UNIVERSITA' DEGLI STUDI
DI MODENA E REGGIO EMILIA

FACOLTA' DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Algoritmi di matching tra schemi XML per la riscrittura di query

Milena Cevolani

Tesi di laurea

Anno Accademico 2002/2003

Relatore : Chiar.mo Prof. Paolo Tiberio

Correlatore : dott.sa Federica Mandreoli

RINGRAZIAMENTI

Ringrazio il Professor Paolo Tiberio per la disponibilità dimostrata.
Ringrazio in modo particolare la Dottoressa Federica Mandreoli e l'Ing. Riccardo Martoglia per il costante aiuto fornito durante lo sviluppo di questa tesi.
Un grosso ringraziamento va a tutta la mia famiglia che in questi anni mi ha sempre sostenuta e incitata.
Un ringraziamento va anche a tutti i miei amici di Università che negli anni hanno condiviso con me le gioie e le preoccupazioni degli esami: Erika Stefanini, Sara Malavasi, Monica Romani, Katia Mazzoni, Maddalena Tomasini, Lisa Fregni, Andrea "Luisa", Francesca Rossi, Claudio Lei, Marco Bergonzini, Massimo Bertacchini, Bergamini Davide, Luca Vezzali, Riccardo Lancellotti, Luca Ridolfi, Roberto Fabbri e a tutti gli altri.
Ringrazio anche le mie amiche Manu, e Marika per il loro sostegno anche al di fuori dell'Università.
.

PAROLE CHIAVE

Schemi
XML
MOMIS
Query
Matching

Indice

Indice delle figure

Indice delle tabelle

Introduzione

Negli ultimi anni il linguaggio *eXtensible Markup Language (XML)* ha ottenuto una sempre maggiore importanza nel campo delle Biblioteche Digitali e nello scambio di dati via Web. Il motivo di questo successo risiede nell'assenza di una struttura specifica dei dati descritti: si parla, infatti, di dati semistrutturati. La forma di un documento XML viene decisa dall'autore del documento stesso, rendendo tale standard molto pratico ed in grado di descrivere qualsiasi cosa. Chiunque abbia dimestichezza con lo standard HTML (attualmente il più popolare linguaggio per la descrizione e la visualizzazione di documenti sul Web) non avrà alcun problema nel comprendere la sintassi di XML. XML si basa su marcatori (*tag*) i cui nomi sono decisi da chi crea il documento ed hanno il duplice compito di descrivere e contenere i dati. Per questi motivi XML risulta adatto per l'implementazione di archivi di dati di grandi dimensioni e riguardanti gli argomenti più disparati.

L'interrogazione di un grande archivio di documenti XML, come può essere il *repository* di una Biblioteca Digitale, non è una cosa semplice, proprio per l'assenza di una struttura fissa che ne contraddistingua i documenti stessi. L'assenza di una struttura fissa rappresenta quindi le due facce di una stessa medaglia: la flessibilità e la facilità d'uso da una parte e la difficoltà a recuperare le informazioni desiderate dall'altra.

Un'interrogazione posta da un utente su un archivio XML, non può essere risolta efficacemente attraverso una *corrispondenza esatta*, cioè recuperando solamente quei documenti descritti dai marcatori presenti nella richiesta. Difficilmente, infatti, un utente potrà essere a conoscenza della struttura esatta di tutti i documenti contenuti nell'archivio. Bisogna considerare, inoltre, che persone diverse possono descrivere la realtà che le circonda un maniera diversa: documenti XML che parlano degli stessi argomenti potrebbero avere strutture e terminologia di marcatori molto differenti fra loro, se creati da due persone distinte. Per questi motivi, allora si parla di *interrogazioni approssimate*, ossia di riscrivere la query in maniera approssimata.

Un'operazione che si può fare prima di riscrivere la query è quella di risolvere le differenze fra gli schemi stessi, ossia effettuare un'operazione di *matching* fra i termini degli schemi. L'operazione di matching prende in ingresso due schemi XML e restituisce una "mappatura" tra gli elementi dei due schemi che corrispondono semanticamente l'uno all'altro. Il matching fra schemi viene effettuato per molte ragioni: ogni schema, per rappresentare concetti identici, può avere una struttura e nomi

differenti, gli schemi possono modellare contenuti simili, ma non identici, possono utilizzare parole simili, ma che hanno significati diversi e così via. Attualmente, il matching tra schemi viene effettuato tipicamente in maniera manuale, magari supportato da un'interfaccia utente grafica.

In questa tesi si è implementato un metodo, basato su ontologie e su un calcolo di fixpoint, per effettuare in maniera automatica il matching fra i diversi schemi XML che descrivono i documenti presenti nell'archivio di una Biblioteca Digitale. L'algoritmo implementato adotta il seguente approccio. In primo luogo gli schemi vengono convertiti in grafi etichettati diretti, secondo le specifiche dell'RDF, quindi, questi grafi, vengono usati in un calcolo iterativo di fixpoint, il cui risultato è rappresentato dalle coppie di nodi simili nei due grafi e dal livello di similarità. Per il calcolo delle similarità ci si basa su due intuizioni: la prima è che si può ottenere un valore iniziale di similarità attraverso un approccio linguistico, applicato ai termini degli schemi XML, la seconda intuizione è che gli elementi di due *modelli* sono simili quando anche i loro elementi adiacenti lo sono.

L'impiego di ontologie permette di assegnare agli schemi che descrivono i documenti XML, significati che non dipendono dal termine specifico impiegato, ma dal vero e proprio concetto rappresentato. In questa tesi si assume che gli schemi siano espressi nel linguaggio XML Schema per la rappresentazione e la validazione di documenti XML, proposto e consigliato dal W3C, e che i termini che descrivono gli schemi siano stati precedentemente *annotati* con i loro significati, come descritto in [24].

La tesi è strutturata nel seguente modo:

- Nel *Capitolo 1* si parla di Biblioteche Digitali, di quale potrebbe essere il loro futuro nella società moderna e di quale struttura potrebbero avere. Viene introdotto, inoltre, il problema della riscrittura di interrogazioni su archivi di documenti semistrutturati e l'operazione di match fra schemi XML. Infine il capitolo presenta una panoramica sullo stato dell'arte riguardo tali argomenti.

- Nel *Capitolo 2* viene presentata la sintassi del linguaggio XML Schema per la descrizione di documenti XML. La lettura di questo capitolo può essere tralasciata da chiunque sia già a conoscenza del linguaggio.

- Nel *Capitolo 3* viene descritto il database lessicale WordNet e i tipi di relazioni in esso usate.

- Nel *Capitolo 4* viene descritto in dettaglio il metodo implementato per effettuare il matching fra schemi XML. Gli schemi XML vengono inizialmente trasformati in grafi etichettati diretti. Viene poi costruita una "mappatura" iniziale fra le coppie di nodi, sfruttando un approccio linguistico, applicato ai termini degli schemi XML, che sfrutta le gerarchie di ipernimi di WordNet. Per il calcolo della similarità fra le coppie, l'algoritmo di matching si basa su un calcolo iterativo di fixpoint dei valori σ di similarità; questo calcolo iterativo corrisponde ai cammini casuali sui grafi secondo una certa distribuzione di probabilità. Il

capitolo presenta inoltre alcune prove sperimentali che dimostrano l'efficacia dell'algoritmo ottenuto.

- Nel Capitolo *Conclusioni e sviluppi futuri* viene riportata una breve panoramica dell'algoritmo implementato e dei risultati ottenuti, dando poi un suggerimento su come l'algoritmo potrebbe essere impiegato in un futuro modulo per la riscrittura di query.

- Nell'*Appendice A* vengono descritte le catene di Markov.

Capitolo 1

Biblioteche Digitali XML

1.1 Introduzione

Una definizione informale di Biblioteca Digitale può essere la seguente:

> *"Una Biblioteca Digitale è una raccolta gestita di informazioni, con servizi associati, in cui l'informazione è memorizzata in formato digitale ed è accessibile su una rete".*

Una parte importante di questa definizione è che l'informazione è gestita, ossia organizzata in maniera sistematica. Il concetto di "Biblioteca Digitale" nasce dalla sua analogia con la biblioteca tradizionale: uno spazio fisico, o deposito, contenente una collezione organizzata di documenti, insieme a sistemi e servizi atti a facilitare l'accesso, fisico ed intellettuale, ai documenti e la loro conservazione. Le Biblioteche Digitali contengono varie raccolte di informazioni, ad uso di molti utenti e, per questo motivo, le loro dimensioni possono variare da minuscole ad enormi, inoltre esse possono utilizzare qualsiasi tipo di apparecchiatura di calcolo e qualsiasi software che sia adatto alla gestione e al reperimento delle informazioni e dei documenti in esse contenute. Le tecnologie necessarie alla creazione di Biblioteche Digitali sono molto diverse da quelle relative alle biblioteche tradizionali. L'informazione digitale, infatti, si trasferisce con una velocità molto maggiore, può essere archiviata su scale di densità più elevata e può integrarsi in nuovi tipi di documenti che includono testo, immagini, grafica, video, audio, ecc.

In sostanza, le Biblioteche Digitali includono le prestazioni offerte dalle biblioteche tradizionali, ma vanno ben oltre in termini di funzionalità, portata e significato. Possiamo quindi definire una Biblioteca

Digitale come un ambiente dove si mettono in relazione collezioni, servizi e persone lungo l'intero ciclo di vita dell'informazione, dalla creazione, divulgazione, utilizzo, fino alla conservazione.

Le tecnologie delle Biblioteche Digitali hanno il potenziale di influenzare profondamente alcuni aspetti che riguardano il modo di lavorare con l'informazione sotto forma di documento. L'impatto dei cambiamenti previsti sarà vasto, ma in alcuni settori sarà anche profondo. Un settore particolarmente interessato a questi cambiamenti è quello che riguarda i nuovi modelli della diffusione/disseminazione dell'informazione scientifica.

Le Biblioteche Digitali nascono e si sviluppano sulla scia del successo, sempre in aumento, di Internet e del Web in generale. Proprio da questa tecnologia nasce l'idea che un documento può esistere non solo nella sua forma cartacea, ma anche, o solo, in formato digitale.

Gli sforzi che sono stati fatti in questi anni per creare standard in grado di permettere scambi di informazioni tra sistemi differenti (si pensi ad esempio a standard come l'ODBC, il TCP\IP oppure CORBA) hanno permesso di compiere grandi progressi anche nel campo delle Biblioteche Digitali. Proprio in questo senso, uno standard che avrà un impatto sempre maggiore sulla tecnologia e sul modo di implementare le Biblioteche Digitali è l'XML(*eXstensible Markup Language*). Si tratta di un linguaggio per la marcatura dei documenti, ideato per rendere le informazioni in essi contenute *self-describing*. I marcatori (o *tag*) vengono decisi dal creatore del documento e possono fornire una descrizione del contenuto del documento stesso, fornendogli, quindi, una semantica.

Risulta comprensibile, quindi, come le Biblioteche Digitali possano trarre vantaggio dall'utilizzo di uno standard come XML, ampiamente utilizzato in Internet e in grado di fornire ogni tipo di informazione digitale (testo, immagini, video, audio, etc.).Si può parlare di *Biblioteche Digitali XML* nei casi in cui lo standard XML viene utilizzato come strumento per la definizione dei metadati ed eventualmente dei documenti digitali, nonché per la specifica delle caratteristiche della Biblioteca Digitale e della sua interfaccia col mondo esterno, in modo da garantire interoperabilità fra le diverse realizzazioni di biblioteche.

1.2 Il progetto ECD

La presente tesi è stata sviluppata nell'ambito del progetto ECD (*Enhanced Content Delivery*) [23], al quale collaborano numerosi enti di ricerca come l'istituto ISTI-CNR di Pisa, il Politecnico di Milano e le Università di Modena e Reggio Emilia, di Padova e di Roma 3. Il progetto si concentra sullo sviluppo di tecnologie e strumenti per offrire contenuti arricchiti (da cui il nome del progetto) agli utenti finali. Ciò consiste nell'identificare materiale digitale presente su fonti diverse, trasformarlo, organizzarlo, aggiungervi metadati e informazioni utili a qualificarlo e far giungere agli utenti il materiale più rilevante per i loro interessi. Le tecnologie di rete e digitali offrono nuovi mezzi di distribuzione di contenuti, in particolare:

- Biblioteche Digitali

- Ricerca ed accesso sul Web.

I servizi di accesso offerti agli utenti di questi mezzi includono i motori di ricerca, i cataloghi tematici, le collezioni a soggetto (audio, video, WAP, ecc.) e i servizi avanzati delle Biblioteche Digitali. Tramite queste tecnologie agli utenti può essere offerta un'infinita quantità di dati ed informazioni, non necessariamente solo sul piano testuale, ma comprensive di audio, filmati, immagini, ecc.

I principali obiettivi che questo progetto si propone di raggiungere sono i seguenti:

- Sviluppo di algoritmi per indicizzare e ricercare documenti in formato compresso.
- Sfruttare tecniche di High Performance Computing per fronteggiare le moli dei dati e il numero di utenti dei servizi.
- Sviluppo di tecniche di Web Mining per determinare:
  - rank o autorevolezza delle fonti;
  - come migliorare le prestazioni di spidering e caching;
  - come classificare i documenti.
- Sviluppo di servizi di ricerca partecipativa e decentralizzata.
- Sviluppo di un'architettura aperta per Biblioteche Digitali distribuite.
- Utilizzo di XML per strutturare documenti ed esprimere metadati.
- Fornire accesso a documenti multimediali nelle Biblioteche Digitali.
- Formulare e rispondere a interrogazioni su schemi XML.
- Sviluppare servizi avanzati per gli utenti quali: annotazioni di documenti, notifica, supporto al lavoro di gruppo.

1.3 Un'architettura aperta per Biblioteche Digitali XML

In questa sezione descriviamo l'architettura aperta per Biblioteche Digitali che verrà implementata nell'ambito del progetto ECD.

Un'architettura per Biblioteche Digitali viene detta *aperta* quando l'insieme delle funzionalità complessive viene partizionato in un gruppo di servizi autonomi, ben definiti, ed in grado di cooperare fra loro. Tali servizi possono essere distribuiti oppure replicati. Un'architettura di questo tipo può essere fondamentale nello sviluppo di Biblioteche Digitali, il cui funzionamento non si limiti alle semplici funzionalità di ricerca remota e distribuita delle informazioni. Al giorno d'oggi, infatti, le caratteristiche che possono essere richieste ad oggetti di questo tipo sono molteplici; si pensi, ad esempio, a funzionalità per il controllo e l'aggiornamento delle versioni dei documenti presenti, al controllo del copyright o a richieste di mediazione ed organizzazione degli oggetti digitali. Risulta chiaro, quindi, come *lo sviluppo di architetture aperte sia necessario* per permettere la facile estensione delle Biblioteche Digitali implementate con nuovi servizi distribuiti in rete e, fatto

anch'esso molto importante, per facilitare la cooperazione e lo scambio di informazioni fra diverse biblioteche. Proprio nell'ottica di un'architettura aperta può essere risolutivo l'utilizzo dello standard XML che, oltre alla definizione dei documenti digitali e dei metadati utilizzati, fornisce le specifiche per la sua interfaccia con il mondo esterno, premessa fondamentale per l'interoperabilità fra le diverse biblioteche.

Un prototipo esistente di sistema aperto per Biblioteche Digitali, basate sullo standard XML, è rappresentato dal sistema OPEN-DLIB [22]. Tale sistema è stato sviluppato dall'istituto ISTI-CNR di Pisa ed è in grado di creare facilmente una Biblioteca Digitale, in accordo con le richieste e le preferenze di una certa comunità di utenti. OPEN-DLIB consiste in un insieme di servizi distribuiti in rete e fra loro interoperanti, che implementano le funzionalità che possono essere richieste ad una Biblioteca Digitale. I diversi servizi implementati da OPEN-DLIB possono essere:

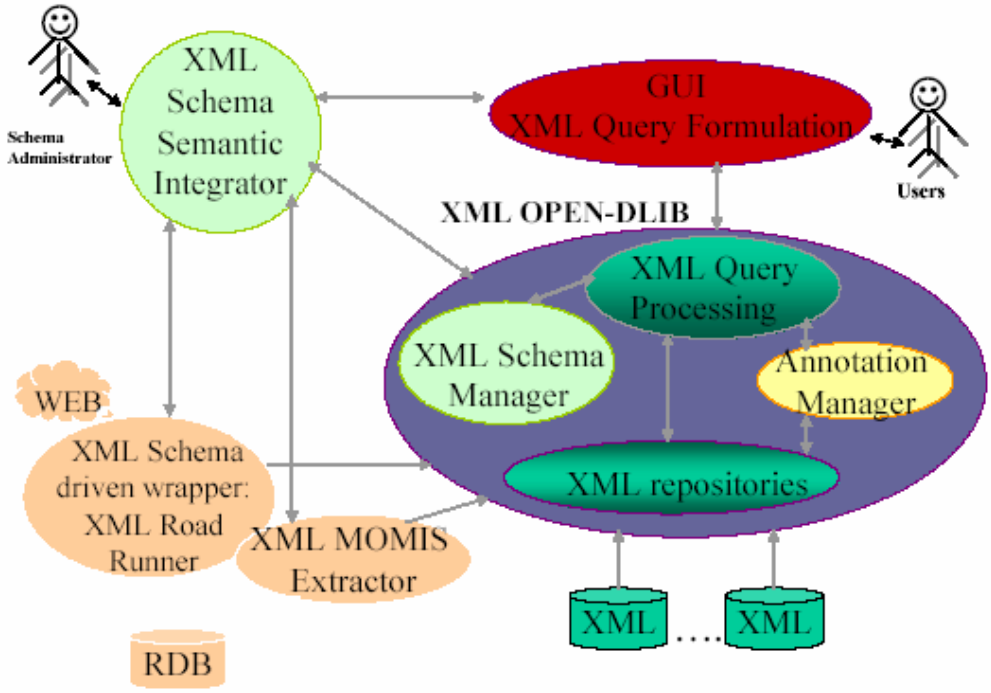

Figura 1.1: L'architettura di riferimento del progetto ECD.

- *Centralizzati*: un servizio centralizzato ha sempre una sola istanza all'interno della Biblioteca Digitale. I servizi centralizzati sono soprattutto quelli riguardanti la sicurezza, servizi di autenticazione e autorizzazione degli utenti in cui la replicazione e la distribuzione può rappresentare un rischio.

- *Distribuiti*: i servizi distribuiti possono essere implementati attraverso molteplici istanze. Ogni istanza del servizio distribuito si occupa di lavorare su un insieme di dati allocati su un particolare server. I servizi distribuiti vengono impiegati soprattutto per la gestione di grandi quantità di dati. Ogni server, quindi, possiede grandi quantità di dati ed è sfruttato da un'istanza dei servizi distribuiti in grado di lavorare su di essi.

- *Replicati*: sono quei servizi, possibilmente collocati su diversi server, in grado, ognuno, di gestire l'intera quantità di dati (nell'intero sistema) per cui è stato progettato. La replicazione di un servizio può peggiorare le prestazioni, ma può fornire una maggiore robustezza e resistenza ai guasti.

La versione attuale di OPEN-DLIB fornisce una selezione di servizi interconnessi che forniscono solamente le attività basilari, quali l'acquisizione di dati, l'archiviazione degli stessi, l'esplorazione e la descrizione delle informazioni, insieme a servizi quali l'autenticazione e l'autorizzazione degli utenti. L'insieme di servizi forniti, seguendo l'ottica dei sistemi aperti, non è fissato a priori, ma può essere facilmente esteso per fornire funzionalità aggiuntive. Nell'ambito del progetto ECD, si è pensato di includere e realizzare i seguenti servizi di OPEN-DLIB:

- Servizio di archiviazione (*XML repositories*): servizio che fornisce i meccanismi e le funzionalità per l'accesso agli oggetti digitali. Tale servizio deve essere in grado di fornire anche una rappresentazione della struttura degli oggetti contenuti. L'archiviazione delle informazioni è, ovviamente, un servizio imprescindibile e deve essere organizzato in maniera robusta e funzionale.

- Servizio di gestione di schemi (*XML Schema Manager*): servizio che fornisce i meccanismi per l'accesso e la gestione degli schemi XML dei dati memorizzati nel repository della Biblioteca Digitale.

- Servizio di gestione delle annotazioni (*Annotation Manager*): servizio che consente di generare in modo automatico sommari e collegamenti tra documenti ed annotazioni. I sommari e i collegamenti, così generati, consentono di ottenere una descrizione sintetica del contenuto semantico del documento e di reperire altri documenti con un contenuto pertinente agli interessi dell'utente.

- Servizio di esecuzione di query XML (*XML Query Processing*): il servizio si occupa dell'esecuzione di query XML espresse in XQuery attraverso un meccanismo di matching esatto.

Le diverse istanze dei servizi comunicano fra loro tramite un protocollo denominato OLP (*OPEN-DLIB Protocol*). Le richieste, attraverso il protocollo OLP, vengono espresse attraverso URL inclusi in richieste http. Tutte le richieste, o le risposte strutturate, sono basate sullo standard XML.

Il progetto ECD si pone inoltre l'obiettivo di estendere l'architettura di OPEN-DLIB con servizi di supporto all'accesso di Biblioteche Digitali eterogenee e distribuite sul web. L'architettura complessiva

del progetto ECD è mostrata in Figura 1.1. Oltre ai servizi per il wrapping di sorgenti HTML in XML (*XML Schema driven wrapper* e *XML MOMIS extractor*) e ad un'interfaccia per la formulazione di interrogazioni (*GUI XML Query Formulation*), l'architettura include un servizio per l'integrazione di schemi XML (*XML Schema Semantic Integrator*). Tale servizio verrà realizzato dall'Università di Modena nell'ambito del progetto. Lo scopo principale del servizio è di fornire un adeguato supporto per la costruzione di una ontologia comune per i diversi metadati in Biblioteche Digitali aperte. Tale ontologia rappresenterà la base per il rilassamento delle interrogazioni poste dall'utente, dove l'obiettivo finale sarà quello di aumentare l'efficacia del processo di esecuzione delle interrogazioni, attraverso una fase di recupero delle informazioni che provvede a riscrivere le interrogazioni in modo compatibile con gli schemi adottati per memorizzare i dati XML.

1.4 XML: un linguaggio di markup

XML (*eXstensible Markup Language*), proposto dal World Wide Consortium (W3C), recentemente si è manifestato come nuovo standard per la rappresentazione di dati e lo scambio di documenti sia nella comunità Web sia nella comunità sociale. XML è un linguaggio per la marcatura di documenti ideato per rendere le informazioni "self-describing". A differenza di HTML, l'attuale standard su Web, XML non si preoccupa dello stile di presentazione del documento, ma della descrizione del contenuto. I marcatori usati in XML possono essere decisi dal creatore del documento e quindi possono essere usati per spiegare il significato delle informazioni (es. prezzo, quantità, colore e così via). XML è stato sviluppato in modo tale da soddisfare i seguenti requisiti:

- Utilizzo su Internet: gli utenti devono potere visualizzare rapidamente e facilmente i documenti in XML quanto quelli in HTML. Questo è già possibile grazie alla presenza di *browser*, come Internet Explorer, che supportano già XML;

- Sostenere un'ampia varietà di applicazioni: XML deve essere indipendente dalla piattaforma in modo tale da poter essere usato da *browser* differenti;

- Facilità nello scrivere programmi in XML: è un punto fondamentale per garantire il successo e la curabilità di XML;

- Il numero di optional in XML deve essere minimo se non addirittura inesistente: in questo modo si evitano difficoltà nella scrittura di programmi, errori e il sorgere inevitabile di problemi di compatibilità;

- Leggibilità e chiarezza dei documenti in XML: in questo modo, anche se non si possiede un *browser* per XML si è in grado di leggere e capire qualunque documento.

XML ha quindi molti vantaggi. Per prima cosa l'indipendenza dalla piattaforma e dal sistema: ciò significa che funziona su qualunque tipo di calcolatore e con qualunque software, a prescindere

dall'azienda che lo crea. In secondo luogo, XML è essenzialmente chiaro e per questo sono state diminuite al minimo i marcatori. In terzo luogo, è possibile creare le proprie modifiche senza bisogno della DTD se il documento è ben formato. Infine, XML ha adottato l'ISO 10646, meglio conosciuto come Unicode: questo standard viene utilizzato per la codifica dei caratteri ed è in grado di supportare la maggior parte delle lingue. Poiché XML segue questo standard, è possibile utilizzare qualunque lingua per il *markup* e il software per XML è in grado di capirlo senza costringere gli utenti ad usare l'inglese come codice per le istruzioni.

In sintesi, le motivazioni che porteranno XML al successo si possono sintetizzare nei seguenti punti:

- Consente di utilizzare documenti strutturati.
- È estensibile, per cui permette di aggiungere sempre nuovi marcatori.
- Offre un ottimo formato di scambio di dati, è strutturato, estensibile, non ambiguo e completamente leggibile (non binario) e sarà comunque riutilizzabile se si considera che anche i programmi ad ogni nuova versione cambiano formato.
- La strutturazione e l'utilizzo di un linguaggio estensibile basato su tag consente una più semplice interazione con altri programmi, compresi i database, e quindi un trattamento dei dati più semplice ed efficace.
- È indipendente dalla piattaforma e dal processore.
- Permette un semplice utilizzo di metadati, come Dublin Core, etc.
- Consente ricerche più semplici e più efficaci: ad esempio, attraverso il controllo sui tag, è possibile effettuare una interrogazione tramite un motore di ricerca assicurandosi una risposta più precisa rispetto a ciò che realmente stiamo cercando.
- Offre un buon meccanismo di rappresentazione, un'ottima capacità di rappresentare dati complessi (notazioni matematiche, interfacce grafiche).
- Offre possibilità di presentazioni superiori a quelle di HTML che, per ottenere risultati simili, deve ricorrere all'utilizzo di Java o altri linguaggi.

1.5 Interrogazione di documenti XML

Occupandoci specificatamente di Biblioteche Digitali in cui i documenti sono scritti in XML e avendo affermato che tali documenti sono semistrutturati, nasce il problema del reperimento dei dati, da parte dell'utente, sull'archivio dei documenti. Un utente della Biblioteca Digitale dovrebbe essere in grado

di formulare una richiesta, o *query*, per cercare documenti contenuti nell'archivio, aiutato possibilmente da un'interfaccia grafica e dovrebbe ricevere una risposta (composta da uno o più documenti) che soddisfi il più possibile la sua richiesta. Quindi, una volta definito lo standard XML, il passo successivo è la definizione di un linguaggio di interrogazione che permetta di formulare query sulla struttura propria dei documenti XML.

La struttura di un documento XML può essere di due tipi diversi:

- *document-centric*: si concentra su applicazioni XML che scambiano documenti (strutturati) nel senso tradizionale, cioè il markup principale serve per esporre la struttura logica di un documento.

- *data-centric*: utilizza XML per scambiare dati in forma strutturata, come le applicazioni EDI (*Electronic Data Interchange*) classiche, i fogli elettronici o i database. Questi documenti hanno una struttura ben definita e sono tipicamente memorizzati in collezioni *omogenee*, ma possono anche essere memorizzati in collezioni *eterogenee*.

Per la ricerca in documenti XML di tipo document-centric possono essere considerate appropriate le tecniche proprie dell'Information Retrieval. Per interrogare, invece, documenti XML di tipo data-centric in collezioni omogenee, sono stati proposti diversi linguaggi di interrogazione per XML (in [10,11] viene presentata un'analisi comparativa fra alcuni linguaggi). Tuttavia, per interrogare collezioni eterogenee di documenti XML di tipo data-centric, né le tecniche dell'IR, né i linguaggi di interrogazione menzionati prima sono idonei.

Negli ultimi anni sono state fatte molte proposte di linguaggi di interrogazione di raccolte eterogenee di documenti semistrutturati, in particolare documenti XML, ma non è ancora stato trovato uno standard effettivo. Il linguaggio XQL [5] segue la rappresentazione di tipo document-centric; altri linguaggi di interrogazione XML, come XML-QL [6], si focalizzano sulla rappresentazione data-centric, offrendo un'ampia varietà di operatori per aggiustare il risultato, come gli operatori di aggregazione, simili ai linguaggi standard di interrogazione dei database, come SQL [7] o OQL [8]. XQuery [1,2], il linguaggio proposto dal W3C, combina le caratteristiche di XQL con quelle di XML-QL e supporta quindi sia la rappresentazione document-centric che quella data-centric. In [12] viene proposto un linguaggio chiamato XIRQL, che comprende le caratteristiche di XQL, e in più ve ne aggiunge altre, proprie dell'Information Retrieval, come la "pesatura", il ranking, la ricerca relevance-oriented, i tipi di dato con predicati vaghi e il relativismo semantico. In [14] viene presentato il linguaggio XXL (*fleXible Xml search Language*); anche questo adotta i concetti base di XML-QL e li estende con condizioni di similarità sugli elementi ed i loro attributi. Un altro metodo interessante per l'interrogazione di dati XML è quello fornito dall'Università di Stanford nel progetto *Lore*. Lore rappresenta un database system completo e multiutente, in grado di trattare dati semistrutturati; all'interno del progetto è stato sviluppato un linguaggio per interrogazioni, detto *Lorel* [13], in grado di gestire richieste per dati semistrutturati e quindi anche per XML.

Un aiuto nell'ambito dell'interrogazione di un archivio di documenti digitali si può avere dall'uso di *metadati*, cioè di schemi che descrivono la struttura dei documenti, limitando in modo considerevole il

numero di dati da valutare per rispondere ad un'interrogazione. Vediamo più in dettaglio di cosa si tratta nel paragrafo successivo.

1.6 I metadati in XML

I metadati sono un concetto di grande importanza nello sviluppo di Biblioteche Digitali. In generale, col termine metadati si fa riferimento a "dati che descrivono dati". Rispetto a tale definizione si evidenzia che i metadati costituiscono una parte integrante dell'informazione, poiché, unitamente ai dati che descrivono, rendono i dati informazioni utilizzabili. In una Biblioteca Digitale possiamo affermare che un metadato è un dato che descrive il contenuto e gli attributi di ogni oggetto presente dentro la biblioteca. Il termine, normalmente associato alle risorse elettroniche, in realtà si riferisce ad un'attività che bibliotecari e indicizzatori svolgono da sempre: quella di applicare ai documenti delle "etichette", principalmente con lo scopo di renderne accessibile l'informazione. L'impiego di metadati è molto importante all'interno di una Biblioteca Digitale, può, infatti, essere considerato la chiave per ritrovare ed usare ogni documento desiderato. Questa affermazione può essere ben compresa se pensiamo agli attuali motori di ricerca per pagine Web, utilizzabili via Internet (Google, Altavista, Yahoo): essi, dopo aver ricevuto un richiesta da parte di un utente, non svolgono una ricerca *full text* (cioè sull'intero corpo del testo di tutti i documenti), poiché una ricerca del genere fornirebbe all'utente migliaia di documenti utili solamente in minima parte. L'impiego di metadati appare così utilissimo per rispondere in maniera appropriata alle richieste poste dagli utenti, specialmente se il numero di documenti che formano la Biblioteca Digitale è molto elevato.

