richiede una scomposizione automatica e una trasformazione delle interrogazioni C-SPARQL in input accettabili per i due tipi di sistemi. Per risolvere questa sfida si è sviluppato un ambiente di esecuzione per query C-SPARQL costruito sopra DSMS relazionali e motori di esecuzione SPARQL usando un approccio plug-in che garantisse estensibilità, portabilità e buone prestazioni.

[1] http://www.linkedlifedata.com/

Grazie ad una precisa caratterizzazione della semantica di C-SPARQL, sono state mappate le interrogazioni C-SPARQL su un modello interno. Vengono usati dei metodi di trasformazione in modo da generare delle interrogazioni che distribuiscano il lavoro tra DSMS e motori di esecuzione SPARQL. Le trasformazioni sono ispirate dalle classiche ottimizzazioni dell'algebra relazionale che vengono eseguite prima della scomposizione della interrogazione. Dopo la trasformazione, l'interrogazione ottiene un risultato frutto dell'orchestrazione dei sistemi DSMS e SPARQL, ognuno dei quali può operare ottimizzazioni a livello di singola query o di gruppo.

Verranno di seguito presentate alcune delle nuove funzionalità di C-SPARQL presenti rispetto a SPARQL quali RDF stream data type, gestione della finestra temporale e timestamps; le loro sintassi, ottenute aggiungendo nuove produzione alla grammatica standard di SPARQL [23] sono accompagnate da esempi di utilizzo riguardanti Urban Computing.

2.2.1 RDF Stream Data Type

C-SPARQL aggiunge **RDF streams** ai tipi di dati suportati da SPARQL.[2] Ogni stream RDF è definito come una sequenza di coppie ordinate composte da una tripla RDF e dal suo timestamp τ:

$$\begin{gathered}\ldots \\ (\langle subj_i, pred_i, obj_i\rangle\ ,\ \tau_i) \\ (\langle subj_{i+1}, pred_{i+1}, obj_{i+1}\rangle\ ,\ \tau_{i+1}) \\ \ldots\end{gathered}$$

I timestamps possono essere considerati come *annotations* delle triple RDF; sono monotonicamente non decrescenti all'interno dello stream ($\tau_i \leq \tau_{i+1}$). Sono non strettamente crescenti perchè i timestamps non devono essere necessariamente unici. Può quindi essere che ci sia un numero grande ma finito di triple consecutive con lo stesso timestamp, che si verificano tutte nello stesso momento, per quanto siano ordinate all'interno dello stream secondo una sequenza posizionale.

Esempio. Nel nostro esempio, preso da uno scenario di Urban Computing, gli streams di dati sono associati ai caselli autostradali. In questi stream ogni tripla corrisponde ad un auto che passa attraverso il casello. 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. Questo setup 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 bindings delle variabili relative alle triple degli streams. Un esempio di stream con cinque auto che passano attraverso tre differenti caselli è fornito di seguito.

[2] Allo stesso modo, il tipo di dato stream è stato introdotto per estendere le relazioni nei DSMS.

triple	Timestamp
c:Distr1 t:registers 156	t_{100}
c:Distr2 t:registers 75	t_{101}
c:Distr1 t:registers 130	t_{102}
c:Distr2 t:registers 95	t_{103}
c:Distr3 t:registers 65	t_{104}

2.2.2 Windows

L'introduzione degli stream di dati C-SPARQL richiede l'abilità di identificare queste sorgenti di dati e di specificare il criterio di selezione sopra di essi.
Per quanto riguarda l'identificazione, si assume che ogni stream sia identificabile da un IRI distinto, rappresentante la localizzazione attuale della sorgente di dati stream; nello specifico, l' IRI rappresenta l'IP e la porta per accedere allo stream di dati.
Per quanto riguarda la selezione, visto che gli stream di dati sono intrinsecamente infiniti, si introduce la nozione di finestra temporale sopra uno stream, le cui caratteristiche si ispirano alle windows presenti nel mondo delle query continue su relational streaming data, come CQL[3].

Identificazione e selezione sono espresse in C-SPARQL dalla clausola FROM STREAM, la cui sintassi è la seguente:

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*

La finestra estrai dallo stream gli ultimi elementi, che vengono poi processati dalla query. Questa estrazione può essere *fisica* (un numero predeterminato di triple) o *logica* (tutte le triple che arrivano durante un dato intervallo di tempo, il numero estratto di tuple quindi può essere variabile nel tempo).

Le finestre logiche si definiscono *sliding* [13] quando avanzano progressivamente di un predeterminato STEP (ad esempio un passo che sia più breve dell'ampiezza della finestra); quando invece due finestre consecutive non si sovrappongono si definiscono *non-overlapping* (o TUMBLING), quando cioè avanzano esattamente di un intervallo temporale pari all'ampiezza della finestra stessa. Con le finestre di tipo tumbling quindi ogni tripla è contenuta in esattamente una finestra, mentre con quelle di tipo sliding può capitare che siano contenute in più di una finestra.

La parola chiave opzionale NAMED funziona esattamente come nello standard SPARQL quando applicata alla clausola FROM per tenere traccia della provenienza delle triple. Non fa altro che collegare IRI dello stream alla variabile che sarà poi accessibili attraverso la clausola GRAPH.

Esempio. Una classica interrogazione di urban computing restituisce l'elenco delle auto che entrano nel centro città attraverso i caselli senza considerare quelli che vi entrano più volte all'interno della stessa fascia oraria; la query considera gli ultimi 60 minuti, mentre la sliding window viene aggiornata ogni minuto.

```
PREFIX  t: <http://linkedurbandata.org/traffic#>

SELECT DISTINCT ?car
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 60m STEP 1m]
WHERE { ?tollgate t:registers ?car . }
```

La query viene eseguita come segue. Prima tutte le accoppiate auto e caselli vengono estratti dalla finestra corrente che opera sullo stream, poi il binding è proiettatto come elenco distinto di auto. La finestra così come è stata impostata implica che avanzi ogni minuti introducendo nuove triple ed eliminando di conseguenza le più vecchie.

2.2.3 Query Registration

Tutte le query su stream dati RDF sono denotate come *continuous queries*, perchè producono continuamente un output sotto forma di tabelle di binding di variabili o di grafi RDF. Ogni query C-SPARQL è registrata attraverso il seguente codice:

Registration ⟶ 'REGISTER QUERY' *QueryName*
['COMPUTED EVERY' *Number TimeUnit*] 'AS'
Query

la clausola opzionale COMPUTED EVERY indica la frequenza alla quale le query *dovrebbero* essere eseguite. Se la frequenza non è specificata, viene impostata dal sistema.[3]

Esempio. Si assuma che un database statico RDF memorizzi (a) i distretti della citta, (b) le strade di ogni distretto e (c) i caselli presenti in ogni strada. Si mostrerà ora una interrogazione che combina la conoscenza statica (dal database di triple) con una dinamica (delle streaming triples) in modo da periodicamente fornire un report delle auto che hanno attraversato ogni distretto negli ultimi 30 minuti. In questo esempio la finestra slitta ogni cinque minuti. Verrà omesso d'ora in avanti i prefissi `c:` and `t:` per brevità

[3]Diversi sistemi di DSMS sono in grado di ottimizzare autonomamente la frequenza di esecuzione delle query registrate, questo anche per mitigare gli inconvenienti di eventuali picchi di carico.

```
REGISTER QUERY CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS

SELECT DISTINCT ?district ?car
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?tollgate c:placedIn ?street .
        ?district c:contains ?street . }
```

La query viene eseguita come segue: come nel caso precedente le coppie di legami tra caselli e auto registrate vengono estratti dalla finestra attuale che opera sullo stream e su di essi viene eseguita un'operazione di join con il graph pattern usato per estrarre dal database RDF le coppie di legami caselli-distretti. A questo punto le distinte coppie di distretti e auto sono proiettati nella soluzione.

2.2.4 Stream Registration

Il risultato può essere un insieme di bindings, ma anche un nuovo stream RDF. Per generare uno stream, la query deve essere registrata nel modo seguente:

Registration ⟶ 'REGISTER STREAM' *QueryName*
['COMPUTED EVERY' *Number TimeUnit*] 'AS'
Query

Solo interrogazioni del tipo CONSTRUCT e DESCRIBE[4] possono essere registrate come generatori di stream RDF, in quanto producono triple RDF associate ad un timestamp come effetto della loro esecuzione.

Esempio. L'esempio seguente mostra la costruzione di un nuovo stream di dati RDF attraverso la registrazione di una query CONSTRUCT .

```
REGISTER STREAM CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS

CONSTRUCT {?district t:has-entering-cars ?cars}
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?district c:contains  ?street .
        ?tollgate c:placedIn  ?street . }
```

Questa interrogazione segue la logica dell'esempio precedente con la differenza che viene creato uno stream di triple RDF come output. Ogni esecu-

[4]Esistono quattro tipi di interrogazioni in SPARQL che differiscono dalla prima clausola: SELECT restituisce il contenuto delle variabili presenti in un query pattern match. CONSTRUCT restituisce un nuovo grafo RDF costruito sostituendo le variabili all'interno di templates contenenti set di triple. ASK restituisce un booleano che indica se il query pattern ha effettuato positivamente il match o no. DESCRIBE restituisce un grafo RDF che descrive la risorsa trovata. Si faccia riferimento a [22] per ulteriori dettagli.

zione della query può produrre un minimo di una tripla fino ad un massimo di un intero grafo, ma il timestamp dipende sempre dal momento di esecuzione della query e viene assegnato dal sistema in maniera monotonicamente non decrescente. Si mostra l'esecuzione della query dopo due volte.

triple	Timestamp
`c:Distr1 t:has-entering-cars AB1234`	t_{400}
`c:Distr2 t:has-entering-cars BC1234`	t_{400}
`c:Distr1 t:has-entering-cars CD1234`	t_{401}
`c:Distr2 t:has-entering-cars DE1234`	t_{401}
`c:Distr3 t:has-entering-cars EF1234`	t_{401}

La prima valutazione viene fatta a t_{400}. Si supponga che solo due sorgenti (`c:Distr1` e `c:Distr2`) siano presenti nella finestra. Vengono quindi generate due trisple con lo stesso timestamp(t_{400}).

La seconda valutazione della query avviene a t_{401}. Si supponga che parte dei dati elaborati precedentemente sia ancora all'interno della finestra mentre i nuovi dati relativi a `uc:Distr3` siano entrati ora. L'esecuzione produce quindi 3 triple con timestamp t_{401}.

2.2.5 Multiple Streams

C-SPARQL può combinare triple che provengono da più sorgenti stream contemporaneamente, come mostrato nell'esempio seguente.

Esempio. Si consideri, in aggiunta ai caselli, anche la presenza delle telecamere come strumento di controllo del traffico posizionato sopra gli incroci. I dati di queste telecamere provengono da un secondo stream. Si consideri la seguente query in cui le auto viste dalle telecamere o che attraversano i caselli siano sommati in modo che vengano restituite le strade che presentano un numero di passaggi maggiore dell'80% della propria capacità massima, tenendo una finestra temporale degli ultimi 5 minuti.

```
REGISTER QUERY FullStreets AS
SELECT ?street (COUNT (?street) AS ?passages)
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 5m TUMBLING]
FROM STREAM <http://streams.org/citycameras.trdf>
            [RANGE 5m TUMBLING]
WHERE {   ?street   c:hasCapacity ?capacity .
  {
  GRAPH <http://streams.org/citytollgates.trdf> {
    ?tollgate t:registers ?car .
    ?tollgate c:placedIn  ?street  . }
  }  UNION  {
  GRAPH <http://streams.org/citycameras.trdf> {
    ?camera   t:registers ?car .
    ?camera   c:placedAt  ?light .
    ?light    c:crossing  ?street . }
}}}
GROUP BY ?street
HAVING (?passages > (0.8 * ?capacity))
```

L'interrogazione esegue i seguenti passi. Vengono estratte le coppie auto e caselli dal primo grafo, usando la finestra sopra lo stream dei caselli, e dal secondo grafo, usando la finestra sopra lo stream delle telecamere. Viene anche estratta dalla base di dati statica RDF la capacità di ogni strada. I bindings sono combinati seguendo la semantica della valutazione del pattern UNION di SPARQL, e raggruppati tramite la clausola GROUP BY per ?street vengono conteggiate le auto registrate dalle telecamere e dai caselli e il risultato legato ad una nuova variabile ?passages. Infine le strade che soddisfano il filtro espresso dalla clausola HAVING sono selezionate e proiettate insieme al loro numero di passaggi.

2.2.6 Timestamp Function

Il timestamp di un elemento dello stream può essere recuperato e legato ad una variabile usando la funzione timestamp che riceve due argomenti come parametri.

- Il primo è il nome della variabile, introdotto nella clausola WHERE e legato alle triple RDF dello stream da un pattern matching.
- Il secondo (opzionale) è IRI dello stream, che può essere ottenuta attraverso la clausola SPARQL GRAPH.

La funzione restituisce il timestamp dell'elemento dello stream RDF producendo un nuovo binding. Se la variabile non è bound, la funzione è indefinita e qualsiasi comparazione che la coinvolga avrà un comportamento non determinato. Se invece la variabile ha molteplici legami, viene restituito il timestamp più recente rispetto al momento di esecuzione della query.

Esempio. Per mostrare l'uso della funzione timestamp all'interno delle interrogazioni, si mostrera una variante dell'esempio precedente. Ora l'obiettivo è di ricercare tutte le auto che si spostano da una certa strada (Palm Street) in un altra (Oak Avenue) tramite la lettura di due telecamere poste allo stesso incrocio. La query è la seguente:

```
REGISTER STREAM AllCarsTurningFromPalmIntoOak
         COMPUTED EVERY 1m AS

SELECT DISTINCT ?car1
FROM STREAM <http://streams.org/citycameras.trdf>
            [RANGE 5m STEP 1m]
WHERE { ?camera1 c:monitors c:Oak-Avenue .
        ?camera2 c:monitors c:Palm-Street.
        ?camera1 c: placedAt ?tr_light .
        ?camera2 c: placedAt ?tr_light .
        ?camera1 t: registers ?car1 .
        ?camera2 t: registers ?car2 .
        FILTER ( timestamp(?car1)>timestamp(?car2)
                 && ?car1 = ?car2 )}
```

Si noti che vengono usate due differenti variabili (`?car1` e `?car2`) per riferirsi alla stessa auto, come dichiarato nella clausola `FILTER`. Questo viene fatto per estrarre i due timestamp differenti e verificare che l'auto sia vista prima da `?camera1` e poi da `?camera2`. In questo modo viene eseguito il match solo per le auto che effettivamente si spostano nella direzione indicata e non in quella opposta.

2.2.7 Execution Enviroment

Per quanto riguarda l'ambiente di esecuzione proposto per C-SPARQL, si è tenuto conto delle precedenti soluzioni ormai altamente ottimizzate per la processazione di query continue su stream relazionali. Perciò l'approccio è stato quello di un architettura di tipo plug-in che sfruttasse le tecnologie preesistenti. Visto che gli stream attualmente disponibili non gestiscono RDF data streams ma bensì stream relazionali, è stato investigato come fosse possibile soddisfare i requisiti di C-SPARQL con le tecnologie relazionali, come STREAM [2], Aurora/Borealis [1] e Stream Mill [4], Esper. La soluzione presa in considerazione è stata Esper.

La figura 2.2 mostra l'architettura proposta per il framework che usa completamente tecnologie esistenti. Mentre uno SPARQL reasoner è usato per la valutazione della parte statica delle interrogazioni, un DSMS relazionale valuta sia gli streams che gli aggregati. Si noti che questo approccio è sostenibile solo se gli aggregati vengono eseguiti dal DSMS. Un parser esegue il parsing della query C-SPARQL e la consegna all'orchestratore. Questo componente è il cuore centrare dell'architettura e traduce la query nelle due parti statica e dinamica. La query statica viene usata per estrarre la conoscenza statica dal reasoner, mentre la query dinamica è registrata nel DSMS. Questa processo è eseguito solo una volta quando la C-SPARQL è registrata, in quanto poi l'esecuzione è effettuata dal DSMS.

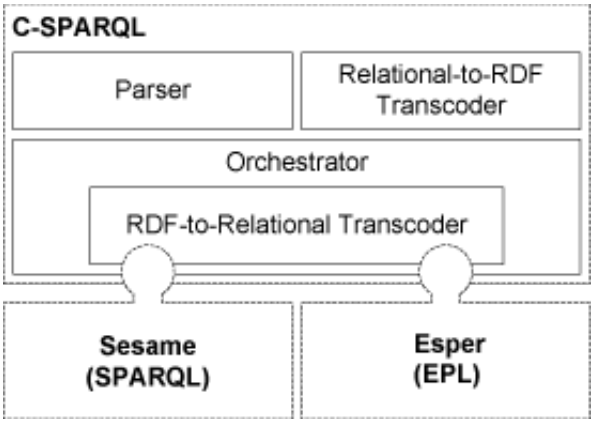

Figura 2.2: Panoramica dell'architettura dell' execution enviroment.

Capitolo 3

Data stream processing

3.1 Relational data stream management systems

3.1.1 Storia e sistemi esistenti

Gli stream di dati sono sequenze continue di elementi variabili nel tempo. Se ne ritrovano in una grande varietà di applicazioni moderne, tra cui il monitoraggio delle reti, le reti di sensori, le applicazioni con tag RFID, le registrazioni di telecomunicazioni e le applicazioni finanziarie. Il processamento di questi stream di dati è stata oggetto di un vasto studio nell'ultima decade [19], sono stati sviluppati dei software di gestione specializzati, i Data Stream Management Systems (DSMS) e le loro funzionalità stanno venendo implementate anche dai grandi colossi del settore come Oracle e DB2.

