SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Seminari di Ingegneria del Sofware

IMDB
The Internet Movie Database DL-Lite$_A$ Ontology

Junio Valerio Franchi

Valerio Del Grande

Indice

1. Introduzione

1.1 Premessa

Il progetto da sviluppare consiste nella scrittura di un'ontologia in DL-LiteA riguardante IMDB (The Internet Movie Database – www.imdb.com) e un opportuno mapping dei dati disponibili sotto forma di DB relazionale.

Tale ontologia sarà poi implementata in Mastro-I, dopo un'opportuna traduzione in formato XML secondo le DTDs del sistema, e interrogata con UCQ (Union of Conjunctive Query) e query EQL (Epistemic Query Language) tramite lo strumento QuOntoEQL.

Parallelamente si procederà alla scrittura della stessa ontologia utilizzando il software Protègè (protege.stanford.edu), strumento open-source che fornisce un editor grafico per la descrizione di ontologie in OWL.

Mediante il plugin OBDA (sviluppato da UNIBZ) Protégé è in grado di tradurre l'ontologia in DL-LiteA e passarla al sistema Mastro e al reasoner QuOnto.

Saranno quindi testate tramite il tool Protégé OBDA le stesse query effettuate in precedenza sul sistema Mastro.

Il seguente elaborato è organizzato in due parti:

la prima parte è di carattere teorico e riguarda: sistemi OBDA, Ontologie, Description Logic DL-LiteA, tecniche di Mapping su dati relazionali, tecnica di Query Answering in sistemi con mapping, tecniche di querying basate sulla chiusura dinamica della conoscenza (EQL);

la seconda parte è invece di carattere implementativo e descrive il lavoro pratico realizzato concentrandosi su alcuni problemi riscontrati e mostrando i risultati ottenuti.

In appendice sono riportati per esteso l'ontologia di IMDB espressa in DL-LiteA (notazione logica) e i mapping sul database utilizzato per i test.

1.2 Ontology-Based Data Access

Lo scopo del data integration è permettere l'accesso a sorgenti dati eterogenee e preesistenti attraverso un punto di accesso unificato che nasconda il data layer sottostante, permettendo agli utenti del sistema di accedere ai servizi offerti (e.g. query answering) senza conoscere la reale struttura dei dati, nè come e dove sono memorizzati.

Per realizzare tali obiettivi è necessario prima di tutto fornire ai client del sistema una descrizione del dominio di interesse ovvero uno schema concettuale globale espresso ad un alto livello di astrazione rispetto ai dati.
La ricerca ha individuato le Ontologie come miglior metodo formale per la descrizione del dominio.
L'approccio seguito sarà quindi denominato Ontology-Based Data Access (OBDA)

Nel seguito sarà presentato un sistema (la cui implementazione è chiamata MASTRO-I) che propone una soluzione completa al problema del data integration.

Il sistema è composto da tre elementi principali <G, S, M>:

1. G: un Global Schema rappresentato mediante una TBox in DL-LiteA, una description logic particolarmente adatta a descrivere ontologie
2. S: le sorgenti dati gestite da un DBMS relazionale (o più di uno)
3. M: un linguaggio di mapping GAV (global-as-view) che mette in relazione le sorgenti dati con il global schema risolvendo il cosiddetto problema dell'*impedence mismatch*, ovvero il fatto che le sorgenti memorizzano valori mentre le istanze dei concetti dell'ontologia sono oggetti. Caratteristica molto importante di tale sistema è che, grazie alla particolare classe di description logic utilizzata, il problema del query answering risulta computazionalmente trattabile, in particolare LOGSPACE rispetto alla dimensione dei dati, e la risposta alle query poste sull'ontologia è delegabile, dopo un opportuna traduzione in SQL, ai DBMS relazionali che, ad oggi, continuano ad essere i sistemi più efficienti per memorizzare ed interrogare quantità elevate di dati.

Un sistema OBDA ha lo scopo di permettere di creare un servizio (query answering nel nostro caso) sopra un insieme di sorgenti dati preesistenti, fornendo agli utenti un unico punto di accesso che rappresenta una vista concettuale del dominio di interesse. Tale descrizione è indipendente dalle sorgenti dati, che esistono autonomamente.

Requisiti fondamentali di un sistema OBDA sono:

1. Il Global Schema fornito ai client del sistema informativo è descritto mediante un ontologia scritta in un linguaggio che permetta un compromesso accettabile tra potere espressivo e complessità computazionale.

2. Poichè la quantità di dati memorizzati nelle sorgenti può essere molto elevata e allo stato attuale l'unica tecnologia che permette di accedervi in modo efficiente sono i DBMS relazionali, è necessario che il formalismo adottato permetta di affidare il query answering ai DBMS del data layer.

3. Poiché le sorgenti dati in generale esistono a priori e non sono state progettate per memorizzare istanze dello schema concettuale desiderato, abbiamo bisogno di un meccanismo che permetta di mettere in corrispondenza i dati nelle sorgenti con gli elementi dell'ontologia.

4. Tale meccanismo deve essere in grado di affrontare il noto problema dell'impedence mismatch: i database memorizzano dati mentre le istanze dei concetti di un ontologia sono oggetti, ciascuno dei quali è identificato da un object identifier

5. I client devono poter esprimere le queries sullo schema concettuale (l'ontologia) senza dover conoscere le sorgenti dati. Il sistema dovrà ragionare sull'ontologia e sui mapping e tradurre la query in input in opportune queries sulle sorgenti in modo del tutto trasparente all'utente.

Sarà analizzato in seguito un importante problema riguardante il query answering sulle ontologie ovvero la valutazione della stessa query sotto due assunzioni diverse: Closed World Assumption (CWA) adottata dai DBMS relazionali e Open World Assumption (OWA) adottata invece dalle ontologie nelle quali l'informazione è incompleta.

2. Ontologie

2.1 Introduzione alle Ontologie

Un ontologia è una concettualizzazione del dominio di interesse di un sistema informativo espressa in un qualche linguaggio formale.

Gli schemi concettuali (UML, ER) sono particolari tipi di ontologie detti *Graph-based* pensati per rappresentare un singolo modello logico (es. Basi di dati)

Esistono diversi linguaggi formali che permettono di descrivere un ontologia ma il nostro obiettivo è trovare un compromesso accettabile tra potere espressivo e complessità computazionale.

Dopo un breve confronto tra alcuni linguaggi verrà evidenziato il motivo che ha portato a scegliere DL-LiteA per descrivere le ontologie nel nostro sistema.

Un ontologia è composta da un livello intensionale e un livello estensionale.

Gli elementi del livello intensionale sono:

Concept: elemento dell'ontologia che rappresenta una collezione di istanze (es. Persona)

Property (attribute): elemento dell'ontologia che qualifica un altro elemento (concept o relationship)

Relationship (role): elemento dell'ontologia che esprime un associazione tra concepts

Axiom (assertion): esprime a livello intensionale una condizione che deve essere soddisfatta a livello estensionale

Al livello estensionale:

Instance: rappresenta un oggetto che è estensione di un Concept

Fact: rappresenta una relazione presente tra due istanze

Ogni linguaggio ontologico deve includere anche uno strumento per esprimere queries.

Query: espressione sul livello intensionale che denota una collezione di individui che soddisfano una data condizione.

2.2 First Order Logic

La logica del primo ordine (FOL) permette di esprimere conoscenze molto articolate ed eseguire complessi ragionamenti ma, proprio a causa del suo elevato potere espressivo, è indecidibile.

Si deve quindi ricercare un linguaggio che sia decidibile e che permetta un buon compromesso tra potere espressivo e complessità computazionale.

Nel seguito si presenta una particolare classe di logiche particolarmente adatte ad assere utilizzate come formalismo per le ontolgie.

2.3 Ontology Web Language (OWL)

Il linguaggio OWL è lo standard del World Wide Web Consortium come linguaggio per il Semantic Web, in tre varianti:

- OWL Lite è decidibile, semplice ma scarsamente espressivo, è una semplificazione sintattica di OWL DL (exptime)

- OWL DL ha lo stesso potere espressivo delle Description Logic, è decidibile e abbastanza espressivo (nexptime)

- OWL Full è oltre la First Order Logic, ad esempio permette l'enumerazione; è molto espressivo ma indecidibile

2.4 Description Logics (Dls)

Le Description Logic sono particolari logiche create appositamente per descrivere e ragionare su conoscenza strutturata (structured knowledge).

Il dominio di interesse è composto da oggetti strutturati in *Concepts* che denotano insiemi di oggetti e *Roles* che denotano relazioni binarie tra istanze di oggetti. La conoscenza è asserita per mezzo di *Assertions.*

Un'ontologia in DL è caratterizzata da:

1. Una TBox che rappresenta il livello intensionale
2. Una Abox che rappresenta il livello estensionale

Lo studio sul compromesso tra potere espressivo e complessità computazione del reasoning sull'ontologia ha mostrato che OWL, senza opportune restrizioni, non è adatto per rappresentare ontologie con una grande quantità di instanze nella Abox poiché risulta essere esponenziale rispetto alla dimensione della ABox.

Un particolare frammento di OWL di interesse pratico per l'OBDA è la famiglia DL-Lite che risulta avere complessità LOGSPACE nella dimensione dei dati.

Ancora più importante DL-Lite permette di delegare il query answering, dopo un opportuna fase di riscrittura, ai DBMS relazionali del data layer.

Esistono due linguaggi della famiglia DL-Lite che sono particolarmente interessanti perché posseggono le proprietà sopra descritte: DL-LiteF e DL-LiteR.

DL-LireF è adatto per specificare le caratteristiche principali dei modelli concettuali: asserzioni cicliche, ISA di concetti, tipizzazione di role, vincoli di partecipazione obbligatoria, restrizioni di funzionalità sui roles.

DL-LiteR ha in più la capacità di esprimere la disgiunzione tra concetti e ruoli, ISA tra role.

In sostanza questi due linguaggi catturano praticamente tutti i costrutti di UML ed ER con alcune eccezioni, ad esempio: non è possibile esprimere la completezza delle gerarchie e vincoli di cardinalità massima e minima diversi da 1 e *.

Il linguaggio ottenuto dall'unione senza alcuna restrizione dei due precedenti (DL-LiteFR) è molto interessante per il suo potere espressivo ma purtroppo perde l'interesse pratico poiché non è più LOGSPACE rispetto alla dimensione dei dati.

Nel seguito sarà quindi presentato un ulteriore linguaggio chiamato DL-LiteA che deriva da DL-LiteFR con opportune restrizioni che lo rendono nuovamente LOGSPACE, continuando a permettere la delega del query answering al data layer.