Per una Biblioteca Digitale, basata su tecnologia XML, è necessario usare metadati che forniscano una descrizione della struttura dei documenti XML che trattano. Un ottimo modo per rappresentare i metadati è, quindi, attraverso l'impiego di linguaggi atti alla costruzione di schemi XML. Tra tutti i vari linguaggi proposti per la costruzione di schemi, lo standard attuale *de facto* è fornito da DTD XML (*Document Type Definition*), ma le sue possibilità espressive sono leggermente ridotte rispetto a quelle degli altri linguaggi proposti. Il linguaggio che, a parere di molti, occuperà il posto delle DTD è XML Schema, nuovo standard per la validazione di documenti XML proposto dal W3C. XML Schema, oltre a risultare più flessibile rispetto alle DTD, presenta l'indiscusso vantaggio di sfruttare lo stesso linguaggio XML, cosa non trascurabile in un ambiente come quello delle Biblioteche Digitali XML in cui, sfruttando XML Schema, sia i dati che i metadati possono essere trattati dagli stessi strumenti. Per un'analisi comparativa dei vari standard di rappresentazione di schemi XML si rimanda a [15]. Nell'ambito di questa tesi, i metadati saranno da considerarsi in formato XML Schema.

1.7 XQuery: un linguaggio per interrogare documenti XML

XML è un linguaggio di markup estremamente versatile, in grado di rappresentare il contenuto di diverse sorgenti dati: database relazionali, ad oggetti, dati semi-strutturati, ecc.

XQuery [1,2] è stato progettato per essere applicabile su tutte diverse sorgenti dati XML. XQuery deriva da Quilt [3], un linguaggio di interrogazione per XML, che a sua volta ha preso le sue funzioni da numerosi altri linguaggi di interrogazione:

- Sintassi per navigare all'interno della struttura di un documento simile a quella di XPath 1.0 [4] e XQL [5].
- Definizioni di variabili globali e locali da XML-QL [6].
- Query come insieme di proposizioni, come in SQL [7].
- Definizione di funzioni da OQL [8].

XQuery è un linguaggio molto flessibile e permette sia interrogazioni facilmente comprensibili dall'utente, sia interrogazioni basate in maniera più specifica sulla sintassi XML e XML Schema. Inoltre risulta essere fortemente tipizzato e il sistema di tipi fornito al suo interno è basato su quello fornito da XML Schema (vedi Capitolo 2). In XQuery una query viene rappresentata come un'espressione. XQuery supporta diversi tipi di espressioni, tuttavia, nell'ambito di questa tesi, verranno fatti degli esempi che utilizzano solo le espressioni FLWR (si pronuncia "flower") e le espressioni di percorso. Vediamole quindi più in dettaglio nei paragrafi seguenti.

1.7.1 Le espressioni di percorso

XQuery usa una sintassi simile a quella di XPath per le espressioni di percorso. Un'espressione di percorso è un insieme di passi: ogni passo rappresenta un movimento all'interno della struttura di un documento e il risultato di ogni passo è una lista di nodi. La Tabella 1.1 riporta un elenco delle espressioni di percorso usate da XQuery.

Simbolo	Descrizione
.	Il nodo corrente
..	Nodo padre del nodo corrente
/	Il nodo radice o il separatore tra diversi passi dell'espressione di percorso
//	I figli del nodo corrente
@	Gli attributi del nodo corrente
*	Qualsiasi nodo
[]	Contengono un'espressione booleana per limitare i nodi selezionati in un certo passo
[n]	Seleziona un elemento figlio da una lista di elementi

Tabella 1.1: Espressioni di percorso di XQuery

1.7.2 Espressioni FLWR

Un'espressione FLWR è formata dalle proposizioni `for`, `let`, `where` e `return`. Come per una query in SQL, queste proposizioni devono apparire in un ordine ben preciso, come riportato in Figura 1.2.

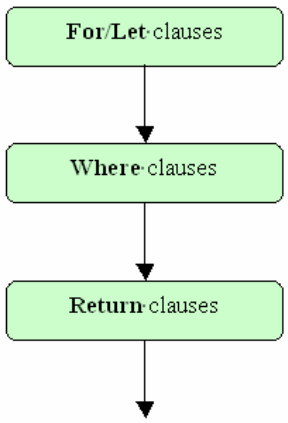

Figura 1.2: Ordine delle clausole FLWR.

- **For/Let clauses**: servono per assegnare un valore ad una o più variabili referenziate nella query. Mentre il `for` assegna un valore alla volta alla variabile, il `let` assegna una lista di nodi. Consideriamo l'Esempio 1.1 di documento XML che, per un certo negozio di cd, memorizza il nome del negozio, il suo indirizzo ed i cd che ha in negozio, ognuno con il proprio titolo, autore e lista delle tracce, inoltre ogni traccia è caratterizzata dal titolo della canzone.

Esempio 1.1

```
<cdstore>
  <name>Fangareggi</name>
  <address>
        <street>via Emilia Est</street>
        <city>Modena</city>
        <state>Italia</state>
   </address>
   <cd>
        <cdtitle>Pipe and Flowers</cdtitle>
        <vocalist>Elisa</vocalist>
        <tracklist>
              <passage>
                 <title>Labirinth</title>
              </passage>
```

```
        </tracklist>
    </cd>
        …..
</cdstore>
```

Un esempio di uso di `for` e `let` potrebbe essere il seguente:

```
for $x in /cdstore/cd
```

comporta tanti assegnamenti ad $x quanti sono i cd in `cdstore`, invece se i usa `let`:

```
let $x:=/cdstore/cd
```

comporta il singolo assegnamento della lista dei cd a $x.

Un espressione FLWR può contenere molte proposizioni `for` e `let`. Il risultato della sequenza di `for` e `let` è una lista di tuple di variabili assegnate.

- **Where clauses**: Solo le tuple per cui le condizioni nella sezione `where` sono vere vengono usate nella sezione `return`. I predicati presenti in questa sezione possono essere collegati con `and`, `or`, e `not`. Considerando sempre l'Esempio 1.1, supponiamo di volere tutti i cd che hanno come artista interprete Elisa, allora la query potrebbe essere formulata come segue:

```
for $x in /cdstore/cd
where $x/vocalist = Elisa
return $x
```

- **Return clauses:** Genera l'output della query, che può essere un nodo, un insieme di nodi o un valore primitivo. Consideriamo l'Esempio 1.2 e supponiamo di voler porre una query per poter trovare il titolo di ogni capitolo di ogni libro il cui argomento sia "XML".

Esempio 1.2

```
...
<Libro>
    <Titolo>XML 1.0 per tutti</Titolo>
    <Autore>Marco Rossi</Autore>
        <Capitolo>
                <Argomento>Introduzione</Argomento>
                <Titolo>Introduzione</Titolo>
            </Capitolo>
            <Capitolo>
                <Argomento>XML</Argomento>
```

```
            <Titolo>Linguaggio</Titolo>
        </Capitolo>
...
</Libro>
...
```

La richiesta avrà la seguente forma:

```
for $x in /Libro/Capitolo
where $x/Argomento = XML
return $x/Titolo
```

Si può notare come attraverso l'utilizzo delle parole chiave `for` e `where`, e l'impiego di variabili ($x) che indicano i percorsi (*path*) desiderati, si esprimano le condizioni da soddisfare per il ritrovamento dell'informazione, mentre, attraverso l'uso della parola chiave `return`, si esprima il risultato da ottenere ed il formato in cui il tale risultato deve comparire. La query può essere visualizzata graficamente tramite una rappresentazione ad albero in cui i nodi intermedi rappresentano gli elementi XML citati, mentre le foglie sono i contenuti degli stessi. Si veda la Figura 1.3 che rappresenta, appunto, la struttura ad albero della query posta sul documento dell'Esempio 1.2:

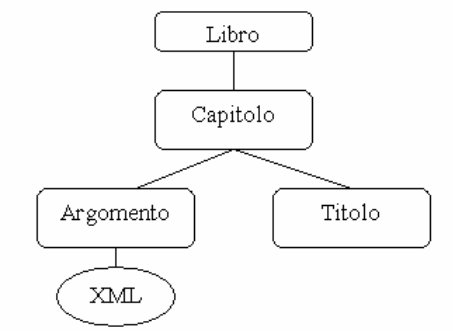

Figura 1.3: Albero rappresentativo della query

Dato che anche un documento XML può essere rappresentato come un albero, la risoluzione di una query XQuery diventa in realtà un problema di confronto fra due alberi e fra i cammini che li compongono.

1.8 Il problema della riscrittura della query

Una volta scelto ed implementato un linguaggio di interrogazione per documenti o schemi XML, non si sono ancora risolti tutti i problemi. Di solito l'archivi di una Biblioteca Digitale, che un utente vuole interrogare, è molto ampio, e quindi l'utente non può conoscere la struttura di tutto l'archivio o di tutti gli schemi dei documenti. Quindi egli dovrà poter porre una query su tutti i documenti dell'archivio e

affinché tale richiesta sia soddisfatta nel modo migliore, bisognerà riscriverla in modo automatico, per ogni documento utile a soddisfare la richiesta dell'utente.

Per comprendere meglio il problema, supponiamo che i documenti presenti in una Biblioteca Digitale si conformino ai due schemi riportati in Figura 1.4, i quali rappresentano l'esempio principale che verrà utilizzato nella presente tesi. Supponiamo ora che un utente voglia ottenere tutti i nomi di negozi che vendono cd di Elisa e che contengano canzoni con "gift" nel titolo. Si supponga anche che tramite l'interfaccia utente sia possibile porre la query nel linguaggio XQuery.

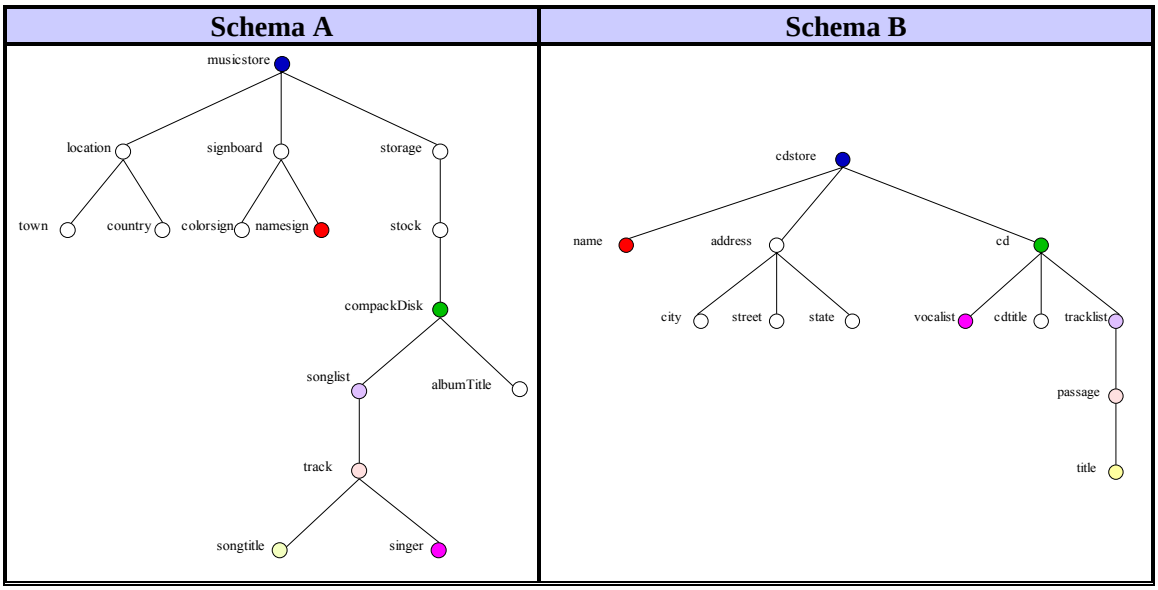

Figura 1.4: Rappresentazione degli alberi dei due schemi.

La query posta dall'utente, che non è a conoscenza della struttura dei documenti, potrebbe essere simile alla seguente:

```
for $x in /cdstore
where $x/cd/singer = ELISA
and $x/cd/song/title = gift
return $x/name
```

La query posta in questo modo non viene soddisfatta da nessuno degli schemi XML di Figura 1.4. Tuttavia tali schemi soddisfano i requisiti della richiesta che l'utente voleva porre nelle sue intenzioni, quindi i documenti contenuti nel repository della Biblioteca Digitale, che si conformano ad essi e che contengono le informazioni richieste, dovrebbero essere restituiti, ma ciò non potrà avvenire se la query non verrà riformulata in maniera adeguata. Cioè, un approccio di corrispondenza totale risulta

restrittivo, poiché limita l'insieme dei risultati rilevanti. In particolare la query sullo schema A dovrebbe essere posta nel seguente modo:

```
for $x in /musicstore
where $x/compacDisk/songlist/track/singer = ELISA
and $x/ compacDisk/songlist/track/songtitle = gift
return $x/signboard/namesign
```

Invece la query sullo schema B dovrebbe essere posta nel modo seguente:

```
for $x in /cdstore
where $x/cd/vocalist = ELISA
and $x/cd/tracklist/passage/title = gift
return $x/name
```

Si vede quindi da questo semplice esempio come sia importante il fatto di poter riscrivere in modo automatico la query posta dall'utente, in modo che la sua richiesta possa essere soddisfatta al meglio, interrogando tutti i documenti, presenti nella Biblioteca Digitale, che contengono concetti vicini a quelli espressi nella query, anche se questi documenti si conformano a schemi diversi. Infatti, come si vede osservando le due strutture ad albero di Figura 1.4, sia i documenti che si conformano allo schema A, che quelli che si conformano allo schema B, in effetti descrivono la stessa realtà, utilizzando però termini e strutture diverse per farlo. Quindi, un'operazione che si potrebbe applicare prima di riscrivere la query è quella di risolvere le differenze fra gli schemi stessi, ossia effettuare un'operazione di *matching* fra i termini degli schemi. Inoltre, l'uso di ontologie, combinato con l'operazione di matching, potrebbe aiutare ulteriormente nella risoluzione del problema della riscrittura della query.

Nel paragrafo successivo diamo una spiegazione di cosa si intende per ontologia e come sia possibile costruire ontologie su schemi XML tramite l'utilizzo di un sistema mediatore, al fine di utilizzarle in un'operazione di matching.

1.9 Ontologie XML

Un aspetto fondamentale nella realizzazione di una Biblioteca Digitale è la modalità di organizzazione dell'enorme quantità di informazione relativa al suo contenuto. Importantissimo, a questo proposito, è l'impiego di schemi (per documenti XML si propone di usare XML Schema) che descrivono il contenuto dell'archivio. Per rappresentare in maniera caratterizzante la conoscenza offerta dai metadati presenti negli schemi, è possibile impiegare ontologie e tecniche di ragionamento basate su di esse. Una *ontologia* può essere vista come un insieme di termini (vocaboli) in grado di definire in modo univoco un determinato concetto. Tramite l'utilizzo di ontologie, dunque, è possibile associare un concetto ad ogni elemento espresso dallo schema (o vista) rappresentante un insieme di documenti in

un archivio. Risulta quindi evidente come l'impiego di ontologie, e di tecniche di ragionamento basate su di esse, possa fornire uno strumento efficace per un accesso selettivo ed efficiente all'enorme quantità di informazioni che possono essere immagazzinate all'interno di una Biblioteca Digitale. Inoltre, utilizzando ontologie assieme ai metadati ed agli schemi degli oggetti contenuti in archivio è possibile, esprimendo i concetti collegati alle viste, risolvere il problema della riscrittura delle query su schemi differenti.

Il punto di partenza del progetto ECD su questo tema di ricerca è rappresentato dal sistema MOMIS [44,45,46] (*Mediating system Environment for Multiple Information Sources*), un sistema progettato e realizzato presso l'Università di Modena e Reggio Emilia a partire dal progetto MURST INTERDATA 97/98. MOMIS è un sistema a mediatori che permette la costruzione di una vista (schema) globale ed integrata di un insieme di sorgenti informative eterogenee e distribuite. La vista globale virtuale (GVV) è ottenuta a partire dalla rappresentazione, tramite l'impiego di ontologie, dei metadati, che descrivono lo schema di ogni sorgente locale. MOMIS supporta l'*annotazione* semiautomatica degli schemi: tramite un'interfaccia grafica, lo schema manager può associare ogni termine presente negli schemi con il significato che il termine assume in quel contesto. In MOMIS, il processo di annotazione si basa su WordNet (vedi Capitolo 3). Tramite l'interazione con questo sistema, MOMIS è in grado di scoprire relazioni fra i termini, basandosi sui concetti che esprimono, descritti nei vari metadati rappresentanti gli schemi delle sorgenti locali. Le relazioni, insieme alle ontologie degli schemi, rappresentano l'ontologia della Biblioteca Digitale. L'ontologia costituisce, quindi, lo strumento principale per mezzo del quale interrogare documenti XML con schemi differenti. La Figura 1.5 illustra bene la procedura di costruzione di una ontologia.

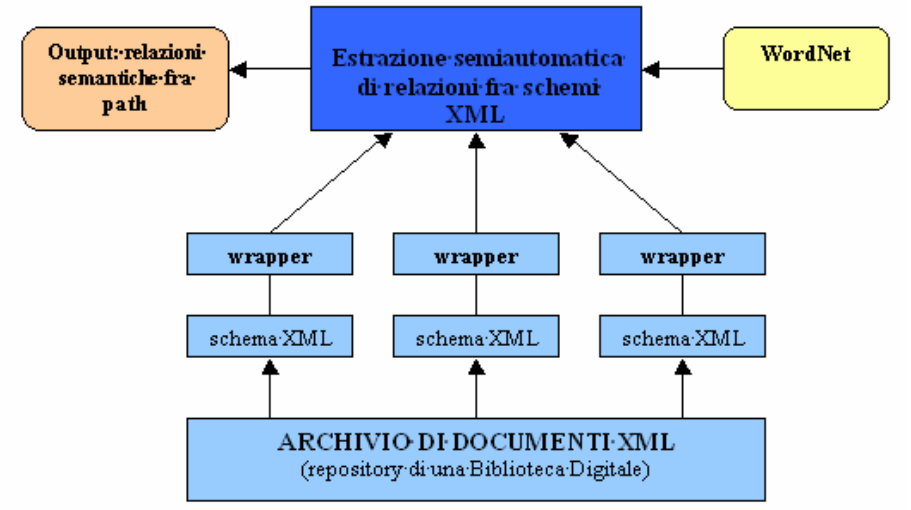

Figura 1.5: Costruzione dell'ontologia.

Sfruttando quindi queste ontologie, è possibile risolvere le differenze fra i diversi schemi utilizzando le annotazioni ricavate tramite l'uso del sistema MOMIS. Partendo quindi da questo punto, è maturata

l'idea che, per riscrivere la query, potrebbe essere utile effettuare prima un'operazione di matching tra i diversi schemi, ricavando dei valori di similarità tra i diversi termini, in modo da riformulare la query solo per quegli schemi i cui termini corrispondono ai termini della query fornendo un valore di similarità che superi almeno un certo valore di soglia.

Si è deciso quindi di sviluppare la tesi in questa direzione, partendo dal lavoro svolto in [24], dove Daniele Miselli propone e realizza un approccio basato su MOMIS per annotare gli schemi XML, ossia per costruire l'ontologia su tali schemi. Si potrebbe, quindi, fare il match fra i termini annotati, considerando la similarità tra i concetti che essi rappresentano, ma questo vorrebbe dire considerare lo schema XML solo come un insieme di termini, mentre in realtà gli schemi XML contengono delle informazioni strutturali che legano i concetti e che non possono essere trascurate in un'operazione di match completa. Nell'approccio sviluppato in questa tesi quindi, oltre alle ontologie, si sfrutteranno le informazioni strutturali fornite dagli schemi XML. Introduciamo, nel paragrafo successivo, l'operazione di match.

1.10 L'operazione di Match

Un'operazione fondamentale nella manipolazione delle informazioni di uno schema è il *match*, il quale prende come input due schemi, S1 e S2, e produce come output una mappatura tra gli elementi dei due schemi che corrispondono semanticamente l'uno all'altro. Definiamo una mappatura in modo tale da essere un insieme di *elementi di mappatura*, ognuno dei quali indica che certi elementi dello schema S1 sono fatti corrispondere con certi elementi nello schema S2. Inoltre, ogni elemento di mappatura può avere *un'espressione di mappatura* che specifica come gli elementi S1 e S2 sono collegati. L'espressione di mappatura può essere direzionale, ad esempio, una certa funzione che va dagli elementi di S1, referenziati dall'elemento di mappatura, agli elementi di S2, referenziati dall'elemento di mappatura. Oppure può essere non direzionale, cioè, una relazione tra una combinazione di elementi di S1 e S2.

L'operazione di match ha un ruolo centrale in numerose applicazioni, come l'integrazione di dati web-oriented, l'e-commerce, l'evoluzione e la migrazione di schemi, l'evoluzione di applicazioni, il data warehousing, ecc. Il matching fra schemi viene effettuato per molte ragioni: ogni schema, per rappresentare concetti identici, può avere una struttura e nomi differenti, gli schemi possono modellare contenuti simili, ma non identici, possono utilizzare parole simili, ma che hanno significati diversi e così via.

Attualmente, il matching tra schemi viene effettuato tipicamente in maniera manuale, magari supportato da un'interfaccia utente grafica. Ovviamente, una tale operazione, effettuata manualmente risulta lunga, noiosa e soggetta ad errori. Inoltre, il livello di sforzo è proporzionale al numero di match da eseguire. Si rende quindi necessario un approccio che sia più veloce e meno laborioso, ossia è necessario un supporto per il matching automatico fra gli schemi. In questa tesi si è implementato un metodo per effettuare, in maniera automatica, il matching fra i diversi schemi XML che descrivono i documenti presenti nel repository di una Biblioteca Digitale.

Per definire un operatore di match, bisogna prima scegliere una rappresentazione per i suoi schemi in ingresso e per la mappatura fornita in uscita, come un modello entity-relationship (ER), un modello object-oriented(OO) oppure un grafo etichettato. In ogni caso c'è una naturale corrispondenza tra i componenti della rappresentazione e le nozioni di elementi e struttura:

- entità e relazioni nei modelli ER;
- oggetti e relazioni nei modelli OO;
- elementi, sotto-elementi e IDREF in XML;
- nodi e rami nei grafi.

Il matching è un'operazione binaria che determina coppie di elementi corrispondenti dai suoi operandi in ingresso, opera su metadati (schemi) e ogni elemento di uno schema, in un risultato di match, può corrispondere più di un elemento dell'altro schema.

1.10.1 Classificazione degli approcci di Schema Matching

In questo paragrafo forniamo una classificazione dei principali approcci di schema matching. La Figura 1.6 mostra parte della classificazione insieme con alcuni esempi di approcci, spiegati in maniera più dettagliata in [25].

Un'implementazione dell'operazione di match può usare più algoritmi di matching o *matcher*. Ciò permette di selezionare il matcher a seconda del tipo di dominio o di schema che si sta trattando. Detto ciò, se si vogliono usare più matcher, si possono presentare due casi:

- Si possono realizzare algoritmi di match, ognuno dei quali calcola individualmente una mappatura, secondo un certo criterio.
- Si può realizzare una combinazione di algoritmi di match, o usando più criteri di match all'interno di un matcher ibrido oppure combinando più risultati di match, prodotti da diversi algoritmi di match.

Per i criteri seguiti dagli algoritmi di match individuali, consideriamo la seguente classificazione:

- *Instance vs. schema*: gli approcci di matching possono tenere conto solo dei dati istanza o solo delle informazioni a livello di schema.
- *Element vs. structure matching*: il matching può essere eseguito per singoli elementi di schema, come gli attributi, o per combinazioni di elementi, come strutture di schema complesse.

- *Language vs. constraint*: un algoritmo di match può far uso di un approccio di tipo linguistico (ad esempio, basato sui nomi e sulle descrizioni testuali degli elementi degli schemi) oppure un approccio basato su vincoli (ad esempio, basato su chiavi e relazioni).

- *Matching cardinality*: ogni elemento della mappatura risultante può far corrispondere uno o più elementi di uno schema con gli elementi dell'altro schema, producendo quatto casi: cardinalità 1:1, cardinalità 1:n, cardinalità n:1 e cardinalità n:m.

- *Informazioni ausiliarie*: molti algoritmi di match utilizzano anche informazioni ausiliarie, come dizionari, schemi globali, precedenti risultati di matching e dati forniti dagli utenti.

L'algoritmo di matching che è stato sviluppato in questa tesi combina due approcci (vedi Capitolo 4):

- Un approccio strutturale: si suppone che, in un grafo etichettato diretto, se due nodi sono simili, la loro similarità si propaga anche ai nodi vicini.

- Un approccio linguistico: si confrontano le gerarchie di ipernimi degli elementi dello schema, ricavando un valore iniziale di similarità

L'esplorazione delle gerarchie di ipernimi richiede l'uso di thesauri o dizionari. Noi utilizzeremo il database lessicale WordNet (vedi Capitolo 3) per ricavare tali gerarchie e le ontologie fornite da [24]. Inoltre, gli schemi XML trattati in questa tesi si considerano scritti utilizzando il linguaggio XML Schema, di cui viene fornita la sintassi nel Capitolo 2.

1.11 Lo stato dell'arte

Forniamo ora una panoramica sui diversi approcci seguiti finora sul problema dell'interrogazione approssimata delle query e sul problema del matching fra schemi.

Alcuni approcci sfruttano il fatto che sia la query che i documenti XML possono essere visti come delle strutture ad albero. Sono stati sviluppati quindi approcci che cercano di far corrispondere l'albero di query con l'albero del documento. In [17] viene proposto un metodo, per la corrispondenza della query, basato sul rilassamento della struttura dell'albero della query. La premessa di questo articolo è: *spesso è più appropriata una corrispondenza approssimata di strutture ad albero di query e la restituzione di una lista graduata di risposte.*

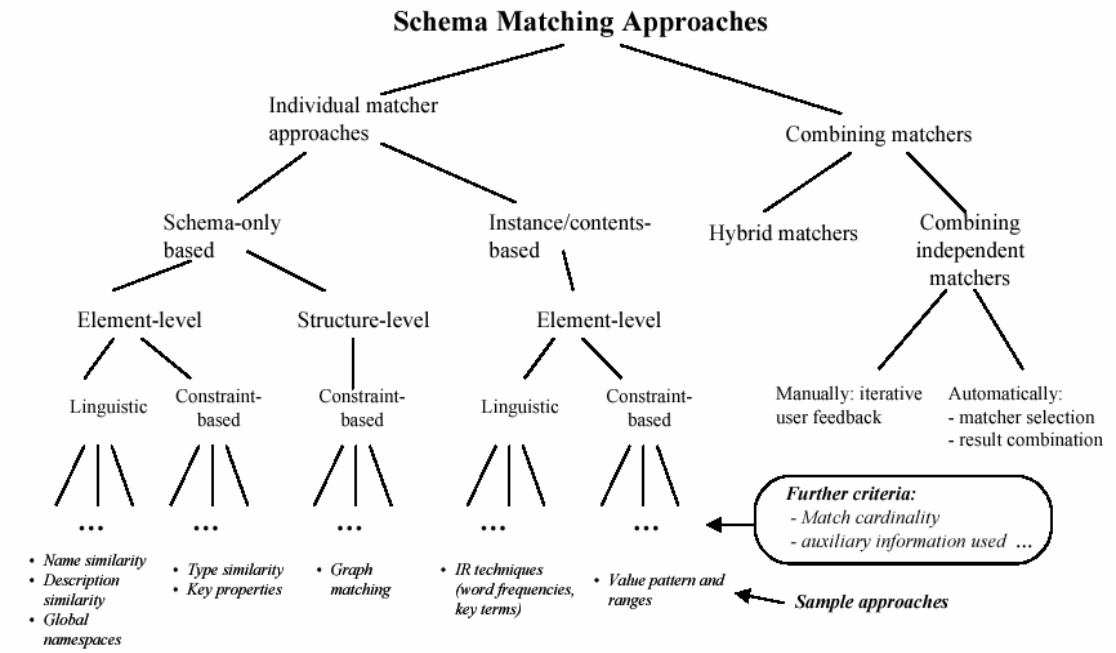

Figura 1.6: Classificazione degli approcci di schema matching esistenti

Un altro approccio è quello proposto in [18,19]. In questo lavoro si dice che a causa dell'eterogeneità delle grandi Biblioteche Digitali e della mancanza di conoscenza dell'organizzazione dei dati, condizioni di richieste di tipo booleane, portano spesso a risultati vuoti; quindi il rilassamento dei requisiti deve essere recepito come un "obbligo". Il risultato ottenuto, allo scopo di trovare e classificare similitudini fra l'albero della query e gli alberi dei documenti, consiste in una funzione di similarità detta SATES (*Scored Approximate Tree Embedding Set*). SATES fornisce come risultato, per ogni query, un insieme di alberi di dati (classificati a seconda della similarità) che approssimano quello della query. Il risultato è ottenuto sfruttando eventuali affinità semantiche presenti fra i nomi delle foglie e dei nodi dell'albero della query e degli alberi dei documenti. Sfruttando questo approccio, anche se computazionalmente pesante, si possono ritrovare informazioni desiderate, ma immagazzinate in archivio tramite termini simili, ma non uguali, a quelli espressi dalla query.

Un altro articolo interessante è [20] di Torsten Schlieder e Felix Naumann. Un utente potrebbe formulare in maniera corretta una query solo se fosse a conoscenza della struttura di tutti i documenti presenti nell'archivio, cosa molto difficile. Per semplificare il suo compito, allora, è necessario operare una trasformazione sulla query, in modo da adattarla alle strutture ad albero rappresentanti i documenti dell'archivio (si tratta esattamente del problema di riscrittura delle query citato in precedenza). A questo scopo viene proposto un nuovo linguaggio per l'interrogazione di documenti XML chiamato *ApproXQL* [21].

Per quanto riguarda lo stato dell'arte a proposito del matching automatico fra schemi, sono stati sviluppati numerosi prototipi. Inizialmente, molto del lavoro fatto si era focalizzato su particolari

domini di applicazione e su determinati formati di schemi. Ad esempio, i seguenti tre prototipi, utilizzati nel dominio dell'integrazione dati:

- SemInt [26,27,28] prodotto dalla Northwestern Università, che crea una mappatura tra i singoli attributi di due schemi. SemInt sfrutta 15 criteri basati sui vincoli e 5 criteri basati sul contenuto. Nel suo approccio principale, SemInt usa le reti neurali per determinare i candidati di match.

- LDS (*Learning Source Descripions*) [29], prodotto dall'Università di Washington, è un sistema che usa delle tecniche di machine-learning per far corrispondere a una nuova sorgente dati uno schema globale precedentemente deciso, producendo una mappatura 1:1 di tipo atomic-level.