I DSMS rappresentano un grande cambiamento di paradigma per il mondo delle basi di dati, in quanto si passa da una concezione di relazioni persistenti e interrogazioni invocate dall'utente singolarmente dall'utente, a dati stream transitori e interrogazioni continue. Le innovazioni maggiori stanno proprio nell'idea di avere degli stream di dati che possono essere utilizzati in tempo reale invece di essere precedentemente memorizzati in maniera persistente e nel fatto che le query possano monitorare costantemente questo flusso di dati e produrre le loro risposte anche senza la necessità di un invocazione. I DSMS supportano la gestione dei risultati di query parallele su dati originati in tempo reale e possono sopportare anche dei picchi di carico adattando a run time il loro comportamento diminuendo l'accuratezza delle risposte agendo sul loro livello di approssimazione.

Nonostante questi DSMS abbiano provato essere degli ottimi strumenti per l'analisi al volo di stream di dati, non sono in grado di effettuare complesse operazioni di reasoning come quelle necessarie a rispondere a domande del tipo *"Quali sono gli argomenti più discussi al momento su Twitter?"*.

I DSMS si basano sull'osservazione che non solo non è impossibile controllare l'ordine in cui arrivano i dati dello stream, ma soprattutto non è sostenibile la loro memorizzazione integrale. Il *Chronicle data model* [16] è

uno dei primi modelli proposti per la gestione degli stream di dati. Introdusse il concetto delle *chronicles*, sequenze ordinate di tuple create aggiungendo le nuove tuple in fondo alla catena. Inoltre introdusse un linguaggio di definizione delle stesse e un algebra che operasse sia sulle chronicles che sulle interrogazioni tradizionali. OpenCQ [17], NiagaraCQ [15] and Aurora [14] sono degli esempi di implementazioni di DSMS che operano su query continue. Il primo linguaggio strutturato appositamente per gli stream di dati, CQL [2] [3], fu il risultato delle ricerche fatte sul problema delle query continue su stream di dati, considerando sia aspetti semantici che riguardanti l'efficienza. Il risultato fu la specifica di un architettura generale e flessibile per il processamento degli streams. Più recentemente è stata messa enfasi sul problema del data mining sugli streams [18]. Hanno quindi concepito e sviluppato Stream Mill [4], che considera e affronta estensivamente queste problematiche, specialmente rispetto all'aspetto dell'aggregazione di dati online e sulle nozioni distintive di operatori bloccanti e non. Il suo linguaggio di interrogazione (ESL) supporta efficientemente le finestre temporali fisiche che logiche (con le opzioni slides e tumbles) sia sulle aggregazioni standard definite nel sistema sia quelle definite dall'utente. I costrutti definiti in ESL hanno esteso la potenza e la generalità dei sistemi DSMS.

3.2 Complex event processing (CEP)

3.2.1 Storia e sistemi esistenti

Introduzione al processamento di eventi complessi

Il processamento degli eventi è stato il cuore di qualsiasi sistema di computing per più di una decade. La sfida comune tra le industrie del settore è stata quella di essere in grado di estrarre dell'intelligenza, eseguibile attraverso delle azioni, da sorgenti di eventi disparati, il tutto possibilmente in tempo reale. Qualsiasi ritardo nel recupero delle informazioni riduce il loro valore e bisognerebbe essere in grado di prevedere quando queste informazioni non sono recuperabili in un certo lasso di tempo. Quindi, su questo insieme di eventi in arrivo di devono poter eseguire operazioni di aggregazioni e filtraggio fissando un periodo temporale di riferimento.

I database relazionali sono difficilmente in grado di gestire dati temporali in real time e interrogazioni continue. Altri sistemi molto conosciuti di processamento di eventi si sono fin'ora focalizzati nell'integrazione delle parti terminali come servizi, introducendo astrazione sopra il trasporto e i protocolli: EAI, MOM e in generale SOA (Service-Oriented Architecture). La parte mancante di intelligenza eseguibile attraverso azioni è un processore di eventi capace di eseguire interrogazioni continue con un liguaggio ad hoc molto potente, in grado di esprimente le richieste più complesse: quelle in cui tempo e casualità sono le variabili fondamentali che entrano in gioco.

Questo è ciò di cui si occupa CEP: l'analisi di dati (eventi) che fluiscono attraverso i sistemi informativi per recuperare informazioni significative in tempo reale. La complessità degli eventi sta nel fatto che possono esistere delle relazioni tra di essi, sia di causalità che di temporalità.

3.3 Esper ed EPL

3.3.1 Introduzione

Esper è un engine CEP open-source scritto interamente in Java e completamente integrabile in qualsiasi processo Java, JEEE, ESB, BPM ecc. Ha da poco raggiunto la versione 2.0, è sostenuto da EsperTech con un modello di doppia licenza sia open-source che professionale e richiama un notevole interesse sia nella comunità Java che in quella .Net. Per questo motivo è stata sviluppata una versione interamente scritta in C# chiamata NEsper. Entrambe le versioni sono liberamente scaricabili dal sito esper.codehaus.org e sono complete di esempi e documentazione esaustiva.

Il motore di Esper funziona quasi come un database al contrario. Invece di memorizzare i dati ed eseguire le interrogazioni su di essi, Esper abilità la possibilità per le applicazioni di memorizzare le query e di far passare i dati attraverso di esse. In questo modo le risposte sono in tempo reale in quanto le interrogazioni vengono eseguite continuamente invece che solo in fase di registrazione.

Una delle caratteristiche peculiari di Esper è la facilità nel formulare le interrogazioni: viene fornito un linguaggio simile a SQL (Event Process Language, EPL) esteso per gestire gli stream di eventi e il match dei patterns. per gli sviluppatori più inclini alla programmazione sfruttando delle API, viene messo a disposizione anche un oggetto che rappresenta la query continua.

Le altre caratteristiche principali sono elencate di seguito:

- Interrogazioni, filtraggio e computazione del risultato in maniera continua.
- Operazioni di Join continue
- Finestre temporali basate su tempo o numero di tuple.
- Logica Followed by e ricerca di eventi mancanti.
- Join continui di stream e dati storici memorizzati in database relazionali tradizionali, effettuando cache locale.
- Rappresentazione degli eventi attraverso oggetti Java, .Net, Map o XML

- Modello di processazione di tipo *continuous listener* e recupero dei dati basandosi su interatore secondo il modello pull.

3.3.2 Esempi di casi d'uso

I casi d'uso per i motori CEP sono numerosi, tra cui applicazioni di tracking di dati finanziari, RFID e più in generale in SOA basati su eventi. Di seguito si descriverà un tipico caso di BAM (Business Activity Monitoring): avviene un processo di vendita tramite pagamento, ad esempio con PayPal, che si connette al sistema di gestione delle richieste di pagamento della banca del cliente. Si consideri che si usi un motore CEP per monitorare la corretta accettazione da parte del sistema della banca e riportare eventiali situazioni problematiche.

Eventi di richieste di pagamento

L'applicazione manda richieste di pagamento al sistema esterno che le processa, punto di ingresso del sistema della banca del cliente. La richiesta avrà le seguenti informazioni:

```
public class PayInqRequest {
  private String customerId;
  private String paymentId;
  private long timestamp;
}
```

Il sistema della banca deve confermare la ricezione della richiesta di pagamento entro 15 minuti dal ricevimento, rispondendo con una conferma che attesti che la richiesta sia presente correttamente all'interno del proprio sistema, come ad esempio indicato di seguito:

```
public class PayInqAck {
  private String customerId;
  private String paymentId;
  private long timestamp;
  private boolean isFound;
}
```

Riportare pagamenti non trovati

Quando il sistema della banca non trova il pagamento nel proprio sistema, deve riportare un messaggio di errore. Per massimizzare l'efficienza operativa però si vuole che questo messaggio venga inviato solo se la condizione si verifica con una certa frequenza, ad esempio per più di tre volte negli ultimi dieci minuti.

Innanzitutto per soddisfare questo requisito dobbiamo filtrare lo status dell'evento `PayInqAck` e conteggiare il totale dei false per ogni cliente. Questa operazione va però eseguita sugli eventi non più vecchi di dieci minuti rispetto al momento attuale, creando una finestra temporale che scorra in avanti

costantemente. Quindi si richiedono le due caratteristiche base gestite da Esper: il setup di eventi e la creazione di una finestra temporale scorrevole sui dati per filtrarli. La query di seguito riportata esegue proprio questa operazione di filtro degli eventi non più vecchi di dieci minuti e totalizzazione per cliente:

```
select customerId, count(*)
  from PayInqAck(isFound = false).win:time(10 minutes)
  group by customer
  having count(*) >= 3
```

Con la sintassi `win:time` usata precedentemente si dichiara una finestra temporale di 10 minuti. La query riporta immediatamente quando la condizione in essa espressa si verifica, migliorando l'efficienza operativa. L'uso di un engine CEP consente di guadagnare molto rispetto ad uno sviluppo a mano di questa logica di controllo. Inoltra consente una grande flessibilità nel cambiarla e arricchirla per rimanere al passo con i cambiamenti delle regole di business.

3.3.3 Esempio Query EPL di Esper

Di seguito verrà mostrato come effettivamente viene usato EPL per registrare e ottenere i risultati delle interrogazioni e come inviare eventi nel sistema affinchè vengano utilizzati per generare i risultati.

Esper può essere configurato sia tramite API che tramite file XML. Di seguito si mostrerà l'utilizzo solo di alcuni API fondamentali. Si tenga presente che le interrogazioni possono essere create e registrate a runtime.

```
EPStatement statement = service.createEQL(
     "select customerId, count(*) as cnt " +
     "from PayInqAck(isFound = false).win:time(10 minutes) " +
     "group by customer " +
     "having count(*) >= 3";
```

Un modulo col ruolo di `observer` si sottoscrive per ricevere i risultati continui. Esper invoca questo modulo e gli restituisce i risultati fortemente tipizzati quando le condizioni dell'interrogazione viene soddisfatta, in tempo reale. In questo modo, una volta che lo stream di eventi è disponibile, è relativamente semplice aggiungere ulteriori interrogazioni attraverso cui far passare lo stream o ampliare e modificare le interrogazioni già registrate, in quanto il modello di esecuzione è continuo e non solo nel momento di sottoscrizione della query. Le interrogazioni vengono espresse come si è visto in un linguaggio molto simile a SQL garantendo la necessità di un basso sforzo di apprendimento per la loro scrittura. Gli eventi vengono utilizzati indipendentemente dalla tecnologia utilizzata per l'integrazione consentendo di fatto la possibilità di riuso di sistemi preesistenti.

3.3.4 Caratteristiche avanzate

- Joins continui : le operazioni di join o join esterni tra più flussi di eventi possono essere utili quando si comparano stream di eventi provenienti da differenti sistemi. I join continui di Esper sono incrementali per loro stessa natura: vengono valutati ogni volta che arriva un nuovo evento in uno qualsiasi degli stream appartenenti al join. I join possono anche includere dati statici provenienti da database relazionali o dati ottenuti attraverso un metodo arbitrario di invocazione, che apre alla possibilità di un ampio spettro di possibilità di integrazione, ad esempio con dati facenti parte di cache distribuita.

- Variabili delle procedure : valori di soglia, come numero di eventi per intervallo di tempo, sono tipicamente rappresentati come variabili in Esper per parametrizzarne le procedure.

- Robustezza e out of heap overflow : EsperTech fornisce una estensione, sotto licenza commerciale, che garantisce alta robustezza e disponibilità (EsperHA) preservando dai crash sui calcoli interni dei risultati intermedi e verificando preventivamente che la la finestra da processare non sia limitata da una mancanza di memoria del sistema.

Capitolo 4

Progettazione parser C-SPARQL

Dopo aver descritto le tecnologie che stanno alla base di questo lavoro, verranno introdotti e descritti da qui in avanti i componenti che sono l'oggetto vero e proprio di questo elaborato e che sono stati sviluppati durante questo lavoro.

In particolare questo capitolo si occupa del componente che sta a monte dell'operazione di traduzione di una query C-SPARQL in quelle SPARQL ed EPL, ovvero del **parser**. Il suo compito è quello di verificare se l'interrogazione che si intende tradurre sia effettivamente una query C-SPARQL valida almeno sintatticamente ed eventualmente semanticamente. Questa operazione di riconoscimento di appartenenza di una stringa di testo rispetto ad un linguaggio[1], in questo caso C-SPARQL, deve essere eseguita automaticamente e soprattutto efficientemente.

Vista la complessità intrinseca del ruolo svolto da questo componente si è rivelato necessaria l'adozione di strumenti e metodologie, quali la definizione di una grammatica del linguaggio C-SPARQL, che verranno di seguito introdotti e giustificati in quanto necessari alla generazione di un parser efficiente ed efficace. Esistono infatti dei tools (es. ANTLR) che permettono di generare automaticamente un parser con le suddette caratteristiche a partire dalla grammatica del linguaggio che deve essere in grado di riconoscere. La progettazione del parser si è quindi tramutata nella progettazione e stesura di una grammatica C-SPARQL che fosse in grado di recepire tutta la sua sintassi[2].

[1] trattata estensivamente nella sezione 4.1.1.6

[2] Per una trattazione estensiva della sintassi C-SPARQL si veda cap 2

4.1 Tecniche traduzione automatica

Per poter tradurre un linguaggio bisogna prima descriverlo attraverso uno strumento formale: la grammatica.

4.1.1 Grammatiche

Le grammatiche permettono di descrivere i linguaggi attraverso un approccio generativo: metodo di costruzione delle stringhe del linguaggio basato sulla riscrittura.

- 1914 Axel Thue studia i primi problemi di riscrittura.
- 1943 Emil Post definisce sistemi di produzione (lavori del 1920).
- 1947 A.A. Markov definisce algoritmi basati su regole di riscrittura.
- 1956 N. Chomsky introduce le grammatiche formali nell'ambito degli studi sul linguaggio naturale.
- 1960 J. W. Backus e P. Naur introducono la BNF per descrivere la sintassi del linguaggio Algol.

4.1.1.1 Grammatiche di Chomsky

Una grammatica formale e' un sistema G = <V_T,V_N,S, $\prod$> caratterizzato da:

- $V_T = \Sigma$ alfabeto finito di simboli detti terminali,
- V_N alfabeto di simboli non terminali (variabili, categorie sintattiche)
- $V = V_T \cup V_N$ simboli terminali e non
- $\prod$, detto insieme di produzioni, è una relazione binaria finita su $V^* \cdot V_N \cdot V^* \times V^*$. Ogni produzione $< \alpha, \beta > \in \prod$ si indica con $\alpha \rightarrow \beta$, α contiene contiene almeno un simbolo non terminale. In generale una produzione che ha molteplici alternative viene indicata nel seguente modo: $\alpha \rightarrow \beta_1$ | ..... | $\alpha \rightarrow \beta_n$
- S$\in V_N$ è detto assioma.

4.1.1.2 Definizione di linguaggio

Il linguaggio generato da una grammatica è l'insieme delle stringhe di terminali ottenibili con una sequenza finita di passi di riscrittura consistenti nell'applicazione delle regole di produzione Esempio.
$G =< a, b, c, S, B, C, S, \prod > con \prod$consistente delle seguenti produzioni $S \rightarrow aS|BB \rightarrow bB|bCC \rightarrow cC|c$ genera L(G) = anbmch | n=0, m,h=1

4.1.1.3 Derivazione

Meccanismo generativo su cui si basano le grammatiche.
Derivazione diretta: relazione su $(V^* \cdot V_N \cdot V^*) \times V^*, < \phi, \psi >$ appartiene alla relazione se $\alpha \in V^{*\circ}V_N^{\circ}V^*$ e $\beta, \gamma, \delta \in V^*$ tali che $\phi = \gamma\alpha\delta, \psi = \gamma\beta\delta$ e $\alpha \rightarrow \beta \in \prod$.
In tal caso si scrive $\phi \Rightarrow \psi$ e si dice che ϕ produce direttamente ψ o che ψ deriva direttamente da ϕ.

Si dice che ϕ produce (indirettamente) ψ o che ψ deriva (indirettamente) da ϕ e si indica con la notazione $\phi \Rightarrow^+ \psi$ se esiste una sequenza di derivazioni dirette $\phi = x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n = \psi$ con n>0 , con n pari alla lunghezza della derivazione.
Con $\phi \Rightarrow^* \psi$ si indica la possibilità che ϕ produca ψ o che ϕ e ψ coincidano.

4.1.1.4 Linguaggio generato da una grammatica

Data una grammatica G=<VT,VN,S, $\prod$ >, una forma di frase è una qualunque stringa x tale che x$\in V^*$ e S $\Rightarrow^*$ x.
Il linguaggio generato da G è l'insieme di particolari forme di frase, costituite di soli terminali.
Il linguaggio generato da G è L(G) = x$\in$VT* | S$\Rightarrow^*$x

Esempio

G=<{a,b,c},{S,B,C}, S, $\prod$ >
1.S $\rightarrow$ aSBC
2.S $\rightarrow$ aBC
3.CB $\rightarrow$ BC
4.aB $\rightarrow$ ab
5.bB $\rightarrow$ bb
6.bC$\rightarrow$ bc
7.cC$\rightarrow$ cc

genera il linguaggio $a^n b^n c^n$ | n=1

4.1.1.5 Classificazione grammatiche

CLASSI DI GRAMMATICHE DI CHOMSKY

- di tipo 0, non limitate
- di tipo 1, contestuali (context sensitive: CS)
- di tipo 2, non contestuali (context free: CF)
- di tipo 3, regolari

Un linguaggio è strettamente di Tipo n se esiste una grammatica di tipo n che lo genera e non esiste nessuna grammatica di tipo m>n che lo genera. Più la grammatica è di livello inferiore e maggiore è la sua potenza espressiva che va di pari passo con la complessità dell'automa che è in grado di riconoscerlo, come mostrato nella tabella seguente.