Una KB in DL-LiteA è una coppia <T, A> dove A è una DL-LiteFR ABox e T è una DL-LiteFR TBox che soddisfa le seguenti condizioni:

1. Per ogni atomic role (o atomic role inverso) Q che appare in un asserzione $\exists Q.C$ dove C è un Concept, in T non possono apparire le asserzioni (funct Q) e (funct Q-)
2. Per ogni role inclusion assertion $Q \sqsubseteq R$, in T non possono apparire le asserzioni (funct R) e (funct R-)

3. Per ogni Concept Attribute Inclusion Assertion Uc ⊑ Vc, l'asserzione (funct Vc) non è in T
4. Per ogni Role Attribute Inclusion Assertion Ur ⊑ Vr, l'asserzione (funct Vr) non è in T.

2.5 DL-LiteA

DL-LiteA è una description logic che risulta particolarmente adatta a modellare ontologie in quanto prende in seria considerazione la distinzione tra oggetti e valori, distinguendo:

- Concepts da Value-domains: un concept è una astrazione di un insieme di oggetti mentre un value-domain ddenota un insieme di valori (data values)
- Attributes da Roles: un ruolo denota una relazione binaria tra oggetti, un concept attribute denota una relazione binaria tra oggetto e valore.

Da notare che in DL-LiteA è possibile specificare Role-Attributes, cosa non possibile in OWL-DL. Un Role-Attribute è una relazione ternaria tra una coppia di oggetti e un valore.

A causa di questa limitazione di OWL-DL, come si vedrà in seguito al capitolo 6, in Protégé non è infatti possibile modellare i role-attribute.

Le espressioni DL-LiteA sono costruite a partire da un alfabeto. Un alfabeto comprende simboli per Atomic Concept, Value domains, Atomic Roles, Atomic Attributes e costanti.

In seguito si denota con Γ l'afabeto delle costanti, partizionato in due insiemi: Γ_V l'insieme dei simboli di costante per i valori e Γ_O l'insieme di simboli di costante per gli oggetti.

Usiamo la seguente notazione:

– A denota un atomic concept, B un basic concept, C un general concept, and $\top_C$ denota l'universal concept.

– E denota un basic value-domain, i.e. il range di un attribute, F una value-domain expression, and $\top_D$ l'universal value-domain.

– P denota un atomic role, Q un basic role, e R un general role.

– U_C denota un atomic attribute (o semplicemente attribute), e V_C un general attribute.

Si dice dominio di U_C , denotato con $\delta(U_C)$, l'insieme di oggetti che U_C mette in relazione con valori. Il dominio di U_C è quindi un concept.

Si dice range di U_C, denotato con $\rho(U_C)$, l'insieme di valori che U_C mette in relazione con oggetti. Il range di U_C è quindi un value-domain.

Grammatica DL-LiteA

1. Concept expressions:

$$B ::= A \mid \exists Q \mid \delta(U_C)$$

$$C ::= \top_C \mid B \mid \neg B \mid \exists Q.C$$

2. Value-domain expressions:

$$E ::= \rho(U_C)$$

$$F ::= \top_D \mid T1 \mid \cdots \mid Tn$$

3. Role expressions:

$$Q ::= P \mid P^-$$

$$R ::= Q \mid \neg Q$$

4. Attribute expressions:

$$VC ::= U_C \mid \neg U_C$$

Il significato di ogni espressione in DL-LiteA è definito dalla sua semantica che viene specifica da interpretazioni di termini in logica del primo ordine, di cui le DL's sono un frammento.

Ogni value-domain Ti è interpretato come l'insieme val(Ti) di valori dei corrispondenti RDF data type. Ogni costante ci è interpretata come uno specifico valore $val(ci) \in val(Ti)$.

Se $i \neq j$ allora $val(Ti) \cap val(ci) = \varnothing$ cioè DL-LiteA adotta la *Unique Name Assunption*.

2.6 Ontologie in DL-LiteA

Un ontologia in DL-Lite è data dalla coppia <T,A> e rappresenta il dominio di interesse in termini di due componenti:

- T: Tbox che rappresenta il livello intensionale della base di conoscenza
- A: Abox che rapprense il livello estensionale della base di conoscenza

La Tbox è formata da un insieme di asserzioni chiamate *intensional assertions* della seguente forma:

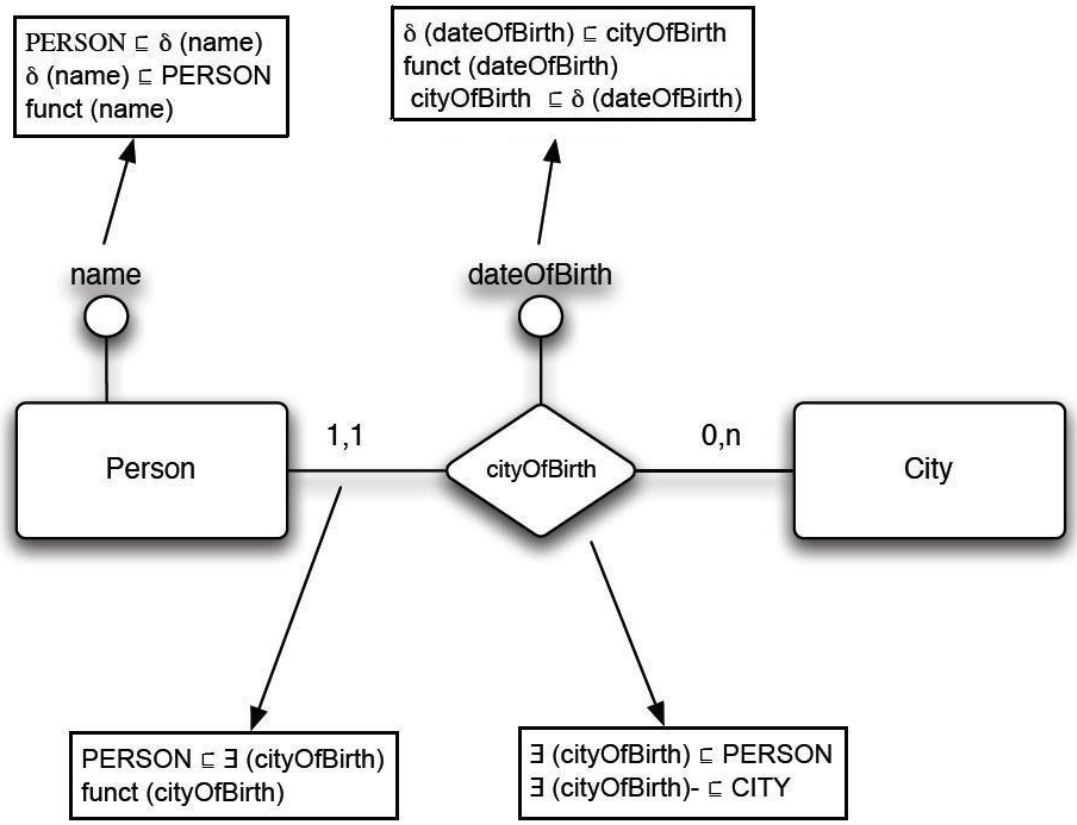
$B \sqsubseteq C$	(concept inclusion assertion)
$Q \sqsubseteq R$	(role inclusion assertion)
$E \sqsubseteq F$	(value-domain inclusion assertion)
$U_C \sqsubseteq V_C$	(attribute inclusion assertion)
(funct Q)	(role functionality assertion)
(funct U_C)	(attribute functionality assertion)

Una DL-LiteA TBox è un insieme finito T di intensional assertions che soddisfano la condizione che ogni identifying property in T è primitiva in T.

Un atomic attribute Uc (o basic role Q) è detto *indentifying property* se T contiene l'asserzione (funct Uc) (o rispettivamente (funct Q)).
Un atomic attribute Uc (o basic role Q) è detto *primitivo* se non appare positivamente nella parte destra di una inclusion assertions del tipo $Y \sqsubseteq Uc$ (o rispettivamente $Y \sqsubseteq Q$) in T, e non appare in una espressione della forma $\exists Q.C$ in T.

In altre parole questo significa che in DL-LiteA non è possibile specializzare le identifying properties.

Sotto si riporta un esempio di Tbox DL-LiteA confrontato con un diagramma ER:



Una Abox in DL-LiteA è un insieme di asserzioni chiamate membership assertions della forma:

A(a), P(a, b), Uc(a, b) dove a e b sono costanti nell'alfabeto Γ.

2.7 Mapping di dati relazionali

Per mappare dati relazionali su un ontologia bisogna ricordare che mentre le sorgenti dati memorizzano valori, le istanze dei Concept sono oggetti, ognuno denotato da un identificatore (*impedence mismatch problem*).

Nel seguito si manterranno sempre separati i valori delle sorgenti dagli identificatori, in particolare sceglieremo come object identifier termini della logica costruiti a partire dai valori.

Un ontologia DL-LiteA con mapping è caratterizzata da una tripla Om = <T, M, DB> tale che:

- T è una TBox DL-LiteA
- DB è un database relazionale
- M è un insieme di *mapping assertions*, partizionato in due insiemi Mt e Ma dove:
  - Mt è un insieme di *typing mapping assertions* della forma $\Phi \rightarrow$ Ti , con Φ query di arità 1 sul DB (proiezione su una colonna) e Ti data type DL-LiteA
  - Ma è un insieme di *data-to-object mapping assertions* della forma $\Phi \rightarrow \Psi$ dove Φ è una query SQL arbitraria sul DB e Ψ è una conjunctive query su T che coinvolge variable terms. Un variable term è del tipo f(z) con f simbolo di funzione di arità n e z ennupla di variabili.

Ricordando che i DB seguono CWA mentre le ontologie seguono OWA è interessante notare che la mapping assertion $\Phi \rightarrow \Psi$ rispetta pienamente questa distinzione, infatti:

- Φ è una query sql che viene valutata direttamente sulla sorgente quindi in CWA
- L'implicazione materiale dell'asserzione impone che tutti i modelli di Φ sono modelli di Ψ ma non dice nulla sul viceversa. Quindi mentre tutte le tuple di Φ soddisfano Ψ, ci potrebbero essere anche altre tuple che soddisfano Ψ (OWA).