- SKAT (*Semantic Knowledge Articulation Tool*) [30], prodotto dall'Università di Stanford, segue un approccio rule-based per determinare in maniera semiautomatica le corrispondenze tra due ontologie (schemi). SKAT viene usato all'interno dell'architettura ONION per l'integrazione di ontologie [31]. In ONION le ontologie vengono trasformate in modelli di database graph-based e object-oriented. Le regole di matching tra ontologie vengono usate per costruire una "ontologia articolata", la quale copre le "intersezioni" di sorgenti di ontologie.

Invece i prototipi ARTEMIS e DIKE sono usati per l'integrazione di schemi:

- ARTEMIS [42,43], prodotto dalle Università di Milano e Brescia, è uno strumento per l'integrazione di schemi. Prima calcola le "affinità", nell'intervallo da 0 a 1, fra attributi, quindi completa l'integrazione di schema raggruppando gli attributi in base alle loro affinità, dopodiché costruisce delle viste basate sui raggruppamenti. L'algoritmo opera su un modello ibrido relational-OO che include il nome, i tipi di dati e le cardinalità di attributi e di tipi di oggetti destinazione di attributi che si riferiscono ad altri oggetti. Esso calcola le corrispondenze tramite una somma pesata di affinità di nome e tipo di dato e affinità strutturale. ARTEMIS è utilizzato come un componente di MOMIS (vedi paragrafo 1.9 sulle ontologie XML).

- DIKE [32,33], prodotto dall'Università di Reggio Calabria, propone algoritmi per determinare automaticamente sinonimi e inclusioni (is-a, ipernimi), relazioni tra oggetti di vari schemi entity-relationship. Gli algoritmi sono basati su un insieme di sinonimo, omonimo e proprietà di inclusione specificati dall'utente, che comprendono un "fattore di plausibilità" numerico (tra 0 e 1) sulla certezza che ci si aspetti che la relazione duri. Per derivare (in modo probabilistico) nuovi sinonimi e omonimi e i fattori di plausibilità associati, gli autori eseguono un confronto a coppie di oggetti negli schemi immessi considerando le proprietà delle somiglianze dei loro "oggetti collegati" (cioè, i loro attributi e l'is-a e le altre relazioni a cui gli oggetti partecipano).

I prototipi TransScm e Cupid sono stati utilizzati per attività di traduzione dei dati, come nel dominio e-commerce.

- TransScm [31], prodotto dall'Università di Tel Aviv, utilizza il matching fra schemi per derivare una traduzione automatica fra istanze di schema. Gli schemi in ingresso vengono trasformati in grafi etichettati. I rami, nei grafi degli schemi, rappresentano la componente di relazione. Tutte le altre informazioni dello schema sono rappresentate come proprietà dei nodi. Il matching viene effettuato nodo per nodo, cominciando dall'alto e supponendo un alto grado di similarità tra gli schemi. Ci sono molti matcher, i quali vengono cercati in un ordine fissato.

- Cupid [41], combina un matcher di nomi con un algoritmo di match strutturale in maniera ibrida sofisticata. I valori di somiglianza dei nomi e dei tipi di dati vengono combinati con regole euristiche di livello di struttura per fornire un risultato migliore. Gli schemi vengono convertiti in alberi e per risolvere le relazioni multiple tra un nodo condiviso e i suoi nodi genitori, vengono aggiunti nodi addizionali.

Un ultimo prototipo che citiamo è dato dal sistema COMA [47], il quale fornisce una libreria estendibile di vari matcher e supporta diversi modi di combinare risultati di match. Attualmente, i matcher sfruttano informazioni di schema a livello di struttura e a livello di elemento. Matcher speciali si occupano di riutilizzare risultati provenienti da match precedenti e da interazioni con gli utenti. Nella rappresentazione interna, gli schemi vengono trasformati in grafici diretti aciclici radicati, su cui operano tutti gli algoritmi di corrispondenza. Ogni elemento di schema è identificato univocamente dal suo percorso completo dalla radice del grafico al nodo corrispondente.

Capitolo 2

Il linguaggio XML Schema

2.1 Origini

Il linguaggio XML (*Extensible Markup Language*) [16], venne pensato, inizialmente, solo per esprimere documenti di testo: libri, manuali, pagine web ecc. Nasce poi l'idea che XML possa servire per qualcosa più che per la sola rappresentazione dei dati, cioè XML può essere (anche) un linguaggio di markup per trasferire dati. XML quindi, non è pensato per la visione umana, ma per essere prodotto ed usato da programmi. Secondo Adam Bosworth, XML è un'interfaccia:

- Un'interfaccia tra autore e lettore, attraverso XSL (*eXtensible Styles Language*) e Xlink (*XML Linking Language*), per portare significato tra creatore ed utente.
- Un'interfaccia tra *applicazione ed applicazione*, attraverso XML Schema, per esprimere contratti sui formati, e verificarne il rispetto.

2.2 Documenti validi e documenti ben formati

Per essere ben formato, un documento XML deve seguire le regole di sintassi stabilite per XML dal consorzio W3C nelle specifiche XML 1.0. In modo informale, la corretta formazione di un documento dal punto di vista grammaticale spesso significa che il documento deve contenere uno o più elementi e che l'elemento radice deve contenere tutti gli altri elementi. L'innestamento dei tag deve essere fatto in maniera corretta. Per esempio, il documento che segue non è ben formato poiché il tag di chiusura </GREETING> segue il tag di apertura <MESSAGE> dell'elemento successivo:

```
<?xml version=”1.0” encoding =”UTF-8”?>
  <DOCUMENT>
     <GREETING>
        Hello from XML
     <MESSAGE>
     </GREETING>
        Welcome to the wild and woolly word of XML.
     </MESSAGE>
   </DOCUMENT>
```

La maggior parte dei browser XML verificherà il documento per controllare se è ben formato. Alcuni browser tuttavia possono anche controllare se è valido. Un documento XML viene detto valido se è ben formato ed inoltre rispetta tutti i vincoli strutturati definiti nella DTD (*Document Type Definition*) o nello schema cui si riferisce.

2.3 XML Schema e DTD

Col passare del tempo molte persone hanno protestato con il consorzio W3C per la complessità delle DTD e hanno richiesto metodi più semplici. W3C ha preso in carico la richiesta e ha istituito un comitato per esaminare il problema e ha prodotto una soluzione che è in verità più complessa delle DTD: gli schemi XML.

D’altro canto gli schemi sono anche molto più potenti e precisi rispetto alle DTD. Con gli schemi non solo è possibile specificare la sintassi di un documento come si farebbe con una DTD, ma è anche possibile specificare il tipo di dati effettivo del contenuto di ciascun elemento, ereditare la sintassi da altri schemi, inserire note negli schemi, usare gli schemi con più namespace, creare tipi di dati semplici e complessi, specificare il numero minimo e massimo di volte che un elemento può essere presente, creare i tipi di liste, creare i gruppi di attributi, limitare le gamme dei valori che gli elementi possono contenere, limitare quali altri schemi possono essere limitati dai propri, unire insieme frammenti di più schemi, richiedere che i valori degli attributi o elementi siano univoci e così via.

Lo scopo di XML Schema è quello di definire una classe di documenti XML, permettendone la validazione. I documenti XML che corrispondono alla descrizione fornita da un XML Schema vengono chiamati documenti ‘istanza’ di quel particolare schema. L’invenzione di tale linguaggio è stata resa necessaria anche dal fatto che le DTD sono fornite con una sintassi a se stante, differente da quella dell’XML vero e proprio, mentre gli schemi si basano su descrizioni fornire in XML, permettendo di lavorare su di essi con strumenti già creati per tale standard e risultando molto più comodi per i programmatori. Le specifiche per gli schemi XML si possono trovare in [38,39,40].

2.4 Primo esempio di documento XML Schema

Per comprendere meglio la struttura di XML Schema iniziamo con l'analizzare un esempio. L'esempio di schema XML che verrà utilizzato in questo capitolo è diverso dai due schemi XML usati per spiegare l'algoritmo di matching (vedi Capitolo 4). Si è fatta questa scelta semplicemente perché lo schema dell'Esempio 4 è più completo e quindi si presta meglio per evidenziare le principali caratteristiche di XML Schema.

Esempio 2.1

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:annotation>
    <xsd:documentation>
          Book borrowing transaction schema.
    </xsd:documentation>
</xsd:annotation>

<xsd:element name="transaction" type="transactionType"/>

<xsd:complexType name="transactionType">
    <xsd:element name="Lender" type="address"/>
    <xsd:element name="Borrower" type="address"/>
    <xsd:element ref="note" minOccurs="0"/>
    <xsd:attribute name="borrowDate" type="xsd:date"/>
</xsd:complexType>

<xsd:element name="note" type="xsd:string"/>

<xsd:complexType name="address">
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" minOccurs="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:attribute name="phone" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:complexType name="books">
    <xsd:element name="book" minOccurs="0" maxOccurs="10">
        <xsd:complexType>
            <xsd:element name="bookTitle" type="xsd:string"/>
            <xsd:element name="pubDate" type="xsd:date" minOccurs="0"/>
            <xsd:element name="replacementValue" type="xsd:decimal"/>
```

```
            <xsd:element name="maxDaysOut">
                <xsd:simpleType base="xsd:integer">
                    <xsd:maxExclusive value="14"/>
                </xsd:simpleType>
            </xsd:element>
            <xsd:attribute name="bookID" type="catalogID"/>
        </xsd:complexType>
    </xsd:element>
</xsd:complexType>

<xsd:simpleType name="catalogID" base="xsd:string">
    <xsd:pattern value="\ d{ 3}  -\ d{ 4}  -\ d{ 3} "/>
</xsd:simpleType>

</xsd:schema>
```

Questo esempio è relativo alla registrazione di libri prestati. In questo caso l'elemento radice del documento,<transaction>, è definito di tipo 'transactionType' e questo elemento può contenere diversi altri elementi, inclusi quelli dei tipi 'address' e 'books'. Lo stesso tipo 'address' è definito per contenere elementi che hanno al loro interno il nome e l'indirizzo di una persona e il tipo 'books' contiene elementi chiamati <book> che descrivono un libro, compreso il titolo, la data di pubblicazione e così via.

Per capire meglio l'utilizzo di questo XML Schema, riportiamo di seguito un possibile documento istanza (book.xml) che utilizza lo schema dell'Esempio 2.1 e che quindi è conforme ad esso:

```
<?xml version="1.0" encoding="UTF-8"?>

<transaction borrowDate="2003-7-31">

  <Lender phone="607.555.2222">
        <name>Doug Glass</name>
        <street>416 Disk Drive</street>
        <city>Medfield</city>
        <state>MA</state>
  </Lender>

  <Borrower phone="310.555.1111">
       <name>Britta Regensburg</name>
       <street>217 Union Drive</street>
       <city>Medfield</city>
       <state>CA</state>
  </Borrower>
```

```
  <note>Lender wants these back in two weeks!</note>

  <books>
        <book bookID="123-4567-890">
              <bookTitle>Earthquakes for Breakfast</bookTitle>
              <pubDate>2003-10-20</pubDate>
              <replacementValue>15.95</replacementValue>
              <maxDaysOut>14</maxDaysOut>
        </book>
        <book bookID="123-4567-891">
              <bookTitle>Avalanches for Lunch</bookTitle></bookTitle>
              <pubDate>2003-10-21</pubDate>
              <replacementValue>11.95</replacementValue>
              <maxDaysOut>14</maxDaysOut>
        </book>
  </books>

</transaction>
```

Vediamo che in questo documento è presente un elemento <transaction> con un attributo "borrowDate". Sono presenti un prestatore e un beneficiario del prestito con i relativi dati 'name', "street", "city" e "state"; è inoltre presente una "note". E' quindi chiaro come uno schema possa definire strutturalmente e, quindi, validare una pagina XML. Vediamo ora, in dettaglio, la struttura e la sintassi degli XML Schema.

2.5 Sintassi di XML Schema

2.5.1 Il tag Schema

Ogni schema XML deve essere completamente contenuto all'interno del tag `schema`. Un generico documento si presenterà quindi con questa struttura:

```
<schema attributoSchema1=" ..." attributoSchema2=" ..." ... >
      Dichiarazione degli elementi
</schema>
```

Gli attributi che possono comparire all'interno di questo elemento, come `xmlns` nell'Esempio 2.1, servono a definire i namespace di riferimento per gli elementi dello schema. Per una trattazione più approfondita si veda il paragrafo sui namespace.

2.5.2 Dichiarazione dei tipi e degli elementi

La cosa fondamentale da comprendere sugli schemi XML è il concetto dell'uso dei tipi semplici e complessi e come si mettono in relazione alle dichiarazioni degli elementi.

A differenza della DTD, si deve specificare il tipo di elementi che si dichiarano con gli schemi. Questo significa che il primo passo nel dichiarare gli elementi è accertarsi di avere i tipi che si desidera questo spesso significa definire nuovi tipi complessi. I tipi complessi possono racchiudere elementi e avere attributi a differenza dei tipi semplici.

E' possibile creare nuovi tipi complessi usando l'elemento <xsd:complexType> negli schemi. Una definizione di tipo complesso normalmente contiene dichiarazioni di elementi, riferimenti ad altri elementi e dichiarazioni di attributi. Si dichiarano gli elementi con l'elemento <xsd:element> e si dichiarano gli attributi con l'elemento <xsd:attribute>.

Le dichiarazioni di elementi specificano la sintassi di un elemento e volendo anche il tipo dell'elemento. Inoltre è possibile specificare il tipo di attributi. Consideriamo un esempio tratto dall'Esempio 2.1 visto precedentemente:

<xsd:element name="transaction" type="transactionType"/>

L'attributo `type` indica il tipo a cui l'elemento <transaction> deve appartenere. Un tipo può essere:

- *Semplice*: al suo interno non può contenere né elementi né attributi.
- *Complesso:* può contenere al suo interno sia elementi che attributi.

Prendiamo in considerazione in maniera più approfondita il tipo "complexType" considerando il seguente esempio:

```
<xsd:complexType name="address">
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" minOccurs="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:attribute name="phone" type="xsd:string" use="optional"/>
</xsd:complexType>
```

In questo caso si dichiara un tipo complesso che contiene gli elementi che costituiscono l'indirizzo di una persona. Si userà "address" come il tipo degli elementi <Lender> e <Borrower> per poter memorizzare gli indirizzi di chi presta e di chi riceve in prestito il libro. Questa dichiarazione appare nel seguente modo:

```
<xsd:complexType name="transactionType">
    <xsd:element name="Lender" type="address"/>
    <xsd:element name="Borrower" type="address"/>
```

```
        <xsd:element ref="note" minOccurs="0"/>
        <xsd:attribute name="borrowDate" type="xsd:date"/>
    </xsd:complexType>
```

Nel tipo “address” si indica che qualsiasi elemento di questo tipo deve possedere cinque elementi e un attributo. Questi elementi sono <name>, <street>, <city> e <state> e l’attributo è “phone”.

La definizione del tipo complesso “address” contiene solo le dichiarazioni basate sui tipi semplici “xsd:string”. D’altra parte, come già detto, I tipi complessi possono contenere elementi che sono basati sui tipi complessi. Si può vedere come questo funziona in “transactionType”, che è il tipo dell’elemento radice <transaction> di book.xml. In questo caso due elementi, <Lender> e <Borrower> sono di tipo “address”. Si può notare che il tipo “transationType” include anche un attributo, “borrowDate”, che è del tipo semplice “xsd:date”. Gli attributi sono sempre di un tipo semplice poiché sono senza contenuto.

Dopo aver definito un nuovo tipo, è possibile dichiarare nuovi elementi di questo tipo. Per esempio, dopo la dichiarazione “transactionType”, si può dichiarare l’elemento <transaction>, che è l’elemento radice del documento, affinché sia di questo tipo:

```
    <xsd:complexType name="transactionType">
        <xsd:element name="Lender" type="address"/>
        <xsd:element name="Borrower" type="address"/>
        <xsd:element ref="note" minOccurs="0"/>
        <xsd:attribute name="borrowDate" type="xsd:date"/>
    </xsd:complexType>
```

Quindi abbiamo visto che se si vuole usare un tipo complesso, è necessario crearlo con l’elemento <xsd:complexType>.

Consideriamo ora con attenzione la dichiarazione dell’elemento <note> nel tipo “transactionType”:

```
    <xsd:complexType name="transactionType">
        <xsd:element name="Lender" type="address"/>
        <xsd:element name="Borrower" type="address"/>
        <xsd:element ref="note" minOccurs="0"/>
        <xsd:attribute name="borrowDate" type="xsd:date"/>
    </xsd:complexType>
```

In questo esempio non è stato dichiarato un nuovo elemento, è stato incluso, invece, un elemento *di riferimento* già esistente. L’elemento <note> esiste già come nell’esempio:

```
    <xsd:complexType name="transactionType">
        <xsd:element name="Lender" type="address"/>
        <xsd:element name="Borrower" type="address"/>
        <xsd:element ref="note" minOccurs="0"/>
```

```
        <xsd:attribute name="borrowDate" type="xsd:date"/>
    </xsd:complexType>

    <xsd:element name="note" type="xsd:string"/>
```

L'uso dell'attributo `ref` permette di includere un elemento che è già stato definito in una definizione di tipo complesso. Tuttavia, non p possibile includere alcun elemento tramite un riferimento. L'elemento di riferimento deve essere stato dichiarato *in modo globale*, non deve, cioè, far parte di nessun altro tipo complesso. Un elemento globale o una dichiarazione di attributo appare come un elemento figlio immediato all'elemento <xsd:schema>.

2.5.2.1 Frequenza della ricorrenza degli elementi

In generale, è possibile specificare il numero minimo di volte che un elemento può apparire con l'attributo `minOccurs` e il numero massimo di volte che può apparire con `maxOccurs`. Per esempio, ecco come si dovrà dichiarare che l'elemento <note> dovrà apparire da zero a cinque volte nel tipo "transactionType":

```
<xsd:element ref="note" minOccurs="0" maxOccurs="5"/>
```

Il valore predefinito di `minOccurs` è 1. Se non si specifica un valore per `maxOccurs`, il valore predefinito è il valore di `minOccurs`. Per indicare che non esiste un limite superiore per l'attributo `maxOccurs`, è necessario impostarlo al valore `unbounded`.

2.5.2.2 Impostazione dei valori predefiniti degli elementi

All'interno degli attributi `minOccurs` e `maxOccurs` è possibile usare i valori `fixed` e `default`, per indicare i valori che l'elemento deve contenere (si userà o l'uno o l'altro, mai insieme).

Per esempio, di seguito è impostato il valore dell'elemento denominato <maxTrials> a 100 ed è specificato che deve essere sempre 100, quando si usa l'attributo `fixed` nell'elemento <xsd:element>:

```
<xsd:element name="maxTrials" type="xsd:integer" fixed="100"/>
```

Nell'esempio che segue viene dato all'elemento il valore di `default` 100 invece di stabilire il valore 100, che può essere utile quando si desidera fornire valore di default che dovranno essere utilizzati se l'utente non specifica un valore alternativo:

```
<xsd:element name="maxTrials" type="xsd:integer" default="100"/>
```

2.5.3 Impostazione dei vincoli e dei valori di default degli attributi

Come per gli elementi, è possibile specificare il tipo di attributo. A differenza degli elementi, tuttavia, gli attributi devono essere di un tipo semplice. Inoltre, gli attributi non usano `minOccurs` e `maxOccurs` perché possono apparire al massimo una volta sola. Invece si usa una sintassi diversa quando si definiscono i vincoli per gli attributi.

Gli attributi si dichiarano con l'elemento <xsd:attribute>. Tale elemento ha un attributo di tipo che fornisce (semplicemente) il tipo dell'attributo. Perciò come si può indicare che un attributo è obbligatorio o facoltativo o se esiste un valore di default o anche se il valore dell'attributo è stabilito a un determinato valore? Si usano gli attributi `use` e `value` dell'elemento <xsd:attribute>.

L'attributo `use` specifica se l'attributo è `required` oppure `optional`. Se l'attributo è `optional`, l'attributo `use` specifica se il valore dell'attributo è `fixed` o se è un valore `default`. Il secondo attributo, `value`, contiene qualsiasi valore necessario.

Per esempio, è stato aggiunto un attributo denominato "phone" al tipo "address". Questo attributo è di tipo "xsd:string" e il suo uso è `optional`:

```
<xsd:complexType name="address">
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" minOccurs="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:attribute name="phone" type="xsd:string" use="optional"/>
</xsd:complexType>
```

Di seguito sono riportati i possibili valori dell'attributo `use`:

- `required`: l'attributo è obbligatorio e può avere qualsiasi valore.
- `optional`: il valore dell'attributo è fisso e può avere qualsiasi valore.
- `fixed`: il valore dell'attributo è fisso e si può impostare il valore con l'attributo `value`.
- `default`: se non esiste l'attributo, il suo valore è quello di default impostato con l'attributo `value`. Se non appare il suo valore, è il valore che è assegnato nel documento.
- `prohibited`: l'attributo non deve apparire.

2.5.4 Creazione di tipi semplici

I tipi semplici si differenziano in due sottocategorie

- *Tipi semplici primitivi*

- *Tipi semplici derivati*

I tipi semplici primitivi,come *string* o *decimal*, sono definiti direttamente all'interno del linguaggio proprio di XML Schema (built-in datatype), mentre i tipi semplici derivati sono ottenuti partendo da quelli di base tramite costrutti quali l'enumerazione, l'unione, le liste o la restrizione di tipo. Nei prossimi paragrafi forniremo una descrizione dettagliata di tutti questi sotto-tipi.

2.5.4.1 I tipi semplici primitivi

I tipi semplici primitivi sono stati definiti direttamente all'interno di XML Schema che, essendo un linguaggio fortemente tipizzato, ne presenta numerosi. Viene riportata di seguito una tabella che li riassume tutti.

Tipo	Descrizione
binary	Contiene valori binary, come 110001
boolean	Contiene valori come True, False, 1, 0
byte	Rappresenta il valore di un byte, come 123; valore Massimo 255
century	Contiene un secolo, come 20
date	Rappresenta una data nel formato YYYY-MM-DD, come 2003-08-27
decimal	Contiene valori decimali, come 5.4, 0, -219.06
double	Rappresenta un numero a virgola mobile a 64-bit a doppia precisione
ENTITIES	Rappresenta il tipo di attributo ENTITIES di XML 1.0
ENTITY	Rappresenta il tipo di attributo ENTITY di XML 1.0
float	Rappresenta un numero a virgola mobile a 32-bit a singola precisione
ID	Rappresenta il tipo di attributo ID di XML 1.0
IDREF	Rappresenta il tipo di attributo IDREF di XML 1.0
IDREFS	Rappresenta il tipo di attributo IDREFS di XML 1.0
int	Rappresenta un intero, come 123456789
integer	Rappresenta un intero
language	Contiene un identificativo di lingua, come de o en-US, come definito in XML 1.0
long	Rappresenta un intero lungo, come 12345678901234
gMonth	Contiene un mese, come 2001-10
Name	Rappresenta il tipo Name di XML 1.0
NCName	Contiene un nome XML senza un prefisso di namespace e punto
negativeInteger	Rappresenta un intero negativo
NMTOKEN	Rappresenta il tipo di attributo NMTOKEN di XML 1.0
NMTOKENS	Rappresenta il tipo di attributo NMTOKENS di XML 1.0
nonNegativeInteger	Rappresenta un intero non negativo
nonPositiveInteger	Rappresenta un intero non positivo
NOTATION	Rappresenta il tipo di attributo NOTATION di XML 1.0
positiveInteger	Rappresenta un intero positivo
Qname	Rappresenta il tipo XML Namespace Qualified Name
gYearDay	Specifica una data ricorrente, come --10-15, che significa ogni 15 ottobre

gDay	Specifica un giorno ricorrente, come ----31
anyURI	Contiene un URI, come http://www.w3.org
short	Rappresenta un intero breve, come 12345
string	Rappresenta una stringa di testo
time	Rappresenta un'ora, come 12:00:00:00
duration	Contiene una durata, come P1Y2M3DT10H30M12.3S
dateTime	Contiene il tempo nel formato seguente: 2001-10-1t12:00:00.000-05:00
unsignedByte	Rappresenta un valore di byte senza segno
unsignedInt	Rappresenta un intero senza segno
unsignedLong	Rappresenta un intero lungo senza segno
unsignedShort	Rappresenta un intero breve senza segno
Base64Binary	Valore codificato tramite Base64 Content-Transfer-Encoding defined in Section 6.8 of [RFC 2045]
token	Come normalizedString, non può avere due spazi consecutivi
hexBinary	Rappresenta un numero binario in formato esadecimale, come 0FB7
normalizedString	Stringa non contenente Carriage Return, Line Feed o Tab
gYear	Contiene un anno, come 2001

Tabella 2.1: Tipi semplici primitivi

2.5.4.2 I tipi semplici derivati

Possiamo raggruppare i tipi semplici derivati in tre categorie:

- *Liste*
- *Unioni*
- *Tipi semplici derivati per restrizione*

Ogni volta che si definisce un tipo semplice, bisogna usare il tag:
<xsd:simpleType name=”NomeNuovoTipoSemplice”>….</simpleType>

Per capire meglio i tipi semplici derivati faremo riferimento ad un nuovo esempio :

Esempio 2.2

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="Order" type="OrderType"/>

    <xsd:element name="Comment" type="xsd:string"/>

    <xsd:complexType name="OrderType">
        <xsd:element name="Address" type="AddressType"/>
```

```
    <xsd:element name="Object" type="ObjectType" minOccurs="1" maxOccurs="unbounded"/>
    <xsd:element ref="Comment" minOccurs="0"/>
    <xsd:attribute name="Date" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="AddressType">
    <xsd:element name="State" type="StateType"/>
    <xsd:element name="City" type="xsd:string"/>
    <xsd:element name="Street" type="xsd:string "/>
    <xsd:element name="Number" type="NumberType"/>
</xsd:complexType>

<xsd:complexType name="ObjectType">
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Quantity" type="QuantityType"/>
    <xsd:element name="PriceEuro" type="xsd:decimal"/>
    <xsd:element name="ColorAvailable" type="ListColor"/>
    <xsd:element ref="Comment" minOccurs="0"/>
</xsd:complexType>

<xsd:simpleType name="QuantityType">
    <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxExclusive value="100"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="StateType">
    <xsd:restriction base="string">
        <xsd:enumeration value=" Italy"/>
        <xsd:enumeration value="USA"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="ListColor">
    <xsd:list itemType="ColorType"/>
</xsd:simpleType>

<xsd:simpleType name="ColorType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Red"/>
        <xsd:enumeration value="Black"/>
        <xsd:enumeration value="Green"/>
    </xsd:restriction>
</xsd:simpleType>
```

```
    <xsd:simpleType name="NumberType">
        <xsd:union memberTypes="xsd:string xsd:positiveInteger"/>
    </xsd:simpleType>

</xsd:schema>
```

Tramite la definizione di nuovi tipi semplici è possibile specificare nell'Esempio 2.2 che i soli valori possibili per l'elemento "State", figlio dell'elemento "AddessType", sono Italy e USA. E' presente inoltre una lista di colori disponibili nella descrizione dell'oggetto, colori che possono essere presi fra quelli definiti dal tipo semplice "ColorType".

2.5.4.3 Il tipo semplice derivato unione

Di solito il valore di un tipo semplice è definito dall'attributo `type`. Nel caso del tipo semplice union invece, il valore dell'elemento può essere uno a scelta fra quelli contenuti nell'attributo `memberType` dell'elemento <union>. Quindi, in una stessa istanza XML potremmo ritrovare elementi di tipo "NumeroType" il cui valore è fornito da un numero positivo intero e altri rappresentati una stringa.

Considerando il seguente pezzo di schema XML dell'Esempio 2.2:

```
    <xsd:simpleType name="NumberType">
        <xsd:union memberTypes="xsd:string xsd:positiveInteger"/>
    </xsd:simpleType>
```

Una parte di documento XML che usa tale schema potrebbe risultare come segue:

```
<Address>
        <State>Italy</State>
        <City>Modena</City>
        <Street>via Puccini</Street>
        <Number>7</Number>
</Address>
…..
<Address>
        <State>Italy</State>
        <City>Modena</City>
        <Street>via Puccini</Street>
        <Number>sette</Number>
</Address>
```

2.5.4.4 Il tipo semplice derivato lista

Per fare in modo che il contenuto di un elemento di tipo semplice non sia composto solamente da un tipo atomico, ma da una lista di valori di tipo atomico, separati da spazi bianchi, viene utilizzato l'elemento <list>. Vediamo un esempio:

```
<xsd:element name=”CodBook” type=”ListNum”/>
<xsd:simpleType name=”ListNum”>
    <xsd:list itemType=”xsd:integer”/>
</xsd:simpleType>
```

Quindi in un documento XML istanza potrebbe comparire l'elemento:
<CodBook> 1 234-3435 0 23 980-4</CodBook>

Osserviamo che l'attributo `itemType` dell'elemento <list> contiene il tipo atomico (che deve essere, proprio come nel caso dell'attributo `memberTypes` per l'unione, di tipo semplice) da cui si ottiene la lista.

Anche nel nostro Esempio 2.2 compare la definizione di un tipo semplice lista:

```
<xsd:simpleType name=”ListColor”>
    <xsd: list itemType=”ColorType”/>
</xsd:simpleType>
```

“ColorType” è un tipo semplice ottenuto da una restrizione enumerazione del tipo primitivo string. Per una descrizione del concetto di restrizione si rimanda al paragrafo successivo.

2.5.4.5 Il tipo semplice derivato per restrizione

In XML Schema è possibile definire nuovi tipi di dati semplici tramite il concetto di restrizione. In questo modo, partendo da un tipo primitivo semplice, si ottiene un nuovo tipo, il cui valore può essere considerato come un sottoinsieme proprio del valore del tipo primitivo.

Il concetto di restrizione viene espresso tramite il tag <xsd:restriction> e il tipo primitivo su cui viene svolta la restrizione è espresso come valore dell'attributo `base`. All'interno del tag <restriction> deve comparire il tipo di restrizione (che viene chiamato in gergo *facet* o *sfaccettature*) da applicare. Consideriamo le seguenti righe tratte dall'Esempio 2.2:

```
<xsd:simpleType name="StateType">
    <xsd:restriction base="string">
        <xsd:enumeration value="Italy"/>
        <xsd:enumeration value="USA"/>
    </xsd:restriction>
```

```
</xsd:simpleType>
```

Nell'esempio precedente viene usato il facet <xsd:enumeration> (enumerazione), che fornisce un elenco di tutti i possibili valori (appartenenti al tipo di base, quindi stringhe) che possono comparire nel tipo "StateType"; in un documento XML istanza, quindi, ogni elemento appartenente al tipo "StateType" potrà contenere solamente il valore stringa Italy oppure il valore stringa USA.

```
<xsd:simpleType name="StateType">
    <xsd:restriction base="string">
        <xsd:enumeration value="Italy"/>
        <xsd:enumeration value="USA"/>
    </xsd:restriction>
</xsd:simpleType>
```

Ci sono numerosi *facets* che possono essere usati nelle restrizioni:

- *Length*: per specificare il numero di caratteri che deve avere una stringa.
- *MaxLength*: specifica il numero massimo di caratteri che può avere una stringa.
- *MinLength*: specifica il numero minimo di caratteri che può avere una stringa.
- *Pattern*: si fornisce un'espressione regolare che deve essere soddisfatta dal valore dell'elemento ottenuto per restrizione.
- *Enumeration*: serve per fornire tutti e soli i valori che possono essere adottati dell'elemento ottenuto per restrizione.
- *WhiteSpace*: specifica che tipo di comportamento bisogna seguire nei confronti dei caratteri spazio in una stringa di caratteri.
- *MaxInclusive*: fornisce il massimo valore inclusivo che può contenere il valore di un nuovo tipo semplice ottenuto per restrizione a partire da un tipo primitivo specificante un numero (int, integer, long, decimal...).
- *MinInclusive*: esattamente come maxInclusive, ma si specifica il valore minimo.
- *MaxExclusive*: come maxInclusive, ma il valore fornito compreso.
- *MinExclusive*: come minInclusive, ma il valore minimo compreso.
- *TotalDigits*: serve per specificare il numero di cifre che può contenere il valore di un nuovo tipo semplice ottenuto per restrizione a partire da un tipo primitivo specificante un numero (int, integer, long, decimal...).