Gerarchia di Chomsky	Grammatica	Linguaggio	Automa **minimo**
Tipo-0	(illimitato)	Ricorsivamente enumerabile	Macchina di Turing
	(illimitato)	Ricorsivo	Decider
Tipo-1	Dipendente dal contesto	Dipendente dal contesto	Automa lineare
Tipo-2	**Libera dal contesto**	Libero dal contesto	Automa a pila ND
Tipo-3	Regolare	Regolare	A stati finiti

Figura 4.1: Teoria degli automi: linguaggi formali e grammatiche formali

Grammatiche di tipo 0

Le grammatiche di Chomsky di tipo 0, coincidono con le grammatiche definite fin qui, cioè non ci sono limitazioni sul formato delle regole di produzioni, che sono quindi del tipo:
$\alpha \rightarrow \beta, \alpha \in V^* \cdot V_N \cdot V^*, \beta \in V^*$.
Le grammatiche di tipo 0 ammettono anche derivazioni che accorciano stringhe cioè $|\alpha| > |\beta|$. I linguaggi di tipo 0 sono tutti i linguaggi generati da grammatiche di tipo 0.

Grammatiche di tipo 1

Le grammatiche di tipo 1, (dette context sensitive o contestuali) sono basate su produzioni del tipo:
$\alpha \rightarrow \beta, \alpha \in V^* \cdot V_N \cdot V^*, \beta \in V+$ con $|\alpha|=|\beta|$
Le produzioni di tipo 1 non riducono mai la lunghezza delle forme di frase tranne che nel caso in cui il linguaggio generato contenga la stringa vuota poiché in questo caso è necessaria la regola S $\rightarrow \epsilon$. I linguaggi di tipo 1 (context sensitive o contestuali) sono i linguaggi generati da grammatiche di tipo 1. Ovviamente ogni linguaggio o grammatica di tipo 1 è anche un linguaggio o grammatica di tipo 0 ma il viceversa non vale sempre. Ad esempio il linguaggio $\{a^n b^n c^n \mid n=1\}$ è di tipo 1.

Grammatiche di tipo 2

Le grammatiche di tipo 2, (dette context free o non contestuali) sono basate su produzioni del tipo:
$\alpha \rightarrow \beta, \alpha \in V_N, \beta \in V^*$
cioè sulla parte sinistra di una regola c'è un solo simbolo che è ovviamente

non-terminale. I linguaggi di tipo 2 (context free o non contestuali) sono i linguaggi generati da grammatiche di tipo 2. Una grammatica di tipo 2 non necessariamente è di tipo 1 poiché potrebbe contenere una produzione vuota $A \rightarrow \epsilon$. Ma è sempre possibile riscriverla in modo che sia di tipo 1. Quindi i linguaggi di tipo 2 sono anche di tipo 1. In pratica, le grammatiche di tipo 2 contengono spesso produzioni vuote poiché sono comode e non rischiose. Il linguaggio $a^n b^n$ | n=1 è di tipo 2 in quanto generato dalla grammatica con le produzioni: S → aSb | ab.

Grammatiche di tipo 3

Le grammatiche di tipo 3, (dette regolari) sono basate su produzioni del tipo (lineare destre):
A→αB oppureA→α, con A, B∈V_N, $\alpha \in V_T^*$
Oppure da produzioni del tipo (lineare sinistre):
A→ B α oppure A→ α, con A, B∈V_N, $\alpha \in V_T^*$
I linguaggi di tipo 3 (regolari) sono i linguaggi generati da grammatiche di tipo 3. Ovviamente le grammatiche di tipo 3 sono anche di tipo 2, quindi un linguaggio di tipo 2 è anche di tipo 3. Il linguaggio $a^n b$ | n=0 è di tipo 3 in quanto è generato dalla grammatica lineare destra: S → aS | b oppure dalla grammativa lineare sinistra S→ Ab|b , A→Aa|a.

4.1.1.6 Riconoscimento di linguaggi

Dato un linguaggio L, il problema del riconoscimento di L è il problema decisionale seguente: data una stringa x, stabilire se essa appartiene ad L. Tale problema è noto anche come problema dell'appartenenza (o membership). La maggior parte dei linguaggi sono indecidibili, cioè il problema dell'appartenenza non è risolvibile in tempo finito da un qualche algoritmo.

- I linguaggi di tipo 0 sono semidecidibili cioè se la stringa appartiene il riconoscimento avviene in tempo finito altrimenti la risposta negativa potrebbe non arrivare mai. La classe dei linguaggi di tipo 0 coincidono con la classe di tutti i linguaggi semidecidibili.

- I linguaggi di tipo 1 sono decidibili nel senso che il problema dell'appartenenza è risolvibile in tempo finito, tuttavia non tutti i linguaggi decidibili sono generati da grammatiche di tipo 1. In effetti non esiste nessun formalismo che definisce tutti e soli i linguaggi decidibili: o ne include qualcuno semidecidibile o ne esclude qualcuno decidibile.

- I linguaggi di tipo 2 e quelli di tipo 3 sono ovviamente decidibili e riconoscibili in maniera efficiente, soprattutto quelli di tipo 3, per i quali il risultato lo si determina con una sola scansione (passata) della stringa da riconoscere.

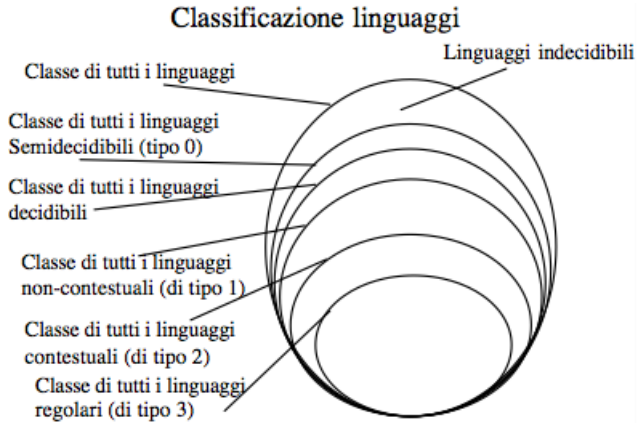

Figura 4.2: Classificazione linguaggi

4.1.2 Traduttore

Per tradurre un linguaggio A in un linguaggio B è necessario procedere in due fasi: una di analisi e una di sintesi.

Nella fase di analisi si possono identificare tre passi principali:

- Analisi lessicale (lexer): converte gruppi di caratteri in token.
- Analisi sintattica (parser): converte gruppi di token in un albero sintattico.
- Analisi semantica: verifica la validità logica dell'albero sintattico.

Nella fase di sintesi viene prodotto la risposta all'input nel linguaggio di destinazione.

4.1.2.1 Lexer

Il lexer ha il compito di leggere stream di caratteri e di produrre uno stream di token corrispondente. Un token è esprimibile con le espressioni regolari (es: DIGIT := [0-9]+;) e quindi implementabile attraverso un automa a stati finiti.
Infatti è in grado di interpretare sequenze precise di caratteri ("uno"), ripetizioni indefinite ("+" / "∗") o scelte finite ("?","[ ]","|").

4.1.2.2 Parser

Il parser è un programma che deve cercare di riconoscere la grammatica di uno stream di token generando come struttura dati di output un albero sintattico. Per fare questo è necessario usare la ricorsione per gestire strutture arbitrarie degli alberi e le grammatiche usate per riconoscere i linguaggi di

programmazione sono quelle libere dal contesto (che permettono riconoscimenti molto efficienti). Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool (such as Antlr) from a grammar written in Backus-Naur form. Il lavoro del parser quindi è essenzialmente quello di determinare se e come l'input può essere derivato dal simbolo iniziale con le regole della grammatica formale. Questo può essere fatto essenzialmente in due modi:

- Analisi top-down - Un parser può partire con il simbolo iniziale e cercare di trasformarlo nell'input. Intuitivamente, il parser parte dal più grande elemento e lo divide in parti sempre più piccole. I parser LL sono esempi di parser top-down.
- Analisi bottom-up - Un parser può partire con l'input e cercare di riscriverlo sino al simbolo iniziale. Intuitivamente, il parser cerca di trovare il più elementare simbolo, quindi elabora gli elementi che lo contengono, e così via. I parser LR sono esempi di parser bottom-up.

Un'altra importante distinzione è quella tra parser che generano per leftmost derivation o per rightmost derivation. I parser LL generano una derivazione leftmost e i parser LR una derivazione rightmost.

```
EXPR := TERM ( "+" TERM | "-" TERM )*
TERM := FACTOR ( "*" FACTOR | "/" FACTOR )*
FACTOR := NUM | "(" EXPR ")" | "-" FACTOR| "+" FACTOR
NUM := "[0-9]+"
```

Ad esempio data la grammatica sopra riportata e l'input "-248" si ottiene un albero risultante mostrato in Figura 4.3, mentre la Figura 4.4 mostra il flusso delle regole utilizzate per il riconoscimento dell'input.

Figura 4.3: Parse Tree

4.2 Una grammatica completa per C-SPARQL

Dopo aver presentato che cos'è una grammatica, in questa sezione viene proposta una possibile grammatica completa per C-SPARQL aggiornata alla

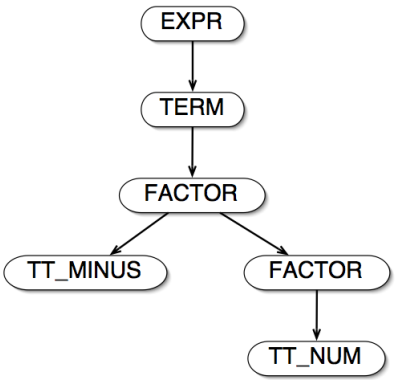

Figura 4.4: Stack chiamate

versione 1.1 del linguaggio prendendo come linee guida le specifiche W3C[3].

Dopo una breve introduzione sul tipo di grammatica usato per C-SPARQL, vengono presentate le sue caratteristiche salienti assieme all'insieme di regole che sono in grado di descriverle. Il filo conduttore che si è seguito per la stesura delle produzioni grammaticali è stato quello ovviamente di implementare la sintassi C-SPARQL, ma anche tenendo conto della facilità di utilizzo in fase di traduzione. Quindi in alcuni casi si è preferito avere delle catene di produzioni leggermente più lunghe ma che consentissero una chiara individuazione delle varie parti della query una volta prodotto l'albero sintattico e ne consentissero quindi una più facile manipolazione.

4.2.1 Tipologia di grammatica

La grammatica di C-SPARQL è una del tipo 2, cioè libera dal contesto. Le grammatiche context-free sono infatti abbastanza potenti da descrivere la sintassi della maggior parte dei linguaggi di programmazione; al tempo stesso, sono abbastanza semplici da consentire un parsing molto efficiente essendo implementabili attraverso un automa a pila non deterministico. Non è infatti possibile descrivere un linguaggio come SPARQL o C-SPARQL con un automa a stati finiti (e quindi una grammatica di tipo 3) in quanto ad esempio non sarebbe possibile riconoscere il bilanciamento delle parentesi all'interno della query.

Di conseguenza la parte sinistra delle regole è rappresentata sempre da un singolo elemento non terminale, ma la parte destra della regola non è regolare. (Es: queryWithReg : registration? query EOF;)

[3]si faccia riferimento alle specifiche del 26 Gennaio 2010, poi aggiornate con delle leggere modifiche, dopo il completamento di questo elaborato, in data 1 Giugno 2010

4.2.2 Processo di sviluppo grammatica C-SPARQL

Si è partiti dalla grammatica di SPARQL già fatta e testata (rif: mettere riferimento alla grammatica antlr) e si è proceduto aggiugendo le componenti specifiche del linguaggio C-SPARQL. Si è quindi seguito un approccio incrementale.

I terminali vengono scritti in maiuscolo e il loro riconoscimento deve essere "case-insensitive"e sono del tipo:

```
QUERY
        :       ('Q'|'q')('U'|'u')('E'|'e')('R'|'r')('Y'|'y')    ;
```

Può essere che in alcuni casi dei terminali vengano ignorati ed esclusi dall'albero sintattico, come nel caso degli spazi bianchi:

```
WS
    : (' '| '\t'| EOL)+ { $channel=HIDDEN; }    ;
```

Per questo motivo nella presentazione delle regole verranno omesse tutte le produzioni dei terminali della grammatica che rientrano nelle spracitate casistiche. Verranno omesse per semplicità di lettura anche le regole facenti parte della grammatica SPARQL originale e che non sono state modificate significativamente. In ogni caso è presente in appendice (A) tutta la grammatica per completezza.

Parte registrazione (SPARQL 1.0)

È stata la prima parte aggiunta, in quanto era di facile inserimento prima della query stessa. Per questo si è modificato leggermente la regola iniziale della vecchia grammatica in modo che potesse riconoscere la parte di registrazione nel caso di una query C-SPARQL, altrimenti saltasse direttamente al resto della grammatica nel caso di una query SPARQL. La parte di registrazione della query / stream è infatti il carattere distintivo delle query C-SPARQL rispetto a quelle SPARQL.

Riadattamento con le regole della grammatica C-SPARQL:

```
queryWithReg
        : registration? query EOF       ;

registration
        : REGISTER (QUERY|STREAM) queryName queryRange? AS      ;
queryName
        : QUERY_NAME    ;
queryRange
        : COMPUTED_EVERY timeRange      ;
timeRange
        : TIME_RANGE    ;
TIME_RANGE
        : INTEGER TIME_UNIT     ;
TIME_UNIT
        : 'ms' | 'h'    | 's' | 'm' | 'd'       ;
QUERY_NAME
        : VARNAME       ;
```

Clausola FROM e finestra selezione (SPARQL 1.0)

Per l'identificazione della sorgente dei dati su cui fare la query si usano i loro riferimenti di tipo iri. Per quanto riguarda invece le operazioni di selezione su di essi si deve introdurre il concetto di finestra, in quanto gli stream sono infiniti. Le finestre possono essere di tipo fisico, caratterizzate da un numero di triple costante, oppure logiche, caratterizzate da una ampiezza determinata in ordine di tempo e da una velocità di avanzamento della finestra (che se è identificata dalla parola chiave TUMBLING indica che l'avanzamento della finestra è pari alla sua ampiezza e non vi è quindi sovrapposizione tra 2 finestre di dati successive).

Di seguito le regole della clausola FROM di una query SPARQL:

```
datasetClause
    : FROM ( defaultGraphClause | namedGraphClause )    ;
```

Di seguito le regole aggiunte e modificate per l'interpretazione della clausola FROM di una query C-SPARQL:

```
datasetClause
    : datasetClauseStd
    | datasetClauseStream    ;
datasetClauseStd
        : FROM ( defaultGraphClause | namedGraphClause )        ;
datasetClauseStream
        : FROM NAMED? STREAM ( defaultGraphClause | namedGraphClause )
            range    ;
range
        : OPEN_SQUARE_BRACE RANGE window CLOSE_SQUARE_BRACE     ;
window
        : physicalWindow
        | logicalWindow         ;
logicalWindow
        :       timeRange windowOverlap ;
physicalWindow
        :       TRIPLES INTEGER ;
windowOverlap
        :       STEP timeRange
        |       TUMBLING        ;
```

Funzione Timestamp (SPARQL 1.0)

Questa funzione serve a recuperare il timestamp di un elemento dello stream e legarlo ad una variabile dichiarata nella funzione stessa in modo che possa essere conseguentemente utilizzato, in generale per operazioni di filtraggio dei dati. La funzione, come si vedrà di seguito nella regola, ha anche un parametro opzionale che rappresenta l'IRI dello stream dei dati da cui si stanno recuperando i timestamps, che può essere ottenuto anche attraverso la clausola GRAPH di SPARQL.

```
timestampCall
        :       TIMESTAMP OPEN_BRACE var (iriRef |
            graphPatternNotTriples)? CLOSE_BRACE ;
```

Questa funzione è stata inserita nella regola *primaryExpression*presente in SPARQL, in quanto rappresenta il punto di arrivo della catena di produzioni riguardanti le espressioni. Questa catena è ovviamente ricorsiva, in modo da esprimere tutta la complessità delle espressioni possibili. La catena di produzioni parte con la regola expression e si sviluppa con un pattern abbastanza regolare evidenziato dalle prime due produzioni della catena incluse di seguito.

```
expression : conditionalOrExpression ;
conditionalOrExpression    :  conditionalAndExpression ( OR
    conditionalAndExpression )*    ;
    ...
primaryExpression
    : brackettedExpression
        ...
    | timestampCall
    | aggregate
    ;
```

Aggregazioni (SPARQL 1.1)

Le funzioni di aggregazione sono state forse una delle aggiunte più richieste dai ricercatori e utilizzatori di SPARQL che fino alla versione 1.1 del linguaggio avevano provveduto a implementare delle versioni personali e parziali per ovviare ai limiti espressivi del linguaggio. Le funzioni di aggregazione si applicano sopra un gruppo (di default singolo) contenente tutte le soluzioni della query. Questo gruppo può anche essere ulteriormente specificato attraverso la sintassi della regola `group by` sotto riportata. La regola permette di raggruppare le soluzioni in base al valore di una variabile e di filtrare ulteriormente sul suo valore attraverso la clausola `having`. La regola `groupBy` è stata inserita nella regola SPARQL `solutionModifier` in quanto serve a modificare l'insieme delle soluzioni ottenute proprio come ad esempio la funzione di ordinamento già presente in SPARQL1.0.

```
solutionModifier
    : groupBy? orderClause? limitOffsetClauses?    ;
groupBy
        : GROUP BY      var+ having?    ;
having
        :       HAVING brackettedExpression     ;
```

Ovviamente la regola che introduce le aggregazioni nella grammatica di C-SPARQL è stata inserita alla fine delle produzioni riguardanti le espressioni, nella regola `primaryExpression` per gli stessi motivi descritti per la funzione Timestamp. Le funzioni aggregate implementate sono : `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`;

```
aggregate       :
  ( countAggregateExpression | sumAggregateExpression
  | minAggregateExpression | maxAggregateExpression
  | avgAggregateExpression) ;
```