3. Reasoning su ontologie

3.1 Assunzioni CWA e OWA

Una query può essere valutata sotto due diverse assunzioni:

- CWA: Closed World Assumption, adottata dai DBMS relazionali
- OWA: Open World Assumption, adottata invece dai sistemi di rappresentazione della conoscenza (ontologie)

Nell'assunzione CWA si assume di avere una conoscenza completa sul mondo, ovvero i dati sono completamente specificati. Nelle interrogazioni si considera vero quello che è presente nella base di dati e falso tutto il resto.
Nell'assunzione OWA invece si assume di avere a che fare con informazione incompleta. Nelle interrogazioni quindi si considera vero tutto ciò che è asserito mentre su tutto il resto non si possono fare ipotesi, cioè potrebbe essere sia vero sia falso.

Vediamo un semplice esempio:

PERSON

Name
Claudio
Ilaria
Giuseppe

STUDENT

Name
Valerio
Ilaria

q(x): Person(x) $\wedge \neg$ Student(x)

Il risultato di questa query è diverso a seconda dell'assunzione
In CWA: {Claudio, Giuseppe}
In OWA: {}

3.2 Query su ontologie DL-LiteA

La FOL (First Order Logic) permette di eseguire complessi ragionamenti e ha un potere espressivo molto elevato. Sotto l'assunzione CWA la FOL risulta decidibile infatti SQL ha lo stesso potere espressivo. Nell'assunzione OWA invece la FOL è indecidibile, poiché il risultato della query dipende dal modello, ovvero può esistere un modello in cui la query è vera e uno in cui è falsa.

Per ragionare sulle ontologie è quindi necessario considerare un sottoinsieme della FOL che risulti decidibile: si è individuato tale sottoinsieme nelle query congiuntive (CQ, Conjunctive Query) e unione di esse (UCQ, Union of Conjunctive Query).

Una conjunctive query è formula del tipo:

$q(\mathbf{x}) \leftarrow conj(\mathbf{x}, \mathbf{y})$ dove:

- q(x) è l'head della query, conj(x,y) è il body
- l'arità di q coincide con l'arità di x
- x è una tupla di variabili non quantificate (libere)
- y è una tupla di variabili quantificate esistenzialmente
- conj è una congiunzione di atomi (and)

La mancanza dell'operatore NOT rende molto limitato il potere espressivo e l'effettiva utilità pratica delle Conjunctive Query.
Questa limitazione ha reso necessaria l'introduzione dei linguaggi di query epistemici, basati sulla chiusura dinamica della conoscenza (EQL, Epistemic Query Language) che verranno descritti nel seguito di questo documento (Capitolo 4).

Una Conjunctive Query q(x) su una ontologia DL-LiteA O = <T, A> è una Conjunctive Query come sopra descritta dove il body conj(**x**,**y**) è una congiunzione di atomi della forma:
A(x), P(x,y), D(x), Uc(x,y), x=y dove A,P,D e Uc sono rispettivamente un Atomic Concept, un Atomic Role, una Atomic Value Domain e un Atomic Attribute e (x,y) sono variabili in **x** o **y** o costanti in Γ .

Data un ontologia O= <T, A>, un interpretazione I per O e una CQ q(x) si dice *answer* di q(x) su I, denotata con qI l'insieme delle tuple t di costanti tale che la valutazione del body della query è vera nell'interpretazione I.

Rispondere a una query q su una ontologia O significa restituire solo le *certain answers* cioè le tuple t tali che per ogni modello I di O abbiamo che $t^I \in q^I$.

3.3 Reasoning su ontologie DL-LiteA

Assumiamo che l'ABox dell'ontologia sia rappresentata da un database relazionale.
Sia O = <T,A> un'ontologia scritta in DL-Lite$_A$, allora possiamo rappresentare A in termini di database relazionale db(A), così definito:

- db(A) contiene una relazione unaria T_A per ogni atomic concept A che appare in T.
  La tupla t è in T_A se e solo se l'asserzione A(t) è in A.

- db(A) contiene una relazione binaria T_P per ogni atomic role P che appare in T.
  La tupla t è in T_P se e solo se l'asserzione P(t) è in A.

- db(A) contiene una relazione binaria T_U per ogni atomic concept attribute U che appare in T.
  La tupla t è in T_U se e solo se l'asserzione U(t) è in A.

Il reasoning su un'ontologia in DL-LiteA può essere ridotto alla valutazione di particolari queries su db(A).
Dal momento che tali query possono essere espresse in SQL e valutate con un DBMS relazionale tradizionale, segue che il reasoning su un ontologia in DL-LiteA ha complessità LOGSPACE.

3.3.1 Soddisfacibilità: *ViolatesDB()*

La soddisfacibilità, di un'ontologia può essere ridotto alla valutazione di un'opportuna query, chiamata *Violates(T)*.

Violates(T) è una query al prim'ordine che si domanda per ogni costante in A se sono violati:

- vincoli espliciti corrispondenti ad asserzioni di funzionalità e asserzioni disgiunzione in T

- vincoli impliciti, che seguono dalla semantica di T, che impongono che ogni Concept è in disgiunzione con ogni dominio e che, per ogni coppia T_i, T_j di rdfDataType, T_i e T_j sono disgiunti.

Chiamiamo *ViolatesDB(T)* la funzione che trasforma la query *Violates(T)* sostituendo ogni predicato X in *Violates(T)* con T_X.
Si dimostra che la nuova query *ViolatesDB(T)* è equivalente all'originale. Essendo espressa su db(A) può essere riscritta in SQL.

3.3.2 Query Answering: *PerfectRef(Q,T)*

Il query answering, cioè il problema di calcolare le certain answers su un'ontologia O soddisfacibile, può essere ridotto alla valutazione di un opportuna Union of Conjunctive Query su db(A) attraverso un algoritmo chiamato *perfect reformulation*.

La funzione *PerfectRef(Q,T)* prende in input una UCQ Q su O_m e una TBox T, e riformula Q in nuova query Q' che è ancora una UCQ e ha la seguente proprietà:
certain answer di Q' rispetto a <∅,M,DB> coincide con certain answer di Q rispetto a <T,M,DB>.
In pratica, attraverso questa fase di riscrittura, le inclusion assertions in T rilevanti per la computazione viene compilata dentro la nuova query.

Denotiamo con *PerfectRefDB(Q,T)* la funzione che trasforma la query *PerfectRef(Q,T)* cambiando ogni predicato X in *PerfectRef(Q,T)* in T_X.
Si dimostra che la nuova query *PerfectRefDB(Q,T)* è equivalente all'originale. Essendo espressa su db(A) può essere scritta in SQL.

3.3 Reasoning su ontologie DL-LiteA con mapping

Esistono due diversi approcci per interrogare un'ontologia in DL-Lite$_A$ con mappings. Il primo è quello di utilizzare questi ultimi per produrre l'ABox e quindi interrogare l'ontologia tramite la TBox creata in precedenza e l'ABox appena creata; questo approccio, di tipo *bottom-up* richiede la stesura dell' ABox a partire dalle sorgenti dei dati, ma così i dati presenti nelle sorgenti vengono duplicati.
In seguito viene applicato l'algoritmo di query answering all'ontologia O=<T, A(M,DB)>.
L'approccio bottom-up crea due problemi:

1) Il procedimento di creazione dell'ABox è PTIME nella dimesione del database
2) Il database e l'ABox diventano indipendenti perciò è necessario un meccanismo per tenerli sincronizzati nel tempo

Si preferisce quindi adottare un secondo approccio, di tipo *top-down*, che utilizza una ABox virtuale, evitando la duplicazione dei dati, i problemi di sincronizzazione e mantenendo la complessità del reasoning LOGSPACE.
Per spiegare in dettaglio questo approccio dobbiamo prima fornire alcune definizioni riguardo la ABox virtuale e la cosiddetta "split version" di un'ontologia.

3.3.1 Splitting mapping

Data un'ontologia O_m= <T,M,DB> in DL-Lite$_A$ costituita da una TBox, un insieme di mapping e una sorgente dati, si dice versione split di O_m, denotata come Split(O_m)=<T,M',DB>, una nuova ontologia ottenuta da O_m costruendo un nuovo insieme di mappings M' tale che:

- ogni typing assertion di M sarà presente anche in M'
- per ogni mapping assertion $\Phi \rightarrow \Psi$ in M e per ogni atomo $X \in \Psi$, M' conterrà il mapping assertion $\Phi' \rightarrow \Psi$ dove Φ' è la proiezione di Φ sulle variabili che occorrono in X.

Si dimostra che ogni ontologia O_m con mapping è logicamente equivalente alla corrispondente split version Split(O_m).

Inoltre il processo di splitting è PTIME nella dimensione dei mapping e non dipende dalla dimensione dei dati.

3.3.2 Virtual ABox

Data un'ontologia O_m= <T, M, DB> in DL-Lite$_{A,}$ la ABox virtuale corrispondente è una ABox dove le asserzioni sono computate applicando i mapping assertions ai dati presenti nel DB, senza memorizzare fisicamente l'ABox (on-the-fly durante il reasoning).

Data un'ontologia O_m= <T, M, DB> in DL-Lite$_A$ con mapping, e una mapping assertion $m \in M$ della forma $\Phi \rightarrow \Psi$, si dice ABox virtuale generata da m partire dalle sorgenti dati (DB) il seguente insieme di asserzioni:

$$A(m, DB) = \{X[x/v] \mid v \in ans(\Phi, DB)\}$$

dove v e Φ sono di arità n e X[x/v] denota il ground atom ottenuto da X(x) sostituendo la n-upla di variabili x con la n-upla di costanti v.

La Abox virtuale di O_m, denotata da A(M, DB) è un insieme di asserzioni del tipo:

$$A(M, DB) = \{A(m,DB) \mid m \in M\}$$

Si dimostra che ogni ontologia O_m= <T, M, DB> è logicamente equivalente alla corrispondente versione con ABox virtuale O_m= <T, A(M, DB)>.

3.3.3 Unfolding

La funzione UnfoldDB(O_m, Q) prende in input un ontologia O_m= <T, M, DB> e una UCQ Q su O_m e restituisce:

- un insieme di query SQL sul DB
- le sostituzioni da applicare al risultato per ottenere la risposta a Q

3.3.4 Soddisfacibilità di Ontologie DL-LiteA con mapping

Algorithm ***Sat(O_m)***

Input: DL-Lite$_A$ ontology with mappings O_m = <T ,M,DB>
Output: true or false
Q^s ←Violates(T);
$S^{'}$ ←UnfoldDB(Q^s,O_m);
Q' ← false;
for each ansθ ← q' ∈ S' **do**
    Q' ← Q' ∪ {q'} ;
return not(ans(Q',DB))

L'algoritmo chiama Violates(T) che restituisce un insieme di query che chiedono se sono violate le asserzioni di funzionalità e disgiunzione in T.
Successivamente UnfoldDB traduce ognuna di queste query in SQL prendendo in considerazione i mapping. Quindi Sat(O_m) restituisce true se e solo se la valutazione di tali query è false.