- *FractionalDigits*: serve per specificare il numero di cifre della parte frazionaria (quella dopo il punto) che può contenere il valore di un nuovo tipo semplice ottenuto per restrizione a partire da un tipo primitivo specificante un numero frazionario (es.: decimal...).

Ovviamente non tutti i facets possono essere applicati nelle restrizioni di tutti i tipi primitivi. Ad esempio non si può usare il facet *minInclusive* associato ad un tipo di base *string*, né usare *length* applicato ad un tipo che rappresenta un numero.

Più facets possono comparire assieme nella stessa restrizione. Consideriamo le seguenti righe:

```
<xsd:element name=”Day” type=”dayOfMonth”/>

<xsd:simpleType name=”dayOfMonth”>
    <xsd:restriction base=”xsd:integer”>
         <xsd:minInclusive value=”1”/>
        <xsd:maxInclusive value=”31”/>
    </xsd:restriction>
</xsd:simpleType>
```

In questo modo è stato creato un tipo semplice “dayOfMonth” che può contenere solo I valori fra 1 e 31, compresi.

Consideriamo ora le seguenti righe, prese dall’Esempio 2.1:

```
  <xsd:attribute name="bookID" type="catalogID"/>
….
  <xsd:simpleType name="catalogID" base="xsd:string">
      <xsd:pattern value="\ d{ 3} -\ d{ 4} -\ d{ 3} "/>
  </xsd:simpleType>
```

Il tipo semplice “catalogID” usa la sfaccettatura *pattern* per specificare un’espressione regolare (cioè, un pattern impostato per effettuare la ricerca del testo nel formato specificato) che i valori delle stringhe di testo di questo tipo devono soddisfare. In questo caso, il testo nel tipo “simpleType” deve corrispondere all’espressione regolare “\ d{ 3} -\ d{ 4} -\ d{ 3}”, che significa tre cifre, un trattino, quattro cifre, un altro trattino e tre cifre. Il tipo “catalogID” p il tipo di attributo “bookID” dell’elemento <book>, perciò è possibile specificare i valori ID di book nel seguente modo, che corrisponde all’espressione regolare usata per questo attributo:

```
<book bookID=”123-4567-890”>
        <bookTitle>Earthquakes for Breakfast</bookTitle>
        <pubDate>2002-10-20</pubDate>
        <replacementValue>15.95</replacementValue>
        <maxDaysOut>14</maxDaysOut>
</book>
```

NOTA: I facet *length*, *maxLength*, *minLength* ed *enumeration* possono essere applicati anche a restrizioni del tipo semplice derivato L*ist*. In questo particolare caso length, maxLength e minLength assumono un significato diverso da quello consueto (cioè indicare il numero di caratteri in un campo stringa); consideriamo il seguente esempio:

```
<xsd:element name=”Numeri” type=”ListaD6iNumeri”/>

<xsd:simpleType name=”ListaDiNumeri”>
    <xsd: list itemType=”xsd: integer ”/>
</xsd:simpleType>

<xsd:simpleType name=”ListaDi6Numeri”>
    <xsd: restriction base=”ListaDiNumeri”>
        <xsd:length value=”6”/>
    </xsd: restriction >
</xsd:simpleType>
```

L’elemento Numeri appartenente al tipo ListaDi6Numeri rappresenta praticamente un array di sei valori di tipo integer. Una possibile istanza potrebbe essere:

```
<Numeri>1 2 -3 234 5 6</Numeri>
```

2.5.5 Il tipo semplice anyType

Esiste, in verità, anche un altro tipo semplice: il tipo *anyType*. AnyType rappresenta il tipo base da cui derivano tutti gli altri tipi semplici e complessi e non pone nessun vincolo sul proprio contenuto (all’interno di un anyType possiamo quindi, per esempio, ritrovare altri elementi semplici o complessi oppure solamente un valore rappresentato da un tipo primitivo). La dichiarazione di un tipo anyType viene svolta in questo modo:

```
<xsd:element name=”QualunqueCosa” typo=”anyType”/>
```

o più semplicemente con la forma abbreviata:

```
<xsd:element name=”QualunqueCosa”/>
```

anyType rappresenta infatti il tipo di default nel caso in cui il tipo non venga direttamente specificato nella clausula *type*.

Lo spazio dei valori di *anySimpleType* può essere pensato come l'unione dello spazio dei valori di tutti i tipi di dati primitivi.

2.5.6 Uso delle definizioni di tipo anonimo

Finora tutte le dichiarazioni dell'elemento usate nello schema dell'Esempio 1 hanno utilizzato l'attributo `type` per indicare il nuovo elemento. Che cosa è necessario fare, tuttavia, se si desidera usare un tipo una sola volta? In questo caso si può usare una *definizione di tipo anonimo* per evitare la definizione completa di un nuovo tipo a cui si farà riferimento una sola volta. Farlo significa semplicemente che si deve racchiudere l'elemento <xsd:simpleType> o <xsd:complexType> all'interno della dichiarazione dell'elemento <xsd:element>. In questo caso, non si dovrà assegnare un valore esplicito all'attributo `type` nell'elemento <xsd:element>, poiché il tipo anonimo che si sta usando non possiede un nome (infatti è possibile indicare che si sta usando una definizione di tipo anonimo, se l'elemento <xsd:complexType> non include un attributo `type`).

Consideriamo un esempio tratto dall'Esempio 2.1; in questo caso, si userà una definizione di tipo anonimo per l'elemento <book>. Questo elemento contiene gli elementi <bookTitle>, <pubDate>, <replacementValue> e <maxDaysOut>. Possiede inoltre un attributo chiamato bookID, perciò si presenta come un buon candidato per la creazione di un tipo complesso. Invece di dichiarare un tipo complesso separato, tuttavia, si inserisce l'elemento <xsd:complexType> all'interno dell'elemento <xsd:element> che dichiara <book>:

```
<xsd:element name=”book” minOccurs=”0” maxOccurs=”10”>
        <xsd:complexType>
          .
          .
        </xsd:complexType>
</xsd:element>
```

Ora è possibile aggiungere qualsiasi elemento all'interno dell'elemento <book> senza doversi preoccupare di definire un tipo complesso separato con un nome:

```
<xsd:element name=”book” minOccurs=”0” maxOccurs=”10”>
        <xsd:complexType>
           <xsd element name=”bookTitle” type=”xsd:string”/>
           <xsd element name=”pubDate” type=”xsd:date” minOccurs=’0’/>
           <xsd element name=”replacementValue” type=”xsd:decimal”/>
        </xsd:complexType>
</xsd:element>
```

E' possibile anche usare tipi semplici anonimi; per esempio l'elemento <maxDaysOut> contiene il numero massimo di giorni che si potrà tenere un libro in prestito. Per impostare a 14 il numero massimo di giorni, si userà nel modo seguente un nuovo tipo semplice anonimo per usare la sfaccettatura *maxExclusive*:

```
<xsd:element name=”book” minOccurs=”0” maxOccurs=”10”>
        <xsd:complexType>
```

```
            <xsd element name=”bookTitle” type=”xsd:string”/>
            <xsd element name=”pubDate” type=”xsd:date” minOccurs=’0’/>
            <xsd element name=”replacementValue” type=”xsd:decimal”/>
            <xsd:element name=”maxDaysOut”>
                    <xsd:simpleType base=”xsd:integer”>
                        <xsd:maxExclusive value=”14”/>
                    </simpleType>
            </xsd:element>
        ….
          </xsd:complexType>
  </xsd:element>
```

E’ possibile includere anche le definizioni dell’attributo nelle dichiarazioni del tipo anonimo:

```
  <xsd:element name=”book” minOccurs=”0” maxOccurs=”10”>
        <xsd:complexType>
           <xsd element name=”bookTitle” type=”xsd:string”/>
           <xsd element name=”pubDate” type=”xsd:date” minOccurs=’0’/>
           <xsd element name=”replacementValue” type=”xsd:decimal”/>
           <xsd:element name=”maxDaysOut”>
                    <xsd:simpleType base=”xsd:integer”>
                            <xsd:maxExclusive value=”14”/>
                    </simpleType>
            </xsd:element>
           <xsd:attribute name=”bookID” type=”catalogID”/>
        </xsd:complexType>
  </xsd:element>
```

2.5.7 Creazione di elementi vuoti

Gli elementi vuoti non hanno alcun contenuto, ma possono avere attributi. Per dichiarare un elemento vuoto, si dichiara un tipo complesso e poi si imposta l’attributo `content` dell’elemento <xsd:complexType> a `empty`.

Si crea un nuovo elemento vuoto chiamato <image>, che può avere tre attributi: `source`, `width` e `height`, nel modo seguente:

```
  <image source=”/image/cover.gif” height=”256” width=”512”/>
```

Si inizia dichiarando questo elemento:

```
  <xsd:element name=”image”>
```

```
</xsd:element>
```

In questa dichiarazione dell'elemento non si è usato l'attributo `type` perché si utilizzerà una definizione di tipo anonimo su cui basare l'elemento. Per trarre il tipo anonimo, si usa un elemento <complexType>; si noti che l'attributo `content` è impostato a `empty`. Infine si aggiungono gli attributi che saranno usati da questo elemento. Ora l'elemento <image> è pronto per essere usato:

```
<xsd:element name="image">
    <xsd:complexType content="empty">
        <xsd:attribute name="source" type="xsd:string"/>
        <xsd:attribute name="width" type="xsd:decimal"/>
        <xsd:attribute name="height" type="xsd:decimal"/>
    </xsd:complexType>
</xsd:element>
```

2.5.8 Creazione di elementi a contenuto misto

In XML Schema è possibile creare anche elementi che supportano contenuto misto, cioè sia testo che altri elementi. In questi elementi i tipi di dati carattere possono apparire allo stesso livello degli elementi figli.

Consideriamo un esempio di documento che mostra come si presentano gli elementi a contenuto mosto usando gli elementi dichiarati nell'Esempio 2.1; in questo caso, si è creato un nuovo elemento chiamato <reminder> che racchiude una lettera di sollecito a chi ha preso in prestito un libro affinché lo restituisca:

```
<?xml version="1.0">
<reminder>
    Dear <name>Britta Regensburg</name>:
        The book <bookTitle>Snacking on Vulcanoes</bookTitle>
    Was supposed to be out for only<maxDaysOut>14</maxDaysOut>
    Days. Please return it of pay
    $<replacementValue>17.99</replacementValue>.
    Thank you.
</reminder>
```

Questo documento usa gli elementi che sono stati definiti in precedenza, i dati di tipo carattere e il nuovo elemento <reminder> che ha un modello di contenuto misto; per dichiararlo in uno schema, si inizierà creando un nuovo tipo anonimo complesso all'interno della dichiarazione di <reminder>. Si ricordi che l'elemento <xsd:complexType> ha l'attributo `content` con cui è possibile specificare un modello di contenuto; in questo caso il modello di contenuto è `mixed`:

```
<xsd:element name=”reminder”>
        <xsd:complexType content=”mixed”>
    …
    </complexType>
</xsd:element>
```

Ora si dovranno aggiungere le dichiarazioni per gli elementi che è possibile usare all'interno dell'elemento <reminder>, nel modo seguente:

```
<xsd:element name=”reminder”>
    <xsd:complexType content=”mixed”>
        <xsd:element name=”name” type=”xsd:string”/>
        <xsd:element name=”bookTitle” type=” xsd:string”/>
        <xsd:element name=”maxDaysOut”/>
            <xsd:simpleType base=”xsd:integer”>
                <xsd:maxExclusive value=”14”/>
            </xsd:simpleType>
        </xsd:element>
        <xsd:element name=”replacementValue” type=”xsd:decimal”/>
    </complexType>
</xsd:element>
```

L'ordine e il numero degli elementi figli devono corrispondere all'ordine e al numero degli elementi figli che si specifica nello schema.

Quando non si specifica nessun modello di contenuto, il modello di default per i tipi complessi si chiama *elementOnly* e può includere solo elementi. In pratica le due definizioni di tipo che seguono sono analoghe:

```
<xsd:element name=”transaction” type=”transactionType”/>

<xsd:complexType name=”transactionType”>
    …..
</xsd:complexType>
```

e lo stesso risultato si ha con

```
<xsd:complexType name=”transactionType” content=”elementOnly”>
    …..
</xsd:complexType>
```

Il modello di contenuto del tipo complesso di default è `elementOnly` eccetto che nel caso in cui si deriva un tipo complesso da un tipo semplice, allora in quel caso il modello di contenuto è `textOnly` e

specifica che il contenuto di questo tipo è testo, cioè il processore XML non applicherà alcuna regola di sintassi al contenuto.

2.5.9 Le notazioni

Gli schemi XML forniscono tre nuovi elementi che si usano per aggiungere note agli schemi:

- *annotation*
- *documentation*
- *appInfo*

<xsd:annotation> è l'elemento contenitore degli elementi <xsd:documentation> e <xsd:appInfo>. L'elemento <xsd:documentation> contiene testo del tipo che ci si potrebbe attendere in un normale commento (cioè testo per gli utenti); l'elemento <xsd:documentation> può presentare l'attributo *xml:lang* che specifica in quale lingua è scritto il commento interno. L'elemento <xsd:appInfo> contiene note adatte alle applicazioni che effettuano letture del documento.

Riportiamo un pezzo di Esempio 2.1 in cui compaiono `annotation` e `documentation`:

```
<xsd:annotation>
   <xsd:documentation>
        Book borrowing transaction schema.
   </xsd:documentation>
</xsd:annotation>
```

E ora un esempio che usa gli elementi `annotation` e `appInfo`:

```
<simpleType name=”string” base=”urSimpleType”>
    <annotation>
        <appInfo>
            <has-facet name=”length”/>
            <has-facet name=”minLength”/>
            <has-facet name=”maxLength”/>
            <has-facet name=”pattern”/>
            …..
        </appInfo>
    </annotation>
</simpleType>
```

2.5.10 I gruppi Sequence, Choice e All

I tag `sequence`, `choice` e `all` permettono di applicare alcune caratteristiche a gruppi di attributi presenti in un modello di contenuto complesso. Vediamo, tramite esempi, il loro utilizzo:

- *Sequenze*: il gruppo sequence forza gli elementi contenuti ad essere rappresentati in un documento XML nello stesso ordine in cui vengono presentati nello schema.

  Supponiamo di modificare l'Esempio 2.1 in modo da consentire di prendere in prestito anche riviste oltre che libri. Per fare questo creiamo un nuovo gruppo chiamato "booksAndMagazine". Per creare il gruppo si usa l'elemento <xsd:group> (che vedremo più avanti) e per essere certi che gli elementi all'interno del gruppo appaiano in un determinato ordine, si usa l'elemento <xsd:sequence> nel seguente modo:

```
<xsd:complexType name="transactionType">
    <xsd:element name="Lender" type="address"/>
    <xsd:element name="Borrower" type="address"/>
    <xsd:element ref="note" minOccurs="0"/>
    <xsd:choice>
        <xsd:element name=”books” type=”books”/>
        <xsd:element ref=”book”/>
        <xsd:group ref=”booksAndMagazine”/>
    </xsd:choice>
    <xsd:attribute name="borrowDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name=”books”>
    <xsd:element ref=”book” minOccurs=”0” maxOccurs=”10”/>
</xsd:complexType>

<xsd:group name=”booksAndMagazine”>
    <xsd:sequence>
        <xsd:element ref=”book”/>
        <xsd:element ref=”magazine”/>
    </xsd:sequence>
</xsd:group>

    <xsd:element name="book">
        <xsd:complexType>
            <xsd:element name="bookTitle" type="xsd:string"/>
            <xsd:element name="pubDate" type="xsd:date" maxOccurs="0"/>
            <xsd:element name="replacementValue" type="xsd:decimal"/>
            <xsd:element name="maxDaysOut">
                <xsd:simpleType base="xsd:integer">
```

```
                    <xsd:maxExclusive value="14"/>
                </xsd:simpleType>
            </xsd:element>
            <xsd:attribute name="bookID" type="catalogID"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="magazine">
        <xsd:complexType>
            <xsd:element name="magazineTitle" type="xsd:string"/>
            <xsd:element name="pubDate" type="xsd:date" maxOccurs="0"/>
            <xsd:element name="maxDaysOut">
                <xsd:simpleType base="xsd:integer">
                    <xsd:maxExclusive value="14"/>
                </xsd:simpleType>
            </xsd:element>
            <xsd:attribute name="bookID" type="catalogID"/>
        </xsd:complexType>
    </xsd:element>
```

- *Choice*: il costrutto choice permette di produrre elementi con una struttura variabile propria dei documenti formati da dati semi-strutturati: in una istanza può apparire un solo elemento figlio del gruppo choice.

  Nell'esempio riportato sopra, appare il costrutto choice. Infatti grazie ad esso, chi chiede il prestito può scegliere alcuni libri o solo uno. Si noti che in questo caso <book> deve essere trasformato in un elemento globale affinché sia possibile fare riferimento all'elemento in questa scelta, per rimuoverlo dalla dichiarazione dell'elemento <books>:

```
<xsd:complexType name="transactionType">
    <xsd:element name="Lender" type="address"/>
    <xsd:element name="Borrower" type="address"/>
    <xsd:element ref="note" minOccurs="0"/>
    <xsd:choice>
        <xsd:element name=”books” type=”books”/>
        <xsd:element ref=”book”/>
        <xsd:group ref=”booksAndMagazine”/>
    </xsd:choice>
    <xsd:attribute name="borrowDate" type="xsd:date"/>
</xsd:complexType
```

- *All*: tutti i suoi elementi possono esser presenti una votala o non esserlo del tutto e apparire in qualsiasi ordine. Questo gruppo va usato al livello più alto del modello di contenuto e i figli del gruppo devono essere singoli elementi, cioè questo gruppo non ne deve contenere altri. Inoltre,

qualsiasi elemento in questo modello di contenuto non può apparire più di una volta (cioè i valori consentiti minOccurs e maxOccurs sono solo 0 e 1).

Nell'esempio che segue si converte il tipo "transactionType" in un gruppo `all`:

```
<xsd:complexType name="transactionType">
    <xsd:all>
      <xsd:element name="Lender" type="address"/>
      <xsd:element name="Borrower" type="address"/>
      <xsd:element ref="note" minOccurs="0"/>
      <xsd:element name=”books” type=”books”/>
    </xsd:all>
    <xsd:attribute name="borrowDate" type="xsd:date"/>
</xsd:complexType
```

2.5.11 Gruppi di elementi e gruppi di attributi

XML Schema permette, per rendere più leggibile uno schema e più riusabili i suoi componenti, di definire gruppi di elementi e di attributi esternamente alle definizioni di tipo complesso. Un gruppo di elementi è definito dal tag `group` e deve possedere un attributo `name` in modo da poter essere referenziato.

```
    <xsd:complexType name="transactionType">
        <xsd:element name="Lender" type="address"/>
        <xsd:element name="Borrower" type="address"/>
        <xsd:element ref="note" minOccurs="0"/>
        <xsd:choice>
            <xsd:element name=”books” type=”books”/>
            <xsd:element ref=”book”/>
            <xsd:group ref=”booksAndMagazine”/>
        </xsd:choice>
        <xsd:attribute name="borrowDate" type="xsd:date"/>
    </xsd:complexType>
….

<xsd:group name=”booksAndMagazine”>
<xsd:sequence>
    <xsd:element ref=”book”/>
    <xsd:element ref=”magazine”/>
</xsd:sequence>
</xsd:group>
```

Abbiamo riportato la parte di schema tratta dall'Esempio 2.1 modificato, visto precedentemente. Abbiamo fatto in modo di consentire di prendere in prestito anche riviste oltre che libri. Per fare questo abbiamo creato un nuovo gruppo chiamato "booksAndMagazine". Per creare il gruppo abbiamo usato <xsd:group> e per essere certi che gli elementi all'interno del gruppo appaiano in un determinato ordine, abbiamo usato <xsd:sequence>.

E' possibile creare anche gruppi di attributi usando l'elemento <attributeGroup>. Per esempio, si supponga di voler aggiungere un certo numero di attributi all'elemento <book> che descrive il libro. Per fare questo si può creare un gruppo di attributi chiamato "bookDescription" e quindi farvi riferimento nella dichiarazione di <book>:

```
<xsd:element name="book">
    <xsd:complexType>
        <xsd:element name="bookTitle" type="xsd:string"/>
        <xsd:element name="pubDate" type="xsd:date" maxOccurs="0"/>
        <xsd:element name="replacementValue" type="xsd:decimal"/>
        <xsd:element name="maxDaysOut">
            <xsd:simpleType base="xsd:integer">
                <xsd:maxExclusive value="14"/>
            </xsd:simpleType>
        </xsd:element>
        <xsd:attributeGroup ref="bookDescription"/>
    </xsd:complexType>
        </xsd:element>
```

Per creare il gruppo degli attributi "bookDescription", si usa l'elemento <xsd:attributeGroup>, che racchiude gli elementi <xsd:attribute> usati per dichiarare gli attributi nell'elemento <xsd:attributeGroup>:

```
<xsd:attributeGroup name="bookDescription">
        <xsd:attribute name="bookID" type="CatalogID"/>
        <xsd: attribute name="numberPages" type="xsd:decimal"/>
        <xsd: attribute name="coverType"/>
            <xsd:simpleType base="xsd:string">
                <xsd:enumeration value="leather"/>
                <xsd:enumeration value="cloth"/>
                <xsd:enumeration value="vinyl"/>
            </xsd:simpleType>
        <xsd:attribute/>
</xsd:attributeGroup>
```

2.5.12 Derivazione di tipi

XML Schema fornisce due metodi per derivare nuovi oggetti a partire da oggetti già definiti:

- *Derivazione per estensione*
- *Derivazione per restrizione*

Alcuni esempi e metodologie di derivazione per restrizione sono già state fornite quando si è parlato di creazione di nuovi tipi semplici per restrizione. E' possibile derivare un nuovo tipo di dato complesso sia tramite la restrizione che tramite l'espansione, mentre per quanto riguarda i dati semplici, è possibile usare solamente la restrizione. Dato che la parte riguardante l'ereditarietà dei tipi semplici è già stata descritta in precedenza, considereremo solamente le regole riguardanti i tipi complessi.

- *Nuovi tipi complessi ottenuti per Estensione*

  Un nuovo tipo complesso deve essere dichiarato a partire dall' elemento `complexType` e successivamente bisogna indicare se si vuole derivare da un tipo semplice (utilizzando l'elemento `simpleContent`) o da un tipo complesso (utilizzando l'elemento `complexContent`) e poi usare l'elemento `extension`. Nell'attributo base di `extension` bisogna inserire il nome del tipo da cui si deriva e all'interno dello stesso tag `extension`, si inseriscono i nuovi elementi e attributi che si desidera aggiungere all'elemento padre. E' importante notare che se si specifica il modello di contenuto come `simpleContent` si possano aggiungere solamente attributi e non elementi. Il nuovo modello di contenuto risulta essere quindi l'unione del contenuto del tipo specificato nell'attributo `base` con il nuovo contenuto compreso nel tag `extension`. Supponiamo di volere estendere il contenuto del tipo complesso "OggettoType" per poter considerare novità che devono ancora uscire sul mercato:

```
<xsd:complexType name=”OggettoType”>
        < xsd:element name =”Nome” type=”xsd:string”/>
        < xsd:element name =”Quantità” type=”QuantitàType”/>
        < xsd:element name =”PrezzoEuro” type=”xsd:decimal”/>
        < xsd:element ref =”Commento” minOccurs=”0”/>
</xsd:complexType>

<xsd:complexType name=”OggettoNovità”>
        <xsd:complexContent>
                <xsd:extension base=”OggettoType”>
                        < xsd:element name =”DataDiUscita” type=”xsd:data”/>
                        < xsd: attribute name =”Novità” type=”xsd:boolean” fixed
                        =”true”/>
                </xsd:extension>
```

```
        </xsd:complexContent>
</xsd:complexType>
```

All'interno di un documento XML istanza un elemento del tipo "OggettoNovità" può essere dichiarato in modo normale oppure comparire al posto di un elemento di tipo "OggettoType", vediamo come:

```
<Ordine>
...
        <Oggetto>
                <Nome>Tastiera</Nome>
                <Quantità>10</Quantità>
                <PrezzoEuro>15.25</PrezzoEuro>
        </Oggetto>
    ...
            <Oggetto xsi: type="OggettoNovità" Novità="true">
                    <Nome>TastieraNuova</Nome>
                    <Quantità>5</Quantità>
                    <PrezzoEuro>26.25</PrezzoEuro>
                    <DataDiUscita>2002-11-15</DataDiUscita>
            </Oggetto>
    ...
    </Ordine>
```

Il tipo derivato usato deve essere comunque specificato tramite l'utilizzo dell'attributo "xsi:type" che fa riferimento al namespace http://www.w3.org/2001/XMLSchemainstance. Si è parlato precedentemente di nuovi tipi complessi ottenuti per estensione di contenuto a partire da un tipo semplice, un esempio di questo processo può essere il seguente:

```
<xsd:complexType name="PrezzoType">
        <xsd:simpleContent>
                <xsd:extension base="xsd:decimal">
                        < xsd: attribute name ="Tipo" type="xsd: string " required ="true"/>
                </xsd:extension>
        </xsd:simpleContent>
</xsd:complexType>
```

Definizioni di questo tipo servono per creare elementi con un contenuto sostanzialmente semplice, ma che possono avere attributi.

- *Nuovi tipi complessi ottenuti per restrizione*

Un oggetto complesso ottenuto per restrizione è molto simile al suo tipo di base, ma le dichiarazioni presenti al suo interno sono maggiormente limitate. Consideriamo ad esempio il seguente tipo complesso "OrdineType":

```
<xsd:complexType name="OrdineType">
        <xsd:element name ="Indirizzo" type="IndirizzoType"/>
        <xsd:element name ="Oggetto" type="OggettoType" minOccurs="1"
                maxOccurs="unbounded"/>
        <xsd:element ref ="Commento" minOccurs="0"/>
        <xsd: attribute name ="Data" type="xsd:date"/>
</xsd:complexType>
```

Una possibile restrizione è la seguente:

```
<xsd:complexType name="OrdineStock">
        <xsd:complexContent>
                <xsd: restriction base="OrdineType">
                        < xsd:element name ="Indirizzo" type="IndirizzoType"/>
                        < xsd:element name ="Oggetto" type="OggettoType"
                        minOccurs="50"
                                maxOccurs="100"/>
                        < xsd:element ref ="Commento" minOccurs="0" />
                        < xsd: attribute name ="Data" type="xsd:date"/>
                </xsd: restriction >
        </xsd:complexContent>
</xsd:complexType>
```

Vediamo che il nuovo tipo complesso "OrdineStock" è più vincolato rispetto a "OrdineType", infatti è possibile avere un numero di "Oggetto" compreso fra 50 e 100, intervallo più ristretto rispetto al precedente che era da 1 a infinito.

2.5.13 I namespace

Il concetto di namespace in XML è molto simile a quello usato in linguaggi di programmazione tipo Java e C++: permette di evitare collisioni tra i nomi di elementi e attributi tramite l'uso di nomi qualificati, consistenti in un prefisso che indica il namespace cui l'elemento appartiene, seguito da un nome locale a tale namespace. Per esempio <ns1:tag1> e <ns2:tag1> sono due elementi distinti, pur avendo lo stesso nome locale tag1.

I namespace sono in realtà identificati da URI (*Uniform Resource Identificator*) e associati a particolari prefissi nel file XML stesso, tramite l'uso dell'attributo *xmlns*:

```
<schema xmlns:ns1="http://www.starpowder.org/ns1">
        <ns1:tag1 ns1:att1="value1"/>
</schema>
```

Una cosa importante riguardo gli XML namespace (e che li distingue da quelli Java e C++) è che non implicano, di per sé, l'esistenza di alcun file o risorsa contenente la dichiarazione degli elementi e attributi che essi contengono. L'URI usato come valore dell'attributo *xmlns* in genere non si risolve in alcuna risorsa effettivamente esistente su qualche server. Esso serve semplicemente a identificare univocamente il namespace, in modo che la applicazioni possano interpretare correttamente i tag che si riferiscono a quel namespace.

La dichiarazione di `targetNamespace` definisce il namespace del documento da validare. Gli attributi `elementFormDefault` e `attributeFormDefault` permettono di controllare se l'uso del prefisso è necessario per i tipi non globali.

```
<schema xmlns=http://www.w3.org/2001/XMLSchema xmlns:po="http://www.example.com/PO1"
          targetNamespace=http://www.example.com/PO1 elementFormDefault="unqualified"
          attributeFormDefault="unqualified">

        <element name="A" type="po:prova"/>

        <element name="C" type="string"/>

        <complexType name="prova">
                <sequence>
                        <element name="B" type="string"/>
                        <element ref="C"/>
                </sequence>
        </complexType>
</schema>
```

Poiché gli attributi `elementFormDefault` e `attributeFormDefault` sono definiti come `unqualified`, solo i tipi globali debbono avere il prefisso, gli altri no.

```
<po:A xmlns:po="http://www.example.com/PO1">
        <B> ... </B>
        <po:C> ... </po:C>
</po:A>
```

Quello che i namespace permettono di fare è di specificare regole di validazione solo su alcuni e non tutti i namespace del documento:

```
<element name="htmlElement">
  <complexType>
```

```
    <sequence>
      <any namespace="http://www.w3.org/1999/xhtml" minOccurs="1"
maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>
```

Il valore dell'attributo `namespace` dell'elemento `any` non deve essere per forza definito da un URI, ma può assumere i seguenti valori:

Valore dell'attributo namespace	Contenuti permessi
##any	Qualunque XML ben formato
##local	Qualunque XML ben formato che non sia qualificato, cioè non dichiarato come appartenente ad un namespace.
##other	Qualunque XML ben formato che non sia del target namespace.
http://www.w3.org/1999/xhtml ##target-Namespace	Ogni XML ben formato che appartenga ad un namespace della lista

Tabella 2.2: Attributi di namespace e valori permessi.