La funzione COUNT non fa altro che determinare la cardinalità del gruppo conteggiando tutte le soluzioni in esso presenti. Il gruppo può essere quello di default di tutte le soluzioni (si usa quindi *) oppure si può specificare il conteggio su una specifica variabile. In entrambi i casi, può essere specificato che il conteggio non consideri elementi ripetuti e si dichiara apponendo la parola DISTINCT, come di seguito riportato.

```
countAggregateExpression
        :       COUNT OPEN_BRACE ( DISTINCT? ( ASTERISK | var ) )
            CLOSE_BRACE   ;
```

La funzione SUM somma tutti i valori scalari di una espressione indicata nei parametri della funzione, facendo riferimento ai gruppi indentificati dalla clausola GROUP BY.

```
sumAggregateExpression
        :       SUM OPEN_BRACE expression CLOSE_BRACE   ;
```

Le funzioni MIN e MAX invece selezionano il valore minimo / massimo assunto dall'espressione identificata nella funzione, sempre rispetto ai gruppi identificati dalla GROUP BY.

```
minAggregateExpression
        :       MIN OPEN_BRACE expression CLOSE_BRACE   ;
maxAggregateExpression
        :       MAX OPEN_BRACE expression CLOSE_BRACE   ;
```

Infine la funzione AVG calcola il valore medio dell'espressione, sul totale della cardinalità del gruppo.

```
avgAggregateExpression
        :       AVG OPEN_BRACE expression CLOSE_BRACE   ;
```

Sottoquery (SPARQL 1.1)

Questa nuova funzionalità del linguaggio consente di innestare delle query una dentro l'altra all'interno della clausola WHERE. È stata quindi inserita la regola per le sottoquery all'interno di quella preesistente del groupGraphPattern che è una delle produzioni della regola che descrive la clausola WHERE.

```
groupGraphPattern
    : OPEN_CURLY_BRACE triplesBlock? ( ( graphPatternNotTriples | filter
        | subquery ) DOT? triplesBlock? )* CLOSE_CURLY_BRACE   ;
subquery
        :       OPEN_CURLY_BRACE selectQuery CLOSE_CURLY_BRACE  ;
```

Negazioni (SPARQL 1.1)

Questa funzionalità serve per testare l'assenza o la presenza di un pattern grazie all'introduzione delle due rispettive parole chiave:

- NOT EXISTS pattern testa che un graph pattern non corrisponda ad alcun risultato presente nel dataset, dati i valori delle variabili in scope. Non viene generato nessun legame aggiuntivo tra dati e variabili.

```
notExistsFunc
        :       NOT_BY_WORDS EXISTS groupGraphPattern   ;
```

- EXISTS pattern testa che il pattern corrisponda ai dati e anche esso non genera nuovi bindings a variabili.

```
existsFunc
        :       EXISTS groupGraphPattern ;
```

Entrambi i costrutti possono essere usati in due contesti diversi della grammatica e sono stati identificati da regole diverse per semplificare l'utilizzo del parser:

- all'interno di un `graph pattern`

```
graphPatternNotTriples
    : optionalGraphPattern | groupOrUnionGraphPattern |
        graphGraphPattern | existElt | nonExistElt      ;
existElt
        :       EXISTS groupGraphPattern        ;
nonExistElt
        :       NOT_BY_WORDS EXISTS groupGraphPattern   ;
```

- all'interno di una espressione di tipo FILTER

```
builtInCall
    :
    ...
    | existsFunc
    | notExistsFunc   ;
existsFunc
        :       EXISTS groupGraphPattern        ;
notExistsFunc
        :       NOT_BY_WORDS EXISTS groupGraphPattern   ;
```

Da notare che la quando si trovano in un filtro le negazioni si applicano a tutto il gruppo in cui il filtro è presente e solo alle variabili precedentemente dichiarate nel pattern.

Espressioni nella clausola SELECT (SPARQL 1.1)

Con questa funzionalità viene data la possibilità di inserire una espressione nella clausola SELECT e creare nuove variabili per identificare il risultato di quelle espressioni. Questa funzionalità apre anche alla possibilità di rinominare semplicemente le variabili nella clausola SELECT, grazie alla parola chiave AS così come specificato nella regola di seguito riportata. L'inserimento di una espressione e la rinomina di variabili vanno espresse includendole tra parentesi tonde, dichiarate con le parola chiavi corrispondenti OPEN/CLOSE_BRACE

che racchiudono la nuova regola `newVarFromExpression` aggiunta per rappresentare le funzionalità sopra descritte. Ovviamente questa regola è stata inserita nella preesistente regola `SELECT`.

```
selectQuery
        : SELECT ( DISTINCT | REDUCED )? ( (var | OPEN_BRACE (
            newVarFromExpression ) CLOSE_BRACE )+ | ASTERISK )
        datasetClause* whereClause solutionModifier     ;
newVarFromExpression
        :       expression AS var       ;
```

4.2.3 Testing della grammatica

Si vuole qui fare notare come la stesura della grammatica C-SPARQL sopra descritta sia stata accompagnata da una frequente e continua fase di testing. Infatti prima ancora di iniziare con la modifica evolutiva della grammatica SPARQL, è stato creato un ambiente di testing che consentisse da un lato di eseguire tutti i test già resi pubblici a corredo della grammatica di partenza [21] e dall'altro di introdurne di nuovi, specifici delle funzionalità C-SPARQL che venivano via via implementate. In questo modo si è garantita costantemente la consistenza della grammatica. Questa necessità di compatibilità a ritroso era dovuta proprio dal fatto che si partiva da una grammatica già esistente e che il linguaggio C-SPARQL include strettamente SPARQL. Non potevano essere quindi accettate situazioni in cui anche uno solo dei casi di test fino ad allora formulati e sintatticamente corretti non venisse superato. Per una trattazione esaustiva dell'implementazione di questo ambiente di test si veda la sezione (6.2).

Capitolo 5

Progettazione del Traduttore

Dopo aver spiegato il ruolo del parser quale analizzatore di una query C-SPARQL per verificarne la sua bontà da un punto di vista sintattico, si è affrontato solo il primo passo nella descrizione di un traduttore di linguaggi. Infatti ci sono altre fasi necessarie per l'ottenimento di due query SPARQL e EPL in uscita: una fase di controllo semantico e una di sintesi finale.

Nella fase di controllo semantico viene verificato il significato dell'interrogazione. Questo controllo infatti non può essere svolto dai parser che hanno un ruolo di controllori sintattici e quindi deve necessariamente essere svolto a posteriori. Il controllo semantico che si effettua all'interno del traduttore rimane comunque limitato a poche regole presenti esplicitamente nelle specifiche W3C, rimandando ad altri componenti dell'architettura LarK controlli più esaustivi.

Nella fase di sintesi si utilizzano gli elementi della query C-SPARQL analizzati tramite il parser per comporre le interrogazioni nei linguaggi di uscita, che in questo caso sono SPARQL e EPL.

In questo capitolo quindi si spiegherà anche come si inserisce il parser automaticamente generato dalla grammatica precedentemente descritta all'interno della struttura del traduttore.

5.1 Richiamo dell'architettura

Si darà di seguito una descrizione dell'architettura del traduttore, dei suoi componenti e dei ruoli che essi hanno nella processazione dell'interrogazione C-SPARQL al fine di ottenere una query SPARQL e una EPL.

5.1.1 Architettura traduttore

Il traduttore è stato diviso in tre componenti fondamentali: parser C-SPARQL, produttore SPARQL, produttore EPL. Il flusso di dati attraverso questi componenti è descritto dettagliatamente in seguito (vedere 5.2.2). I singoli

componenti invece verranno illustrati di seguito nelle rispettive sottosezioni. Si è deciso di avere tre componenti separati all'interno del traduttore per ognuna delle sue macro-funzionalità in modo da consentire la massima flessibilità per quanto riguarda la loro interazione e la possibilità di evoluzione e modifica futura.

5.1.1.1 Parser C-SPARQL

Il modulo del traduttore che si occupa di effettuare il parsing di C-SPARQL è stato arricchito di alcune funzionalità oltre al semplice riconoscimento di una interrogazione. Per questo motivo, il parser generato automaticamente dalla grammatica C-SPARQL è stato incluso in questo modulo e usato come servizio. Al suo interno questo modulo svolge infatti tre operazioni fondamentali: una di preparsing, una di parsing e una di postparsing.

L'operazione di preparsing è infatti necessaria per elaborare la query per sostituire i *code point* UNICODE codificati in UTF8 con gli equivalenti caratteri (come da specifiche SPARQL).

L'operazione di parsing è invece quella principale in cui avviene il riconoscimento sintattico della query da parte del parser generato dalla grammatica e che genera l'albero sintattico come risultato in caso di corretta esecuzione.

L'operazione di postparsing infine è necessaria per applicare alcuni dei controlli di validazione semantica di alcune regole previste dalle specifiche SPARQL (questa fase verrà estesamente trattata nella sezione 5.3).

5.1.1.2 Produttore SPARQL

Il produttore è un modulo che deve recuperare le informazioni di suo interesse contenute all'interno di un albero sintattico rappresentante la query C-SPARQL di ingresso ed essere in grado di produrre una interrogazione SPARQL sotto forma di stringa. Nel caso specifico, essendo SPARQL un linguaggio strettamente contenuto in C-SPARQL e vista la sua sintassi aggiunta in maniera evolutiva e non invasiva rispetto a quella di SPARQL, per la generazione dell'interrogazione è stato sufficiente eliminare quelle parti della query C-SPARQL riguardanti gli stream. Di fatto si tratta ricercare ed eliminare i rami dell'albero riguardanti quelle parti.

5.1.1.3 Produttore EPL

Il produttore EPL invece parte riceve come input lo stesso albero sintattico ricevuto dal produttore di SPARQL, ma deve essere in grado di recuperarne le parti continue, riguardandi cioè gli stream e le finestre temporali da usare per la registrazione delle query EPL. L'output è sempre una stringa contenente l'interrogazione direttamente eseguibile dal motore scelto come DSMS.

5.2 Principi e pipeline

5.2.1 Pattern usati: DECORATOR

Introduction

Il pattern *decorator*, una volta impostato, serve a fornire la possibilità di estendere le funzionalità di un oggetto a runtime. Per fare ciò è necessario boxare l'oggetto della classe originale (A) con la classe decoratrice (B). Questa operazione è schematizzabile nei seguenti passi:

- la classe B deve estendere la classe A.
- la classe B deve contenere un attributo che contenga il puntatore ad un oggetto della classe A. Questo attributo viene inizializzato passando al costruttore della classe B un oggetto della classe A.
- la classe B redireziona tutti i metodi della classe A all'oggetto contenuto nel suo puntatore.
- la classe B fa override di tutti i metodi della classe A di cui deve essere cambiato il comportamento.

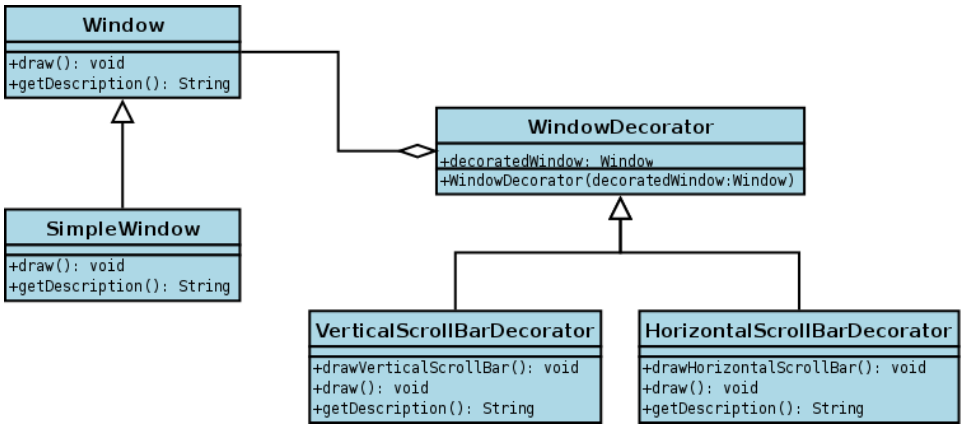

Figura 5.1: Design pattern decorator: class diagram

In questo modo è possibile racchiudere uno dentro l'altro molteplici classi decoratrici, ogni volta aggiungendo nuove funzionalità ai metodi sovrascritti. Questo pattern è un'alternativa all'operazione di `subclassing`, che invece opera a compile time e vale per tutte le istanze della classe, mentre con il pattern decorator è possibile estendere il singolo oggetto. Questa caratteristica si rivela vincente quando ci sono diversi modi alternativi ed equivalenti di estendere le stesse funzionalità con combinazioni non prevedibili in fase di design. Sarebbe quindi troppo oneroso creare una classe per ogni possibile combinazione dei comportamenti dei metodi della classe da decorare.

Motivazioni: TreeBox per superare limiti della classe ParseTree di Antlr

Vista la massima flessibilità fornita dal pattern è sembrato la scelta ottimale per aggiungere funzionalità alla classe che rappresenta il nodo degli alberi sintattici. Questa classe, fornita dal generatore di parser Antlr, offriva un ventaglio di funzionalità limitate e non del tutto implementate che si è avuto la necessità di completare e diversificare. Sarà poi possibile in futuro aumentare il numero le classi decoratrici (che al momento è solo una) per sfruttare al massimo le potenzialità del design pattern.

5.2.2 Pipeline

Verrà di seguito descritto il flusso dei dati dell'applicazione come reppresentato in Figura 5.2.

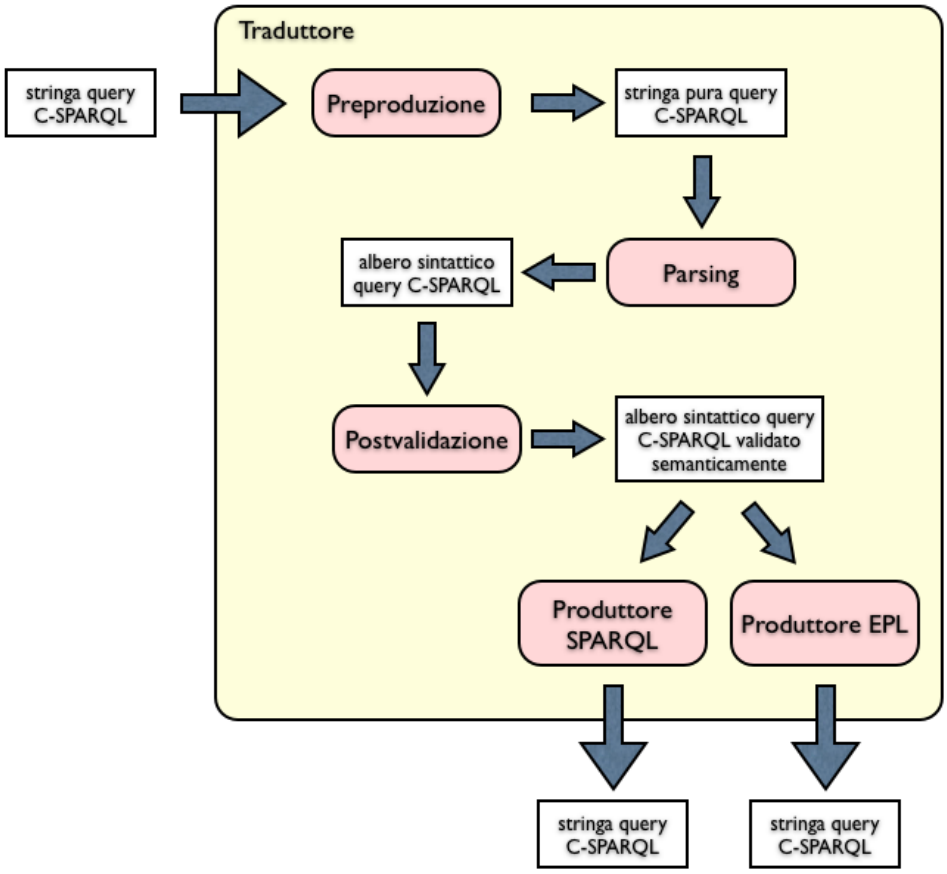

Figura 5.2: Fasi di elaborazione del flusso dati dell'applicazione

Il dato di partenza è una query scritta in C-SPARQL. Quello che si vuole ottenere sono due query, una scritta in SPARQL (che interroghi le

triple RDF) e una in EPL (che lavori sulla parte continua della query C-SPARQL). Per ottenere questo risultato l'ingresso deve subire una serie di elaborazioni.

- Fase preproduzione: lo standard SPARQL prevede che le query sia codificata come stringa di caratteri Unicode e per compatibilità con le future versioni di Unicode, la stringa può contenere caratteri espressi con il loro *code point*in UTF8. Vi è quindi la necessità di sostituire questi caratteri espressi in forma esadecimale con i loro equivalenti (Es: \U0050 viene sostituito da P).

- Fase parsing: la query è pronta per essere elaborata dal Lexer (che trasformerà la stringa di caratteri in ingresso in una stringa di tokens rappresentanti le singole parole della query) e dal Parser che ne verificheranno l'aderenza alla grammatica C-SPARQL effettuando un analisi sintattica dello stream di token. Nel caso in cui dovesse essere riconosciuto come appartenente al linguaggio C-SPARQL, il parser genererà un albero sintattico contenente nelle foglie le parole riconosciute e nei nodi intermedi le regole utilizzate per il riconoscimento dei singoli token.

- Validazione semantica: l'albero sintattico ottenuto dalla fase precedente potrebbe rappresentare una query C-SPARQL, ma non eseguibile. Infatti, così come non tutte le frasi grammaticalmente corrette hanno un significato in italiano, così potrebbero non essere rispettate alcune delle regole semantiche espresse nello standard SPARQL. Questo parziale processo di validazione semantica viene trattato approfonditamente in una sezione (5.3) ad hoc in seguito.