Data O_m= <T, M, DB> si dimostra che Sat(O_m) termina e l'ontologia è soddisfacibile se e solo se l'algoritmo restituisce true.
La terminazione dell'algoritmo segue dalla terminazione della funzione UnfoldDB.

3.3.5 Query Answering su Ontologie DL-LiteA con mapping

Algorithm ***Answer(Q, O_m)***

Input: DL-Lite$_A$ ontology with mappings O_m = <T ,M,DB>, UCQ Q over O_m
Output: set of tuples R^s
if O_m is not satisfiable
then return AllTup(Q,O_m)
else
$Q^p \leftarrow \cup_{qi \in Q}$ PerfectRef(q_i,T);
S' ←UnfoldDB(Q^p,O_m)
$R^s \leftarrow \emptyset$;
for each ansθ ← q' ∈ S' **do**
$R^s \leftarrow R^s \cup$ ans(q',DB)θ ;
return R^s

L'algoritmo *Answer(Q, O_m)* prende in input un'ontologia O_m in DL-Lite$_A$ con mappings e una UCQ Q su questa ontologia. Se l'ontologia non è soddisfacibile, allora restituisce l'insieme di tutte le possibili tuple di elementi $\Gamma_0 \cup \Gamma_V$ denotato come *AllTup(Q,O_m)* con la stessa arità della query Q.
Se invece l'ontologia è soddisfacibile, l'algoritmo computa la *Perfect Reformulation* Q^p di Q, poi l'*Unfolding* S' di Q^p. Per ogni query q' presente nell'insieme S' restituito da UnfoldDB, estrae la conjunctive query presente nel body, la valuta sul DB e processa la risposta in accordo con le sostituzioni che occorrono nell'head.

Teorema: Sia O_m =<T,M,DB> un'ontologia in DL-Lite$_A$ con mappings, e Q una UCQ su O_m.
Allora l'algoritmo Answer(Q, O_m) termina. Inoltre, sia R^s l'insieme delle tuple restituite da Answer(Q, O_m), e sia t una tupla di elementi in $\Gamma_0 \cup \Gamma_V$. Allora t ∈ ans(Q, O_m) se e solo se t ∈ R^s.

La terminazione dell'algoritmo segue dalla terminazione di PerfectRef e dalla funzione UnfoldDB.

L'algritmo Answer ricostruisce il risultato a partire dai risultati ottenuti dalla valutazione delle query SQL sul DB. Quello che accade in pratica è che la fase di ricostruzione viene delegata al motore SQL. L'implementazione pratica dell'algoritmo sostituisce il loop **for each** con un passo che, a partire dall'insieme S' generato nella fase di Unfold, crea una query SQL che, una volta valutata sulle sorgenti, computa direttamente la risposta alla query originale Q.

3.4 Considerazioni sulla complessità computazionale

Le mapping assertions sono della forma $\Phi \rightarrow \Psi$ dove Φ è una query SQL sul database relazionale sottostante. Si assume che tale query appartenga alla First Order Logic (ad esempio non si potranno utilizzare costrutti come groupby) e in quanto tale è LOGSPACE rispetto alla dimensione dei dati nel DB.

Sia O_m =<T, M, DB> un ontologia in DL-Lite$_A$ con mapping e Q una UCQ su O_m.
La funzione UnfoldDB(Q, O_m) impiega tempo essponenziale rispetto alla dimesione di Q e polinomiale rispetto alla dimensione di M

Teorema: Data un'ontologia O_m =<T, M, DB> in DL-Lite$_A$ con mappings, l'algoritmo *Sat(O_m)* è LOGSPACE nella dimensione del DB (data complexity) e viene eseguito in tempo polinomiale nella dimensione di M e in tempo polinomiale nella dimensione di T.

Proof:

- Violates(T) restituisce una UCQ su db(A(M, DB)) la cui dimensione è polinomiale nella dimensione di T
- Ogni query Q contiene due atomi e l'applicazione di UnfoldDB ad ogni query è polinomiale nella dimensione di M e costante nella dimensione dei dati.
- La valutazione di un'unione di queries SQL su un database è LOGSPACE rispetto alla dimensione del database.

Teorema: Data un'ontologia O_m =<T,M,DB> in DL-Lite$_A$ con mappings e una UCQ Q su tale ontologia, l'algortitmo *Answer(Q, O_m)* è LOGSPACE nella dimensione dei dati (data complexity), PTIME nella dimensione di M, EXPTIME nella dimensione di Q e PTIME nella dimensione di T.

Proof:

1. il massimo numero di atomi nel body di una CQ generata da PerfectRef è uguale alla lunghezza della query iniziale Q.
2. PerfectRef(Q,T) viene eseguito in tempo polinomiale nella dimensione di T.
3. l'applicazione di UnfoldDB su ogni query generata da PerfectRef ha costo esponenziale nella dimensione della query e polinomiale nella dimensione di M

4. la valutazione di un'unione di queries SQL su un database è LOGSPACE rispetto alla dimensione del database.

In conclusione il reasoning su un ontologia in DL-LiteA può essere sempre ricondotto alla valutazione di queries SQL appartenenti alla FOL. Quindi:

- la complessità computazionale è LOGSPACE rispetto alla dimensione dei dati
- il query answering, dopo il processo di riformulazione, può essere delegato al RDBMS

4. Linguaggi di query epistemici

In 3.2 abbiamo visto che una query in FOL su un ontologia con Open World Assumption risulta indecidibile. Per questo motivo ci siamo limitati a considerare solo Conjunctive Query.

Il potere espressivo di questa classe di query è notevolmente limitato e non permette di esprimere interrogazioni complesse che sono di grande interesse pratico, come ad esempio query in cui compare l'operatore NOT.

Per risolvere questa limitazione si introducono i lunguaggi di query epistemici mediante i quali è possibile avere lo stesso potere espressivo della FOL anche sulle ontologie, mantenendo una complessità computazionale accettabile.

4.1 Chiusura dinamica del dominio ed EQL

Introduciamo ora un nuovo linguaggio di query, una variante della logica modale del primo ordine: il linguaggio EQL.
Questo nuovo linguaggio adotta l'operatore di conoscenza ***K*** che serve a formalizzare lo stato epistemico della base di conoscenza.
Con $K\phi$ intendiamo dire che "si sa che ϕ è valida dalla base di conoscenza".
A questo punto possiamo effettuare query sulla base di conoscenza pur avendo una informazione incompleta.

EQL = FOL + operatore epistemico (di conoscenza minimale) su KB

Essendo una variante della logica modale sappiamo che ogni modello rappresenta un mondo.
Un mondo è una interpretazione della FOL sul dominio di interesse.
Un'interpretazione epistemica è una coppia E,w dove E è l'insieme di tutti i modelli della Base di conoscenza, e w è un modello in E.

Sintassi EQL:
ϕ::= A(t) | P(t1...tn) | t1=t2 | $\neg\phi$ | $\phi 1 \wedge \phi 2$ | $\exists x.\phi$ | Kϕ

Definizione induttiva della verità di ϕ in un'interpretazione E,w :

E,w $\models$ $c_1=c_2$ sse $c_1=c_2$

E,w $\models$ P(**c**) sse w $\models$ P(c)

E,w $\models$ $\phi_1 \wedge \phi_2$ sse E,w $\models$ ϕ_1 AND E,w $\models$ ϕ_2

E,w $\models$ $\neg\phi$ sse E,w $\not\models$ ϕ

E,w $\models$ $\exists x.\psi$ sse E,w $\models$ ψ_{c}^{I} per qualche costante c

E,w $\models$ $\mathbf{K}\psi$ sse E,w' $\models$ ψ per ogni w' $\in$ E

Tutte le formule che contengono ***K*** sono dette soggettive, in quanto si basano su ciò che la base di conoscenza conosce e non su verità assolute. Quindi se si vuole verificare se un'interpretazione modella una formula soggettiva bisogna prendere in considerazione solo E, tralasciando w, in quanto ci stiamo chiedendo cosa la base di conoscenza conosce.

Tutte le formule che non contengono ***K*** sono invece dette oggettive, in quando si basano su ciò che è vero. Quindi quando valutiamo una formula oggettiva possiamo prendere in considerazione solo w, senza E.

E' facile verificare che tutte le asserzioni nella KB sono formule oggettive.

Tramite questo nuovo operatore non interroghiamo le informazioni presenti nel mondo, bensì lo stato epistemico della base di conoscenza.

Di tutte le interpretazioni epistemiche noi siamo interessati a quella che rappresenta lo stato epistemico minimale della DL KB, ovvero lo stato in cui la KB ha conoscenza minimale.

Quindi data una DL KB Σ , e denotato Mod(Σ) l'insieme di tutte le FOL-interpretazioni che sono modelli di Σ , allora una Σ -EQL-interpretazione è un'interpretazione epistemica E,w per la quale E= Mod(Σ).
Una formula ϕ è EQL-logicamente implicata da Σ , ovvero Σ $\models$ EQLϕ, se ogni EQL-interpretazione E,w implica logicamente ϕ.

4.2 EQL-Lite(Q)

EQL-Lite(Q) è un frammento di EQL ed è parametrizzato rispetto a un linguaggio di query immerso Q, anch'esso sottoinsieme di EQL.
Una Query EQL-Lite(Q) è una possibile EQL-Lite formula aperta che rispetta la seguente sintassi:

$$\phi ::= \mathbf{K}\rho \mid x_1 = x_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists x.\phi$$

dove ρ è una query espressa nel linguaggio di query immerso Q.
Di fatto EQL-Lite(Q) non è altro che il linguaggio di query FOL con uguaglianza in cui gli atomi sono formule epistemiche: gli atomi epistemici sono formule $\mathbf{K}\rho$ presenti nella query EQL-Lite(Q).

L'operatore **K** non può essere utilizzato al di fuori degli atomi epistemici, e anzi, utilizzandolo al di fuori di tale contesto non si aumenta la potenza espressivo del linguaggio.
Si può infatti dimostrare che ogni query espressa nel linguaggio EQL-Lite(Q)$^+$, dove EQL-Lite(Q)$^+$ è l'estensione di EQL-Lite(Q) ottenuta aggiungendo alla sintassi la regola $\psi ::= \mathbf{K}\psi$, può essere ridotta a una query equivalente in EQL-Lite(Q) in tempo lineare semplicemente spostando internamente l'operatore **K** e non facendo nulla nel caso di atomi epistemici.