2.5.14 Include ed Import

Gli elementi `include` e `import` servono per poter utilizzare all'interno dello schema XML definizioni fornite su file separati da quello contenente lo schema stesso. Il funzionamento dei due elementi è leggermente diverso: `include` serve per importare definizioni esterne al documento provenienti da uno schema che si riferisce allo stesso target namespace (è quindi come avere uno schema diviso fra più file), mentre `import` viene usato per importare elementi da differenti target namespaces. Se, per esempio, vogliamo aggiungere ad uno schema le dichiarazioni fornite su di un altro schema XML con lo stesso target namespace basta aggiungere la riga di codice:

```
<xsd:include schemalocation="http :// ferrari .mo.it /SchemiXml/schema1.xsd/"/>
```

in cui il valore dell'attributo `schemalocation` rappresenta l'URI cui ritrovare il file con le dichiarazione desiderate. Gli elementi importati dal file schema1.xsd possono essere usati apponendo loro lo stesso prefisso impiegato per indicare le entità appartenenti al target namespace (come già citato, infatti, i due target namespaces devono essere coincidenti).

Per usare un elemento definito in un documento referente un diverso namespace, invece, si usa l'espressione:

<xsd:import namespace=”http :// ferrari .mo.it /SchemiXml/namespace1/”/>

e si aggiunge alla dichiarazione dell’elemento schema un attributo xmlns per indicare un prefisso da apporre agli elementi importati.

Capitolo 3

Il database lessicale WordNet

3.1 Introduzione

Il database lessicale WordNet[9] è stato sviluppato presso l'Università di Princeton sotto la direzione del professore George A. Miller. WordNet 1.7.1 è disponibile gratuitamente presso il sito http://www.cogsci.princeton.edu/wn/. La licenza d'uso permette l'utilizzo gratuito del database anche a fini commerciali e al di fuori della ricerca, purché vengano citati gli autori ed il sito ufficiale del progetto.

Poiché le frasi significative sono composte di parole significative, qualsiasi sistema che speri di elaborare linguaggi naturali, come fanno le persone, deve avere delle informazioni sulle parole e sui loro significati. Generalmente, queste informazioni vengono fornite attraverso dizionari cartacei e dizionari machine-readable, ora ampiamente disponibili. Ma le voci del dizionario sono ordinate per essere comode per le persone fisiche, non per le macchine. WordNet fornisce una combinazione più efficiente di informazioni lessicografiche tradizionali e di elaborazioni moderne. WordNet è progettato per essere usato sotto il controllo di programmi. I termini, infatti non sono disposti seguendo l'ordine alfabetico, ma per affinità di significato.

3.2 Definizioni del linguaggio

Definiamo il *vocabolario* di una lingua come un insieme W delle coppie (f, s), dove una *forma* f di una parola è una stringa su un alfabeto limitato e un *significato* s è un elemento di un insieme dato di

significati. Le forme f possono essere espressioni composte di una stringa di fonemi o iscrizioni composte di una stringa di caratteri. Ogni forma con un significato in una lingua viene chiamato una *parola* in quella lingua. Un *dizionario* è un elenco alfabetico di parole.

L'utilizzo di una parola è l'insieme C di contesti linguistici in cui la parola può essere utilizzata. La sintassi della lingua suddivide C in *categorie sintattiche*. Le parole presenti nel sottoinsieme N sono sostantivi, parole presenti nel sottoinsieme V sono verbi, e così via. All'interno di ogni categoria di contesti sintattici, ci sono ulteriori categorie di contesti semantici - l'insieme dei contesti in cui una particolare forma può essere utilizzata per esprimere un particolare significato. WordNet tratta quattro categorie sintattiche: nomi, verbi, aggettivi ed avverbi. Ogni categoria sintattica è suddivisa in diversi insiemi di sinonimi; ad ognuno di questi insiemi è associato un unico significato, condiviso da tutti i termini associati a un certo insieme. Un termine, ovviamente, può possedere più di un significato ed essere, quindi, presente in molti di questi insiemi ed anche in più di una categoria sintattica. Nel gergo usato all'interno del progetto WordNet, un insieme di vocaboli che condivide uno stesso significato viene chiamato *synset*.

Elenchiamo di seguito un insieme di termini significativi propri della terminologia di WordNet:

- *Categoria sintattica*: sono le grandi categorie in cui sono suddivisi i termini (ed anche i file in cui sono contenuti) di WordNet. Le categorie sintattiche trattate sono quattro: nomi, verbi, aggettivi, avverbi.

- *Lemma*: è la parola, il termine a cui viene associato uno o più significati. A volte un lemma è costituito da due o più parole ed in tal caso i singoli termini sono uniti dal carattere underscore (_).

- *Synset*: rappresenta il significato che viene associato ad un insieme di lemmi appartenenti alla stessa categoria sintattica. In pratica è corretta l'affermazione che ad un synset appartengono un certo numero di lemmi. Un synset, infatti, può essere rappresentato, oltre che tramite la sua Gloss, anche tramite l'insieme dei sui lemmi.

- *Gloss*: una Gloss (o italianizzandola Glossa) è una descrizione a parole di uno specifico significato, ogni synset oltre a contenere un insieme di sinonimi possiede anche una glossa.

- *Relazione semantica*: si tratta di una relazione presente fra due synset appartenenti alla stessa categoria sintattica; i diversi tipi di relazioni semantiche saranno trattate in seguito.

- *Relazione lessicale*: è una relazione che collega due lemmi appartenenti a due synset distinti (ma sempre relativamente alla stessa categoria sintattica); i diversi tipi di relazioni lessicali saranno trattate in seguito.

3.3 La matrice lessicale

Il punto di partenza per la semantica lessicale è capire che esiste un'associazione fra la forma f della parola ed il significato s ad essa associato. La corrispondenza fra forma della parola e significato non è un'associazione di tipo uno a uno, ma bensì di tipo molti a molti, dando luogo ai concetti di

- *Sinonimia*: proprietà per cui uno stesso significato è esprimibile usando due o più parole distinte.

- *Polisemia*: proprietà per cui ad una parola sono associati due o più significati distinti.

	W_1	**W_2**	**W_3**	**W_4**	**W_5**
M_1	$E_{1,1}$				
M_2		$E_{2,2}$			
M_3		$E_{3,2}$	$E_{2,2}$		
M_4					
M_5				$E_{5,4}$	
M_6					$E_{6,6}$

Tabella 3.1: La matrice lessicale.

Le relazioni fra f ed s possono essere rappresentate dalla cosiddetta *matrice lessicale*. In questa matrice, le righe rappresentano i vari significati che una forma f può assumere, mentre le colonne sono formate dai diversi termini. In pratica, se si vuole leggere la matrice lessicale tramite la terminologia di WordNet, ad ogni riga è associato un synset, mentre ad ogni colonna è associato un lemma. Ogni elemento non nullo che compare all'interno della matrice implica che il particolare lemma, o termine, situato in quella riga, può essere usato per rappresentare lo specifico significato associato a quella colonna. Nel caso in una colonna sia presente più di un elemento, si ha un caso di polisemia, mentre se in una riga è presente più di un elemento, si ha un caso di sinonimia.

In WordNet il concetto della matrice lessicale viene espresso tramite la separazione tra synset e lemmi, ossia, mantenendo separati significati e termini. Nei file utilizzati da WordNet, un synset viene espresso tramite l'insieme dei termini che sono ad esso associati. Tuttavia, nella maggior parte dei casi un insieme di parole non basta per descrivere un significato, così viene associato a ciascun synset anche una descrizione del significato, tramite una frase in inglese, detta Gloss.

3.4 Le relazioni

WordNet presenta due grandi gruppi di relazioni: le *relazioni lessicali*, quando entrambi gli operandi sono entrambi dei lemmi e le *relazioni semantiche*, quando invece gli operandi sono synset. Non possono esistere relazioni fra un lemma e un synset o fra operandi appartenenti a diverse categorie

sintattiche (ad esempio fra un nome ed un verbo). All'interno dei file originali di WordNet tutte le relazioni (con l'unica eccezione riguardante la relazione di sinonimia) sono rappresentate tramite puntatori e tramite caratteri speciali che indicano il tipo di relazione specificata.

3.4.1 Relazioni Semantiche

Le relazioni di tipo semantico coinvolgono sempre due concetti (synset) e non semplicemente due termini (o lemmi). Ci sono otto diverse relazioni di tipo semantico.

3.4.1.1 Iponimia (Hyponimy)

La relazioni di iponimia (sub-name) e la sua inversa, *ipernimia* (*Hypernymy* o super-name), sono relazioni transitive tra synset. Poiché di solito c'è solo un ipernimo, questa relazione semantica organizza i significati dei sostantivi in una struttura gerarchica. Queste relazioni possono essere considerate l'equivalente delle gerarchie di specializzazione/generalizzazione per database relazionali, o dell'ereditarietà per modelli ad oggetti. Diremo che un synset X è un iponimo di un synset Y se è valida l'affermazione *"X is a kind of Y"*. Mentre la relazione opposta, di ipernimia, lega un concetto ad uno più particolare, più specializzato. In pratica si può affermare che un synset X rappresenta un ipernimo di un synset Y se Y presenta tutte le caratteristiche di X più almeno una sua caratteristica particolare aggiuntiva.

3.4.1.2 Meronimia (Meronymy)

La relazione di meronimia (part-name) e la sua inversa, *olonimia* (*holonymy* o whole-name), sono relazioni semantiche complesse. Un concetto X è detto meronimo di un concetto Y se è lecita l'affermazione '*X is a part of Y*'. Anche la relazione di meronimia, come quella di iponimia, può essere sfruttata per costruire una gerarchia sui synset di WordNet, in cui uno risulta essere parte dell'altro. Le relazioni di meronimia e iponimia vengono formulate sulla categoria sintattica dei nomi.

3.4.1.3 Implicazione (Entailment)

La relazione di implicazione è posta fra due verbi. La relazione di *entailment* può essere ritenuta simile a quella di meronimia posta sui nomi. Questa relazione è verificata se è vera la seguente affermazione: "*un verbo X implica un verbo Y seX non può verificarsi a meno che non sia verificato (o non si stia*

verificando) Y". L'implicazione non è solo una relazione semantica, ma è possibile avere anche implicazioni lessicali fra verbi (fra singoli termini).

3.4.1.4 Relazione causale (Cause to)

La relazione causale è simile alla relazione di implicazione, ma senza inclusione temporale.

3.4.1.5 Raggruppamento di verbi (Verb Group)

Questa relazione viene utilizzata per produrre raggruppamenti nella categoria sintattica dei verbi. In un gruppo formato in tale maniera, i synset hanno tutti un significato semantico molto simile.

3.4.1.6 Similarità (Similar to)

La relazione di similarità è utilizzata solo nella categoria sintattica degli aggettivi. Molti synset di questa categoria sono raggruppati in coppie legate da una relazione di antinomia, tali synset vengono chiamati synset principali (o *head synset*). A questi synset principali sono collegati, per similarità, dei synset *satelliti*, che condividono indirettamente la relazione di antinomia insieme al significato principale cui sono legati.

3.4.1.7 Attributo (Attribute)

La relazione di attributo rappresenta il legame, che intercorre fra un aggettivo ed un nome, di cui esprime il valore. Gli aggettivi in grado di descrivere il valore di un attributo sono gli aggettivi descrittivi.

3.4.1.8 Coordinazione

La coordinazione non un tipo di relazione base, bensì derivata. Due synset sono detti coordinati se possiedono lo stesso ipernimo, cioè se risultano essere la specializzazione del medesimo concetto.

3.4.2 Relazioni Lessicali

Le relazioni lessicali, a differenza di quelle semantiche, coinvolgono sempre due synset. WordNet prevede i seguenti tipi di relazioni lessicali.

3.4.2.1 Sinonimia (Synonymy)

La relazione di sinonimia è una relazione di base di WordNet, poiché WordNet utilizza degli insiemi di sinonimi (*synset*) per rappresentare il significato delle parole. La sinonimia (syn stessi, onyma nomi) è una relazione simmetrica fra forme di parola. Per ogni coppia di termini appartenenti allo stesso synset esiste dunque, in maniera implicita, una relazione di sinonimia. Una possibile definizione di sinonimia potrebbe essere la seguente: *"due termini sono sinonimi, all'interno di un contesto linguistico C, se la sostituzione di un termine con l'altro, all'interno di C, non varia il valore della frase"*.

3.4.2.2 Antinomia (Antynomy)

La relazione di antinomia (opposing-name) è una relazione semantica simmetrica fra forme di parola, particolarmente importante nell'organizzazione dei significati di aggettivi e avverbi. L'antinomia è una relazione lessicale fra due lemmi. Due termini legati da una relazione di antinomia sono l'uno il contrario dell'altro.

3.4.2.3 Relazione di Pertinenza (Pertainym)

La relazione di pertinenza concerne gli aggettivi relazionali. Un aggettivo relazionale svolge un ruolo che può essere riassunto in un espressione come: *associato con*, oppure *pertinente a* o, semplicemente *di* in relazione ad un nome. L'aspetto di un aggettivo relazionale risulta molto simile a quello del nome cui è legato, leggermente modificato.

3.4.2.4 Vedi anche (See also)

Questa relazione lega singoli lemmi di synset differenti. I motivi di tale relazione possono essere molto differenti fra loro.

3.4.2.5 Relazione Participiale (Participle)

Questa relazione lega gli aggettivi, detti participiali, ai nomi da cui derivano.

3.4.2.6 Derivato da (Derived to)

Alcuni aggettivi relazionali derivano da antichi nomi Greci o Latini. Questa affermazione risulta essere vera sia per la lingua italiana che per quella inglese. La relazione *derivato da* lega gli aggettivi ai nomi stranieri da cui derivano.

Capitolo 4

Un algoritmo per il matching fra schemi XML

4.1 Introduzione

In questo capitolo verrà spiegato in dettaglio il procedimento seguito per l'implementazione dell'algoritmo di matching. Tale algoritmo è stato sviluppato basandosi sull'algoritmo Similarity Flooding [34]. L'algoritmo esegue in maniera automatica il matching fra coppie di *modelli* che gli vengono forniti in input e produce in uscita una "mappatura" fra gli elementi dei modelli. Come già detto nel Capitolo 1, definiamo una "mappatura" in modo tale da essere un insieme di *elementi di mappatura*, ognuno dei quali indica che certi elementi del primo schema sono fatti corrispondere con certi elementi nel secondo schema. Col termine *modello* si vuole indicare una certa struttura dati, che in questa tesi è data dagli schemi XML.

Come prima cosa gli schemi XML in ingresso, cioè i modelli, vengono convertiti in grafi etichettati diretti. Successivamente, questi grafi vengono utilizzati in un calcolo iterativo di fixpoint, il cui risultato ci fornisce la "mappatura finale", in cui il punteggio di similarità fra le varie coppie di nodi ci dice quale elemento nel primo grafo è più simile ad un altro elemento nel secondo grafo. Per il calcolo della similarità ci si è basati sull'intuizione che elementi di due modelli distinti sono simili quando i loro elementi adiacenti sono simili. In altre parole, una parte della similarità di due elementi si propaga ai rispettivi elementi adiacenti. Al risultato finale prodotto, ossia alla mappatura, verrà poi applicato un filtro, per limitare il numero di mappature riportate. L'algoritmo qui presentato si concentra su matching binari non direzionali. Per una classificazione più ampia dei possibili obiettivi e approcci di matching, si veda [25].

4.2 Specifiche funzionali dell'algoritmo

Prima di descrivere in modo dettagliato l'algoritmo di match, diamo una panoramica generale dell'approccio seguito. Supponiamo di avere i due schemi XML (scritti in XML Schema e annotati precedentemente) riportati in Figura 1.4 e supponiamo che il nostro obiettivo sia quello di trovare i valori di similarità fra le coppie di nodi dei due alberi, rappresentanti le strutture dei due schemi, che sono identificati dagli stessi colori. La sequenza di passi che ci permette di determinare tali valori, può essere espressa, sinteticamente e in maniera semplificata, come segue:

1. G_A = XMLDOMGraph(schemaA); G_B = XMLDOMGraph(schemaB);
2. initialMap = StringMatcher(G_A, G_B);
3. multimapping = match(G_A, G_B, initialMap);
4. result = ThresholdFilter(multimapping);

Come primo passo, traduciamo gli schemi XML dal loro formato nativo(XML Schema) nei grafi G_A e G_B. La traduzione in grafi degli schemi è fatta utilizzando la API DOM e le specifiche dei modelli RDF (vedi paragrafo successivo). Come secondo passo, otteniamo una mappatura iniziale tra G_A e G_B, usando un operatore `StringMatcher`, che utilizza le gerarchie di ipernimi di WordNet per trovare un valore iniziale di similarità fra i nodi dei due grafi (vedi paragrafo 4.2.2). Come terzo passo, applichiamo un operatore `match` per affinare la mappatura tra G_A e G_B. Questo operatore si basa su un calcolo iterativo di fixpoint, che sfrutta le informazioni strutturali che derivano dagli schemi XML originali (vedi paragrafo 4.2.3). Come ultimo passo, applichiamo un filtro per selezionare un sottoinsieme di coppie di nodi da `multimapping`, che corrisponde ai match migliori (vedi paragrafo 4.2.4). I vari passi appena descritti, sono riassunti graficamente in Figura 4.1.

4.2.1 Conversione degli schemi XML

Come è stato introdotto nel Capitolo 1, per definire un operatore di match, bisogna prima scegliere una rappresentazione per i suoi schemi in ingresso. Si è scelto di rappresentare gli schemi XML come grafi etichettati diretti. In particolare, per arrivare a una tale rappresentazione, si è fatto prima il parsing degli schemi XML, utilizzando la API DOM (*Document Object Model*) e successivamente si è costruito il grafo etichettato seguendo lo standard RDF (*Resource Description Framework*) [36].

Diamo prima alcune definizioni di richiamo sui grafi e poi vediamo come si è arrivati a rappresentare gli schemi come grafi.

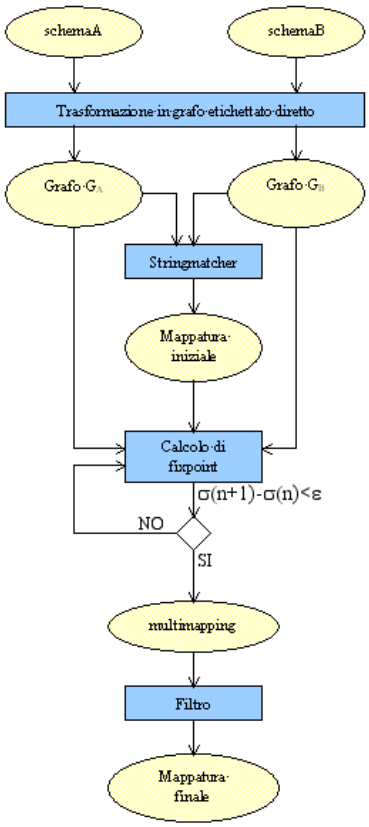

Figura 4.1: Struttura dell'approccio.

Definizione 4.1 (Grafo orientato). Un grafo orientato G = (V,A) è definito da un insieme finito $V(G)=\{v_1, \ldots, v_n\}$ di elementi detti vertici e da un insieme $A(G) = \{a_1, \ldots, a_m\} \subseteq V \times V$ di coppie ordinate di vertici dette archi. Dato l'arco $a = (v,u)$, il primo vertice nella coppia (v) è detto *coda*, e si dice che l'arco a è uscente da v. Il secondo vertice (u) è detto *testa*, e si dice che l'arco a è entrante in u. L'arco a si dice orientato dal nodo v al nodo u.

I grafi da noi considerati sono naturalmente dei grafi orientati, ossia del tipo rappresentato in Figura 4.2.

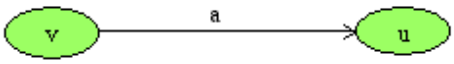

Figura 4.2: Esempio di grafo orientato.

Definizione 4.1 (Grafo Totale Diretto). Un grafo G = (V,A,s,t) è un *grafo totale diretto* se V è un *insieme di vertici*, A è un *insieme di archi* ed s e t sono *funzioni totali* da A a V, che assegnano ad ogni arco i loro vertici sorgente(*source*) e destinazione(*target*), rispettivamente.

Nel nostro grafo rappresentativo dello schema XML, due nodi possono essere collegati da un solo ramo. La definizione seguente introduce le etichette (*label*) sui nodi e sui rami.

Definizione 4.2 (Grafo Etichettato Diretto). Sia **L** un insieme arbitrario di etichette. Un grafo G=(V,A,s,t,l) è un (**L** -)*grafo etichettato diretto* se (V,A,s,t) è un grafo totale diretto e l: $V \cup A \rightarrow$ **L** è una *funzione totale di etichetta* che assegna ad ogni nodo e ad ogni ramo un'etichetta da **L**.

4.2.1.1 I parser XML

Con il consolidamento di XML, come strumento per l'interscambio di informazioni tra applicazioni e come linguaggio di programmazione general-purpose, si è resa necessaria la creazione di strumenti, nella fattispecie di parser XML sotto forma di API, che permettessero la lettura e l'analisi di documenti XML.

Un parser XML è un modulo software che si colloca tra l'applicazione e il documento XML. Esso permette all'applicazione di accedere al contenuto e alla struttura del documento XML. Esistono due tipi di parser: validanti e non validanti. I primi, oltre a controllare se un documento è ben-formato, cioè che ogni elemento sia racchiuso tra due tag (uno di apertura e uno di chiusura), controlla pure se esso è un documento XML valido, cioè se è fedele alle regole definite nella sua DTD (o schema). I parser non validanti, invece, si preoccupano solo di vedere se un documento è ben formato. Il parser utilizzato nella tesi, per gli schemi XML, è non validante, ossia controlla solo che lo schema sia ben formato e rispetti le norme del W3C.

Ci sono due modi per interfacciare un parser con una applicazione: usando un'interfaccia object-based oppure usando un interfaccia event-based. Con l'approccio object-based, il parser costruisce esplicitamente in memoria un albero che contiene tutti gli elementi del documento XML. Con l'approccio event-based, invece, il parser genera un evento ogni qual volta incontra, durante la lettura del documento XML, un elemento, un attributo, o del testo; ci sono eventi per i tag di apertura e di chiusura, per gli attributi, per il testo, per le entità e così via.

Nella tesi è stato usato un approccio di tipo object-based, utilizzando la API DOM. Riprendiamo l'esempio di riferimento introdotto nel Capitolo 1 (in Figura 1.4), e riportiamo di seguito il codice dello schema XML rappresentato dall'albero di destra (schemaB):

Esempio 4.1

```
<?xml version="1.0" encoding="UTF-8"?>
<schema>
    <element name="cdstore" type="cdstoreType"/>
    <complexType name="cdstoreType">
        <element name="name" type="string"/>
        <element name="address" type="addressType"/>
```

```
        <element name="cd" type="cdType"/>
    </complexType>
    <complexType name="addressType">
        <element name="city" type="string"/>
        <element name="street" type="string"/>
        <element name="state" type="string"/>
    </complexType>
    <complexType name="cdType">
        <element name="vocalist" type="string"/>
        <element name="cdtitle" type="string"/>
        <element name="tracklist" type="tracklistType"/>
    </complexType>
    <complexType name="tracklistType">
        <element name="passage" type="passageType"/>
    </complexType>
    <complexType name="passageType">
        <element name="title" type="string"/>
    </complexType>
</schema>
```

Il codice Java utilizzato per eseguire il parsing dello schema è il seguente:

```
import javax.xml.parsers;

import org.w3c.dom;
import org.xml.sax.SAXException;
import java.io.IOException;

…
String xmlFile = "file:///schemaB.xsd";
Node doc;
try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder parser = factory.newDocumentBuilder();

        doc = parser.parse(schema);

} catch (SAXException e)
       { e.printStackTrace(); }
  catch (IOException ioe)
      { ioe.printStackTrace(); }
```

Nel paragrafo seguente vediamo più in dettaglio le caratteristiche di DOM.

4.2.1.2 La API DOM

Il Document Object Model, DOM in breve, è una struttura dati astratta che permette di accedere alle informazioni immagazzinate in un documento XML come in un modello ad oggetti gerarchico. DOM crea un albero di nodi basato sulla struttura e sulle informazioni del documento XML. Le informazioni testuali nel documento XML vengono trasformate in un insieme di nodi di albero, come illustrato, molto semplicemente, nella Figura 4.3.

DOM è basato su un modello di dati implicito, che è simile, ma non del tutto, ai modelli di dati utilizzati da altre tecnologie XML come XPath, Infoset XML e SAX. Vediamo ora come DOM interpreta un documento XML.

Secondo DOM, un documento XML è un albero composto da nodi di diversi tipi. L'albero ha un singolo nodo radice e tutti i nodi in questo albero, salvo la radice, hanno un singolo nodo padre. Inoltre, ogni nodo ha un elenco di nodi figli. Può succedere che questo elenco di figli sia vuoto, in tal caso il nodo è chiamato un *nodo foglia*.

Ci possono anche essere nodi che non fanno parte della struttura ad albero. Per esempio, ogni nodo attributo appartiene ad un nodo elemento, ma non è pensato per essere un figlio di quell'elemento. Inoltre, i nodi possono essere rimossi dall'albero o creati, ma non inseriti nell'albero. Un documento completo DOM è quindi composto da un albero di nodi, da vari nodi che sono associati in qualche modo ad altri nodi nell'albero(ma non sono essi stessi parte dell'albero) e da un assortimento casuale di nodi scollegati.

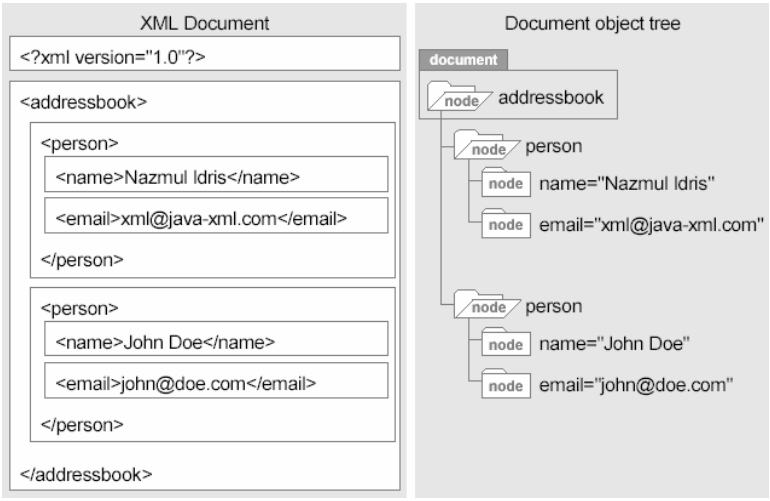

Figura 4.3: L'albero DOM basato sul modello ad oggetti per le informazioni in un documento XML.

Oltre ai suoi collegamenti di albero, ogni nodo ha un nome locale, un namespace URI e un prefisso, sebbene per numerosi tipi di nodo, questi possono essere nulli. Per esempio il nome locale, il

namespace URI e il prefisso di un commento sono sempre nulli. Ogni nodo ha inoltre un *nome di nodo*. Per un elemento o per un attributo, il nome del nodo è il nome premesso. Per altri concetti, come le notazioni o le entità, il nome del nodo è il nome della notazione o dell'entità. Per nodi senza nomi, come i nodi di testo, il nome del nodo è il valore, preso dal seguente elenco, che corrisponde al tipo di nodo:

- #document
- #comment
- #text
- #cdata-section
- #document-fragment

Infine ogni nodo ha un *valore* stringa. Per nodi di testo e commenti, questo tende a essere il testo del nodo. Per gli attributi il valore stringa è il valore normalizzato dell'attributo. Per tutti gli altri tipi di nodo, compresi elementi e documenti, il valore è nullo.

Nel nostro caso, avendo a che fare con schemi XML e non con documenti XML, i soli tipi di nodo che si hanno sono quelli di tipo Element, Document e Attribute.

4.2.1.3 Rappresentazione RDF degli schemi XML

Una volta effettuato il parsing dello schema XML, si è percorso l'albero DOM e, secondo le specifiche RDF, si è costruito il modello, formato da triple di valori del tipo (*s, p, o)* chiamate *statement*. Il modello finale ottenuto rappresenta il grafo etichettato diretto dello schema XML di partenza. Vediamo questo passaggio in maniera più approfondita.

Sia **U** l'alfabeto Unicode e **U*** l'insieme di stringhe definite su **U**. L'insieme delle entità **E** e l'insieme degli statement **V** sono definiti usando la seguente definizione ricorsiva:

1. $\mathbf{U}^* \times \mathbf{U}^* \subset \mathbf{E}$: ogni tupla formata da due stringhe è un'entità. La prima stringa della tupla è chiamata *tipo* o *namespace* dell'entità, la seconda stringa viene indicata come il *nome* dell'entità.

2. $\mathbf{E} \times \mathbf{E} \times \mathbf{E} \subset \mathbf{V}$: ogni tupla formata da tre stringhe costituisce uno *statement*.

3. $\mathbf{V} \subset \mathbf{E}$: ogni statement è un'entità.

4. **V** ed **E** sono i più piccoli insiemi aventi le proprietà precedenti.

Un sottoinsieme di **V** è chiamato *modello*. La definizione e la terminologia utilizzate sono basate sullo standard RDF. Come suggeriscono le definizioni ricorsive di **V** ed **E**, gli statement possono essere annidati, cioè uno statement può essere usato come elemento di un altro statement. Nel nostro

modello di dati interno, tuttavia, ciò non si verifica, quindi facciamo un'assunzione per semplificare, cioè che $\mathbf{E} = \mathbf{U}^* \times \mathbf{U}^*$ e $\mathbf{V} = \mathbf{E}^3$. Quindi, un modello è un sottoinsieme di $\mathbf{E}^3$.

I modelli possono essere rappresentati graficamente, come illustrato nella Figura 4.4, che mostra una parte della rappresentazione in grafo etichettato diretto di schemaB riportato nell'Esempio 4.1. Si è scelta una rappresentazione di tipo XML/DOM, in cui le relazioni gerarchiche fra gli elementi vengono evidenziate utilizzando l'etichetta `child`. In questa rappresentazione, le entità corrispondono ai nodi del grafo, mentre gli statement rappresentano gli archi. Per ogni statement (*s*,*p*,*o*), *s* rappresenta il *soggetto* (*subject*), ossia il nodo sorgente, *o* rappresenta l'*oggetto* (*object*), cioè il nodo destinazione e *p* rappresenta il *predicato* (*predicate*), cioè l'etichetta dell'arco. Un insieme di statement con lo stesso predicato definisce una relazione binaria sulle entità.

Nella Figura 4.4 distinguiamo due diversi tipi di nodo: quelli rappresentati da ovali identificano delle *risorse*, quelli rappresentati da rettangoli identificano dei *letterali* e appartengono al sottoinsieme delle entità $\mathbf{L} = \{\text{"letterali"}\} \times \mathbf{U}^*$. La direzione della freccia è importante: l'arco inizia sempre dal soggetto e punta l'oggetto, mai il contrario.