- Produzione SPARQL: questa fase è relativamente semplice, in quanto C-SPARQL è un linguaggio che include strettamente SPARQL. Per questo è direttamente ottenibile una query SPARQL a partire da una C-SPARQL rimuovendo le parti della query legate alla registrazione della query / stream e alla dichiarazione degli stream e delle finestre temporali nella clausola <FROM>. Questa operazione viene fatta direttamente sull'albero sintattico, tagliando i rami necessari e ristampando le foglie per ottenere di nuovo una query sotto forma di stringa pronta per essere eseguita sullo specifico motore SPARQL.

- Produzione EPL: in questa fase viene prodotta una query per ogni stream usato nella query C-SPARQL in modo che il motore di esecuzione delle query continue (ESPER) possa recuperare i dati dagli stream e analizzarli con la query SPARQL. Per questo si analizza l'albero sintattico e si recuperano i riferimenti degli stream e delle finestre degli stessi.

5.3 Sistema modulare per postparsing

Le specifiche SPARQL prevedono anche alcuni vincoli semantici per le query che non possono essere verificati in fase di parsing, in cui vengono controllati i soli vincoli sintattici. Si è reso necessario quindi creare un meccanismo a posteriori che permettesse di effettuare dei controlli semantici sull'albero sintattico creato in fase di parsing. Questo sistema di controllo a posteriori dove possedere due caratteristiche fondamentali:

- espandibilità: deve essere possibile aggiungere nuovi test in futuro.
- automaticità: devono essere eseguiti a run time sempre tutti i test disponibili per ogni query.

5.3.1 Reflection

Per fare questo si è pensato di usare la reflection, una funzionalità del linguaggio Java che permette di ispezionare le classi per ottenere informazioni sui loro attributi, sui loro metodi ed anche di eseguirli, senza conoscere a compile-time il tipo della classe che si andrà ad ispezionare e istanziare. L'idea è stata quindi quella di fare in modo da uniformare l'interfaccia dei controlli in modo che fosse possibile tramite la reflection recuperare tutte le classi con la stessa interfaccia, istanziarle ed eseguire il loro test specifico. Tutto questo senza la necessità di conoscere quante classi di test sono presenti nel package nè quale sia lo specifico controllo implementato, in quanto in caso di fallimento viene lanciata una eccezione comune a tutti i controlli. Nel caso invece in cui i controlli vengono superati, semplicemente viene restituito l'albero sintattico di partenza in quanto corretto anche da un punto di vista semantico.

5.3.2 Controlli affrontati

Per il momento, sono state soddisfatte le specifiche riguardo i seguenti controlli:

- prefisso non dichiarato o duplicato: controlla che tutti i prefissi usati all'interno della query siano stati dichiarati univocamente. Prefissi con etichetta diversa riferiti alla medesima iri sono invece consentiti così come prefissi dichiarati ma non utilizzati.

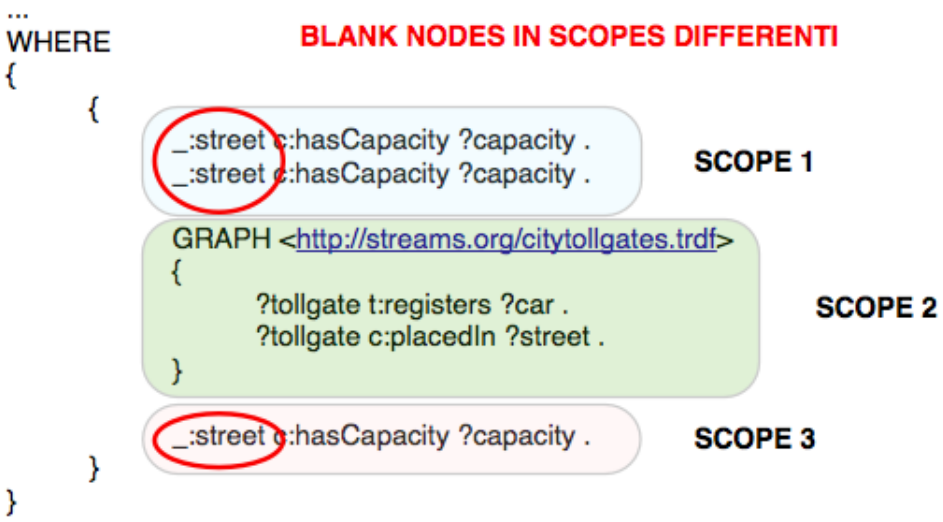

Figura 5.3: Presenza stesso blank node in scope differenti in una query C-SPARQL

- label nodo blank ripetuto: non ci possono essere due nodi blank identificati dalla stessa etichetta in due in scope diversi all'interno della stessa query. (vedere Figura 5.3)

Capitolo 6

Implementazione

Dopo aver illustrato la progettazione del traduttore e del suo parser nei capitoli precedenti, si vuole ora descriverne in dettaglio alcuni aspetti implementativi salienti.

Innanzitutto il tool che ha reso possibile la generazione automatica del parser una volta definita la grammatica C-SPARQL (e ne ha facilitato anche la stesura). La scelta e lo studio di questo strumento sono infatti stati eseguiti fin dall'inizio del lavoro in quanto punto di partenza per l'implementazione del traduttore. Nonostante la scelta sia stata fatta abbastanza velocemente viste le qualità del prodotto scelto, il suo apprendimento ha richiesto un discreto impegno soprattutto per comprendere come usare al meglio il parser che veniva generato e ad un secondo livello come modificarne la generazione in modo da ottenere il comportamento desiderato soprattutto per quanto riguarda la gestione delle eccezioni. Va detto che se da un lato la documentazione reperibile online si sia dimostrata sufficiente, le API messe a disposizione degli sviluppatori si sono rivelate talvolta lacunose per quanto riguarda la loro implementazione e documentazione e cioè ha rallentato lo sviluppo del lavoro soprattutto in fase iniziale e ha portato alla formulazione di alcune scelte progettuali (es. l'utilizzo del pattern decorator, 5.2.1).

6.1 Tecnologie utilizzate

6.1.1 ANTLR

ANTLR è un generatore di parser che fa uso del sistema di parsing LL(k). Il nome ANTLR sta per ANother Tool for Language Recognition (in italiano un altro strumento per il riconoscimento del linguaggio). Dato che ANTLR è in competizione con i generatori di parser LR, la lettura alternativa ANT(i)-LR può non essere accidentale. Le regole in ANTLR sono espresse in un formato deliberatamente simile all'EBNF. Più in generale ANTLR è uno strumento a tutto tondo per i linguaggi, fornendo un framework per

creare riconoscitori, interpreti, compilatori e traduttori a partire da descrizioni grammaticali. Fornisce anche la possibilità di inserire nella grammatica azioni in diversi linguaggi di programmazione (gli stessi in cui verrà creato il parser) tra cui: Java, C++, C# , Python. Infine fornisce un buon supporto per la costruzione e la visita degli alberi, oltre a impostare un predefinito meccanismo di reporting e recovery degli errori.

Motivi adozione

ANTLR è un tool che ha delle alternative, primo fra tutti JavaCC. Di seguito verranno elencati senza volontà di essere esaustivi, alcuni dei pro che hanno spinto per l'utilizzo di ANTLR.

Pro ANTLR:

- molto facile da usare grazie ad ANTLRWorks: è un ambiente grafico di scrittura e test delle grammatiche. Permette di visualizzare anche anteprime degli alberi ottenuti dal parsing di un ingresso di test oltre a fornire un controllo degli errori immediato mentre si scrive la grammatica.
- ha una base di utenza molto vasta, che mette a disposizione una gran quantità di casi uso e documentazione. Ma soprattutto, caso quasi unico per un software libero, esiste un intero libro di riferimento scritto dallo stesso autore di ANTLR.
- permette di generare parser in molteplici linguaggi di programmazione oltre Java, permettendo in caso di conversione del software di generare un nuovo parser senza dover modificare la grammatica.

6.1.2 Linguaggio query continue

Attualmente, come motore per la parte continua delle query C-SPARQL è stato scelto ESPER, che si avvale del linguaggio EPL per la scrittura delle interrogazioni. Queste ultime hanno una struttura ricorrente costante tranne per gli elementi tra <> che invece sono dinamici e vengono recuperati dalla query C-SPARQL.

```
select subject, predicate, object, timestamp from STREAM <iri> <window>
```

6.2 Implementazione e testing grammatica

Per quanto riguarda la stesura della grammatica, preso atto del linguaggio di destinazione scelto per il parser, Java, si è optato, per soddisfare le stringenti esigenze di testing della grammatica, per l'adozione di un ambiente di testing separato basato su JUnit. Questa scelta se da un lato ha reso più robusto

il processo di sviluppo della grammatica, dall'altro ne ha complicato la fase iniziale in cui è stato necessario fare il setup di questo ambiente.

Infatti di default, il parser generato automaticamente da ANTLR intercettava le eccezioni in caso di problemi mascherandole all'esterno e impedendone la gestione da parte delle classi di test. Per modificare quindi questo comportamento, oltre alle regole è stato necessario aggiungere tramite ANTLRWorks del codice Java innestato nel file della grammatica per modificare il parser che veniva generato a valle. È stato anche aggiunto del codice per per l'intestazione delle classi lexer e parser (riguardante il package). Grazie a questi innesti resi possibili dalle potenzialità di ANTLR è stato agevole testare velocemente la grammatica C-SPARQL durante la sua stesura ed evoluzione. Va detto che l'implementazione della modifica della gestione delle eccezioni da parte del parser ha richiesto un discreto tempo per essere messa in atto a causa della complessità e scarsa chiarezza delle API di ANTLR.

Ecco di seguito il codice inserito nel file della grammatica:

```
options
{
        output=AST;
}
@header //not autocopied in Lexer, just in Parser
{
        package querytester;
}
@lexer::header
{
        package querytester;
}
@members
{
        protected void mismatch(IntStream input, int ttype, BitSet
            follow) throws RecognitionException
        {
                throw new MismatchedTokenException(ttype, input);
        }

        protected Object recoverFromMismatchedToken(IntStream input, int
             ttype, BitSet follow) throws RecognitionException
    {
        // must throw the exception to notify junit
        throw new MismatchedTokenException(ttype, input);
    }
}

// Alter code generation so catch-clauses get replace with this action.
@rulecatch {
catch (RecognitionException e) {
        throw e;
}
}
```

6.2.1 Descrizione ambiente test

Per quanto riguarda l'ambiente di test, è composto da una classe contente un metodo per ogni caso di test. Le query da testare sono contenute ognuna su un singolo file di testo che viene letto a runtime dal metodo relativo e viene eseguito il parsing dell'interrogazione che contiene. I casi presenti possono essere di due tipologie: positivi, nel caso in cui la query contenuta nel file sia corretta; negativi, nel caso in cui la query usata come caso di test sia volutamente errata. I metodi che devono eseguire i due tipi di test sono di conseguenza differenti: nel primo caso propagheranno l'eccezione nel caso in cui si verifichi, mentre nel secondo verrà mascherata, come mostrato nel codice che segue.

```
/**
* Test of @nameFile@ method. #POSITIVE#
*/
@Test
public void @nameMethod@() throws Exception {
    System.out.println();
    System.out.println("@nameFile@");
    String fileName = positivePath+"@nameFile@";
    String query = readQuery(fileName);
    System.out.println(query);
    //Parse query
    TreeBox tb = Helper.parse(query);
}

/**
* Test of @nameFile@ method. #NEGATIVE#
*/
@Test
public void @nameMethod@() throws Exception {
    System.out.println();
    System.out.println("@nameFile@");
    String fileName = negativePath+"@nameFile@";
    String query = readQuery(fileName);
    System.out.println(query);
    try{
        //Parse query
        TreeBox tb = Helper.parse(query);
    }
    catch (Exception e){
        System.out.println(e.toString());
        return;
    }
    throw new org.antlr.runtime.RecognitionException ();
}
```

Preso però atto della grande mole di casi di test forniti inizialmente a corredo della grammatica SPARQL (intorno ai 200 casi), si è creato un meccanismo che fosse in grado di generare automaticamente questa classe di test con tutti i suoi metodi a partire dalla presenza o meno a runtime dei file stessi che contenevano le query e seguendo i due template mostrati precedentemente. In questo modo è stato possibile creare nuovi casi ti test semplicemente scrivendone le interrogazioni in file di testo e rigenerando la

classe di test.

6.3 Uso reflection

Come descritto nel capitolo 5, una volta eseguito correttamente il parsing ed ottenuto un albero sintattico, vi è la necessità di effettuare su di esso alcuni controlli semantici. Si è voluto quindi trovare un modo con cui essere in grado di istanziare le classi che contengono i metodi di controllo semantico ed eseguirli automaticamente, senza essere a conoscenza a priori di quante siano queste classi e di che controllo implementino. Per soddisfare questa specifica è stata creata una gerarchia di classi che vede nell'uso della reflection il suo fulcro, come di seguito riportato.

Si è iniziato con il creare un'interfaccia che accomunasse per tutte le classi il metodo di controllo, come di seguito riportato:

```
/**
 * The Interface describe the method needed to post validate a syntactic
      tree resulted from a query parsing.
 * @author Marco
 */
public interface TreeCheckerInterface {
    /**
     * The method validate the tree according to a semantic rule of the
        language.
     * If the tree doesn't pass the check an exception will be thrown
        with the reason as text.
     * @param t Tree to be validated
     * @throws Exception Generic exception that contains the text which
        explain the reason of the fail check.
     */
    public void treeCheck (TreeBox t) throws PostProcessingException;
}
```

Grazie a questa interfaccia tutti le classi che la implementeranno in futuro vedranno eseguito automaticamente il loro metodo di check specifico. Ad esempio si riporta di seguito il metodo che controlla il prefisso della query C-SPARQL.

```
public class PrefixChecker implements TreeCheckerInterface{
    /**
     * Check if all the prefix used in the query are declared and if
        there are not prefix with same name.
     */
    public void treeCheck (TreeBox t) throws PostProcessingException {
        //Get all the prefix
        List <TreeBox> prefixList = t.getNodesByText("prefixDecl");
        //Map use id as key and iri as value
        HashMap prefixMap = new HashMap();
        //Build up the map and check if the same id is declared twice
        for (TreeBox tbx : prefixList) {
            if (prefixMap.containsKey(tbx.getChild(1).getText()))
                //There is yet a prefix with this name
                throw new PostProcessingException("Prefix "+tbx.getChild
                    (1).getText()+"already declared twice in the query")
                    ;
```

```
            else
                //Add the prefix id as key and iri as value
                prefixMap.put(tbx.getChild(1).getText(), tbx.getChild(2)
                    .getText());
        }

        //Now check everytime a prefix is used in the query if it's
            present in the prefixMap
        List <TreeBox> prefixedNames = t.getNodesByText("prefixedName");

        for (TreeBox tbx : prefixedNames) {
            Pattern pattern = Pattern.compile(".*:");
            Matcher matcher = pattern.matcher(tbx.getChild(0).getText())
                ;
            if (matcher.find()) {
                //Check if the prefix has been declared before
                if (!prefixMap.containsKey(matcher.group())) {
                    //Prefix used without declaration
                    throw new PostProcessingException("Prefix  used
                        without declaration");
}}}}}
```

Una volta che si hanno le classi che implementano l'interfaccia TreeCheckerInterface, la classe che effettua il parsing richiede tutte queste classi e le istanzia, eseguendone poi il metodo di check.

```
/**
     * Run the method <code>treeCheck(TreeBox tbx)</code> of all the
         classes in the package which implements <code>
         TreeCheckerInterface</code>.
     * The classes are found through reflection.
     * @param tbx
     * @throws Exception, in case <code>tbx</code> don't pass at least
         one of the checks.
     */
    private static void postProcessing (TreeBox tbx)throws
        PostProcessingException {
        //Post Validate the tree with all classes which implements
            TreeCheckerInterface
        List<Class> classList = ClassFinder.find(Helper.class.getPackage
            ().getName(),"TreeCheckerInterface");
        for (Class c : classList){
            try {
            //Make an instance of the class
            TreeCheckerInterface i = (TreeCheckerInterface)c.newInstance
                ();
            //Call his method
            i.treeCheck(tbx);
            }
            catch (InstantiationException ie) {
                ie.toString();
            }
            catch (IllegalAccessException iae) {
                iae.toString();
            }
        }
    }
```

Come sopra mostrato, la ricerca di queste classi è affidata ad una classe ad hoc, `ClassFinder`, attraverso il metodo `ClassFinder.find()` a cui viene

passato come unico parametro l'interfaccia da ricercare nelle classi. È infatti la classe `ClassFinder` che usa effettivamente la reflection, come di seguito riportato nel blocco try, cercando tra le interfacce delle classi all'interno del package se sia presente `TreeCheckerInterface`.

```
...
try     {
                        // Try to create an instance of the object
                        Class tempClass = Class.forName(pckgname+"."+
                            classname);
                        // Get interfaces implemented by each class
                            found
                        Class [] interfaces = tempClass.getInterfaces();
                        for (Class c : interfaces){
                            if ( c.getSimpleName().equals(interfaceName)
                                 ) {
                                //System.out.println("finder-"+c.
                                    getSimpleName());
                                //Found a class which implements the
                                    target interface
                                result.add(tempClass);
                            }
        }
```

6.4 Schemi UML

6.4.1 Class diagram completo

Verranno ora descritte le parti più significative dell'applicazione attraverso degli schemi di tipo classdiagram corredati da una breve descrizione delle classi stesse.