La proprietà computazionale interessante di EQL-Lite(Q) è che per valutare una query si può separare il ragionamento necessario per rispondere agli atomi epistemici dal ragionamento necessario per rispondere all'intera query.

4.3 EQL-Lite(UCQ) su ontologie in DL-Lite

Introduciamo EQL-Lite(UCQ), linguaggio di query derivato dal precedente dove come linguaggio di query immerso Q si utilizzano UCQ.
Ricordiamo che il query answering in DL-Lite è LOGSPACE rispetto alla complessità dei dati.
Si dimostra che EQL-Lite(UCQ) è ancora LOGSPACE nella complessità dei dati nel caso di query answering.

La DL-Lite family si basa sulla FOL-riducibilità di UCQs; la FOL-riducibilità è valida anche per query EQL-Lite(UCQ) indipendenti dal dominio.

Ricordiamo che la FOL-riducibilità significa che il query answering si riduce alla valutazione di query in logica del prim'ordine su una finita FOL-interpretazione basata sulla ABOX di una base di conoscenza.
Questo risultato è molto forte in quanto se il query answering può essere ridotto alla valutazione di query in FOL indipendenti dal dominio, allora una query può essere espressa in algebra relazionale, e quindi in SQL, e quindi si possono sfruttare tutti i vantaggi dei DBMS in uso.

Più formalmente:

Data una ABOX che comprende solo asserzioni su concetti atomici e ruoli definiamo la sua interpretazione come segue:

- $a^{I_A} = a$ per ogni costante a
- $A^{I_A} = \{a \mid A(a) \in \mathcal{A}\}$ per ogni concetto atomico A
- $a^{I_P} = \{(a1,a2) \mid P(a1,a2) \in \mathcal{A}\}$ per ogni ruolo atomico P

Il queries answering in un linguaggio di query *L* (contenuto in EQL) su una KB espressa in una DL *DL* è FOL-riducibile se per ogni query $q \in L$ e ogni TBOX T espressa in *DL*, esiste una query FOL rdc(q) tale che per ogni ABOX $\mathcal{A}$, abbiamo che $ans(q,(T,\mathcal{A})) = eval(rdc(q),I_A)$.

Possiamo quindi concludere con il seguente teorema:

Rispondere a queries EQL-Lite(Q) è FOL-riducibile. Inoltre, se la query q in EQL-Lite(UCQ) è indipendente dal dominio, allora rdc(q) è tale anch'esso.

4.4 SPARSQL

Per superare i limiti espressivi delle Conjunctive Queries è stato proposto, come abbiamo visto, il linguaggio EQL basato sul principio che si può assumere di avere informazione completa su ciò che si conosce, ricostituendo così in un certo qual modo l'assunzione CWA.

Il linguaggio implementato per esprimere le query in EQL è SparSQL.

Sintassi di SparSQL:

SparSQL = SQL + SPARQL

La sintassi è basata su:

- la sintassi standard di SQL
- la sintassi di SPARQL

Lo schema è quello di una query SQL ma nella clausola FROM ci sono una o più tabelle ottenute valutando queries SPARQL sull'ontologia.

QuerySparSQL :: =

SELECT *ListaAttributiOEspressioni*

FROM (*sparqltable(< QuerySparql >) alias*) +

[**where** *CondizioniSemplici*]

[**group by** *ListaAttributiDiRaggruppamento*]

[**having** *CondizioniAggregate*]

[**order by** *ListaAttributiDiOrdinamento*]

A ogni sparqltable deve essere associato un alias utilizzato in seguito nella query SparSQL.

In SparSQL non viene utilizzata tutta la sintassi di SPARQL in quanto le queries SPARQL devono avere la stessa espressività delle queries congiuntive e quindi molte clausole non possono essere utilizzate.
Il soggetto di una tripla SPARQL può essere una variabile, un nome di concetto, un'istanza di concetto tra apici oppure un'istanza di attributo. Il soggetto può essere anche una tripla nel caso si voglia esprimere un attributo di ruolo.
Il predicato di una tripla può essere un nome di ruolo o la stringa "rdf:type"; oppure un nome di attributo di concetto o nome di attributo di ruolo.
L'oggetto di una tripla può essere una variabile, un nome di concetto, un'istanza di concetto o un'istanza di attributo.
Ci sono delle regole da rispettare per scrivere queries in EQL:

- La query in input deve terminare con un ";"
- Gli attributi della clausola SELECT della query SparSQL devono avere nomi coerenti con le variabili restituite dalle queries SPARQL della clausola FROM
- Tutto ciò che è presente nelle queries SparSQL è case sensitive ad eccezione di nomi di concetti, ruoli, attributi di concetti, attributi di ruoli e relative istanze
- Ogni tabella SPARQL deve avere un alias utilizzato nella query SparSQL
- Quando si usa il predicato "rdf:type" in una tripla sparql l'oggetto della tripla deve essere indicato tra apici
- Se si vuole far riferimento ad un'istanza specifica sia nella query sql che nella query sparql il nome dell'istanza deve essere inserito tra apici ' '.
- Il punto finale dopo una tripla sparql, nel caso si tratti dell'ultima tripla di un GroupGraphPattern inserita nella query sparql, è opzionale;
- Le variabili restituite dalla query sparql devono essere effettivamente utilizzate nella rispettiva query altrimenti viene lanciata un'eccezione;
- Nella query EQL e nell'ontologia non possono essere utilizzate per i nomi dei concetti, ruoli, attributi di concetto, attributi di ruolo le parole chiave di seguito elencate:

EOF, ALL, AND, ANY, AS, ASC, BOOLEAN, AVG, BINARY_INTEGER, BETWEEN, BY, CHAR, COMMENT, COMMIT, CONNECT, COUNT, DELETE, DESC, DISTINCT, EXCLUSIVE, EXISTS, EXIT, FLOAT, FOR, HAVING, FROM, GROUP, IN, INSERT, INTEGER, INTERSECT, INTO, IS, LIKE, LOCK, MINUS, MAX, MIN, MODE, NATURAL, NOT, NOWAIT, NULL, NUMBER, OF, ONLY, OR, ORDER, PRIOR, QUIT, READ, REAL, ROLLBACK, ROW, SELECT, SET, SHARE, SMALLINT, START, SUM, TABLE, TRANSACTION, UNION, UPDATE, VALUES, VARCHAR2, VARCHAR, WHERE, WITH, WORK, WRITE, SPARQLTABLE, rdf:type, REDUCED, CREATE, VIEW, NUMBER, FLOAT, INTEGER, DIGIT.

Semantica di SparSQL:

SparSQL è il linguaggio che implementa EQL-Lite(UCQ), e quindi una query espressa in SparSQL non è altro che:

$$q(\mathbf{x}) : -\phi(\boldsymbol{K}\alpha_1, \ldots, \boldsymbol{K}\alpha_n)$$

Dove:

- ϕ è rappresentata dalla query SQL
- $\mathbf{K}\alpha_i$ corrisponde ad una UCQ che nel caso di SparSQL è formata da una query SPARQL
- **x** è il vettore delle variabili che devono essere restituite dalla query

La clausola FROM estrae la conoscenza dell'ontologia attraverso tabelle (sparqltable) ottenute tramite queries SPARQL che corrispondono ad UCQs.
Quindi le sparqltable delle queries SparSQL corrispondono alle queries soggettive in EQL che utilizzano l'operatore ***K***.
In sostanza attraverso le sparqltable andiamo ad estrarre la stato epistemico della base di conoscenza.

4.5 SPARQL

SPARQL è un linguaggio di query per ontologie che è in grado di esprimere Union of Conjunctive Query e per questo motivo viene utilizzato da SparSQL per estrarre informazioni sullo stato epistemico dell'ontologia.

La sintassi di SPARQL ricorda molto quella di SQL:

- **PREFIX:** dichiara prefissi e namespace
- **SELECT:** definisce le variabili di ricerca da prendere in considerazione nel risultato
- **FROM:** specifica il set di dati su cui dovrà operare la query
- **WHERE:** definisce il criterio di selezione specificando tra parentesi graffe uno o più “triple patterns” separati da un punto
- **PH:** per specificare più set di dati

Come in SQL è presente un comando Select che contiene l’elenco delle variabili da ritornare, e quindi non è altro che una proiezione sui dati di interesse; abbiamo poi clausola From che elenca i documenti sui quali stiamo effetuando la query; e poi Where che contiene tra parentesi graffe i criteci di selezioni, ovvero i “triple pattern”. L’incognita dell’interrogazione viene scritta con ? davanti.

Tutte le stringhe sono racchiuse tra apici singoli o doppi, e possono avere dei tag opzionali o il riferimento al tipo di dato.
La clausola SELECT restituisce i risultati sottoforma di un set di binding per le variabili cioè di associazioni variabili-valore. E’ l’unica clausola utilizzata nel linguaggio SparSQL.
Come in SQL, è prevista la selezione di tutte le variabili presenti nella clausola WHERE attraverso l’operatore ’*’ e l’unicità dei risultati attraverso la clausola DISTINCT.

5. Protégé: un editor per ontologie

5.1 Descrizione del sistema

Protégé è un editor di ontologie free ed open-source, nonchè un framework per basi di conoscenza che supporta gli utenti nella realizzazione di applicazioni che facciano uso di ontologie.
Tra le principali funzionalità consente:

- la creazione, la manipolazione e la visualizzazione di ontologie;
- la personalizzazione delle varie funzionalità;
- l'import / export delle ontologie in vari formati (OWL, RDF Schema, XML Schema, ecc...);
- l'esecuzione di ragionatori automatici come classificatori di Description Logic.
- I suoi principali vantaggi sono:
- l'architettura basata su Java plug-in che lo rende facilmente estendibile: grazie al Programming Development Kit (PDK) è possibile disporre della documentazione e delle Application Programming Interfaces (APIs) per accedere al codice; si presenta come una collezione di plug-ins singolarmente o interamente sostituibili per modificarne il comportamento e l'interfaccia;
- la vasta community che contribuisce a renderlo aggiornato: al giorno d'oggi consta di uno staff costantemente impegnato nello sviluppo e di oltre 75.000 utenti registrati che contribuiscono alla sua estensione condividendo i plug-in realizzati.