Considerando ancora la Figura 4.4, i valori indicati da b1, b2, b3, ecc., sono degli identificativi di nodo. Nel seguito saranno indicati, per chiarezza come [*tag*:*name*], quindi dall'esempio risulta che b1 può essere rappresentato con [`element:cdstore`], b2 con [`complexType:cdstoreType`] e così via. In realtà quei nodi non avrebbero nessun identificativo proprio, poiché rappresentano ciò che è contenuto tra un tag di apertura e uno di chiusura. Infatti, come si fa se vogliamo fornire meta-informazioni su una meta-informazione? Ad esempio, come posso esprimere che <tag2> è figlio di <tag1>? Bisogna considerare la meta-informazione come una risorsa da descrivere. Questa procedura si chiama *reificazione* (riduzione ad oggetto) dello statement. Dopo aver reificato lo statement, è possibile esprimere ulteriori proprietà su di esso.

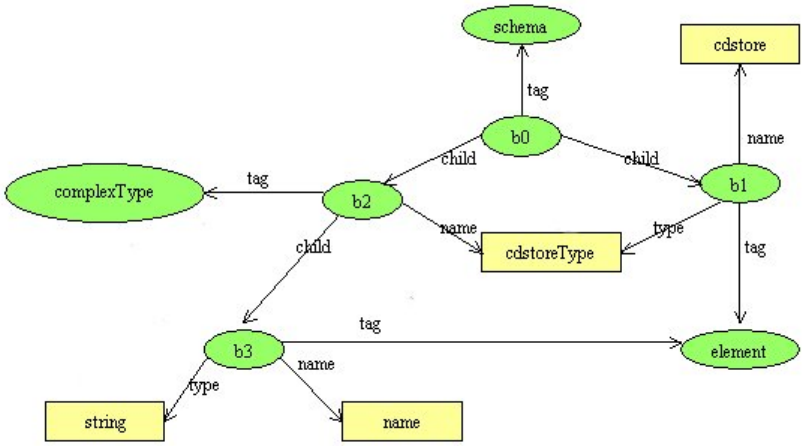

Figura 4.4 : Una porzione del grafo G_B rappresentativo di schemaB.

4.2.2 Creazione della mappatura iniziale

Una volta che gli schemi XML sono stati trasformati in grafi, il passo successivo è di creare una *mappatura iniziale*, cioè un valore iniziale di similarità σ^0 fra i nodi dei grafi. Si cercherà poi di migliorare tale valore utilizzando una formula iterativa, che ad ogni iterazione ricalcala il valore di similarità σ fra due nodi, solo se la differenza fra la σ all'iterazione corrente e la σ all'iterazione precedente, è maggiore di un certo valore ε, cioè se si è ottenuto un miglioramento. Consideriamo, oltre a schemaB dell'Esempio 4.1, anche schemaA, relativo alla struttura ad albero di sinistra in Figura 1.4, il cui codice è riportato nell'Esempio 4.2.

Esempio 4.2

```
<?xml version="1.0" encoding="UTF-8"?>
<schema>
 <element name="musicstore" type="musicstoreType"/>
 <complexType name="musicstoreType">
     <element name="signboard" type="signboardType"/>
     <element name="location" type="locationType"/>
     <element name="storage" type="storageType"/>
 </complexType>
 <complexType name="locationType">
     <element name="town" type="string"/>
     <element name="country" type="string"/>
 </complexType>
 <complexType name="signboardType">
     <element name="namesign" type="string"/>
     <element name="colorsign" type="string"/>
 </complexType>
 <complexType name="storageType">
     <element name="stock" type="stockType"/>
 </complexType>
 <complexType name="stockType">
     <element name="compackDisk" type="compackDiskType"/>
 </complexType>
 <complexType name="compackDiskType">
     <element name="albumTitle" type="string"/>
     <element name="songlist" type="songlistType"/>
 </complexType>
 <complexType name="songlistType">
     <element name="track" type="trackType"/>
 </complexType>
 <complexType name="trackType">
     <element name="songtitle" type="string"/>
     <element name="singer" type="string"/>
```

```
 </complexType>
</xs:schema>
```

Trasformiamo anche questo schema in grafo. Ora abbiamo i due grafi G_A, relativo a schemaA e G_B, relativo a schemaB, di cui vogliamo eseguire il match. Per calcolare il valore iniziale, σ^0, della mappatura costruiamo prima il *grafo di connettività a coppie* (*Pairwise Connectivity Graph* o PCG), definito come segue: se nel grafo G_A ho lo statement (a, p, a'), e nel grafo G_B ho lo statement (b,p b'), allora nel grafo di connettività a coppie posso unire i due statement in uno solo, dato da ((a,b),p,(a',b')), poiché i due nodi sorgenti e i due nodi destinazione sono uniti da un arco avente la stessa etichetta p. In modo formale, il grafo di connettività a coppie è definito come segue:

$$((a,b), p, (a',b')) \in \text{PCG} \quad \Leftrightarrow \quad (a, p, a') \in \text{A e } (b, p, b') \in \text{B}$$

Ad esempio, dati i due grafi, A e B, riportati in Figura 4.5a, il corrispondente grafo di connettività a coppie sarà come quello riportato in Figura 4.5b.

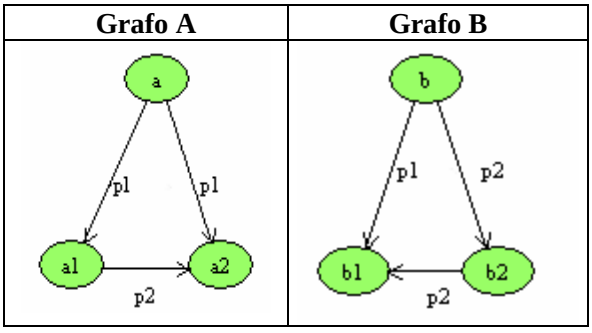

Figura 4.5a

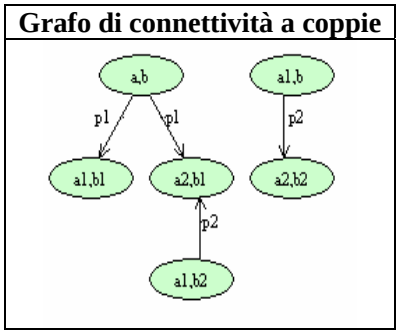

Figura 4.5b

Ogni nodo nel grafo di connettività a coppie è un elemento da A × B e prende il nome di *coppia di mappe*. L'intuizione che sta sotto agli archi che connettono le coppie di mappe è la seguente. Consideriamo ad esempio le coppia di mappe (a,b) e (a_1,b_1) in Figura 4.5b. Se a è simile a b, allora probabilmente a_1 è simile a b_1. La prova di questa affermazione è fornita dall'arco p_1 che connette a con a_1 nel grafo A e b con b_1 nel grafo B. Questa prova viene ben resa nel grafo di connettività a coppie dall'arco p_1 che porta da (a,b) in (a_1,b_1). Chiamiamo (a,b) e (a_1,b_1) nodi *vicini* (*neighbor*).

Consideriamo ora una porzione del grafo rappresentativo di schemaA, riportato in Figura 4.6:

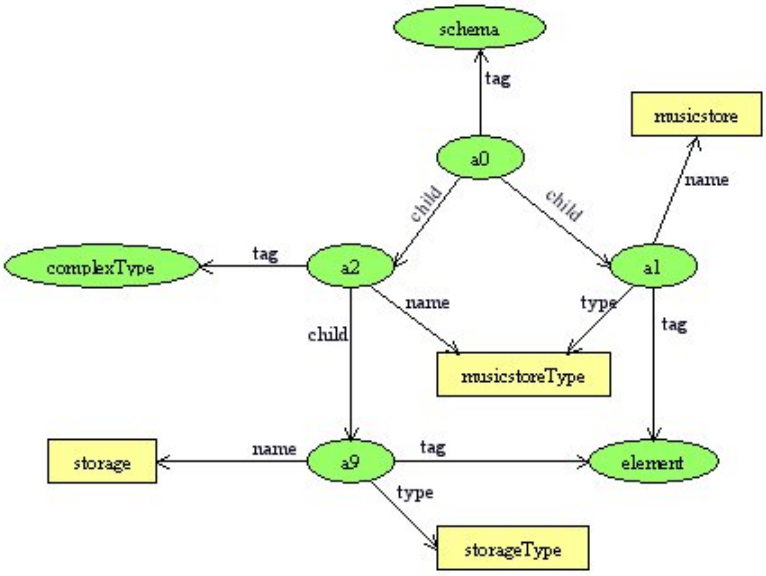

Figura 4.6: Una porzione del grafo G_A rappresentativo di schemaA.

Il grafo di connettività che si ottiene dai due grafi riportati in Figura 4.4 e Figura 4.6, è caratterizzato da statement del tipo $((a,b), p, (a',b'))$, di cui una parte è riportati in Tabella 4.1.

$s_A s_B$	p	$o_A o_B$
(a2,b0)	child	(a9,b1)
(a2,bo)	child	(a9,b2)
(a2,b1)	name	(musicstoreType,cdstore)
(a9,b1)	name	(storage,cdstore)
(a9,b2)	name	(storage,cdstoreType)
(a9,b3)	name	(storage,name)
(a2,b1)	tag	(complexType,element)
(a9,b1)	tag	(element,element)
(a9,b2)	tag	(element,complexType)
(a9,b3)	tag	(element,element)
(a9,b1)	type	(storageType,cdstoreType)
(a9,b3)	type	(storage,cdstoreType)

Tabella 4.1: Statement del grafo di connettività a coppie fra schemaA e schemaB.

Il corrispondente grafo di connettività a coppie, relativo agli statement di Tabella 4.1, è riportato di seguito in Figura 4.7.

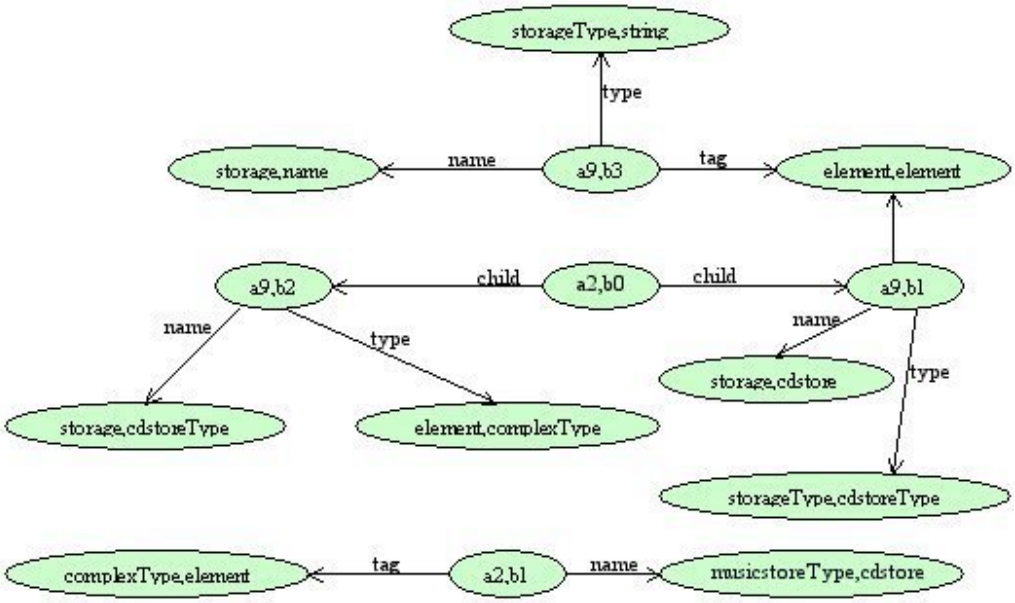

Figura 4.7: Grafo di connettività a coppie di G_A e G_B

Per ogni coppia di mappe (corrispondente ai nodi del PCG) così ottenuta, viene calcolato il valore di similarità iniziale σ^0, compreso fra 0 e 1, dove 1 indica che i due termini esprimono lo stesso concetto. L'insieme di tutti i σ^0 ci dà la mappatura iniziale. Per il calcolo di σ^0 si è proceduto in due modi a secondo dei tipi di nodo considerati:

- Due letterali: partendo dai significati associati ai letterali nella fase di annotazione, si è usata una formula ricavata dal *modello Vector-Space Generalizzato* [37].

- Due risorse: se i due nodi risorsa risultano uguali, σ^0 viene posto a 1, altrimenti ad un valore minimo di similarità, che è stato fissato a 0.01.

- Una risorsa e un letterale: σ^0 viene posto al valore minimo di similarità, 0.01.

Si è preferito porre un valore minimo di similarità diverso da zero per un miglior funzionamento dell'algoritmo di matching (come spiegato nel paragrafo 4.4.1). La formula utilizzata nel primo caso, invece, viene spiegata in dettaglio nel paragrafo successivo.

4.2.2.1 Il modello Vector-Space generalizzato

Il modello Vector-Space Generalizzato, come si deduce dal nome, è una generalizzazione del modello Vector-Space ideato da Salton nel 1975 e ormai largamente diffuso nel campo dell'Information

Retrieval. Questo modello è rappresentato da uno spazio t-dimensionale, in cui ogni dimensione corrisponde ad un elemento del dominio di interesse (ad esempio, un concetto o termine contenuto in un documento), mentre ogni oggetto del dominio di interesse (ad esempio, un documento) è rappresentato da un vettore, le cui componenti sono i "pesi" associati agli elementi che descrivono gli oggetti. I pesi esprimono l'importanza dell'elemento nella caratterizzazione di un certo dominio di interesse.

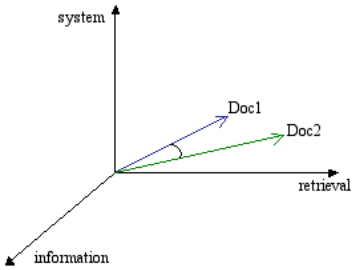

Figura 4.8: Modello Vector-Space tradizionale.

Nel modello Vector-Space tradizionale, la somiglianza tra due vettori è espressa dal coseno dell'angolo tra di essi. In questo modello tutti i versori, che caratterizzano un elemento del dominio, sono fra loro ortogonali, come si vede nell'esempio di Figura 4.8, dove i versori rappresentano i concetti di *system*, *retrieval* e *information* e questo significa che il prodotto scalare fra due versori qualsiasi è nullo, mentre il prodotto scalare del versore con se stesso è pari a 1.

$$\vec{a} \cdot \vec{b} = \begin{cases} 1 & (\vec{a} = \vec{b}) \\ 0 & (\vec{a} \neq \vec{b}) \end{cases}$$

Nella generalizzazione del modello Vector-Space invece, si suppone che due versori distinti non siano esattamente perpendicolari l'uno all'altro, come si vede in Figura 4.9, quindi il prodotto scalare fra di essi è un valore compreso fra 0 e 1.

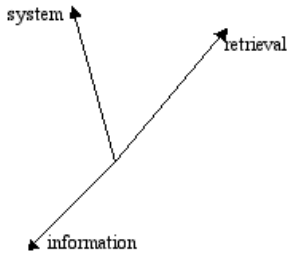

Figura 4.9: Modello Vector-Space Generalizzato.

Ad esempio, se esiste una gerarchia in cui sono rappresentati tutti gli elementi del dominio, allora, per ogni coppia di elementi (*a*,*b*) nel modello Vector-Space Generalizzato, potremmo avere che il prodotto scalare fra di loro non sarà più espresso come nella formula precedente, ma dalla formula che segue:

$$\vec{a} \cdot \vec{b} = \frac{2 \cdot depth(LCA(a,b))}{depth(a) + depth(b)}$$

Questa formula è stata utilizzata per il calcolo del σ^0 iniziale:

$$\sigma^0(a,b) = \frac{2 \cdot depth(LCA(a,b))}{depth(a) + depth(b)}$$

Essa, sfrutta le gerarchie di ipernimi di WordNet. Infatti, i termini degli schemi XML, che vengono forniti in input, sono stati "annotati" precedentemente (vedi [24]) e tale annotazione ci fornisce il significato della parola, tramite un codice univoco, riconosciuto da WordNet. Al denominatore della formula viene messa la somma delle profondità delle due gerarchie, mentre al numeratore viene presa due volte la profondità dell'antenato comune ai termini *a* e *b* più basso nella gerarchia di ipernimi (*Lowest Common Ancesto*r). Per capire meglio come funziona, vediamo un esempio con i termini *singer*, di schemaA, e *vocalist*, in schemaB. La gerarchia di ipernimi per il termine *singer*, preso col primo senso, è riportata di seguito, con indicato a fianco il livello, partendo dal basso:

1. singer, vocalist, vocalizer, vocaliser
livello 8 =>singer, vocalist, vocalizer, vocaliser
livello 7 =>musician, instrumentalist, player
livello 6 =>performer, performing artist
livello 5 =>entertainer
livello 4 =>person, individual, someone, somebody, mortal, human, soul
livello 3 =>organism, being
livello 2 =>living thing, animate thing
livello 1 =>object, physical object
livello 0 =>entity, physical thing
livello 3 =>causal agent, cause, causal agency
livello 2 =>entity, physical thing

Facciamo la stessa cosa per la gerarchia di ipernimi del termine *vocalist*, che ha un solo senso:

1. singer, vocalist, vocalizer, vocaliser
livello 8 =>singer, vocalist, vocalizer, vocaliser
livello 7 =>musician, instrumentalist, player
livello 6 =>performer, performing artist
livello 5 =>entertainer
livello 4 =>person, individual, someone, somebody, mortal, human, soul

livello 3 =>organism, being
livello 2 =>living thing, animate thing
livello 1 =>object, physical object
livello 0 =>entity, physical thing
livello 3 =>causal agent, cause, causal agency
livello 2 =>entity, physical thing

In questo caso i due termini sono sinonimi, quindi il valore di similarità iniziale tra i due termini risulterà pari a 1.0.

$$\sigma^0 = \frac{2 \cdot depth(LCA(\text{singer}, \text{vocalist}))}{depth(\text{singer}) + depth(\text{vocalist})} = \frac{2 \cdot 8}{8+8} = 1.0$$

Consideriamo ora il caso di due termini che non sono sinonimi, come *singer* e *cd*. Vediamo la gerarchia di ipernimi di *cd*, preso con il senso 4:

4. compact disk, compact disc, CD
livello 8 =>compact disk, compact disc, CD
livello 7 =>recording
livello 6 =>memory device, storage device
livello 5 =>device
livello 4 =>instrumentality, instrumentation
livello 3 =>artifact, artefact
livello 2 =>object, physical object
livello 1 =>entity, physical thing
livello 2 =>whole, whole thing, unit
livello 1 =>object, physical object
livello 0 =>entity, physical thing

Il valore ottenuto con la formula è molto basso dato che l'antenato comune è al livello 1.

$$\sigma^0 = \frac{2 \cdot depth(LCA(\text{singer}, \text{cd}))}{depth(\text{singer}) + depth(cd)} = \frac{2 \cdot 1}{8+8} = 0.125$$

4.2.3 Calcolo iterativo di fixpoint

Sia $\sigma(x,y) \geq 0$ la misura di similarità, dei nodi $x \in G_A$ e $y \in G_B$, definita come una funzione totale sul prodotto $G_A \times G_B$, partendo da una mappatura iniziale σ^0, così come è stata definita nel paragrafo precedente. L'algoritmo di matching si basa, per il calcolo della similarità fra le coppie di mappe, su un calcolo iterativo dei valori σ; σ^n rappresenta il valore della mappatura fra le coppie di mappe alla n-

esima iterazione. In ogni iterazione, il valore σ, per una coppia di mappe (a,b), viene incrementato dai valori σ delle coppie di mappe vicine, moltiplicati per i valori dei coefficienti di propagazione. Intuitivamente, si cerca di affinare la similarità tra le coppie di mappe, considerando le coppie di mappe vicine e il "peso" di tali coppia nel calcolo della similarità, espresso appunto dal coefficiente di propagazione associato all'arco che le collega. Le varie formule di fixpoint che sono state considerate sono riportate in Tabella 4.2.

Sigla	Formula di Fixpoint
FTF	$\sigma^{n+1} = normalize\ (\varphi(\sigma^0 + \sigma^n))$
TFF	$\sigma^{n+1} = normalize\ (\sigma^0 + \varphi(\sigma^n))$
FFT	$\sigma^{n+1} = normalize\ (\sigma^n + \varphi(\sigma^n))$
TTF	$\sigma^{n+1} = normalize\ (\sigma^0 + \varphi(\sigma^0 + \sigma^n))$
FTT	$\sigma^{n+1} = normalize\ (\sigma^n + \varphi(\sigma^0 + \sigma^n))$
TFT	$\sigma^{n+1} = normalize\ (\sigma^0 + \sigma^n + \varphi(\sigma^n))$
TTT	$\sigma^{n+1} = normalize\ (\sigma^0 + \sigma^n + \varphi(\sigma^0 + \sigma^n))$

Tabella 4.2: Varianti della formula di fixpoint.

In generale, la mappatura σ^{n+1} viene calcolata dalla mappatura σ^n e σ^0 (ipotizzando di utilizzare la formula TTT) come segue (la normalizzazione viene omessa per chiarezza):

$$\sigma^{n+1}(x,y) = \sigma^0(x,y) + \sigma^n(x,y) + \sum_{(a_u,p,x)\in G_A,(b_u,p,y)\in G_B} (\sigma^0(a_u,b_u) + \sigma^n(a_u,b_u)) \cdot w((a_u,b_u),(x,y)) + \sum_{(x,p,a_v)\in G_A,(y,p,b_v)\in G_B} (\sigma^0(a_v,b_v) + \sigma^n(a_v,b_v)) \cdot w((a_v,b_v),(x,y))$$

dove con $w((a_u,b_u),(x,y))$ si indica il 'peso' del ramo che va dalla coppia (a_u,b_u) alla coppia (x,y). Il calcolo viene effettuato iterativamente, finché la lunghezza Euclidea del vettore residuo $\Delta(\sigma^n,\sigma^{n+1})$ diventa minore di un certo ε, per $n > 0$. Se il calcolo non converge, lo terminiamo dopo un certo numero massimo di iterazioni.

Il punto centrale della formula di fixpoint si basa sulla funzione φ, che prende come parametro in ingresso una mappatura σ e fornisce in uscita una mappatura θ. Per ogni coppia di modelli, G_A e G_B, la funzione φ è definita come segue:

$$\varphi(\sigma) = \theta \Leftrightarrow \forall (a,b) \in G_A \times G_B:\ \theta(a,b) = \sum_{(a,p,x)\in G_A,(b,q,y)\in G_B} \sigma(x,y) \cdot \pi_r(\langle x,p,G_A\rangle,\langle y,q,G_B\rangle) + \sum_{(x,p,a)\in G_A,(y,q,b)\in G_B} \sigma(x,y) \cdot \pi_l(\langle x,p,G_A\rangle,\langle y,q,G_B\rangle)$$

La funzione φ descrive come la similarità dei *neighbor* della coppia di mappe (a,b) influenza la similarità di (a,b) stesso. La funzione π definisce i coefficienti di propagazione per una coppia di mappe (x,y), rispetto agli archi etichettati con p, nel modello G_A e rispetto agli archi etichettati con q, nel modello G_B. I coefficienti di propagazione vengono utilizzati per dare un "peso" agli archi nel grafo di propagazione.

Il *grafo di propagazione della similarità* è una struttura dati ausiliaria, utilizzata nel calcolo di fixpoint, ricavato dal grafo di connettività a coppie semplicemente aggiungendogli, per ogni arco, un arco aggiuntivo, che va nella direzione opposta. Il peso sull'arco viene chiamato, appunto, *coefficiente di propagazione*, varia tra 0 ed 1 inclusi, e può essere calcolato utilizzando differenti formule (vedi Tabella 4.3).

Approccio	p = q	p ≠ q
Inverse product	$\frac{1}{card_{\{l,r\}}(x,p,A)\cdot card_{\{l,r\}}(y,q,B)}$	0
Inverse average	$\frac{2}{card_{\{l,r\}}(x,p,A)+card_{\{l,r\}}(y,q,B)}$	0
Inverse total product	$\frac{1}{card_{\{l,r\}}(p,A)\cdot card_{\{l,r\}}(q,B)}$	0
Inverse total average	$\frac{2}{card_{\{l,r\}}(p,A)+card_{\{l,r\}}(q,B)}$	0
Combined inverse average	$\frac{4}{(card_{\{l,r\}}(p,A)+card_{\{l,r\}}(q,B))\cdot(card_{\{l,r\}}(x,p,A)+card_{\{l,r\}}(y,q,B))}$	0
Equal	1.0	0

Tabella 4.3: Formule per il calcolo dei coefficienti di propagazione.

Nelle varie formule per il calcolo dei coefficienti di propagazione, *card*(p,M) fornisce il numero di rami che portano l'etichetta p, nel modello M, mentre *card*(x,p,M) fornisce il numero di rami che portano l'etichetta p, entranti ed uscenti, in un nodo x, nel modello M, ossia, in modo formale:

$\forall M \in \mathbf{V}^3, \forall x \in \mathbf{E}, \forall p \in \mathbf{E}$: $card_l(x,p,M) = \left\| \{(x,p,t) \mid \exists t : (x,p,t) \in M\} \right\|$

$$card_r(x,p,M) = \left\| \{(t,p,x) \mid \exists t : (t,p,x) \in M\} \right\|$$

Considerando ancora l'esempio relativo ai grafi G_A e G_B, il grafo di propagazione che si ottiene dal grafo di connettività a coppie, mostrato in Figura 4.7, utilizzando, ad esempio, la formula *inverse product* per il calcolo dei coefficienti di propagazione sui rami diretti e inversi, è riportato in Figura 4.10.

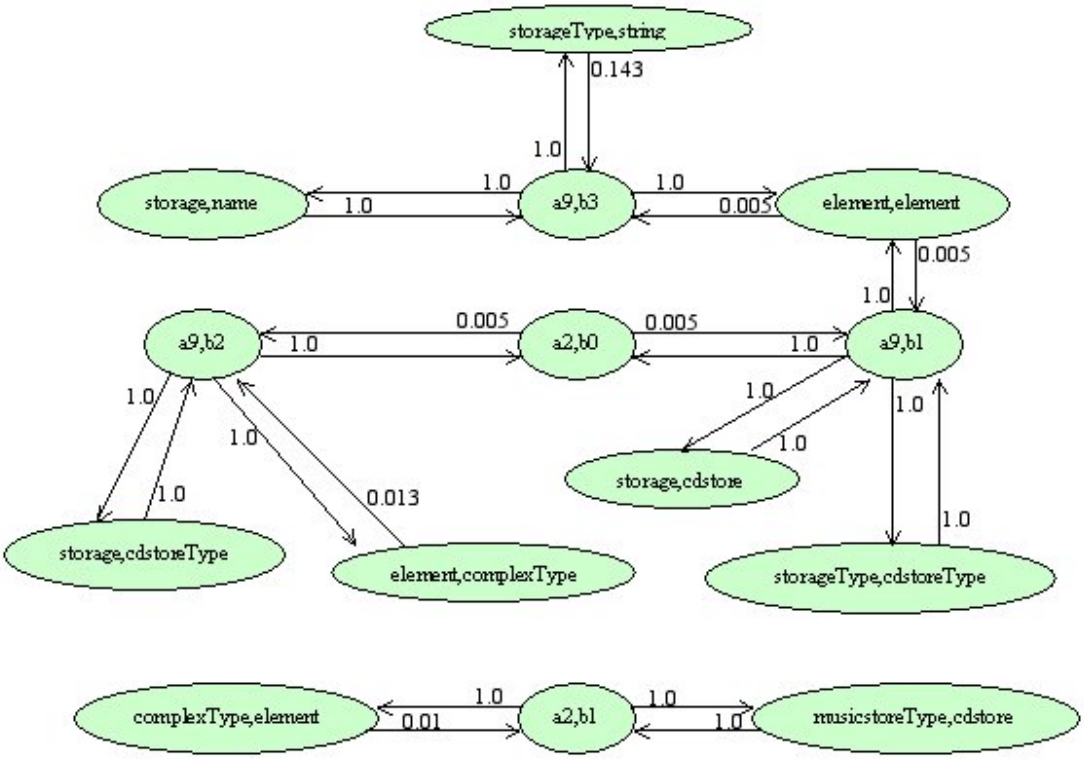

Figura 4.10: Grafo di propagazione indotta fra G_A e G_B.

La funzione θ(a,b) ottenuta, viene poi normalizzata tramite la funzione *normalize*, proiettando i valori della mappatura nell'intervallo [0,1], cioè la normalizzazione corrisponde a dividere il vettore σ per un valore scalare che rappresenta il più alto valore di similarità σ nell'iterazione corrente:

$$\theta = normalize(\sigma) \Leftrightarrow \forall (a,b) \in A \times B: \quad \sigma(a,b) = \frac{\sigma(a,b)}{\max\{s \mid \exists x, y : \sigma(x,y) = s\}}$$

4.2.3.1 Significato matematico della formula di fixpoint

Il calcolo di fixpoint dell'algoritmo di match, può essere espresso come il calcolo di autovettori che illustriamo di seguito.

Sia T la matrice quadrata corrispondente al grafo di propagazione G, ottenuto dai grafi G_A e G_B; se in G c'è un ramo che va dalla coppia di mappe j = (x,y) alla coppia di mappe i = (x',y'), con coefficiente di propagazione sul ramo pari a c, allora l'elemento t_{ij} della matrice T avrà valore c. Notate

che i coefficienti di propagazione in G, corrispondono alle probabilità di transizione, nel caso che T sia una matrice di transizione. Il calcolo iterativo di fixpoint converge quando T è una matrice *aperiodica* ed *irriducibile.*

Definizione 4.1 (Matrice riducibile e irriducibile): Una matrice quadrata $A = (a_{ij})_{i,j=1,\ldots,n}$, di ordine n, si dice *riducibile* se gli indici 1,2,…n possono essere divisi in due insiemi disgiunti, non vuoti i_1, $i_2,\ldots i_\mu$ e $j_1, j_2,\ldots, j_\nu$ (con $\mu + \nu = n$) tale che:

$$a_{i_\alpha j_\beta} = 0$$

per $\alpha = 1,..,\ \mu$ e $\beta = 1,\ldots,\ \upsilon$. Una matrice quadrata che non è riducibile si dice *irriducibile*.
La matrice T è irriducibile se e soltanto se il grafo di connettività G associato è *strettamente connesso* (cioè, ogni nodo è raggiungibile da ogni altro nodo). Un approccio, per assicurare questa proprietà, consiste nell'introdurre in G un "cappio" per ogni nodo. Si noti che, in Tabella 4.2, le formule che assicurano l'irriducibilità di G sono quelle in cui viene utilizzato il valore σ^0 nel calcolo della funzione φ, per esempio come nelle formule TTF, FTT e TTT; un approccio di questo tipo viene riferito, nella letteratura, col termine "dampening". Se σ^0 assegna un valore maggiore di zero ad ogni coppia di mappe in $G_A \times G_B$ allora, l'aggiunta di σ^0 equivale a modificare G in G', in cui tutti i nodi sono interconnessi, con determinati coefficienti di propagazione. Sia ora T' la matrice quadrata di incidenza associata a G'. Ora, il calcolo dell'autovettore si può esprimere come segue. Sia S un vettore di coppie di mappe che, ad ogni posizione, contiene un valore di similarità da σ, per un ordine fissato di coppie di mappe. Un'iterazione della formula di fixpoint corrisponde al prodotto matrice per vettore $T' \times S$. Moltiplicare ripetutamente S per T' produce l'autovettore S^* della matrice T', tale che:

$$T' \times S^* = \lambda S^*$$

dove λ è l'*autovalore dominante* di T'. Nella formula di fixpoint, la normalizzazione corrisponde a dividere $T' \times S^*$ per l'autovalore λ.