Nella prima figura (6.1) sono mostrate le classi centrali del traduttore. Ovvero la classe `Helper` che è quella che è adibita a svolgere la funzionalità di parsing della query C-SPARQL che riceve in ingresso attraverso il metodo `parse(String query)`. All'interno di questo metodo che è l'unico metodo pubblico, vengono chiamati in sequenza il metodo `preprocessQuery` per sostituire i *code point* UNICODE codificati in UTF8 con gli equivalenti caratteri, i metodi delle classi `CSparqlLexer` e `CSparqlParser` generate da ANTLR per effettuare il parsing vero e proprio generando l'albero sintattico che viene poi passato al metodo `postProcessing()` per la validazione semantica che restituisce solo un eccezione in caso di fallimento di uno dei test.

L'albero sintattico così generato è fatto da nodi di tipo `TreeBox` che sono decorati da un etichetta rappresentata dalla classe `Label` che memorizza il posizionamento dei nodi all'interno della struttura complessiva. TreeBox ha poi tutta una fitta serie di metodi che permettono la navigazione all'interno della sua struttura e il recupero delle informazioni rilevanti per i produttori a valle, SPARQL ed EPL (es. `set <StreamInfo> getStreams()`).

In figura 6.2 ci sono invece le classi che sono coinvolte nella validazione semantica dell'albero sintattico effettuato all'interno del metodo `parse()`

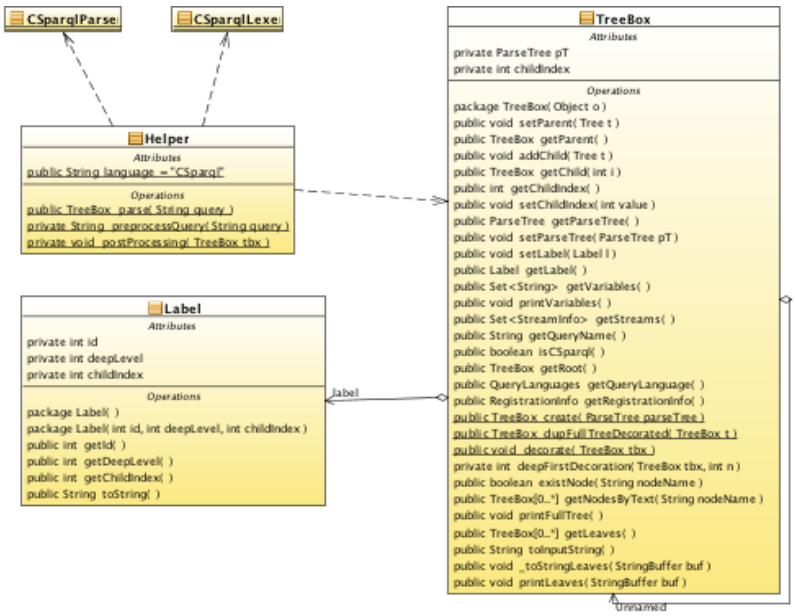

Figura 6.1: ClassDiagram: nucleo del parser

della classe Helper. La classe `ClassFinder` attraverso il metodo `find()` consente il recupero, tra tutte le classi presenti all'interno del package passato come parametro, di quelle che implementano l'interfaccia anch'essa passata come parametro. Questa operazione viene eseguita con l'ausilio della reflection precedentemente spiegata (6.3). Sono anche mostrate le due classi che attualmente implementano l'interfaccia generica `TreeChecker` e la relativa eccezione che generano in caso di mancato superamento del loro specifico test.

In figura 6.3 vengono mostrate le classi che realizzano il traduttore SPARQL. L'impostazione scelta è stata quella che consentisse una massima flessibilità in futuro, quindi è presente un interfaccia di riferimento `SparqlProducer` che contiene due metodi, uno, `produceSparql(TreeBox)`, per tagliare l'albero sintattico delle parti continue che sono estranee a SPARQL e l'altro, `getTextWithSymbols(TreeBox)`, che serve ad ottenere la stringa corrispondente al token memorizzato in un nodo ripristinando quei caratteri speciali che non vengono memorizzati all'interno dell'albero sintattico (ad esempio identificatori di variabili "?"). Quest'ultimo viene implementato per comodità in una classe astratta `DefaultSparqlProducer` estesa poi a sua volta dalla classe `SparqlProducer1_0` che implementa poi concretamente il metodo di produzione dell'interfaccia di riferimento.

Infine, in figura 6.4 è mostrata la gerarchia di classi che implementa la generazione della query EPL a partire dall'albero sintattico della que-

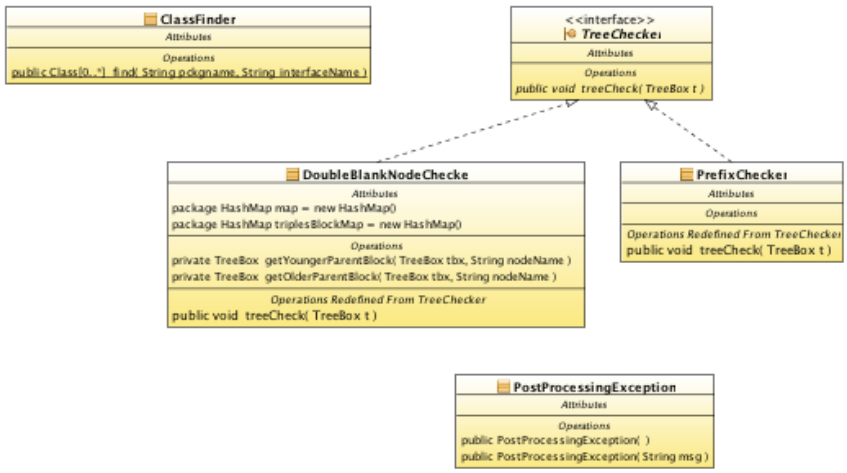

Figura 6.2: ClassDiagram: postparsing

ry C-SPARQL. La gerarchia, composta dall'interfaccia `EplProducer`, dalla sua implementazione astratta `DefaultEplProducer` e concreta `EplProducer1_0` rispecchia quella già spiegata per il produttore di SPARQL. L'unica differenza degna di nota è che in questo caso vengono richiesti alla classe TreeBox le parti continue della query che vengono oggettivati nelle classi `StreamInfo` e `RegistrationInfo`.

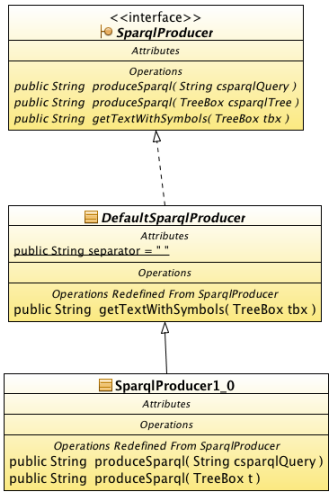

Figura 6.3: ClassDiagram: produttore SPARQL

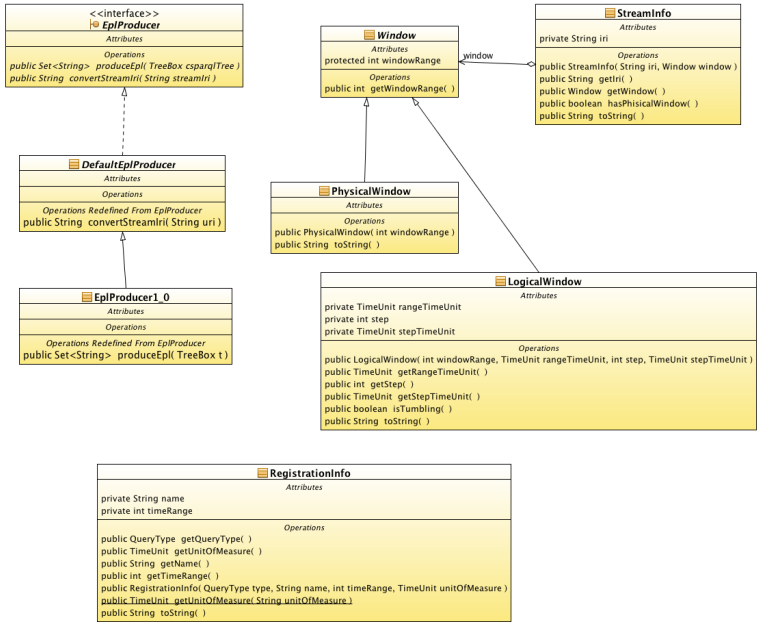

Figura 6.4: ClassDiagram: produttore EPL

Capitolo 7

Validazione sperimentale

In questo capitolo verranno mostrati degli esempi applicativi del traduttore C-SPARQL descritto nei capitoli precedenti (6 , 5 , 4). In particolare verrà mostrato prima un esempio che metta in luce le caratteristiche principali di C-SPARQL 1.0 e poi una serie di piccoli esempi sulle funzionalità aggiunte in C-SPARQL 1.1. Si darà una breve spiegazione testuale dell'interrogazione per poi passare ad evidenziare le regole grammaticali peculiari coinvolte nel riconoscimento e il relativo pezzo di albero sintattico.

7.1 Parte C-SPARQL 1.0

Per presentare tutte le caratteristiche salienti di C-SPARQL 1.0 si è scelto l'esempio seguente, in cui lo scopo dell'interrogazione è quello di estrarre, per tutte le strade, la lista delle automobili in transito, confrontando contemporaneamente i dati provenienti da due stream differenti, uno proveniente dai caselli e uno dalle telecamere.

```
REGISTER QUERY FullStreets AS
PREFIX c: <http://linkedurbandata.org/city#>
PREFIX t: <http://linkedurbandata.org/traffic#>
SELECT DISTINCT ?street ?car
FROM STREAM <http://streams.org/citytollgates.trdf> [ RANGE 5m TUMBLING]
FROM STREAM <http://streams.org/citycameras.trdf> [ RANGE TRIPLES 4 ]
WHERE
{
        {
                GRAPH <http://streams.org/citytollgates.trdf>
                {
                        ?tollgate t:registers ?car .
                        ?tollgate c:placedIn ?street .
                }
        }
        UNION
        {
                GRAPH <http://streams.org/citytollgates.trdf>
                {
                        ?camera t:registers ?car .
                        ?camera c:placedAt ?light .
```

```
                        ?light c:crossing ?street .
                }
        }
}
```

7.1.1 Regole specifiche C-SPARQL usate

Parte che deve riconoscere la registrazione della query

```
queryWithReg
        :       registration? query EOF
        ;
registration
        : REGISTER (QUERY|STREAM) queryName queryRange? AS
        ;
query
    : prologue ( selectQuery | constructQuery | describeQuery | askQuery
        )
    ;
queryName
        : QUERY_NAME
        ;
queryRange
        : COMPUTED_EVERY timeRange
        ;
```

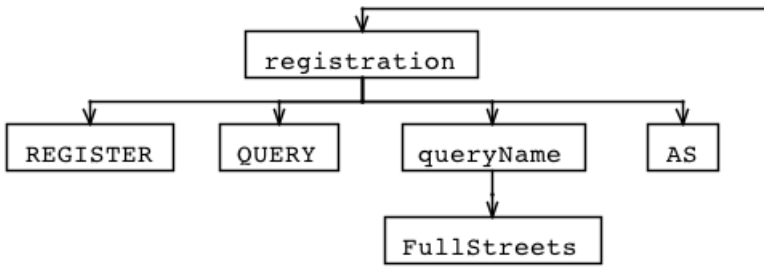

Figura 7.1: Parte registration della query

Parte della grammatica si occupa del prologo contenente la dichiarazione dei prefissi usati nella query in sostituzione di lunghe iri.

```
prologue
    : baseDecl? prefixDecl*     ;
baseDecl
    : BASE IRI_REF    ;
prefixDecl
    : PREFIX PNAME_NS IRI_REF    ;
```

Regole che lavorano sul tipo di query SELECT e in particolare sulla clausola FROM che sono peculiari di SPARQL e C-SPARQL.

```
selectQuery
    : SELECT ( DISTINCT | REDUCED )? ( (var | OPEN_BRACE (
        newVarFromExpression ) CLOSE_BRACE )+ | ASTERISK ) datasetClause
        * whereClause solutionModifier    ;
```

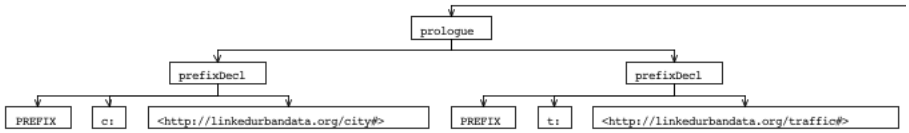

Figura 7.2: Parte prologue della query

```
datasetClause
    : datasetClauseStd
    | datasetClauseStream    ;
datasetClauseStream
        : FROM NAMED? STREAM ( defaultGraphClause | namedGraphClause )
            range    ;
range
        : OPEN_SQUARE_BRACE RANGE window CLOSE_SQUARE_BRACE     ;
window
        : physicalWindow
        | logicalWindow         ;
logicalWindow
        :       timeRange windowOverlap ;
physicalWindow
        :       TRIPLES INTEGER ;
windowOverlap
        :       STEP timeRange
        |       TUMBLING        ;
timeRange
        : TIME_RANGE    ;
TIME_RANGE
        : INTEGER TIME_UNIT     ;
TIME_UNIT
        : 'ms'
        | 'h'
        | 's'
        | 'm'
        | 'd'   ;
```

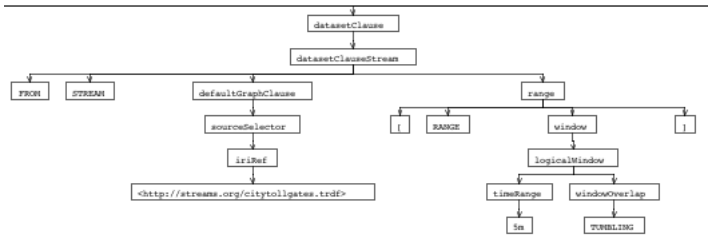

Figura 7.3: Parte FROM della query

7.2 Parte C-SPARQL 1.1

Si presenterà ora un esempio per dimostrare come vengano elaborate dal parser C-SPARQL delle interrogazioni che sfruttano le peculiarità introdotte

nella versione 1.1 del linguaggio SPARQL. Questo esempio si baserà su dati statici RDF definiti qui di seguito:

```
@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 .
```

7.2.1 Aggregazioni - espressioni nella clausola SELECT

Si presenta ora un interrogazione che utilizza l'aggregazione `sum` insieme all'inserimento di una espressione all'interno della clausola SELECT. L'interrogazione ha il significato di recuperare per ogni organizzazione, la somma dei prezzi dei libri degli scrittori affiliati, ponendo come condizione che il totale sia maggiore di 10.

QUERY

```
PREFIX : <http://books.example/>
SELECT ?org (SUM(?lprice) AS ?totalPrice)
WHERE {
  ?org :affiliates ?auth .
  ?auth :writesBook ?book .
  ?book :price ?lprice .
}
GROUP BY ?org
HAVING (SUM(?lprice) > 10)
```

In figura 7.4 è riportato un estratto dell'albero sintattico prodotto dal parser. Come è evidenziato, l'aggregazione viene riconosciuta come tale, ma essendo una delle possibili produzioni della regola `expression`, può essere inserita nella clausola SELECT e legata ad una nuova variabile. Infatti nella parte superiore dell'albero viene riconosciuta l'espressione e l'assegnamento del risultato ad una nuova variabile.

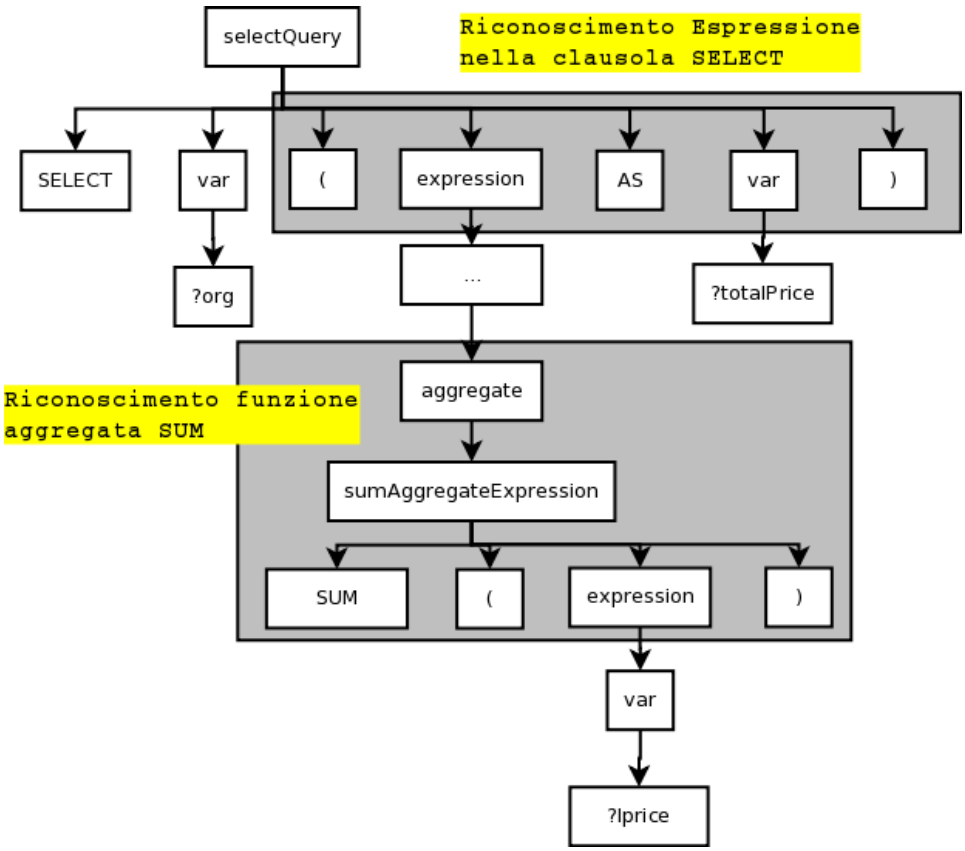

Figura 7.4: Esempio Aggregazione

7.3 Produttore SPARQL

In questa sezione verrà mostrato il risultato ottenuto con l'utilizzo del produttore SPARQL fornendo in ingresso l'albero sintattico prodotto dalla query usata nella sezione 7.1. Come si può vedere questa è l'interrogazione C-SPARQL privata della parte continua riguardante gli streams.

```
PREFIX c: <http://linkedurbandata.org/city#>
PREFIX t: <http://linkedurbandata.org/traffic#>
SELECT DISTINCT ?street ?car
WHERE {
        {
        GRAPH <http://streams.org/citytollgates.trdf>
                { ?tollgate t:registers ?car . ?tollgate c:placedIn ?
                    street .   }
        }
        UNION
        {
        GRAPH <http://streams.org/citytollgates.trdf>
                { ?camera t:registers ?car . ?camera c:placedAt ?light .
                     ?light c:crossing ?street . }
}       }
```

7.4 Produttore EPL

Per mostrare il risultato del produttore EPL si è utilizzato sempre la query C-SPARQL della sezione 7.1.

```
[select subject, predicate, object, timestamp from STREAM
    http___streams_org_citycameras_trdf.win:length(4), select subject,
    predicate, object, timestamp from STREAM
    http___streams_org_citytollgates_trdf.win:time_batch(300 seconds)]
```

Capitolo 8

Conclusione e sviluppi futuri

Il linguaggio C-SPARQL si è rivelato essere un perfetto connubio tra il mondo dei DSMS e quello del web semantico legato alle interrogazioni su dati RDF tramite SPARQL. In seguito al lavoro che è stato presentato è ora possibile, a partire da una interrogazione espressa in linguaggio C-SPARQL, ottenere una query SPARQL e una EPL da eseguire sui rispettivi engine.