5.2 The OBDA Plugin for Protégé

L'obiettivo dell' Ontology-Based Data Access è quello di utilizzare un'ontologia come mezzo per l'accesso alle sorgenti dati. Il valore aggiunto, rispetto all'accesso diretto, è che un'ontologia fornisce una descrizione semantica delle informazioni immagazzinate nelle sorgenti.
Inoltre la risposta alle queries dell'utente possono essere arricchite sfruttando i vincoli espressi nell'ontologia, così da compensare eventuale incompletezze di informazioni che possono essere presenti nei dati.
I sistemi OBDA hanno una struttura comune: presentano un livello di semantic layer,

ovvero nient'altro che un' Ontologia scritta in un certo linguaggio ontologico; permettono in un qualche modo di accedere a una o più sorgenti dati, anche di tipi diversi; gestiscono insiemi di mappings che esprimono la corrispondenza tra le sorgenti dati e gli oggetti presenti nel livello semantico; permettono agli utenti di esprimere queries, prendendo come input l'ontologia, i mappings e i dati presenti nella sorgenti.
Tutti i componenti che il reasoner usa sono già preconfigurati: ovvero quando si interroga il reasoner sottostante l'utente ha già modellato l'ontologia per il suo dominio applicativo, i mapping e ha già caricato i dati.
Da qui nasce l'idea di questo plug-in: offrire un tool in grado di modellare e verificare i vari componenti di un sistema OBDA.
Allo stato attuale sono offerte funzionalità per descrivere sorgenti gestite da DBMS relazionali; descrivere mappings del tipo $\psi \rightarrow \phi$, dove ϕ è una query congiuntiva sull'ontologia, mentre ψ è una qualunque query SQL sulle sorgenti dati; interrogare tramite UCQs rispettando la sintassi SPARQL l'OBDA-enabled reasoner e vedere il risultato; e in ultimo esaminare e manipolare direttamente le sorgenti relazionali.
Il plugin fornisce un'interfaccia per la creazione e manipolazione di mapping sulle sorgenti, nonchè la possibilità di interrogare le sorgenti direttamente. Al momento supporta sorgenti di dati relazionali e mapping di tipo GLAV. Inoltre fornisce metodi addizionali per il querying che permettono all'utente di interrogare la base di dati tramite un reasoner.
In sostanza questo editor, appoggiandosi a un reasoner permette all'utente di costruire, testare e dispiegare un sistema di tipo OBDA.
Esistono diversi reasoners che differiscono per molti aspetti, ad esempio, nel tipo di mapping ammessi, o il tipo di linguaggio per esprimere tali mappings, o ancora il numero e il tipo di srogenti dati che sono in grado di manipolare, il linguaggio con cui viene espressa l'ontologia, il linguaggio con cui vengono espresse le queries, etc...
In Protégé il linguaggio di ontologie è l'OWL-DL, un linguaggio che bene si adatta ai requisiti della maggior parte degli utenti.
Per quanto riguardo il linguaggio delle queries, si fa uso della classe delle UCQs(Unione di Queries Congiuntive), sopra l'ontologia.
Per quanto riguarda le sorgenti dati, il Plug-in si limita a supportare sorgenti di dati relazionali.

5.3 DIG-Mastro: OBDA-enabled reasoner

L'interfaccia del DIG è un'interfaccia standardizzata HTTP/XML per i reasoner delle Description Logic, ed è stata sviluppata dal DL Implementation Group con lo scopo di migliorare la comunicazione tra i tools che fanno uso uso dei DL reasoners. In particolare DIG-Mastro fornisce servizi standard sulle ontologie DL, come il check della consistenza e il query answering.
In generale gli OBDA-enabled reasoners forniscono questi servizi standard su ontologie che presentano un insieme di mapping, che come visto sono un insieme di asserzioni che stabiliscono le relazioni tra gli elementi dell'ontologia e i dati delle sorgenti.
DIG-Mastro è il frutto dello studio di ricerca sul linguaggio DL-$Lite_{A,}$ che come visto è un frammento di OWL-DL. DL-$Lite_A$ è abbastanza espressivo da riuscire a catturare i costrutti standard dell'UML e dell'ER.
Ricordiamo che il reasoning in DL-$Lite_A$ è LOGSPACE nella complessità dei dati.
Ad oggi Mastro è l'unico reasoner DL che oltre ad implementare i servizi tradizionali di reasoning, incorpora funzionalità per specificare i dati sorgenti e i mappings e li prende come input al momento del reasoning.
DIG-Mastro permette a Mastro di interagire con qualsiasi client che è conforme con l'estensione OBDA del DIG. Inoltre estende le funzionalità del querying di Mastro offrendo un insieme di nuovi servizi per l'answering delle UCQ.
Di fatto DIG-Mastro è un componente del sistema OBDA che ben si adatta a particolari istanze di problemi in cui è presente una gran quantità di dati immagazzinati nelle sorgenti, in cui è richiesto un efficiente reasoner.

Ora mostriamo come attraverso l'uso di DIG-Mastro, Protégé e il Plug-in, di cui sopra abbiamo parlato, è possibile realizzare query tramite un sistema OBDA.

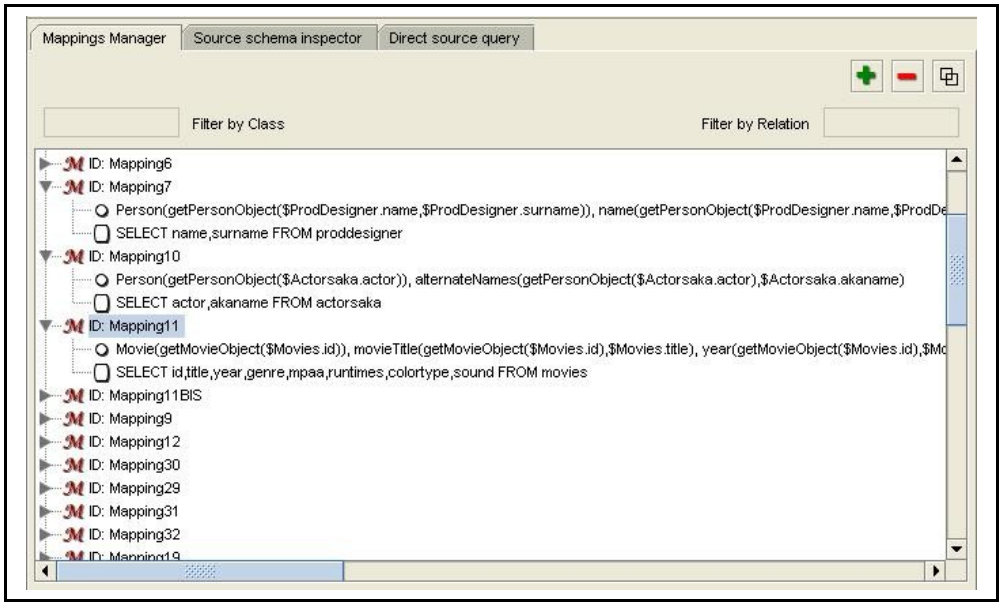


Figura 1: RDMS-Ontolgy Mapping Pane

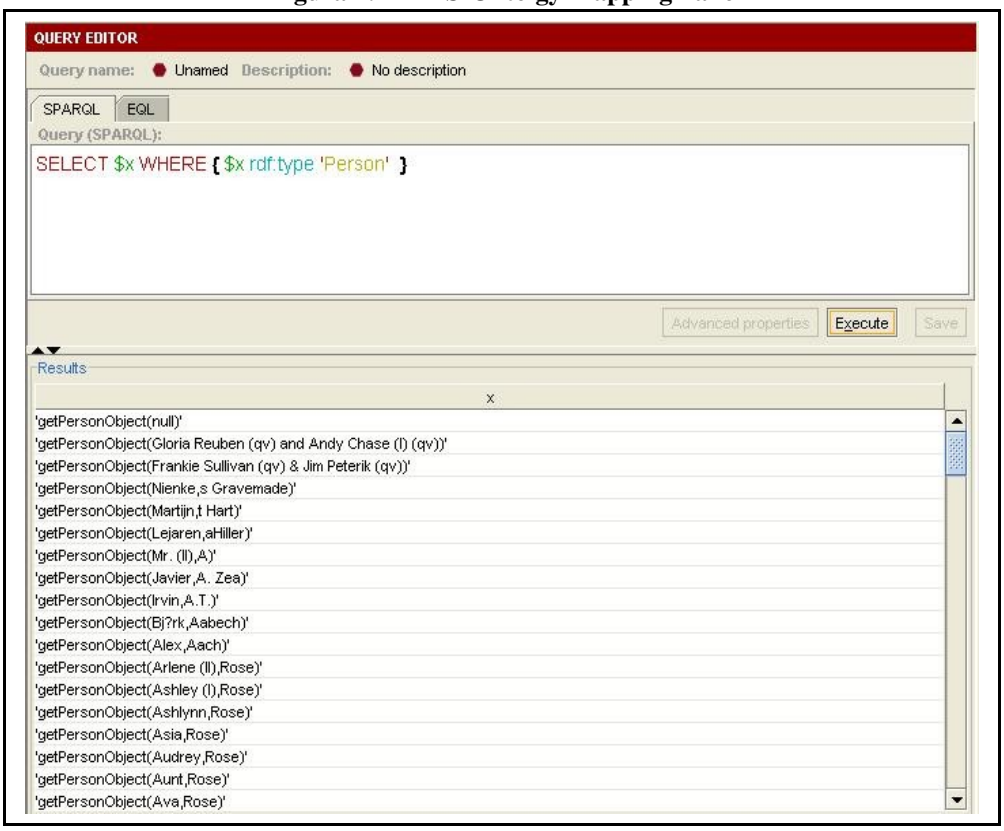


Figura 2: SPARQL UCQs Pane

Ricordiamo che l'ontologia è scritta in OWL-DL, e DIG-Mastro lavora con ontologie scritte in DL-LiteA, quindi quando il Plug-in interroga il reasoner, se possibile riscrive assiomi espressi in linguaggio NON DL-LiteA in assiomi equivalenti espressi in DL-LiteA, e scarta quegli assiomi che vanno oltre DL-LiteA, lanciando un avviso ogni volta che questo accade.
Come abbiamo detto il Plug-in permette all'utente di specificare un database relazionale come sorgente dati, e un insieme di mapping tra i dati e l'ontologia.

La sintassi è quella mostrata in figura; la semantica dipende dal reasoner utilizzato: nel nostro caso, un mapping del tipo $\psi \rightarrow \phi$, esprime che per ogni tupla t che soddisfa ψ, esiste un insieme di oggetti che sono costruiti a partire da t tramite funtori adatti (Skolem), che soddisfano ϕ nell'ontologia.