Il calcolo di fixpoint corrisponde al calcolo delle *catene di Markov* su T (vedi Appendice A). Poiché T corrisponde alla matrice di transizione sul grafo di propagazione G, la misura di similarità che si ottiene, può essere vista come la distribuzione stazionaria di probabilità su una coppia di mappe, prodotta da un cammino casuale da una coppia di mappe ad un'altra. Questo cammino casuale corrisponde al processo di matching manuale eseguito da un progettista umano sui grafi G_A e G_B, cioè il progettista, iniziando da una data coppia di mappe, deduce la similarità di un'altra coppia basandosi sulle proprietà strutturali di G_A e G_B.

La convergenza delle iterazioni nel calcolo di fixpoint può essere misurata usando il *vettore residuo*, calcolato come:

$$R_i = \frac{T' \times S_i}{\lambda_i} - S_i$$

dove $\frac{T' \times S_i}{\lambda_i} = S^*$ rappresenta l'autovettore dominante della matrice di transizione T'. Possiamo considerare $||R_i||$ come un indicatore che ci dice quanto bene S_i approssima S*. Nel nostro caso siamo

interessati solo all'ordine dei risultati delle coppie di mappe e non ai valori assoluti dei coefficienti di similarità. Quindi, in questo caso, le iterazioni possono essere interrotte quanto l'ordine in un certo sottoinsieme di una mappatura, con il più alto valore di similarità, si è stabilizzato, cioè non cambia tra σ^{n-1} e σ^{n}. In uno scenario reale possiamo dire che questo criterio è soddisfatto per

$$||R_i|| = \sqrt{\sum (\sigma^{n} - \sigma^{n-1})^2} < 0.05$$

4.2.4 Filtraggio dei risultati

In questo paragrafo esaminiamo alcuni metodi per scegliere i migliori candidati di match dalla lista di coppie di mappe restituita dall'algoritmo. Di solito, per ogni elemento appartenente ai modelli di cui si è fatto il matching, l'algoritmo restituisce un insieme molto ampio di possibili candidati di match, quindi il risultato immediato del calcolo di fixpoint potrebbe essere molto voluminoso. Indichiamo il risultato fornito dal calcolo di fixpoint col termine *multimappatura* (o *multimapping*), in quanto contiene molte mappature potenzialmente utili come sottoinsiemi.

La scelta dei criteri da usare per limitare il multimapping non è immediata, dipende dagli obiettivi che ci si prefigge. Un ulteriore problema è dato dal numero di multimapping restituiti, infatti da un insieme di *n* coppie di mappe si possono formate fino a 2^n sottoinsiemi diversi; se si considera l'esempio relativo ai grafi G_A e G_B il grafo di propagazione ottenuto contiene ben 4688 nodi (coppie di mappe), quindi si possono formare fino a 2^{4688} possibili sottoinsiemi diversi.Per illustrare meglio il problema della selezione dei risultati, consideriamo il risultato di match ottenuto da due piccoli modelli A e B riportato sulla sinistra di Figura 4.11. Il multimapping M contiene quattro coppie di mappe con similarità $\sigma(a1,b1) = 1.0$, $\sigma(a2,b2) = 0.54$, $\sigma(a1,b2) = 0.81$, $\sigma(a2,b2) = 0.27$. Per l'insieme formato dalle quattro coppie di mappe, si possono selezionare $2^4 = 16$ possibili sottoinsiemi e ognuno di questi sottoinsiemi può essere un'alternativa valida per il risultato di match finale che viene restituito.

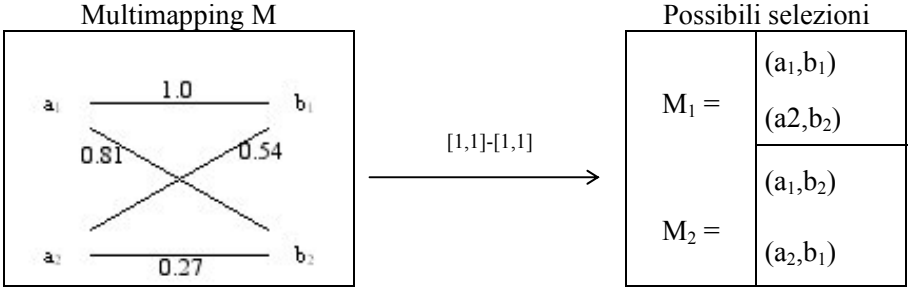

Figura 4.11: Similarità cumulativa vs. "matrimonio stabile"

Per risolvere il problema della selezione dei risultati migliori di match, abbiamo utilizzato i seguenti criteri:

- vincoli specifici per l'applicazione per ridurre la dimensione del multimapping:
  - typing;
  - cardinalità.

- Tecniche di selezione sviluppate, nell'ambito del matching, in grafi bipartitici per scegliere il sottoinsieme di risultati da restituire:
  - Problema del "matrimonio stabile";
  - Problema dell'assegnamento;
  - Valore di soglia (similarity threshold).

4.2.4.1 Vincoli

Il primo vincolo che è stato usato per ridurre il numero di multimapping è un vincolo di typing. Siccome il nostro obiettivo è quello di effettuare il match fra schemi XML al fine di trovare la similarità fra i diversi termini che li compongono, ciò che ci interessa è il matching fra coppie di mappe del tipo [`element:`*`name`*] e [`complexType:`*`name`*], cioè possiamo ignorare i match fra i soli letterali e fra letterali e risorse.

Il secondo vincolo utilizzato riguarda la cardinalità dei risultati di match. In generale, il multimapping fornito dall'algoritmo ha una cardinalità di tipo [0,n]–[0,n], cioè il numero valido di candidati di match, per ogni elemento di G_A, è compreso fra 0 ed n e lo stesso vale per gli elementi di G_B. Questo significa che sia elementi di G_A che elementi di G_B possono avere zero, uno o più candidati di match.

Sfortunatamente il solo uso di vincoli di typing e di cardinalità non sempre è sufficiente per limitare in modo adeguato il numero di candidati di match. Consideriamo ancora l'esempio riportato in Figura 4.11, e restringiamo la cardinalità dei risultati restituiti a [1,1]–[1,1], cioè una mappatura uno-a-uno, limitiamo la scelta a due sottoinsiemi di coppie di mappe, invece di sedici (vedi lato destro di Figura 4.11). Quindi, anche dopo aver applicato un vincolo di cardinalità molto più ristretto ad un modello molto piccolo, rimaniamo comunque con più di una scelta. Di seguito vediamo possibili metri di selezione per prendere una decisione fra le alternative rimanenti.

4.2.4.2 Metriche di selezione

Per fare una scelta appropriata su ciò che costituisce la mappatura 'migliore' c'è bisogno di un'intuizione. Nella letteratura riguardante il problema del matching tra grafi, un *matching* viene definito come una mappatura con cardinalità [0,1]–[0,1], cioè un insieme di archi in cui non ce ne sono due che incidono sullo stesso nodo. Un *grafo bipartitico* è un grafo i cui nodi formano due parti

disgiunte, in modo che non ci siano archi che connettono due nodi qualsiasi che si trovano nella stessa parte. Quindi una mappatura può essere vista come un grafo bipartitico indiretto, pesato.

Un'intuizione che verrà usata nelle strategie di selezione è quella del cosiddetto *problema del matrimonio stabile* [35]. Un esempio di dimensione n del problema del matrimonio stabile implica due insiemi disgiunti di dimensione n: gli uomini e le donne. Ad ogni persona è associata una *lista di preferenze strettamente ordinata*, contenete tutti i membri del sesso opposto. Una persona *p* preferisce *q* a *r*, quando *q* e *r* sono di sesso opposto a *p,* se e soltanto se *q* precede *r* nella lista delle preferenze di *p*. Per un tale esempio, un matching M è una corrispondenza biunivoca tra gli uomini e le donne. Se un uomo *m* e una donna *w* vengono corrisposti in M, allora *m e w* vengono chiamati *partner* in M e scriveremo che $m = p_M(w)$, $w = p_M(m)$, dove $p_M(w)$ è l'M-partner di *w* e $p_M(m)$ è l'M-partner di *m.* Il problema base del matrimonio stabile implica la determinazione di un matching stabile. Un certo matching è *instabile* se ci sono due parti, non fatte coincidere l'uno con l'altro, ognuna di delle quali preferisce rigorosamente l'altra parte rispetto al suo partner nella corrispondenza. Un matching *stabile* è una corrispondenza che non è instabile. Supponiamo che in Figura 4.8 gli elementi a1 e a2 corrispondano alle donne, mentre gli elementi b1 e b2 siano gli uomini, allora la mappatura M_1 soddisfa la condizione di matrimonio stabile, al contrario, la mappatura M_2 no, poiché in M_2 la donna a1 e l'uomo b1 preferiscono qualcun altro rispetto ai loro partner e questo mette in pericolo il loro matrimonio.

La proprietà del matrimonio stabile fornisce un criterio plausibile per selezionare le mappature desiderate da un multimapping. Tuttavia i candidati per le mappature chieste possono essere ricavati dai seguenti criteri di selezione e dai ben noti problemi di matching:

- Il *problema dell'assegnamento*, consiste nel trovare un matching M_i in un grafo bipartitico pesato M, che massimizzi il peso totale (similarità cumulativa) $\sum_{(x,y)\in M_i} \sigma(x,y)$. Visto come un matrimonio, tale matching massimizza la soddisfazione totale di tutti gli uomini e di tutte le donne. In Figura 4.11 si ha:

  $$\sum_{M_2} \sigma = 0.81 + 0.54 = 1.35 \qquad \sum_{M_1} \sigma = 1.0 + 0.27 = 1.27$$

  quindi, M_2 massimizza la soddisfazione totale di tutti gli uomini e di tutte le donne, ma M2 non è un matrimonio stabile. Il problema dell'assegnamento produce una mappatura di tipo [0,1]-[0,1], cioè dei matrimoni monogami e può essere risolto utilizzando algoritmi con complessità computazionale di tipo polinomiale[48,49].

- L'algoritmo di matching implementato assegna, al più, un solo valore di similarità per ogni coppia di mappe (x,y). Chiamiamo questo valore *similarità assoluta*. La similarità assoluta è simmetrica, cioè x è simile a y esattamente come y a x. Secondo l'interpretazione del matrimonio, ciò significa che, in ogni coppia, gli eventuali partner si piacciono l'uno con l'altro nella stessa misura.

- Il valore di *soglia di similarità* è l'ultimo criterio considerato. Per un dato valore di soglia di similarità *th*, selezioniamo dal multimapping solo quelle coppie il cui valore di similarità sia

uguale o maggiore al valore dato. Per esempio, in Figura 4.8,con un valore di soglia *th* = 0.5 a2 trova b2 accettabile (0.54) mentre non succede con b1 (0.27).

Riassumendo, i filtri che sono stati implementati per estrarre dal multimapping, i migliori candidati di match sono i seguenti:

- *ThresholdFilter*: restituisce una mappatura con cardinalità [0,1]-[0,1], ossia produce società monogame, utilizzando un valore di soglia di similarità assoluto pari a *th* = 0.4.

- *BestFilter*: restituisce una mappatura con cardinalità [0,1]-[0,1] e utilizza una metrica di selezione che corrisponde al problema dell'assegnamento. L'implementazione del filtro usa una regola euristica "greedy". Per il successivo elemento non corrisposto viene scelto un candidato di match che massimizzi la similarità cumulativa.

4.3 Valutazione dell'algoritmo

In questo paragrafo riportiamo i casi sperimentali analizzati. Il *caso base* utilizza come schemi schemaA e schemaB dell'esempio di riferimento, le cui strutture ad albero sono mostrate in Figura 4.12; successivamente questi schemi sono stai in parte modificati (producendo i casi 1, 2 e3), per vedere come ciò influiva sui valori di similarità fra coppie di mappe. Queste prove ci hanno inoltre permesso di esaminare come i diversi parametri dell'algoritmo influenzano i risultati di match.

Il primo parametro dell'algoritmo di match che è stato valutato è dato dalla formule per il calcolo dei coefficienti di propagazione. Sono state considerate sei diverse formule (vedi Tabella 4.3). Tenendo conto dei vincoli di typing imposti, le formule che hanno dato i risultati migliori sono state solo tre: *inverse product*, *inverse average* ed *equal*. Le altre tre formule davano i punteggi maggiori a coppie di mappe formate da letterali invece che risorse e questo non è il risultato che ci si aspettava.

Il secondo parametro considerato è dato dalla formula di fixpoint, le cui variazioni sono riportate in Tabella 4.2. Le prove sperimentali confermano l'analisi analitica presentata nel paragrafo [4.2.3.1], infatti i risultati confermano come le formule che non considerano il valore iniziale di similarità σ^0 nel calcolo della funzione φ, producono dei risultati molto peggiori, in quanto non garantiscono la stretta connessione del grafo di connettività. Delle restanti formule, i risultati migliori, per quanto riguarda il numero di matching restituiti, si sono ottenuti con le formule TTT e TFT.

Per limitare il numero di iterazioni eseguite con la formula di fixpoint, si sono considerati tre criteri di terminazione:

- Si è fissato un numero di iterazioni compreso fra 3 e 10 , entro il quale si è visto che in tutte le prove le formule di fixpoint convergevano.

- Si è fissato un tempo massimo di elaborazione per ogni iterazione, pari a 30 secondi, superato il quale il calcolo viene interrotto.

- Alla fine di ogni iterazione viene calcolata la lunghezza del *vettore residuo*:

La mappatura finale è stata ottenuta applicando il filtro *ThresholdFilter* con un valore di soglia pari a 0.4. I valori evidenziati in verdino, nelle varie tabelle di riferimento per le mappature finali, rappresentano, invece, i risultati restituiti applicando il filtro *BestFilter*. I risultati riportati dopo la linea grigia indicano coppie di mappe non restituite dal primo filtro ($\sigma < 0.4$), ma che riguardano comunque elementi di interesse per questo caso. Bisogna tenere conto che nelle tabelle di riferimento, sono state riportate solo le coppie di nodi relative agli elementi di interesse per i casi analizzati e non tutte mappature restituite come risultato finale.

In Tabella 4.4 è riportata la media aritmetica dei tempi, impiegata dalle due formule, per effettuare le 10 iterazioni. Si vede che la formula TTT impiega meno tempo rispetto alla TFT in tutti e tre i casi; inoltre il tempo medio migliore si è avuto con la formula *inverse product*.

	Media dei tempi	
Approccio	**TTT**	**TFT**
Product	13.65	14.621
Avg	13.84	13.902
Equal	14.365	16.11

Tabella 4.4: Media dei tempi per effettuare 10 iterazioni con la formula di fixpoint.

4.3.1 Caso base

Il *caso base* rappresenta l'esempio principale, che è stato sviluppato nel corso del capitolo, riportato in Figura 4.12. In particolare, qui vogliamo determinare i valori di similarità fra alcuni elementi dei due schemi. Osserviamo le due strutture ad albero prestiamo particolare attenzione ai nodi colorati.

Nodi in schemaA	**Nodi in schemaB**	σ^0
musicstore	cdstore	1.0
songlist	tracklist	1.0
compackDisk	cd	1.0
namesign	name	1.0
track	passage	0.5
songtitle	title	1.0
singer	vocalist	1.0

Tabella 4.5: Mappatura iniziale fra i nodi, con i medesimi colori, di schemaA e schemaB.

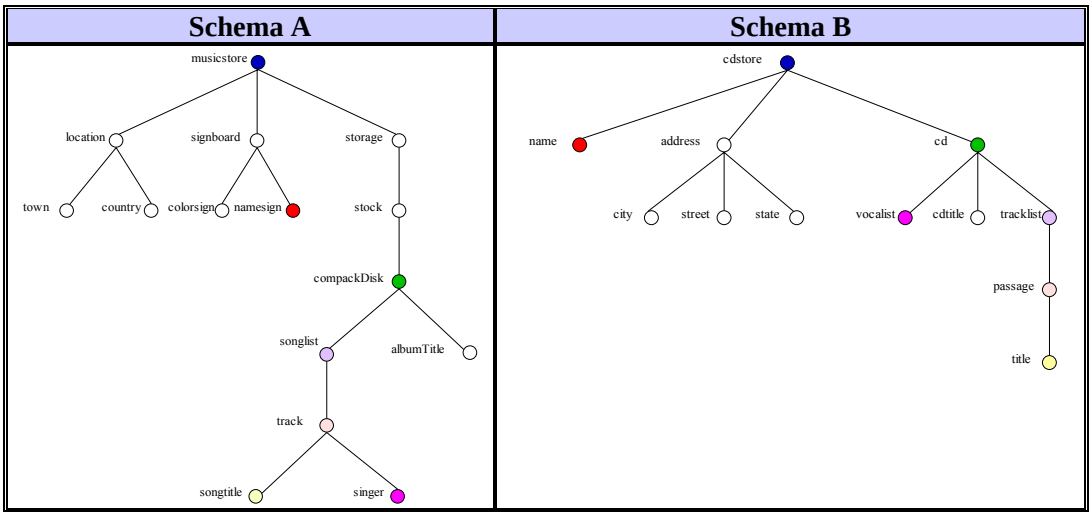

Figura 4.12: Strutture ad albero degli schemi schemaA e schemaB.

Dai valori iniziali di similarità σ^0, riportati in Tabella 4.5, si vede che le coppie di termini che identificano nodi evidenziati con i medesimi colori, sono caratterizzate da un valore σ^0 molto alto. A questo punto vogliamo vedere come questi valori iniziali, fra i singoli termini, e le proprietà strutturali delle due gerarchie, influenzano i valori di similarità fra i vari identificativi di nodo (che rappresentano gli elementi di interesse nel caso di matching fra schemi XML, come indicato nel paragrafo [4.2.4.1]), per mezzo del calcolo iterativo di fixpoint. Ricordiamo che il valore iniziale di similarità fra gli identificativi di nodo è pari a 1 solo se sono identici, altrimenti è pari al valore minimo di similarità, cioè 0.01.

I risultati ottenuti sono riportati in Tabella 4.6. Si vede subito come, ad esempio la similarità fra [`element:musicstore`] ed [`element:cdstore`] si notevolmente aumentata, passando da 0.01 a 0.637 (con la formula Product per il calcolo dei coefficienti di propagazione), così come quella fra [`element:compackDisk`] ed [`element:cd`]. Non sono invece aumentati di molto i valori di similarità fra la coppia di nodi ([`element:songtitle`], [`element:title`]), rimasta praticamente invariata, e la coppia di nodi ([`element:singer`], [`element:vocalist`]), aumentata ad un misero 0.005. Questi valori sono giustificati dalla posizione nelle due gerarchie, dei singoli nodi. Basta considerare `singer` e `vocalist`: in schemaA `singer` rappresenta l'interprete di ogni singola traccia di un cd, mentre `vocalist`, in schemaB, rappresenta l'unico interprete del cd, quindi hanno proprio due significati diversi.

Nodi in A	Nodi in B	Product	Avg	Equal
[element:musicstore]	[complexType:cdstoreType]	1.0	1.0	1.0
[complexType:compackDiskType]	[element:cd]	1.0	1.0	1.0
[complexType:songlistType]	[element:tracklist]	1.0	1.0	1.0
[element:compackDisk]	[complexType:cdType]	0.779	0.795	0.914
[element:compackDisk]	[element:cd]	0.755	0.755	0.755
[complexType:musicstoreType]	[element:cdstore]	0.755	0.755	0.755
[element:namesign]	[element:name]	0.755	0.755	0.755
[element:musicstore]	[element:cdstore]	0.637	0.637	0.637
[element:songlist]	[element:tracklist]	0.506	0.507	0.511
[element:songlist]	[complexType:tracklistType]	0.506	0.507	0.511
[element:track]	[complexType:passageType]	0.504	0.505	0.505
[element:track]	[element:passage]	0.504	0.505	0.505
[complexType:songlistType]	[complexType:tracklistType]	0.02	0.02	0.02
[complexType:trackType]	[complexType:passageType]	0.016	0.017	0.017
[element:songtitle]	[element:title]	0.011	0.014	0.014
[element:singer]	[element:vocalist]	0.005	0.05	0.05
[complexType:compackDiskType]	[complexType:cdType]	0.005	0.005	0.005

Tabella 4.6: Risultati di match per gli schemi schemaA e schemaB.

4.3.2 Caso 1

In questo caso schemaB è stato modificato, come si vede in Figura 4.13 sulla destra. L'elemento `vocalist` che in schemaB era figlio dell'elemento `cd`, è stato spostato più in basso, diventando ora figlio dell'elemento `passage`. Quindi, mentre in schemaB un `cd` era relativo ad un singolo cantante, ora in schemaB1 si può avere un cd con una lista di tracce (`tracklist`), in cui le varie tracce (`passage`) possono essere interpretate da cantanti diversi. In schemaB1 `vocalist` assume quindi lo stesso ruolo di `singer` in schemaA. Quello che ci aspettiamo dai valori di similarità fra le varie coppie di nodi presenti in Tabella 4.6, relative al *caso base*, è che in generale aumentino.

In Tabella 4.7 sono riportati i risultati del *caso 1*. Le frecce accanto ai valori stanno ad indicare come si è modificato il valore di similarità della coppia di mappe rispetto al *caso base*: freccia in su indica un aumento del valore, freccia in giù una diminuzione e freccia orizzontale indica un valore invariato. L'andamento dei valori è lo stesso per tutte e tre le formule relative ai coefficienti di propagazione, per questo motivo le frecce compaiono solo accanto ai valori della colonna di Product.

Come ci si aspettava i valori di similarità, in generale, sono cresciuti, soprattutto quello relativo alla coppia di nodi ([`element:singer`], [`element:vocalist`]), che è passat da 0.005 ad un valore pari a 0.999 (con la formula Product). Anche la similarità della coppia di nodi ([`element:songtitle`], [`element:title`]) è salita, influenzata dalla similarità dei nodi vicini, aumentata anch'essa.

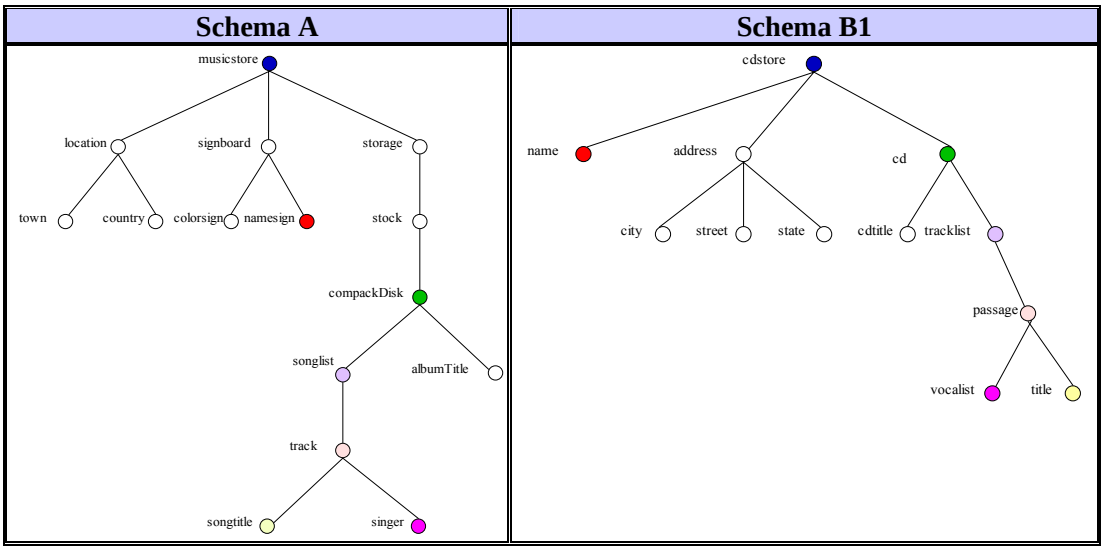

Figura 4.13: Strutture ad albero degli schemi schemaA e schemaB1.

Nodi in A	Nodi in B1	Product	Avg	Equal
[element:musicstore]	[complexType:cdstoreType]	← 1.0	1.0	1.0
[element:singer]	[element:vocalist]	↑ 0.999	0.995	1.0
[element:songlist]	[element:tracklist]	↑ 0.998	0.983	0.851
[element:namesign]	[element:name]	↑ 0.998	0.983	0.851
[element:musicstore]	[element:cdstore]	↑ 0.878	0.872	0.826
[complexType:songlistType]	[complexType:tracklistType]	↑ 0.754	0.746	0.675
[element:songlist]	[complexType:tracklistType]	↑ 0.754	0.746	0.675
[complexType:songlistType]	[element:tracklist]	↓ 0.604	0.599	0.555
[element:compackDisk]	[element:cd]	↓ 0.52	0.516	0.478
[complexType:compackDiskType]	[element:cd]	↓ 0.505	0.502	0.469
[complexType:musicstoreType]	[element:cdstore]	↓ 0.505	0.502	0.472
[element:songtitle]	[element:title]	↑ 0.504	0.501	0.469
[element:track]	[element:passage]	←0.504	0.496	0.429
[complexType:trackType]	[complexType:passageType]	← 0.344	0.342	0.328

Tabella 4.7: Risultati di match per gli schemi XML riportati in Figura 4.9.

4.3.3 Caso 2

In questo caso si è modificato schemaB abbassando `cdstore` di due livelli. Quello che ci si aspetta di trovare è che i nodi in verde, relativi alla coppia di nodi (`compackDisk`, `cd`), e in viola, relativi alla coppia di nodi (`singer,vocalist`), aumentino i loro valori di similarità, poiché vengono a trovarsi allo stesso livello nei due alberi, mentre dovrebbe diminuire la similarità fra i nodi in blu, ossia tra `musicstore` in schemaA e `cdstore` in schemaB2.

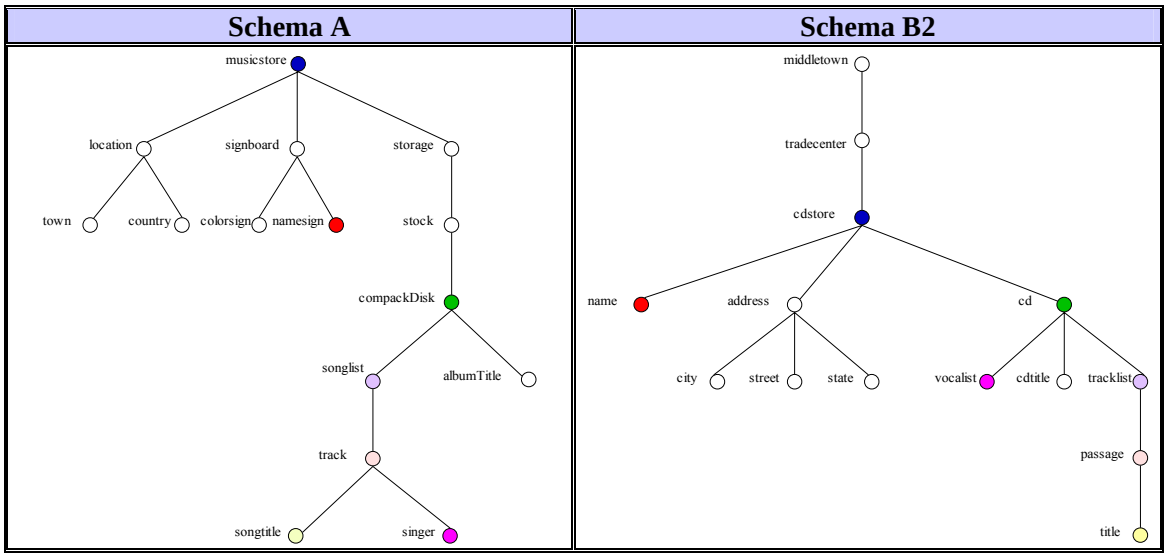

Figura 4.14: Strutture ad albero degli schemi schemaA e schemaB2.

I risultati del *caso 2* sono riportati in Tabella 4.8. I simboli e i colori utilizzati hanno lo stesso significato dei casi precedenti. Facendo riferimento ai valori relativi alla formula Product per il calcolo dei coefficienti di propagazione, si vede che la tendenza generale dei valori non ha deluso le aspettative: le similarità delle coppie di mappe ([`element:compackDisk`], [`element:cd`]) e ([`element:songlist`],[`element:tracklist`]) sono aumentate, mentre la similarità di ([`element:musicstore`], [`element:cdstore`]) è diminuita.

L'abbassamento del livello dei nodi `vocalist e tracklist` in schemaB2, e l'aumento della similarità fra le due coppie di nodi ([`element:compackDisk`], [`element:cd`]) e ([`element:songlist`],[`element:tracklist`]), hanno portato ad un aumento anche dei valori di similarità fra le coppie di nodi ([`element:songtitle`],[`element:title`]) e ([`element:singer`],[`element:vocalist`]).

Nodi in A	Nodi in B2	Product	Avg	Equal
[element:compackDisk]	[element:cd]	↑ 1.0	1.0	1.0
[element:compackDisk]	[complexType:cdType]	↑ 1.0	1.0	1.0
[element:songlist]	[element:tracklist]	↑ 1.0	1.0	1.0
[complexType:songlistType]	[complexType:tracklistType]	↑ 0.914	0.914	0.914
[element:singer]	[element:vocalist]	↑ 0.758	0.774	0.914
[element:musicstore]	[complexType:cdstoreType]	↓ 0.757	0.771	0.914
[complexType:musicstoreType]	[element:cdstore]	← 0.755	0.755	0.755
[element:songlist]	[complexType:tracklistType]	↑ 0.755	0.755	0.755
[element:musicstore]	[element:cdstore]	↓ 0.526	0.526	0.533
[element:namesign]	[element:name]	↓ 0.506	0.507	0.511
[complexType:compackDiskType]	[element:cd]	↓ 0.505	0.505	0.512
[element:songtitle]	[element:title]	↑ 0.505	0.505	0.512
[element:track]	[element:passage]	↓ 0.467	0.497	0.497
[element:track]	[complexType:passageType]	↓ 0.382	0.382	0.382
[complexType:trackType]	[complexType:passageType]	← 0.02	0.02	0.02

Tabella 4.8: Risultati di match per gli schemi XML riportati in Figura 4.10.