Questa operazione di traduzione ha richiesto anche lo studio e l'implementazione preliminare di un componente che fosse in grado di riconoscere una query C-SPARQL come tale, cioè ne verificasse l'aderenza sintattica e semantica alle specifiche del linguaggio. Per questo motivo si è rivelata necessaria l'esigenza di implementare un parser, generato automaticamente tramite il tool ANTLR, a partire dalla scrittura della sua grammatica. Quest' ultima è stata aggiornata in corso d'opera dal W3C ad una nuova versione del linguaggio, ora 1.1, che ha introdotto delle consistenti novità a livello di costrutti e sintassi.

8.1 Obiettivi raggiunti

8.1.1 Scrittura grammatica C-SPARQL 1.1

È stata redatta una grammatica completa per C-SPARQL aderente alla versione 1.1 del linguaggio. La stesura è stata evolutiva rispetto a quella già pubblicata per SPARQL e ha richiesto l'aggiunta delle sole regole che implementavano le funzionalità peculiari di C-SPARQL.

La grammatica ottenuta è rimasta necessariamente consistente con le specifiche SPARQL ed è in grado di riconoscere una query in quel linguaggio, in quanto è strettamente incluso in quello C-SPARQL. La garanzia di questa proprietà è stata ottenuta attraverso la creazione di un ambiente di test che ha permesso di validare la grammatica durante il suo sviluppo attraverso l'esecuzione di una notevole mole di casi di test sia SPARQL che C-SPARQL.

La grammatica ottenuta, oltre che corretta, si è anche rivelata anche comoda da utilizzare al momento della traduzione, in quanto attraverso le

sue produzioni si genera un albero sintattico in cui è facile identificare e quindi utilizzare le componenti continue delle query C-SPARQL rispetto a quelle statiche. Grazie a questa grammatica è stato quindi possibile generare automaticamente tramite il tool ANTLR un parser efficiente ed efficace.

8.1.2 Progettazione e implementazione del traduttore

La seconda parte del lavoro svolto e descritto in questo documento è stato quello di implementare due traduttori che fossero in grado, a partire da un albero sintattico rappresentante una query C-SPARQL, di ottenere rispettivamente una interrogazione SPARQL (contenente la parte statica della query di partenza) e una EPL (contenente la parte continua). L'approccio utilizzato per la progettazione è stato quello di dare massima flessibilità, soprattutto per quanto riguarda da un lato lo sviluppo futuro del linguaggio, dall'altro le possibili ottimizzazioni proposte in [8] che possono essere messe in pratica. Per questo i traduttori sono stati implementati separatamente in modo da poter essere facilmente modificati ed estesi uno indipendentemente dall'altro.

Il traduttore SPARQL si è rivelato di immediata realizzazione, in quanto al momento opera direttamente sull'albero sintattico ricercando e tagliando le parti continue dell'interrogazione C-SPARQL e rimanendo con un albero sintattico SPARQL. Questo modo di procedere si è rivelato molto efficace ed efficiente, sfruttando appieno le possibilità offerte dall'oggetto `TreeBox` che implementa l'albero e che è stato creato ex novo proprio per consentire questa flessibilità di utilizzo.

Il traduttore EPL invece compone semplicemente una stringa a partire dagli elementi continui che l'albero sintattico gli fornisce. Questi elementi (ad esempio lo stream dati) sono stati oggettivizzati per garantire la massima flessibilità futura nella modifica del linguaggio SPARQL in quando non si deve preoccupare di recuperare le informazioni sugli stream e sulla registrazione della query in quanto gli vengono fornite sempre attraverso gli stessi oggetti (es. `StreamInfo`). Gli aggiornamenti del linguaggio andranno quindi ad impattare esclusivamente sulla classe `TreeBox` che ne è direttamente legata in quanto ne rappresenta le interrogazioni.

8.2 Sviluppi futuri

8.2.1 Aggiornamento grammatica C-SPARQL con specifiche SPARQL

Il lavoro di riconoscimento e traduzione di interrogazioni ha le sue fondamenta nelle specifiche del linguaggio SPARQL (di cui C-SPARQL è una evoluzione). Sarà quindi necessario aggiornare il linguaggio e di conseguenza la grammatica che dovrà riconoscerlo qualora escano delle nuove specifiche.

In particolare a fine lavoro è stata rilasciata una versione draft delle nuove specifiche[1] in cui viene aggiunta una modalità di espressione delle negazioni. Viene data la possibilità di rimuovere delle soluzioni da un binding attraverso la parola chiave MINUS di cui si riporta uno spaccato di un esempio:

```
SELECT * { ?s ?p ?o MINUS { ?x ?y ?z } }
```

Questa modifica impatterà, esclusivamente sul parser per quanto riguarda il suo riconoscimento, in quanto non va a coinvolgere la parte continua che è quella critica invece per i produttori.

8.2.2 Ottimizzazioni con traduzioni dipendenti dal contesto

Per ragioni di scelta delle priorità si è deciso di effettuare una traduzione molto aderente all'interrogazione C-SPARQL di partenza, senza cioè effettuare alcun tipo di ottimizzazione. Sarebbe possibile infatti studiare e implementare delle ottimizzazioni nell'esecuzione dei filtri o delle aggregazioni.

Per quanto riguarda i filtri, si potrebbe provare a effettuarli direttamente sugli streams attraverso la query EPL, in modo da aumentare le prestazioni di esecuzione complessiva della query C-SPARQL limitando i dati su cui deve poi operare la query SPARQL. Alcuni dati sperimentali [9] hanno dimostrato come l'incremento delle prestazioni sia tangibile sia all'aumentare della finestra dei dati da considerare sia all'aumentare della dimensione dei dati da processare.

Per quanto riguarda le aggregazioni, si potrebbe effettuare anche qui un'operazione di push, eseguendole direttamente sugli streams. Questa pratica si reputa essere sempre vantaggiosa, specialmente nel caso di presenza del modificatore DISTINCT nella SELECT, perchè può essere applicato subito sui risultati parziali degli aggregati che arrivano dai diversi streams. Va detto che a tale scopo le funzioni aggregate possono essere classificate nelle tre tipologie esposte di seguito.

- Completamente distribuibili: una funzione aggregata in cui gli argomenti e le variabili di raggruppamento fanno riferimento a uno o più streams, ma ogni possibile binding delle variabili appartiene ad un solo stream. Quindi le funzioni sono distribuite sui vari streams e il la query C-SPARQL ritorna l'unione dei risultati parziali. Un esempio potrebbe essere il calcolo della somma degli importi delle transazioni raggruppate per venditore in ogni mercato, avendo uno stream per ogni mercato.

- Parzialmente distribuibili: è il caso in cui un aggregazione abbia gli argomenti e solo un sottoinsieme delle variabili di raggruppamento che

[1] draft version del 1 giugno 2010

appartiene a uno o più streams e c'è una distribuzione della computazione della funzione aggregata sugli streams su parti delle relazioni[2]. Un esempio può essere calcolare la somma di tutte le transazioni eseguite raggruppate per venditore, rispetto a tutti i mercati, mantenendo però gli streams associati ad un solo mercato. In questo caso quindi si ottiene solo una somma per venditore per ogni mercato e la somma delle somme porta al risultato ricercato.

- Non distribuibili: quando un aggregato non rientra in nessuno dei casi precedenti. Un esempio può essere il conteggio degli importi distinti di varie transazioni.

Viste quindi le potenzialità in termini prestazionali delle ottimizzazioni precedentemente proposte si è dato ampio margine di miglioramento in tal senso progettando e implementando un traduttore estremamente flessibile in modo da garantire la massima aggiornabilità dei suoi componenti.

[2]nel senso espresso in [8]