Il Plug-in offre tutte la possibilità di effettuare query answering e la possibilità di interagire con il reasoner. Tramite il pannello SPARQL UCQs vengono passate le UCQs al reasoner.
DIG-Mastro prende in input l'ontologia, i dati e i mappings, a questo punto implementa le tecniche di riscrittura che traducono le UCQs prese passate in un insieme di queries sulla sorgente dati, e l'unione dei risultati di tali queries costituiranno la risposta alla query originale.
Alla fine il Plug-in mostra il risultato e permette all'utente di manipolare ulteriormente i dati, esportandoli o salvandoli.

6. Implementazione Ontologia IMDB

Il lavoro svolto è articolato in diverse fasi:

1. Scrittura ontologia in DL-LiteA (vedi Appendice A)
2. Mapping dei dati relazionali sull'ontologia (vedi Appendice B)
3. Implementazione di ontologia e mapping in XML per MastroI
4. Implementazione di ontologia e mapping in Protege
5. Interrogazione dell'ontologia con query SPARQL e EQL
6. Test per comprendere i motivi della scarsa efficienza riscontrata al punto 5
7. Test con le varie versioni del plugin OBDA di Protege e modifiche al sistema QuOnto

6.1 Ontologia IMDB

Mediante la navigazione del sito web www.imdb.com è stato possibile estrapolare i concetti chiave del dominio applicativo. Data la vastità del sito, si è scelto di limitarsi ad alcuni aspetti tralasciando i dettagli minori. Tale analisi ha portato alla scrittura di un insieme di Inclusion Assertions in DL-LiteA, scritte in notazione logica, che costituiscono la TBox dell'ontologia, riportata integralmente in Appendice A.

Successivamente si è preso in considerazione un database relazionale contenente dati sui film, costruito a partire da alcuni dati forniti in formato testuale da IMDB. A causa della natura poco chiara dei dati di partenza, nel database sono presenti molte informazioni scorrette, oltre a numerose stringhe “null” che non sono correttamente interpretate da MySQL come valori NULL. Nel seguito non ci preoccuperemo più del problema dei dati sporchi, fermo restando che le query di test restituiranno spesso valori dal significato scorretto o difficilmente interpretabile.

Il mapping di tale DB sull'ontolgia scritta precedentemente permette di popolare una minoranza dei concept e roles, tuttavia alcuni mapping sono di particolare interesse, essendo il risultato di importati scelte progettuali.

Problema: nel database sono presenti diverse tabelle che rappresentano ciascuna una diversa categoria di persona, ad esempio le tabelle actor, cinematographer, composer, designer, etc...

Naturalmente le persone possono ricoprire diversi ruoli, nel corso della propria carriera, quindi in tale tabelle saranno presenti molte tuple in comune.

Nella nostra ontologia invece esiste il Concept “Person” che racchiude tutte le persone, qualunque sia il loro lavoro. Esistono poi svariati Role che rappresentano il ruolo svolto dalla persona in un film (es. directedBy, producedBy, ecc...)

Una persona può quindi partecipare in più role.

In una prima revisione dei mapping è stato commesso un grave errore, corretto poi in seguito:

Person(x) ∧ name(x,y) ∧ surname(x,z) ← **SELECT** name, surname **FROM** cinematographer
x → cin(name,surname)

Person(x) ∧ name(x,y) ∧ surname(x,z) ← **SELECT** name, surname **FROM** composer
x → com(name,surname)

Nell'ontologia il concept Person rappresenta tutte le persone, indipendentemente dal lavoro che svolgono. Ci sono poi svariati role per rappresentare il ruolo svolto da una persona in un film (es. editedBy, producedBy, etc..) . Una persona può svolgere diversi ruoli nel corso della propria carriera.
Nel database invece sono presenti diverse tabelle, una per ogni ruolo (es. actor, cinematographer, composer, etc...): se una stessa persona svolge più ruoli allora c'è una tupla in ogni tabella.
Con i mapping scritti inizialmente si verificava un grave problema: se una persona svolgeva più ruoli, cioè era presente in diverse tabelle del db, nel Concept Person venivano mappati duplicati e la stessa persona era identificata ogni volta con i diversi funtori.

Soluzione 1:
Lasciare invariati i funtori e modificare le query sql dei mapping in modo tale che restituiscano solo le tuple che non sono già state mappate in Person da altre tabelle.
Questa soluzione presenta diversi problemi in quanto non è possibile stabilire un ordine in cui vengono richiamati i mapping e comunque le query sql diventerebbero molto compplesse. Inoltre non è semanticamente corretto che una persona che è attore , producer e cinematographer sia identificata dal funtore cin(): perchè non act() o prod()?

Soluzione 2:
Tutte le persone sono identificate dal funtore per(name, surname) indipendentemente dal lavoro svolto, cioè dalla tabella in cui si troavano. In questo modo vengono evitati i duplicati in quanto la prima volta che viene trovata una persona con name= a e surname = b viene mappato in Person per(a,b). Se successivamente in un altra tabella del db si ritrova la stessa persona, poiché verrebbe identificata sempre da per(a,b) non viene inserita in quanto riconosciuta come duplicato.
E' stata scelta la soluzione 2 e sono stati modificati i mapping come segue:

Person(x) ∧ name(x,y) ∧ surname(x,z) ← **SELECT** name, surname **FROM** cinematographer
x → per(name,surname)

Person(x) ∧ name(x,y) ∧ surname(x,z) ← **SELECT** name, surname **FROM** composer
x → per(name,surname)

Il mapping degli attori comportava un problema in più: nel db gli attori sono identificati da una chiave id e non dalla coppia (name, surname) come tutte le altre persone. Inizialmente avevamo scritto questo mapping:

Actor(x) ∧ name(x,y) ∧ surname(x,z) ← **SELECT** id, name, surname **FROM** composer
x → act(id)

Poiché nell'ontologia Actor ISA Person si ottenevano di nuovo dei duplicati, ad esempio se la persona con name=a e surname=b era presente anche nella tabella actor con id=1, in Person sarebbero stati mappate due istanze della stessa persona, identificate rispettivamente da act(1) e per(a,b).

La soluzione a questo problema ricalca la precedente e consiste nell'utilizzare anche per gli attori il funtore per(name,surname), ignorando gli id che sono tra l'altro delle chiavi surrogate, in quanto in actor esiste una sola tupla per ogni (name, surname) essendo gli omonimi identificati sempre con un suffisso numerico del cognome.

Actor(x) ∧ name(x,y) ∧ surname(x,z) ← **SELECT** name, surname **FROM** composer
x → per(name, surname)

Un altro mapping interessante è quello delle TV Series:
nel database le serie tv sono inserite nella stessa tabella dei film e sono distinte da un tag (TV) all'interno del titolo, quindi per mapparle correttamente è necessario effettuare operazioni sulla stringa title per estrapolare informazioni sul titolo della serie e sul

titolo del singolo episodio.

TvSerie(x) ∧ movieTitle(x,y) ∧ episodeTitle(x,e)
←
SELECT id, SUBSTRING(SUBSTRING_INDEX(title, ':', 1),5) as titolo_serie,
SUBSTRING_INDEX(title, ':', -1) as titolo_episodio
FROM movies
WHERE title like '%(TV)%'

x → mov(ID)
y → titolo_serie xsd:String
e → titolo_episodio xsd:String

6.2 Codifica XML

Il sistema Mastro/QuOnto prende in input la TBox e i Mapping dell'ontologia opportunamente codificati in XML secondo le regole imposte dalle DTDs.

DTD TBox:

```
<!-- level 1
**************************************************************************-->

<!ELEMENT ontology (alphabet, tbox)>

<!-- level 2
**************************************************************************-->

<!ELEMENT alphabet (atomicC | atomicV | atomicR | atomicCA | atomicRA)+>

<!ELEMENT tbox (inclusionAssertion | funct)*>

<!-- level 3
**************************************************************************-->

<!ELEMENT atomicC (#PCDATA)>

<!ELEMENT atomicV (#PCDATA)>

<!ELEMENT atomicR (#PCDATA)>

<!ELEMENT atomicCA (#PCDATA)>

<!ELEMENT atomicRA (#PCDATA)>

<!ELEMENT inclusionAssertion ((basicC, generalC+) |  (basicV, generalV+) | (basicR,
generalR+) |
```

```
  (atomicCA, generalCA+) | (atomicRA, generalRA+))>

<!ELEMENT funct (basicR | atomicCA | atomicRA)>

<!-- level
4****************************************************************************-->

<!ELEMENT basicC (atomicC | exists | CADomain)>

<!ELEMENT generalC (topC | signedC | qualifiedExists | CAQualifiedDomain |
existsRAQualifiedDomain)>

<!ELEMENT basicV (atomicV | ARange)>

<!ELEMENT generalV (topD | signedV | predefinedV)>

<!ELEMENT basicR (atomicR | RADomain)>

<!ATTLIST basicR dir (direct | inverse) #REQUIRED>

<!ELEMENT generalR (signedR | RAQualifiedDomain)>

<!ELEMENT generalCA (atomicCA)>

<!ATTLIST generalCA sign (positive | negative) #REQUIRED>

<!ELEMENT generalRA (atomicRA)>

<!ATTLIST generalRA sign (positive | negative) #REQUIRED>

<!-- level 5
****************************************************************************-->

<!ELEMENT exists (basicR)>

<!ELEMENT CADomain (atomicCA)>

<!ELEMENT topC EMPTY>

<!ELEMENT signedC (basicC)>

<!ATTLIST signedC sign (positive | negative) #REQUIRED>

<!ELEMENT qualifiedExists (basicR, generalC)>

<!ELEMENT CAQualifiedDomain (atomicCA, generalV)>

<!ELEMENT existsRAQualifiedDomain (RAQualifiedDomain)>

<!ELEMENT ARange (atomicCA | atomicRA)>

<!ELEMENT topD EMPTY>

<!ELEMENT signedV (basicV)>

<!ATTLIST signedV sign (positive | negative) #REQUIRED>

<!ELEMENT predefinedV (#PCDATA)>

<!ELEMENT RADomain (atomicRA)>

<!ELEMENT signedR (basicR)>

<!ATTLIST signedR sign (positive | negative) #REQUIRED>

<!ELEMENT RAQualifiedDomain (atomicRA, generalV)>

<!ATTLIST RAQualifiedDomain dir (direct | inverse) #REQUIRED>

<!--
*****************************************************************************_
->
```