4.3.4 Caso 3

In questo caso è stato modificato schemaB1, mantenendo inalterata tutta la gerarchia al di sotto del nodo `cd`, ma modificando completamente la gerarchi ad di sopra, cioè è diverso il contesto nel quale vengono considerati i vari cd. Ora in schemaB3 non si ha più un negozio di cd, ma un catalogo, diviso per categorie e nella categoria `music`, si registrano i cd e le rispettive informazioni.

Quello che ci aspettiamo è che i valori di similarità relativi alle coppie di nodi (compackDisk, cd), (`songlist,tracklist`), (`singer,vocalist`) e (`songtitle,title`) in generale aumentino, come per il *caso 1*, essendo rimasta invariata la struttura, e quindi la similarità, dei nodi adiacenti, mentre invece dovrebbe diminuire notevolmente il valore di similarità fra il nodo `musicstore` e il nuovo nodo radice in schemaB3, `catalog`.

I risultati del caso 3 sono riportati nella Tabella 4.9. Si vede come la coppie di nodi ([`element:catalog`],[`element:musicstore`]) abbia un valore di similarità veramente basso, pari a 0.03, proprio come ci si aspettava.

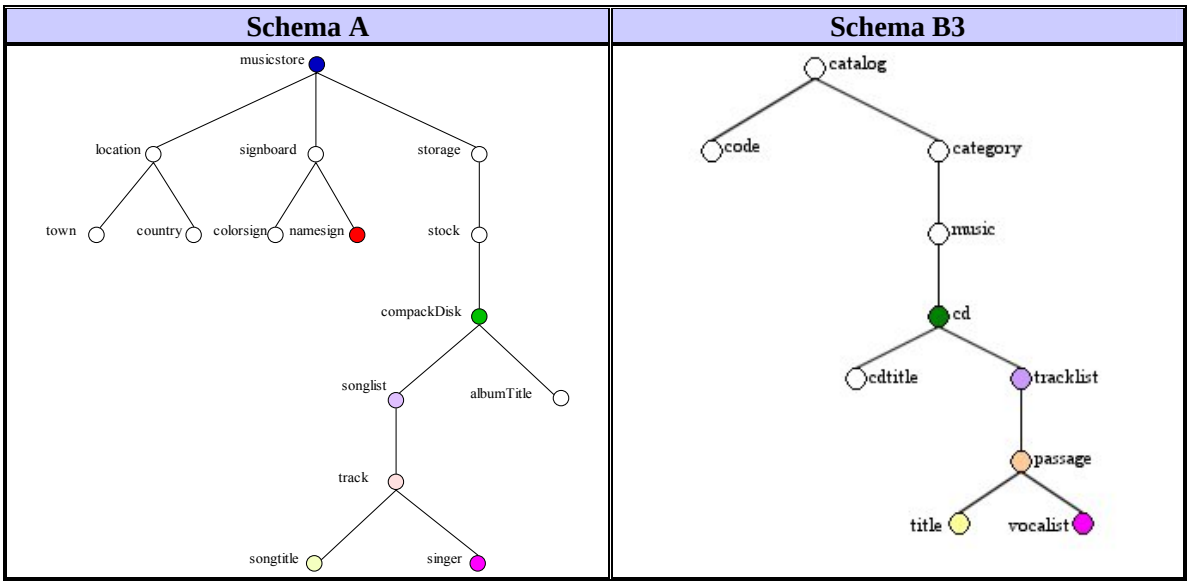

Figura 4.15: Strutture ad albero degli schemi schemaA e schemaB3.

Nodi in A	Nodi in B3	Product	Avg	Equal
[element:compackDisk]	[complexType:cdType]	↑ 1.0	1.0	1.0
[complexType:songlistType]	[complexType:tracklistType]	↑ 0.755	0.755	0.755
[element:singer]	[element:vocalist]	↑ 0.755	0.755	0.755
[element:songlist]	[complexType:tracklistType]	↑ 0.755	0.755	0.755
[complexType:songlistType]	[element:tracklist]	↓ 0.605	0.606	0.622
[element:compackDisk]	[element:cd]	↓ 0.509	0.509	0.515
[element:songtitle]	[element:title]	↑ 0.505	0.506	0.512
[complexType:compackDiskType]	[element:cd]	↓ 0.505	0.506	0.512
[element:catalog]	[element:musicstore]	0.03	0.03	0.03
[complexType:trackType]	[complexType:passageType]	↑ 0.02	0.02	0.02
[element:track]	[complexType:passageType]	↓ 0.007	0.007	0.007
[element:track]	[element:passage]	↓ 0.005	0.005	0.005

Tabella 4.9: Risultati di match per gli schemi XML riportati in Figura 4.11.

Conclusione e sviluppi futuri

In questa tesi si è implementato un metodo per effettuare in maniera automatica il matching fra i diversi schemi XML che descrivono i documenti presenti nell'archivio di una Biblioteca Digitale XML. La tesi è stata svolta nell'ambito del progetto ECD (*Enhanced Content Delivery*) atto a fornire, usando mezzi come biblioteche digitali e sistemi mediatori, contenuti arricchiti agli utenti finali, in modo da rispondere più efficacemente alle loro aspettative.

Il metodo proposto si basa sull'utilizzo di ontologie, in grado di arricchire semanticamente i metadati utilizzati per descrivere i documenti, e sull'utilizzo delle informazioni strutturali fornite dagli schemi XML. Sfruttando le ontologie, è possibile risolvere le differenze, fra i termini nei diversi schemi, utilizzando le annotazioni ricavate tramite l'uso del sistema MOMIS. MOMIS è un sistema a mediatori che permette l'annotazione manuale dei termini degli schemi.

Assumendo, quindi, che gli schemi XML considerati siano stati precedentemente annotati, questi sono successivamente convertiti in grafi etichettati diretti (secondo le specifiche dell'RDF), i quali costituiscono la rappresentazione interna scelta per l'input dell'operazione di match fra schemi. Dopo la trasformazione, dai grafi ottenuti, si ricava una mappatura iniziale fra le coppie di nodi, cioè trovata un valore iniziale di similarità fra le coppie di nodi. Il passo successivo consiste nel cercare di affinare tale valore, sfruttando le informazioni strutturali, degli schemi XML, che legano fra loro i vari concetti. Per fare ciò, viene utilizzato un calcolo iterativo di fixpoint. L'intuizione che sta alla base del calcolo della similarità è che gli elementi di due *modelli* distinti sono simili quando i loro elementi adiacenti sono simili, in altre parole, una parte della similarità di due elementi si propaga ai rispettivi elementi adiacenti. Il calcolo di fixpoint viene reiterato fintanto che si ottiene un miglioramento nel valore della similarità fra una coppia di nodi; quando questo miglioramento non si ha più, significa che il valore di similarità tende a convergere e la distanza Euclidea fra la similarità dell'iterazione corrente e quella dell'iterazione precedente, è minore di una certa soglia ε.

Infine sono stati implementare dei filtri in grado di diminuire il numero di mappature restituite dall'algoritmo di match, in modo da avere solo i migliori risultati di match. Naturalmente, ciò che rappresenta il "miglior risultato di match" dipende anche dall'obiettivo che ci si prefigge. A questo proposito, sono state analizzati diversi parametri di "modulazione" dell'algoritmo di match, che sono rappresentati dalle diverse formule di fixpoint e da quelle per il calcolo dei coefficienti di

propagazione. Anche per il filtraggio delle mappature sono stati considerati diversi vincoli e metriche di selezione. Questi vincoli e metriche possono essere combinate in modi diversi, ottenendo vari filtri, a secondo dell'obiettivo.

Il modulo software progettato e costruito nell'ambito di questa tesi al fine di trovare i valori di similarità fra termini appartenenti a schemi diversi (all'interno di un insieme di schemi annotati), potrà, in futuro, essere sfruttato in un modulo, ancora da implementare, di gestione delle query. Una volta che questo sarà stato implementato risulterà possibile effettuare la riscrittura della richiesta, posta dall'utente (in un linguaggio di interrogazione per XML come XQuery), solo per quei documenti, contenuti nell'archivio della Biblioteca Digitale, che si conformano a schemi i cui termini presentano i valori di similarità migliori con i termini della query.

Un criterio per riscrivere la query, sui singoli schemi XML, che permetterebbe di individuare, in ogni schema, i path (o cammini) che meglio rappresentano i vincoli della query, potrebbe essere quello del cosiddetto problema del *matrimonio stabile*. In questo problema, ognuna delle *n* donne e degli *n* uomini elenca i membri del sesso opposto, in ordine di preferenza. L'obiettivo è di trovare il miglior match tra uomini e donne. Secondo questo criterio, la query sarebbe riscritta solo per quegli schemi in cui le coppie di termini (x,y) rappresentano un matrimonio stabile, cioè non succede che esista un'altra coppia (*x*',*y*') tale che *x* preferisca *y*' a *y* e *y*' preferisca *x* a *x*'.

Inoltre, al fine di rendere significativo il verso di "attraversamento" degli schemi (cioè la similarità fra schemaA e schemaB non rappresenta la stessa di quella fra schemaB e schemaA), al posto della similarità assoluta calcolata con il nostro algoritmo, si potrebbe considerare la *similarità relativa*. La similarità relativa è asimmetrica e viene calcolata come frazione delle similarità assolute dei migliori candidati di match per ogni elemento dato. Nell'esempio di Figura 4.5, b_1 è il miglior candidato di match per a_2, quindi poniamo $\vec{\sigma}_{rel}(a_2, b_1) = 1.0$. La similarità relativa per tutti gli altri candidati di match di a_2 viene calcolata come frazione di $\sigma(a_2,b_1)$, quindi, ad esempio

$$\vec{\sigma}_{rel}(a_2, b_2) = \frac{\sigma(a_2, b_2)}{\sigma(a_2, b_1)} = \frac{0.27}{0.54} = 0.5$$

Bibliografia

[1] World Wide Web Consortium Working Draft. *XQuery 1.0: An XML Query Language*. Agosto 2003. http://www.w3.org/TR/xquery/.

[2] World Wide Web Consortium Working Draft. *XML Syntax for XQuery 1.0 (XQueryX)*. Giugno 2001. http://www.w3.org/TR/xqueryx/.

[3] Don Chamberlin, Jonathan Robie, e Daniela Florescu. *Quilt: an XML Query Language for Heterogeneous Data Sources*. In *Lecture Notes in Computer Science*, Springer-Verlag, Dicembre 2000. http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf.

[4] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*. W3C Recommandation, Novembre 1999. http://www.w3.org/TR/xpath.html.

[5] J. Robie, J. Lapp, D. Schach. *XML Query Language (XQL)*. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[6] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *A Query Language for XML*. http://www.research.att.com/~mff/files/final.html.

[7] International Organization for Standardization (ISO). *Information Technology-Database Language SQL*. Standard No. ISO/IEC 9075:1999. (Disponibile presso American National Standards Institute, New York, NY 10036, (212) 642-4900).

[8] Rick Cattell et al. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann Publishers, San Francisco, 1996.

[9] George A. Miller. *Wordnet: A Lexical Database for English*. Communications for the ACM, 38(11):39-41, 1995. Il sito di WordNet si trova presso http://www.cogsci.princeton.edu/wn/.

[10] A. Bonifati e S. Ceri. *Comparative analysis of five XML languages*. SIGMOD Record, 29(1), Marzo 2000.

[11] S. Cluet, A. Deutsch, D. Florescu, A. Levi, D. Maier, J. McHugh, J. Robie, D. Sucio, J. Widom. *XML query languages: experiences and examples*. 1999.

[12] N. Fuhr, K. Großjohann. *XQIRL: A query language for information Retrieval in XML documents.* In Proceeding of SIGIR, 2001.

[13] S.Abiteboul, D.Quass, J.McHoug, and J.Widom. *The lorel query language for semistructured data*. International Journal on Digital Libraries, pages 68–88, Aprile 1997.

[14] A. Theobald, G. Weikum. *Adding relevance to XML.* In 3rd International Workshop on the Web and Databases *(WebDB)*, 2000. http://www-dbs.cs.uni-sb.de/papers/webdb2000.ps.

[15] Dongwon Lee and Wesley W.Chu. *Comparative analysis of six xml schema languages*. *ACM SIGMOD Record*, Settembre 2000. http://www.w3.org/TR/REC-xml.

[16] World Wide Web Consortium Working Draft. *Extensible Markup Language (XML) 1.0*. Ottobre 2000.

[17] S. Amer-Yahia, S. Cho, D. Srivastava. *Tree Pattern Relaxation*. In Proc. of Int. Conf. on Extending Database Technology (EDBT 2002), Marzo 2002.

[18] P. Ciaccia, W. Penzo. *Relevance ranking tuning for similarity queries on XML data.* Proceedings of the 28th VLDB Conference,2002.

[19] P. Ciaccia, W. Penzo. *Adding flexibility to structure similarità queries on XML data.* FQAS 2002, LNAI 2522. pagine 124-139, 2002.

[20] T. Schlieder, F. Naumann. *Approximate tree embedding for querying XML data.* Proceeding of the workshop on the Web Information and Data.

[21] Torsten Schlieder. *ApproXQL: design and implementation of an approximate pattern matching language for XML*. Technical Report B01-01, Freie Universitat, Berlin, 2001.

[22] Istituto CNR di Pisa. *Open dlib official site*. Informazioni presso http://opendlib.iei.pi.cnr.it/.

[23] *Enhanced Content Delivery*. Informazioni presso il sito http://www-ecd.cnuce.cnr.it.

[24] Daniele Miselli. *Riscrittura di interrogazioni XML: un approccio basato sull'analisi semantica degli schemi.* Tesi di laurea, anno 2001/2002.

[25] E. Rahm, P.A. Bernstein. *On Matching Schemas Automatically.* Techical Report MSR-TR-2001-17, http://www.research.microsoft.com/pubs/, Feb. 2001.

[26] Li, W., C. Clifton. *Semantic Integration in Heterogeneous Databases Using Neural Networks.* Proc. VLDB 1994, 1-12.

[27] Li, W., C. Clifton. *SemInt: A tool for identifying attribute correspondences in heterogeneous databases using neural network.* Data and Knowledge Engineering, 33 (1), 200.

[28] Li, W., C. Clifton, S. Liu. *Database Integration Using Neural Networks: Implementation and Experiences.* Knowledge and Information Systems, 2(1), 2000.

[29] Doan, A.H., P. Domingos, A. Levy. *Learning Sources Descripions for Data Integration.* Proc. WebDB 2000, pp 81-92.

[30] Mitra, P., G. Wiederhold, J. Jannink: *Semi-automatic Integration of Knowledge Sources*. Proc. of Fusion '99, Sunnyvale, USA, July 1999.

[31] Milo, T., S. Zohar: *Using Schema Matching to Simplify Heterogeneous Data Translation*. Proc. VLDB 1998

[32] L. Palopoli., D. Sacca, D. Ursino. *Semi-Automatic, Semantic Discovery of Properties from Database Schemas*. Proc. IDEAS, 1998, 244-253.

[33] L. Palopoli, D. Sacca, G. Terracina, D. Ursino: *A Unified Graph-Based Framework for Deriving Nominal Interscheme Properties, Type Conflicts and Object Cluster Similarities.* Proc.4th CoopIS, IEEE Computer Society, 1999, 34-45.

[34] S. Melnik, H. Garcia-Molina, E. Rahm. *Similarity Flooding: A Versatile Graph Matching Algorithm*. Extended Technical Report, http://dbpubs.stanford.edu/pub/2001-25, 2001.

[35] D. Gusfield, R. Irving. *The Stable MarriagePproblem: Structure and Algorithms.* MIT Press, Cambridge, MA, 1989.

[36] O. Lassila, R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification.* http://www.w3.org/TR/REC-rdf-syntax/, 1998.

[37] P. Ganesan, H. Garcia-Molina, J. Widom. *Expoiting Hierarchical Domain Structure to Compute Similarity*. 2003.

[38] Word Wide Web Consortium. *XML Schema Part 0: Primer.* Maggio 2001. http://www.w3.org/TR/xmlschema-0/.

[39] Word Wide Web Consortium. *XML Schema Part 1: Structure.* Maggio 2001. http://www.w3.org/TR/xmlschema-1/.

[40] Word Wide Web Consortium. *XML Schema Part 2: Datatype.* Maggio 2001. http://www.w3.org/TR/xmlschema-2/.

[41] J. Madhavan, P.A. Bernstein, and E. Rahm. *Generic schema matching with Cupid*. Proceedings of the 27th VLDB Conference 2001, pag 49-58.

[42] S. Castano, V. De Antonellis, S. De Capitani di Vemercati. *Semantic Integration of Heterogeneous Data Sources*. Trans. On Data and Knowledge Eng., to appear.

[43] S. Castano; V. De Antonellis. *A Schema Analysis and Reconciliation Tool Environment*. Proc. IDEAS '99, IEEE, 1999.

[44] S. Bergamaschi, S. Castano, M. Vincini. *Semantic Integration of Semistructured and Structured Data Source*. SIGMOND Record 28(1),1999.

[45] D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, M. Vincini. *Information Integration: the MOMIS Project Demonstration.* Proc. VLDB 2000, 611-614.

[46] Bergamaschi, S., S. Castano, M. Vincini, D. Beneventano: *Retrieving and Integrating Data from Multiple Sources: the MOMIS Approach*. Special Issue on Intelligent Information Integration, Data & Knowledge Engineering, Elsevier Science B.V. (to appear).

[47] H.H. Do, E. Rahm: *COM: A System for Flexible Combination of Schema Matching Approach*. VLDB 2002.

[48] L. Lovasz, M. Plummer. *Matching Theory.*North-Holland, Amsterdam. 1986.

[49] R. Motwani, P. Raghavan. *Randomized Algoritm.* Cambridge University Press, 1995.

Appendice A

Catene di Markov

La costruzione di una catena di Markov richiede due elementi di base: una matrice di transizione e una distribuzione iniziale. Iniziamo con la definizione della matrice di transizione. Assumiamo un insieme limitato S = {1,..,m}di stati. Assegniamo ad ogni coppia di stati (i,j)$\in S^2$ un numero reale p_{ij} tale che le proprietà:

$$p_{ij} \geq 0 \quad \forall (i,j) \in S^2 \qquad (F1)$$
$$\sum_{j \in S} p_{ij} = 1 \quad \forall i \in S \qquad (F2)$$

sono soddisfatte e definiamo la matrice di transizione $\mathbb{P}$ come:

$$\mathbb{P} = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & P_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{pmatrix}$$

Sia $(X_n)_{n \in N_0}$ una sequenza di variabili casuali con valori in S. Qui, *n* indica il tempo a cui lo stato X_n si verifica.

Definizione 1 (Catena di Markov). La sequenza $(X_n)_{n \in N_0}$ è chiamata una *catena omogenea di Markov* con tempo discreto, spazio degli stati S e matrice di transizione $\mathbb{P}$, se per ogni $n \in N_0$ la condizione

$$P[X_{n+1} = j \mid X_0 = i_0,\ldots,X_n = i_n] = P[X_{n+1} = j \mid X_n = i_n] = p_{i_n j} \qquad \text{(F.3)}$$

è soddisfatta, per tutti gli $(i_0,\ldots,i_n,j) \in S^{n+2}$, per cui $P[X_0 = i_0,\ldots,X_n = i_n] > 0$.

La prima identità in (F.3) è chiamata anche "proprietà di Markov ", definisce la "memoria" o "l'ordine" della catena. In questo caso, l'ordine è uguale a uno poiché le probabilità di transizione sono determinate completamente dallo stato precedente. Come vedremo, la limitazione a catene di ordine uno, non è una limitazione grave, dato che i processi con la memoria arbitraria limitata *s* possono essere interpretati come catene di Markov di ordine uno sullo spazio di prodotto S^s. La seconda identità in (F.3) è chiamata "condizione di omogeneità" ed assicura che le probabilità di transizione non varino con il tempo *n*.

Finora, abbiamo solo specificato gli elementi per l'evoluzione delle probabilità per tutto il tempo. Per completare la costruzione di una catena di Markov abbiamo bisogno di specificare una *distribuzione iniziale*. Quindi indichiamo con D_S l'insieme delle distribuzioni discrete su *S*,

$$D_S = \{\mathbf{P} = (P_i)_{i\in S} : P_i \geq 0,\ \sum_{i\in S} P_i = 1\}$$

in cui rappresentiamo le distribuzioni come vettori riga. Chiamiamo $\mathbf{P_0} = (P_{0i})_{i\in S} \in D_S$ la *distribuzione iniziale della catena* $(X_n)_{n\in N_0}$ se $P[X_0=i] = P_{0\,i}$ per tutti gli stati $i \in S$.

Consideriamo ora le cosiddette *probabilità di transizione di ordine n-esimo*

$$p^{(n)}{}_{ij} = P[X_{m+n} = j \mid X_m = i],$$

$n \in N$, le quali sono calcolate sommando su tutti i possibili stati intermedi, cioè, su tutti i percorsi di lunghezza $n+1$, con stato iniziale *i* e stato finale *j*,

$$p_{ij}^{(n)} = \sum_{(i_1,\ldots i_{n-1})\in S^{n-1}} P\left[X_{m+n} = j \middle| X_m = j, X_{m+1} = i_1,\ldots,X_{m+n-1} = i_{n-1}\right] = \sum_{(i_1,\ldots i_{n-1})\in S^{n-1}} p_{ii_1} \cdot \prod_{j=1}^{n-2} p_{i_j i_{j+1}} \cdot p_{i_{n-1} j}$$

dove la matrice $(p_{ij}^{(n)})_{(i,j)\in S^2}$ è uguale alla potenza n-esima $\mathbb{P}^n$ della matrice di transizione $\mathbb{P}$. Definiamo inoltre $p_{ij}^{(0)} = \delta_{ij}$ tale che $\mathbb{P}^0$ è uguale alla matrice identità I_m. Data la distribuzione $\mathbf{P_0}$ di X_0, la distribuzione di X_n è quindi calcolata come $\mathbf{P_0}\mathbb{P}^n$, con $n \in N_0$. Come vedremo, la formula di Perron dimostra di essere uno strumento potente per l'analisi di potenze della matrice $\mathbb{P}$.

La costruzione di uno spazio di probabilità (Ω,A,μ), su cui una data catena $(X_n)_{n\in N_0}$ può essere realizzata, è dato dal Teorema di Esistenza. Possiamo pensare che Ω rappresenti tutti i possibili percorsi $(x_0,x_1,x_2,\ldots)$ in S^{N_0}.

Definizione 2 (Catena Indipendente). Sia $\mathbf{P} = (P_1,\ldots,P_m) \in D_S$ e definiamo una catena di Markov con spazio degli stati S, matrice di transizione $\mathbb{P} = (p_{ij})_{(i,j)\in S^2}$ data da $p_{i\,j} = P_j$, con $i \in S$, e una distribuzione iniziale arbitraria $\mathbf{P_0} \in D_S$. Allora, per tutti gli $n \in N_0$,

$$P[X_{n+1} = j \mid X_0 = i_0,\ldots,X_n = i_n] = P[X_{n+1} = j \mid X_n = i_n] = P_J$$

Chiamiamo questa la "catena indipendente" rispetto a **P**.

Corollario 1. In una catena indipendente, la sequenza di stati $(X_n)_{n\in N_0}$ è una sequenza di variabili casuali indipendenti. Le catene indipendenti non hanno alcuna memoria, quindi vengono anche chiamate catene di Markov di ordine zero.

È abbastanza chiara come quanto detto sopra generalizza il caso di spazio degli stati numerabile S=N. Ricordiamo le seguenti definizioni e lemmi che aiutano a identificare i vari tipi di catene. Essi sono discussi in maniera più dettagliata in ogni libro di testo che tratta le catene di Markov .

Definizione 3 (Ricorrenza e Transitorietà). Consideriamo la probabilità

$$r_{ij} = P\left[\bigcup_{n\in N}\{X_n = j \mid X_0 = i\}\right]$$

di un'eventuale visita allo stato j partendo da i. Uno stato i è chiamato *ricorrente* (o persistente) se $r_{ii}=1$, *transitorio* se $r_{i\,i} < 1$.
Una catena è chiamata ricorrente (transitoria) se tutti gli stati sono ricorrenti (transitori).

Le seguenti condizioni alternative seguono per il Lemma di Borel-Cantelli:

Lemma 2. Uno stato i è ricorrente se $P[X_n = i \text{ infinitamente spesso} \mid X_0 = i] = 1$ o, analogamente, se

$$\sum_{n\in N} p_{ij}^{(n)} = \infty .$$

Al contrario, è transitorio se $P[X_n = i \text{ infinitamente spesso} \mid X_0 = i] = 0$ o, analogamente, se $\sum_{n\in N} p_{ij}^{(n)} < \infty$. Nel caso di uno spazio degli stati S finito, uno stato i è transitorio se e soltanto se è assorbente, cioè, se $p_{i\,i}=1$.

Definizione 4 (Irriducibilità). Un catena di Markov è chiamata *irriducibile* se, per tutte le coppie degli stati $(i,j) \in S^2$, esiste un intero n tale che

$$p^{(n)}{}_{i\,j} > 0.$$

Proposizione 3. Una catena di Markov è irriducibile se e soltanto se per un $k \in N$ arbitrario e una coppia di stati arbitraria $(i,j) \in S^2$ esiste un intero $n = n_{ij} \geq k$ tale che

$$p_{ij}^{(n)} > 0.$$

Una catena di Markov irriducibile non può essere scomposta in parti che non interagiscono. Si può mostrare che tutti gli stati di una catena di Markov irriducibile sono transitori, oppure tutti gli stati sono ricorrenti. In questo caso, la catena è essa stessa o ricorrente o transitoria. Le catene irriducibili *limitate* sono sempre ricorrenti. Nel seguito assumeremo l'irriducibilità, tranne dove indicato diversamente.

Nelle catene irriducibili può ancora esistere una struttura periodica tale che, per ogni stato $i \in S$, l'insieme dei possibili tempi restituiti a *i,* quando si inizia da *i*, è un sottoinsieme dell'insieme $p \cdot N = \{p, 2p, 3p, \ldots\}$, contenere quasi un insieme limitato di questi elementi. Il più piccolo numero p con questa proprietà è chiamato *periodo della catena*

$$p = ged\{n \in N : p_{ii}^{(n)} > 0\}$$

Per $p = 1$, si hanno le seguenti definizioni.

Definizione 5 (Aperiodicità). Una catena irriducibile è detta *aperiodica* (o *aciclica*) se il periodo p è uguale a 1, o analogamente, se per tutte le coppie di stati $(i,j) \in S^2$ esiste un intero n_{ij} tale che, per tutti gli $n \geq n_{ij}$, si ha che la probabilità

$$p_{ij}^{(n)} > 0.$$

Lo spazio degli stati S di una catena aperiodica, irriducibile con periodo $p > 1$ può essere partizionato in p classi reciprocamente disgiunte $\mathfrak{P}_0, \ldots, \mathfrak{P}_{p-1}$ di stati, tali che, se il sistema è nello stato $i \in \mathfrak{P}_l$ al tempo n, sarà nello stato $j \in \mathfrak{P}_k$ al tempo $n+1$, dove $k \equiv l+1 \pmod{p}$.

Il seguente lemma dà una classe di catene irriducibili e aperiodiche:
Lemma 4. Se $\mathbf{P} = (P_1, \ldots, P_m)$ è strettamente positivo, $P_i > 0$, $i \in S$, allora la catena indipendente rispetto a **P** è irriducibile e aperiodica.

Definizione 6 (Distribuzione Stazionaria). Una distribuzione $\mathbf{P} \in D_S$ che soddisfa la condizione

$$\sum_{i \in S} P_i p_{ij} = P_j$$

per tutti gli stati $j \in S$, oppure $\mathbf{P}\,\mathbb{P} = \mathbf{P}$ in notazione matriciale, è detta una *distribuzione stazionaria*.

Per induzione, prendiamo subito $\mathbf{P}\,\mathbb{P}^n = \mathbf{P}$ per tutti gli $n \in N$, così da avere i seguenti lemmi:
Lemma 5. Se la distribuzione iniziale di una catena è una distribuzione stazionaria, allora il processo $(X_n)_{n \in N_0}$ di stati è un processo stazionario, cioè ogni X_n ha la stessa distribuzione **P**.

Lemma 6. Per una catena indipendente rispetto a **P**, **P** è una distribuzione stazionaria.

Per analizzare, in seguito, distribuzioni stabili, poniamo:

$$T_i = \begin{cases} \infty & : if\ X_n \neq i \forall n \in N \\ \min\{n \in N : X_n = i\} & : altrimenti \end{cases}$$

l'istante della prima visita della catena nello stato *i* e stia $E_i[T_i]$ ad indicate l'aspettativa condizionale su $X_0 = i$, cioè, il tempo di ritorno previsto per lo stato *i*.

Definizione7 (Ricorrenza Positiva). Uno stato i∈S è detto *ricorrente positivo* (ricorrente zero) se $E_i[T_i] < \infty$ ($E_i[T_i] = \infty$).
Una catena ricorrente è chiamata positiva ricorrente, se tutti gli stati sono positivi ricorrenti.

Notate che non contiamo sulla irriducibilità nella precedente definizione di una catena ricorrente positiva.

Lemma 7 (Esistenza e unicità di distribuzioni stabili). La ricorrenza positiva di una catena (S, $\mathbb{P}$) garantisce l'esistenza di una distribuzione stabile **P**. Se, in più, (S, $\mathbb{P}$) è irriducibile allora la distribuzione stabile è unica e può essere scritta nella forma **P** = (P_1, P_2,…) con

$$P_i = \frac{1}{E_i[T_i]}, \qquad i \in S.$$

In particolare, ogni catena *limitata* è positiva ricorrente e ha così una distribuzione stabile. Per catene irriducibili, *aperiodiche* e ricorrenti positive, si ha il seguente risultato limitante:
Lemma 8. Sia (S, $\mathbb{P}$)irriducibile, ricorrente positivo aperiodico, con distribuzione stabile **P**. Allora **P** = (P_1,P_2,…) è dato da

$$P_j = \lim_{n\to\infty} p_{ij}^{(n)}$$

per tutte le coppie di stati (i,j)∈S^2.

In questo caso, $\mathbb{P}^n$ converge alla matrice di transizione della catena indipendente rispetto alla distribuzione stabile **P**. Per il corrispondente risultato l_1 sulla sequenza delle distribuzioni $\mathbf{P_0}\mathbb{P}^n$, con $\mathbf{P_0} \in D_S$ una distribuzione iniziale arbitraria. Se S è limitato inoltre, abbiamo il seguente lemma sulla velocità di convergenza:
Lemma 9. Sia (S, $\mathbb{P}$) una catena di Markov aperiodica, irriducibile con distribuzione stazionaria **P**. Allora esistono $c \geq 0$ e $0 \leq \rho \leq 1$, tali che:

$$\left| p_{ij}^{(n)} - P_j \right| \leq c\rho^n, \quad \forall (i, j) \in S^2, \forall n \in N.$$

Qualsiasi distribuzione iniziale arbitraria converge esponenzialmente a **P** in questo caso. Questa classe di catene permette una particolare forma semplice del Teorema del limite centrale per il numero di visite in ogni stato.