Appendice A

Grammatica completa C-SPARQL 1.1 per ANTLR

```
//Estensione grammatica Sparql.g

//Author: Marco Regaldo

grammar CSparql;

options
{
output=AST;
}
@header //not autocopied in Lexer, just in Parser
{
package querytester;
}
@lexer::header
{
package querytester;
}
@members
{
protected void mismatch
(IntStream input, int ttype, BitSet follow)
throws RecognitionException
{
throw new MismatchedTokenException(ttype, input);
}

protected Object recoverFromMismatchedToken
(IntStream input, int ttype, BitSet follow)
throws RecognitionException
    {
        // must throw the exception to notify junit
        throw new MismatchedTokenException(ttype, input);
    }
}

// Alter code generation so catch-clauses
//get replace with this action.
@rulecatch {
catch (RecognitionException e) {
throw e;
}
}

// $<Parser
queryWithReg
: registration? query EOF ;

registration//Idenfity CSparql query
: REGISTER (QUERY|STREAM) queryName queryRange? AS ;

query
    : prologue ( selectQuery | constructQuery
    | describeQuery | askQuery )    ;
```

```
queryName
: QUERY_NAME ;

queryRange
: COMPUTED_EVERY timeRange  ;

prologue
    : baseDecl? prefixDecl*    ;

baseDecl
    : BASE IRI_REF    ;

prefixDecl
    : PREFIX PNAME_NS IRI_REF    ;

selectQuery
    : SELECT ( DISTINCT | REDUCED )?
    ( (var | OPEN_BRACE (newVarFromExpression )
    CLOSE_BRACE )+ | ASTERISK ) datasetClause*
    whereClause solutionModifier    ;

newVarFromExpression
: expression AS var ;

constructQuery
    : CONSTRUCT constructTemplate
    datasetClause* whereClause solutionModifier    ;

describeQuery
    : DESCRIBE ( varOrIRIref+ | ASTERISK )
    datasetClause* whereClause? solutionModifier    ;

askQuery
    : ASK datasetClause* whereClause   ;

datasetClause
    : datasetClauseStd
    | datasetClauseStream    ;

datasetClauseStd
: FROM ( defaultGraphClause | namedGraphClause ) ;

datasetClauseStream
: FROM NAMED? STREAM
( defaultGraphClause | namedGraphClause ) range ;

range
: OPEN_SQUARE_BRACE RANGE
window CLOSE_SQUARE_BRACE ;

window
: physicalWindow
| logicalWindow  ;

logicalWindow
: timeRange windowOverlap ;

physicalWindow
: TRIPLES INTEGER ;

windowOverlap
: STEP timeRange
| TUMBLING ;

timeRange
: TIME_RANGE ;

defaultGraphClause
    : sourceSelector    ;

namedGraphClause
    : NAMED sourceSelector    ;

sourceSelector
    : iriRef    ;

whereClause
    : WHERE? groupGraphPattern   ;

solutionModifier
    : groupBy? orderClause? limitOffsetClauses?   ;

groupBy
: GROUP BY var+ having?;
```

```
having
: HAVING brackettedExpression ;

limitOffsetClauses
    : ( limitClause offsetClause?
    | offsetClause limitClause? );

orderClause
    : ORDER BY orderCondition+    ;

orderCondition
    : ( ( ASC | DESC ) brackettedExpression )
    | ( constraint | var )    ;

limitClause
    : LIMIT INTEGER    ;

offsetClause
    : OFFSET INTEGER    ;

groupGraphPattern
    : OPEN_CURLY_BRACE triplesBlock?
     ( ( graphPatternNotTriples | filter | subquery )
     DOT? triplesBlock? )* CLOSE_CURLY_BRACE    ;

subquery
: OPEN_CURLY_BRACE selectQuery CLOSE_CURLY_BRACE ;

triplesBlock
    : triplesSameSubject ( DOT triplesBlock? )?    ;

graphPatternNotTriples
    : optionalGraphPattern | groupOrUnionGraphPattern
    | graphGraphPattern | existElt | nonExistElt    ;

existElt
: EXISTS groupGraphPattern ;

nonExistElt
: NOT_BY_WORDS EXISTS groupGraphPattern ;

optionalGraphPattern
    : OPTIONAL groupGraphPattern    ;

graphGraphPattern
    : GRAPH varOrIRIref groupGraphPattern    ;

groupOrUnionGraphPattern
    : groupGraphPattern ( UNION groupGraphPattern )* ;

filter
    : FILTER constraint    ;

constraint
    : brackettedExpression | builtInCall | functionCall ;

functionCall
    : iriRef argList    ;

argList
    : ( OPEN_BRACE CLOSE_BRACE | OPEN_BRACE expression
    ( COMMA expression )* CLOSE_BRACE )    ;

constructTemplate
    : OPEN_CURLY_BRACE constructTriples? CLOSE_CURLY_BRACE ;

constructTriples
    : triplesSameSubject ( DOT constructTriples? )?    ;

triplesSameSubject
    : varOrTerm propertyListNotEmpty | triplesNode
    propertyList    ;

propertyListNotEmpty
    : verb objectList ( SEMICOLON ( verb objectList )? )* ;

propertyList
    : propertyListNotEmpty?    ;

objectList
    : object ( COMMA object )*    ;

object
```

```
    : graphNode    ;

verb
    : varOrIRIref
    | A    ;

triplesNode
    : collection
    | blankNodePropertyList    ;

blankNodePropertyList
    : OPEN_SQUARE_BRACE propertyListNotEmpty
    CLOSE_SQUARE_BRACE    ;

collection
    : OPEN_BRACE graphNode+ CLOSE_BRACE    ;

graphNode
    : varOrTerm | triplesNode    ;

varOrTerm
    : var
    | graphTerm    ;

varOrIRIref
    : var | iriRef    ;

var
    : VAR1
    | VAR2    ;

graphTerm
    : iriRef
    | rdfLiteral
    | numericLiteral
    | booleanLiteral
    | blankNode
    | OPEN_BRACE CLOSE_BRACE    ;

expression
    : conditionalOrExpression    ;

conditionalOrExpression
    : conditionalAndExpression
    ( OR conditionalAndExpression )*   ;

conditionalAndExpression
    : valueLogical ( AND valueLogical )*    ;

valueLogical
    : relationalExpression    ;

relationalExpression
    : numericExpression ( EQUAL numericExpression
    | NOT_EQUAL numericExpression | LESS numericExpression
    | GREATER numericExpression
    | LESS_EQUAL numericExpression
    | GREATER_EQUAL numericExpression )?   ;

numericExpression
    : additiveExpression    ;

additiveExpression
    : multiplicativeExpression (  (MINUS | PLUS )
    multiplicativeExpression  | numericLiteralPositive
    | numericLiteralNegative )* ;

multiplicativeExpression
    : unaryExpression
    ( (ASTERISK | DIVIDE) unaryExpression )* ;

unaryExpression
    : NOT primaryExpression
    | PLUS primaryExpression
    | MINUS primaryExpression
    | primaryExpression   ;

primaryExpression
    : bracketted Expression
    | builtInCall
    | iriRefOrFunction
    | rdfLiteral
    | numericLiteral
    | booleanLiteral
```

```
    | var
    | timestampCall
    | aggregate    ;

timestampCall
: TIMESTAMP OPEN_BRACE var
(iriRef | graphPatternNotTriples)? CLOSE_BRACE ;

brackettedExpression
    : OPEN_BRACE expression CLOSE_BRACE    ;

builtInCall
    : STR OPEN_BRACE expression CLOSE_BRACE
    | LANG OPEN_BRACE expression CLOSE_BRACE
    | LANGMATCHES OPEN_BRACE expression
    COMMA expression CLOSE_BRACE
    | DATATYPE OPEN_BRACE expression CLOSE_BRACE
    | BOUND OPEN_BRACE var CLOSE_BRACE
    | SAMETERM OPEN_BRACE expression
    COMMA expression CLOSE_BRACE
    | ISIRI OPEN_BRACE expression CLOSE_BRACE
    | ISURI OPEN_BRACE expression CLOSE_BRACE
    | ISBLANK OPEN_BRACE expression CLOSE_BRACE
    | ISLITERAL OPEN_BRACE expression CLOSE_BRACE
    | regexExpression
    | existsFunc
    | notExistsFunc    ;

existsFunc
: EXISTS groupGraphPattern ;

notExistsFunc
: NOT_BY_WORDS EXISTS groupGraphPattern ;

aggregate
:
( countAggregateExpression
| sumAggregateExpression
| minAggregateExpression
| maxAggregateExpression
| avgAggregateExpression
) ;

countAggregateExpression
: COUNT OPEN_BRACE
( DISTINCT?  ( ASTERISK | var ) ) CLOSE_BRACE ;

sumAggregateExpression
: SUM OPEN_BRACE expression CLOSE_BRACE ;

minAggregateExpression
: MIN OPEN_BRACE expression CLOSE_BRACE ;

maxAggregateExpression
: MAX OPEN_BRACE expression CLOSE_BRACE ;

avgAggregateExpression
: AVG OPEN_BRACE expression CLOSE_BRACE ;

regexExpression
    : REGEX OPEN_BRACE expression
    COMMA expression ( COMMA expression )? CLOSE_BRACE ;

iriRefOrFunction
    : iriRef argList?    ;

rdfLiteral
    : string ( LANGTAG | ( REFERENCE iriRef ) )?    ;

numericLiteral
    : numericLiteralUnsigned | numericLiteralPositive
    | numericLiteralNegative    ;

numericLiteralUnsigned
    : INTEGER
    | DECIMAL
    | DOUBLE   ;

numericLiteralPositive
    : INTEGER_POSITIVE
    | DECIMAL_POSITIVE
    | DOUBLE_POSITIVE    ;

numericLiteralNegative
```

```
    : INTEGER_NEGATIVE
    | DECIMAL_NEGATIVE
    | DOUBLE_NEGATIVE    ;

booleanLiteral
    : TRUE
    | FALSE    ;

string
    : STRING_LITERAL1
    | STRING_LITERAL2
    | STRING_LITERAL_LONG1
    | STRING_LITERAL_LONG2    ;

iriRef
    : IRI_REF
    | prefixedName    ;

prefixedName
    : PNAME_LN
    | PNAME_NS    ;

blankNode
    : BLANK_NODE_LABEL
    | OPEN_SQUARE_BRACE CLOSE_SQUARE_BRACE    ;

// $>

// $<Lexer

WS
    : (' '| '\t'| EOL)+ { $channel=HIDDEN; }    ;

AS
: ('A'|'a')('S'|'s') ;

TIME_RANGE
: INTEGER TIME_UNIT ;

TIME_UNIT
: 'ms'
| 'h'
| 's'
| 'm'
| 'd';

STREAM
: ('S'|'s')('T'|'t')('R'|'r')('E'|'e')('A'|'a')('M'|'m');

RANGE
: ('R'|'r')('A'|'a')('N'|'n')('G'|'g')('E'|'e') ;

STEP
: ('S'|'s')('T'|'t')('E'|'e')('P'|'p') ;

TRIPLES
: ('T'|'t')('R'|'r')('I'|'i')('P'|'p')('L'|'l')
('E'|'e')('S'|'s') ;

TUMBLING
: ('T'|'t')('U'|'u')('M'|'m')('B'|'b')
('L'|'l')('I'|'i')('N'|'n')('G'|'g') ;

REGISTER
:  ('R'|'r') ('E'|'e')('G'|'g')('I'|'i')('S'|'s')
('T'|'t')('E'|'e')('R'|'r') ;

QUERY//MARCO 23.11.09
: ('Q'|'q')('U'|'u')('E'|'e')('R'|'r')('Y'|'y')
;

TIMESTAMP
:  ('T'|'t')('I'|'i')('M'|'m')('E'|'e')('S'|'s')
('T'|'t')('A'|'a')('M'|'m')('P'|'p') ;

EXISTS
: ('E'|'e')('X'|'x')('I'|'i')('S'|'s')('T'|'t')('S'|'s');

NOT_BY_WORDS
: ('N'|'n')('O'|'o')('T'|'t') ;

COUNT
: ('C'|'c')('O'|'o')('U'|'u')('N'|'n')('T'|'t') ;
```

```
SUM
: ('S'|'s')('U'|'u')('M'|'m') ;

MIN
: ('M'|'m')('I'|'i')('N'|'n') ;

MAX
: ('M'|'m')('A'|'a')('X'|'x') ;

AVG
: ('A'|'a')('V'|'v')('G'|'g') ;

GROUP
: ('G'|'g')('R'|'r')('O'|'o')('U'|'u')('P'|'p') ;

HAVING
: ('H'|'h')('A'|'a')('V'|'v')('I'|'i')('N'|'n')('G'|'g') ;

PNAME_NS
    : p=PN_PREFIX? ':'    ;

PNAME_LN
    : PNAME_NS PN_LOCAL    ;

BASE
    : ('B'|'b')('A'|'a')('S'|'s')('E'|'e')    ;

PREFIX
    : ('P'|'p')('R'|'r')('E'|'e')('F'|'f')
    ('I'|'i')('X'|'x')    ;

SELECT
    : ('S'|'s')('E'|'e')('L'|'l')('E'|'e')
    ('C'|'c')('T'|'t')    ;

DISTINCT
    : ('D'|'d')('I'|'i')('S'|'s')('T'|'t')
    ('I'|'i')('N'|'n')('C'|'c')('T'|'t')    ;

REDUCED
    : ('R'|'r')('E'|'e')('D'|'d')('U'|'u')
    ('C'|'c')('E'|'e')('D'|'d')    ;

CONSTRUCT
    : ('C'|'c')('O'|'o')('N'|'n')('S'|'s')('T'|'t')
    ('R'|'r')('U'|'u')('C'|'c')('T'|'t')    ;

DESCRIBE
    : ('D'|'d')('E'|'e')('S'|'s')('C'|'c')('R'|'r')
    ('I'|'i')('B'|'b')('E'|'e')    ;

ASK
    : ('A'|'a')('S'|'s')('K'|'k')    ;

FROM
    : ('F'|'f')('R'|'r')('O'|'o')('M'|'m')    ;

NAMED
    : ('N'|'n')('A'|'a')('M'|'m')('E'|'e')('D'|'d') ;

WHERE
    : ('W'|'w')('H'|'h')('E'|'e')('R'|'r')('E'|'e') ;

ORDER
    : ('O'|'o')('R'|'r')('D'|'d')('E'|'e')('R'|'r') ;

BY
    : ('B'|'b')('Y'|'y')    ;

ASC
    : ('A'|'a')('S'|'s')('C'|'c')    ;

DESC
    : ('D'|'d')('E'|'e')('S'|'s')('C'|'c')    ;

LIMIT
    : ('L'|'l')('I'|'i')('M'|'m')('I'|'i')('T'|'t') ;

OFFSET
    : ('O'|'o')('F'|'f')('F'|'f')('S'|'s')
    ('E'|'e')('T'|'t')    ;

OPTIONAL
    : ('O'|'o')('P'|'p')('T'|'t')('I'|'i')('O'|'o')
```

```
    ('N'|'n')('A'|'a')('L'|'l')    ;

GRAPH
    : ('G'|'g')('R'|'r')('A'|'a')('P'|'p')('H'|'h') ;

UNION
    : ('U'|'u')('N'|'n')('I'|'i')('O'|'o')('N'|'n') ;

FILTER
    : ('F'|'f')('I'|'i')('L'|'l')('T'|'t')
    ('E'|'e')('R'|'r')    ;

A
    : 'a'    ;

STR
    : ('S'|'s')('T'|'t')('R'|'r')    ;

LANG
    : ('L'|'l')('A'|'a')('N'|'n')('G'|'g')    ;

LANGMATCHES
    : ('L'|'l')('A'|'a')('N'|'n')('G'|'g')('M'|'m')
    ('A'|'a')('T'|'t')('C'|'c')('H'|'h')('E'|'e')('S'|'s');

DATATYPE
    : ('D'|'d')('A'|'a')('T'|'t')('A'|'a')('T'|'t')
    ('Y'|'y')('P'|'p')('E'|'e')    ;

BOUND
    : ('B'|'b')('O'|'o')('U'|'u')('N'|'n')('D'|'d')    ;

SAMETERM
    : ('S'|'s')('A'|'a')('M'|'m')('E'|'e')('T'|'t')
    ('E'|'e')('R'|'r')('M'|'m')    ;

ISIRI
    : ('I'|'i')('S'|'s')('I'|'i')('R'|'r')('I'|'i')    ;

ISURI
    : ('I'|'i')('S'|'s')('U'|'u')('R'|'r')('I'|'i')    ;

ISBLANK
    : ('I'|'i')('S'|'s')('B'|'b')('L'|'l')('A'|'a')
    ('N'|'n')('K'|'k')    ;

ISLITERAL
    : ('I'|'i')('S'|'s')('L'|'l')('I'|'i')('T'|'t')
    ('E'|'e')('R'|'r')('A'|'a')('L'|'l')    ;

REGEX
    : ('R'|'r')('E'|'e')('G'|'g')('E'|'e')('X'|'x')    ;

TRUE
    : ('T'|'t')('R'|'r')('U'|'u')('E'|'e')    ;

FALSE
    : ('F'|'f')('A'|'a')('L'|'l')('S'|'s')('E'|'e')    ;

IRI_REF
    : LESS ( options {greedy=false;} : ~(LESS | GREATER
    | '"' | OPEN_CURLY_BRACE | CLOSE_CURLY_BRACE
    | '|' | '^' | '\\' | '`' | ('\u0000'..'\u0020')) )*
    GREATER
    { setText($text.substring(1, $text.length() - 1)); } ;

BLANK_NODE_LABEL
    : '_:' t=PN_LOCAL { setText($t.text); }    ;

VAR1
    : '?' v=VARNAME { setText($v.text); }    ;

VAR2
    : '$' v=VARNAME { setText($v.text); }    ;

LANGTAG
    : '@' PN_CHARS_BASE+ (MINUS (PN_CHARS_BASE DIGIT)+)* ;

INTEGER
    : DIGIT+    ;

DECIMAL
    : DIGIT+ DOT DIGIT*
    | DOT DIGIT+    ;
```

```
DOUBLE
    : DIGIT+ DOT DIGIT* EXPONENT
    | DOT DIGIT+ EXPONENT
    | DIGIT+ EXPONENT    ;

INTEGER_POSITIVE
    : PLUS INTEGER    ;

DECIMAL_POSITIVE
    : PLUS DECIMAL    ;

DOUBLE_POSITIVE
    : PLUS DOUBLE    ;

INTEGER_NEGATIVE
    : MINUS INTEGER    ;

DECIMAL_NEGATIVE
    : MINUS DECIMAL    ;

DOUBLE_NEGATIVE
    : MINUS DOUBLE    ;

fragment
EXPONENT
    : ('e'|'E') (PLUS|MINUS)? DIGIT+    ;

STRING_LITERAL1
    : '\'' ( options {greedy=false;} :
     ~('\u0027' | '\u005C' | '\u000A' | '\u000D')
     | ECHAR )* '\''    ;

STRING_LITERAL2
    : '"'  ( options {greedy=false;}
    : ~('\u0022' | '\u005C' | '\u000A' | '\u000D')
    | ECHAR )* '"'    ;

STRING_LITERAL_LONG1
    :   '\'\'\'' ( options {greedy=false;}
    : ( '\'' | '\'\'' )?
    ( ~('\''|'\\') | ECHAR ) )* '\'\'\''    ;

STRING_LITERAL_LONG2
    :   '"""' ( options {greedy=false;} : ( '"' | '""' )?
    ( ~('"'|'\\') | ECHAR ) )* '"""'    ;

fragment
ECHAR
    : '\\' ('t' | 'b' | 'n' | 'r'
    | 'f' | '\\' | '"' | '\'')    ;

fragment
PN_CHARS_U
    : PN_CHARS_BASE | '_'    ;

fragment
VARNAME
    : ( PN_CHARS_U | DIGIT ) ( PN_CHARS_U | DIGIT | '\u00B7'
    | '\u0300'..'\u036F' | '\u203F'..'\u2040' )*    ;

fragment
PN_CHARS
    : PN_CHARS_U
    | MINUS
    | DIGIT
    | '\u00B7'
    | '\u0300'..'\u036F'
    | '\u203F'..'\u2040'    ;

fragment
PN_PREFIX
    : PN_CHARS_BASE ((PN_CHARS|DOT)* PN_CHARS)?    ;

fragment
PN_LOCAL
    : ( PN_CHARS_U | DIGIT ) ((PN_CHARS|DOT)* PN_CHARS)? ;

fragment
PN_CHARS_BASE
    : 'A'..'Z'
    | 'a'..'z'
    | '\u00C0'..'\u00D6'
    | '\u00D8'..'\u00F6'
```

```
    | '\u00F8'..'\u02FF'
    | '\u0370'..'\u037D'
    | '\u037F'..'\u1FFF'
    | '\u200C'..'\u200D'
    | '\u2070'..'\u218F'
    | '\u2C00'..'\u2FEF'
    | '\u3001'..'\uD7FF'
    | '\uF900'..'\uFDCF'
    | '\uFDF0'..'\uFFFD'    ;

fragment
DIGIT
    : '0'..'9'    ;

COMMENT
    : '#' ( options{greedy=false;} : .)*
    EOL { $channel=HIDDEN; }    ;

fragment
EOL
    : '\n' | '\r'    ;

REFERENCE
    : '^^' ;

LESS_EQUAL
    : '<='    ;

GREATER_EQUAL
    : '>='    ;

NOT_EQUAL
    : '!='    ;

AND
    : '&&'    ;

OR
    : '||'    ;

OPEN_BRACE
    : '('    ;

CLOSE_BRACE
    : ')'    ;

OPEN_CURLY_BRACE
    : '{'    ;

CLOSE_CURLY_BRACE
    : '}'    ;

OPEN_SQUARE_BRACE
    : '['    ;

CLOSE_SQUARE_BRACE
    : ']'    ;

SEMICOLON
    : ';'    ;

DOT
    : '.'    ;

PLUS
    : '+'    ;

MINUS
    : '-'    ;

ASTERISK
    : '*'    ;

COMMA
    : ','    ;

NOT
    : '!'    ;

DIVIDE
    : '/'    ;

EQUAL
    : '='    ;
```

```
LESS
    : '<'    ;

GREATER
    : '>'    ;

ANY : .    ;

QUERY_NAME
: VARNAME ;

COMPUTED_EVERY
: ('C'|'c') ('O'|'o')('M'|'m')('P'|'p')('U'|'u')('T'|'t')
('E'|'e')('D'|'d')
WS ('E'|'e')('V'|'v')('E'|'e')('R'|'r')('Y'|'y') ;


// $>
```

Bibliografia

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. Intl. Conf. on Innovative Data Systems Research (CIDR 2005)*, 2005.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proc. ACM Intl. Conf. on Management of Data (SIGMOD 2003)*, page 665, 2003.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A Data Stream Language and System Designed for Power and Extensibility. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM 2006)*, pages 337–346, 2006.

[5] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Continuous queries and real-time analysis of social semantic data with c-sparql. In *SDoW2009*, volume 520 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009. online http://ceur-ws.org/Vol-520/paper02.pdf.

[6] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: Sparql for continuous querying. In Quemada et al. [24], pages 1061–1062.

[7] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: A continuous query language for rdf data streams. International Journal of Semantic Computing (IJSC), 2010.

[8] Stefano Ceri and Giuseppe Pelagatti. Correctness of Query Execution Strategies in Distributed Databases. *ACM Trans. Database Syst.*, 8(4):577–607, 1983.

[9] Stefano Ceri Davide Francesco Barbieri, Daniele Braga and Michael Grossniklaus. An execution environment for c-sparql queries. EDBT, 2010.

[10] John Domingue, Dieter Fensel, and Paolo Traverso, editors. *Future Internet - FIS 2008, First Future Internet Symposium, FIS 2008, Vienna, Austria, September 29-30, 2008, Revised Selected Papers*, volume 5468 of *Lecture Notes in Computer Science*. Springer, 2009.

[11] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc., 2007.

[12] Lukasz Golab, David DeHaan, Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Identifying Frequent Items in Sliding Windows over On-line Packet Streams. In *Proc. Intl. Conf. on Internet Measurement (IMC 2003)*, pages 173–178, 2003.

[13] Lukasz Golab and M. Tamer Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB 2006)*, pages 500–511, 2003.

[14] Balakrishnan H., Balazinska M., Carney D., C. etintemel U., Cherniack M., Convey C., Galvez E., Salz J., Stonebraker M., Tatbul N., Tibbetts, R., and Zdonik S. Retrospective on aurora. page 370?383. VLDB, 2004.

[15] Chen J., DeWitt D.J., Tian F., and Wang Y. Niagaracq: A scalable continuous query system for internet databases. page 379?390, 2000.

[16] H.V. Jagadish, I.S. Mumick, Silberschatz, and A. View maintenance issues for the chronicle data model. pages 113–124, 1995.

[17] Liu L., Pu C., and Tang W. Continual queries for internet scale event-driven information delivery. page 610?628, 1999.

[18] Yan-Nei Law and Carlo Zaniolo. An Adaptive Nearest Neighbor Classification Algorithm for Data Streams. In *Proc. Europ. Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005)*, pages 108–120, 2005.

[19] Garofalakis M., Gehrke J., and Rastogi R. Springer-data stream management: Processing high-speed data streams (data-centric systems and applications). Verlag New York, Inc., Secaucus, NJ,USA, 2007.

[20] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *Proc. Intl. Semantic Web Conf. (ISWC 2006)*, pages 30–43, 2006.

[21] Juergen Pfundt, Michele Mostarda, and Simone Tripodi. SPARQL Grammar for ANTLR v3. `http://www.antlr.org/grammar/1200929755392/index.html`.

[22] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

[23] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF Grammar. `http://www.w3.org/TR/rdf-sparql-query/#sparqlGrammar`.

[24] Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors. *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*. ACM, 2009.

[25] Emanuele Della Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. A first step towards stream reasoning. In Domingue et al. [10], pages 72–81.

[26] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.