Frammento della TBox (7.153 righe):

```
<!-- ACTOR ⊑ PERSON -->
<inclusionAssertion>
        <basicC>
                <atomicC>Actor</atomicC>
        </basicC>
        <generalC>
                <signedC sign="positive">
                        <basicC>
                                <atomicC>Person</atomicC>
                        </basicC>
                </signedC>
        </generalC>
</inclusionAssertion>

<!-- PERSON ⊑ δ (name) -->
<inclusionAssertion>
        <basicC>
                <atomicC>Person</atomicC>
        </basicC>
        <generalC>
                <signedC sign="positive">
                        <basicC>
                                <CADomain>
                                        <atomicCA>name</atomicCA>
                                </CADomain>
                        </basicC>
                </signedC>
        </generalC>
</inclusionAssertion>
```

DTD GeneralMapping:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ******** level 1 ******** -->
<!ELEMENT abox (type_of_abox, type_of_db, mappings, db_info)>
<!-- ******** level 2 ******** -->
<!ELEMENT type_of_abox (direct_abox | general_abox)>
<!ELEMENT type_of_db (local_db | friend_db | external_db)>
<!ELEMENT mappings (mapping)+>
<!-- ******** level 3 ******** -->
<!ELEMENT direct_abox EMPTY>
<!ELEMENT general_abox EMPTY>
<!ELEMENT local_db EMPTY>
<!ELEMENT friend_db EMPTY>
<!ELEMENT external_db EMPTY>
<!ELEMENT mapping (head, (map)+, body)>
<!-- ******** level 4 ******** -->
<!ELEMENT head (CQBody)>
<!ELEMENT map (objMap | valueMap)>
<!ELEMENT body (#PCDATA)>
<!ELEMENT db_info (manager, driver, db_url, db_name, user, password)>
<!-- ******** level 5 ******** -->
<!-- all variables in CQBody are distinguished -->
<!ELEMENT CQBody (atom)+>
<!ELEMENT objMap (dtVar, sqlObjVarList)>
<!ELEMENT valueMap (dtVar, sqlValueVar)>
<!ELEMENT sqlQuery (#PCDATA)>
<!ELEMENT manager (#PCDATA)>
<!ELEMENT driver (#PCDATA)>
<!ELEMENT db_url (#PCDATA)>
<!ELEMENT db_name (#PCDATA)>
<!ELEMENT user (#PCDATA)>
<!ELEMENT password (#PCDATA)>
<!-- ******** level 6 ******** -->
<!ELEMENT atom (AtomicConcept | AtomicDomainValue | AtomicRole | AtomicConceptAttribute
| AtomicRoleAttribute)>
<!ELEMENT dtVar (#PCDATA)>
<!ELEMENT sqlObjVarList (sqlObjVar)+>
<!ATTLIST sqlObjVarList
      funct   CDATA #REQUIRED
```

```
>
<!ELEMENT sqlObjVar (#PCDATA)>
<!ELEMENT sqlValueVar (#PCDATA)>
<!ATTLIST sqlValueVar
        type CDATA #REQUIRED
        relation CDATA #IMPLIED
>
<!-- ******** level 7 ******** -->
<!ELEMENT AtomicConcept (term)>
<!ATTLIST AtomicConcept name CDATA #REQUIRED>
<!ELEMENT AtomicDomainValue (term)>
<!ATTLIST AtomicDomainValue name CDATA #REQUIRED>
<!ELEMENT AtomicRole (term , term)>
<!ATTLIST AtomicRole name CDATA #REQUIRED>
<!ELEMENT AtomicConceptAttribute (term , term)>
<!ATTLIST AtomicConceptAttribute name CDATA #REQUIRED>
<!ELEMENT AtomicRoleAttribute (term , term , term)>
<!ATTLIST AtomicRoleAttribute name CDATA #REQUIRED>
<!-- ******** level 8 ******** -->
<!ELEMENT term (const | var)>
<!-- ******** level 9 ******** -->
<!ELEMENT const (#PCDATA)>
<!ELEMENT var EMPTY>
<!ATTLIST var name CDATA #REQUIRED>
<!-- ************************ -->
```

Frammento dei mapping (2926 righe):

```
<!-- Actor(x) ∧ name(x,y) ∧ surname(x,z) ∧ sex(x,w) →
        SELECT name, surname, sex FROM actor
        x → per(name, surname)
        y → name xsd:String
        z → surname xsd: String
        w → sex xsd: Char -->
<mapping>
        <head>
                <CQBody>
                        <atom>
```

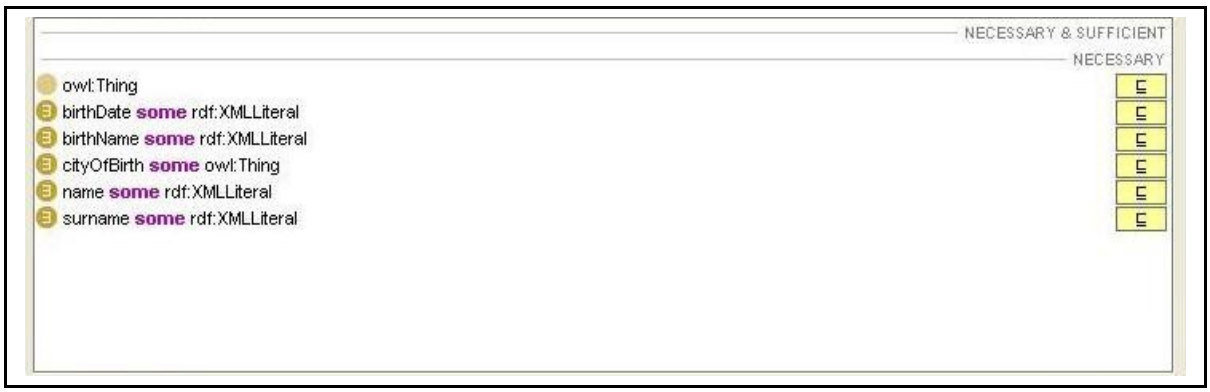
```
                        <AtomicConcept name="Person">
                                <term>
                                        <var name="x"/>
                                </term>
                        </AtomicConcept>
                </atom>
                <atom>
                        <AtomicConceptAttribute name="name">
                                <term>
                                        <var name="x"/>
                                </term>
                                <term>
                                        <var name="y"/>
                                </term>
                        </AtomicConceptAttribute>
                </atom>
                <atom>
                        <AtomicConceptAttribute name="surname">
                                <term>
                                        <var name="x"/>
                                </term>
                                <term>
                                        <var name="z"/>
                                </term>
                        </AtomicConceptAttribute>
                </atom>
                <atom>
                        <AtomicConceptAttribute name="sex">
                                <term>
                                        <var name="x"/>
                                </term>
                                <term>
                                        <var name="w"/>
                                </term>
                        </AtomicConceptAttribute>
                </atom>
        </CQBody>
</head>
<map>
        <objMap>
```

```
                <dtVar>x</dtVar>
                <sqlObjVarList funct="per">
                        <sqlObjVar>name</sqlObjVar>
                        <sqlObjVar>surname</sqlObjVar>
                </sqlObjVarList>
        </objMap>
</map>
<map>
        <valueMap>
                <dtVar>y</dtVar>
                <sqlValueVar type="xs:string">name</sqlValueVar>
        </valueMap>
</map>
<map>
        <valueMap>
                <dtVar>z</dtVar>
                <sqlValueVar type="xs:string">surname</sqlValueVar>
        </valueMap>
</map>
<map>
        <valueMap>
                <dtVar>w</dtVar>
                <sqlValueVar type="xs:char">sex</sqlValueVar>
        </valueMap>
</map>
<body>
        SELECT name, surname, sex FROM actor
</body>
</mapping>
```

6.3 Implementazione in Protege

Il passo successivo del lavoro è stato riportare tutte le inclusion assertions della nostra TBox in Protégé, eccezion fatta per le inclusioni relative ai role attribute, dato che non è possibile modellarli in Protégé.
L'interfaccia di Protégé presenta delle schede: OWL Classes, dove è possibile definire i Concept, con relative partecipazioni obbligatorie, e ISA fra essi, ad esempio per esprimere:

PERSON ⊑ δ (name)
PERSON ⊑ δ (surname)
PERSON ⊑ δ (birthDate)
PERSON ⊑ δ (birthName)
PERSON ⊑ ∃ (cityOfBirth)



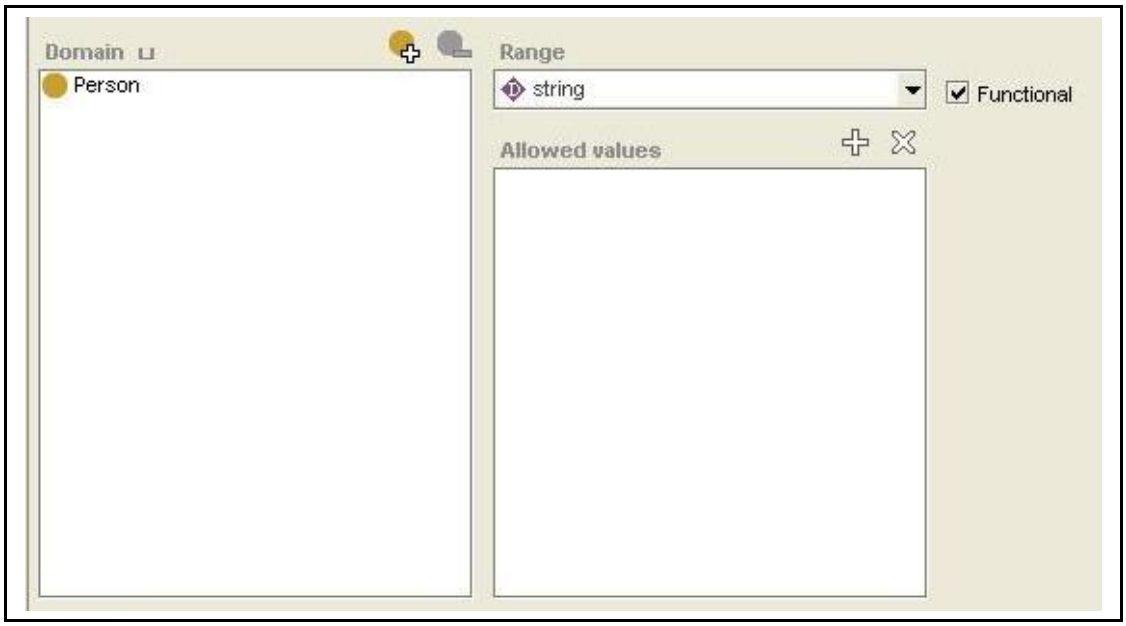
ACTOR ⊑ PERSON
ACTOR ⊑ ∃ (actor2character)

Actor ISA Person impone che ogni istanza di Actor erediti le proprietà di Person; questo

viene riportato nella schermata sotto la voce “Inherited”.

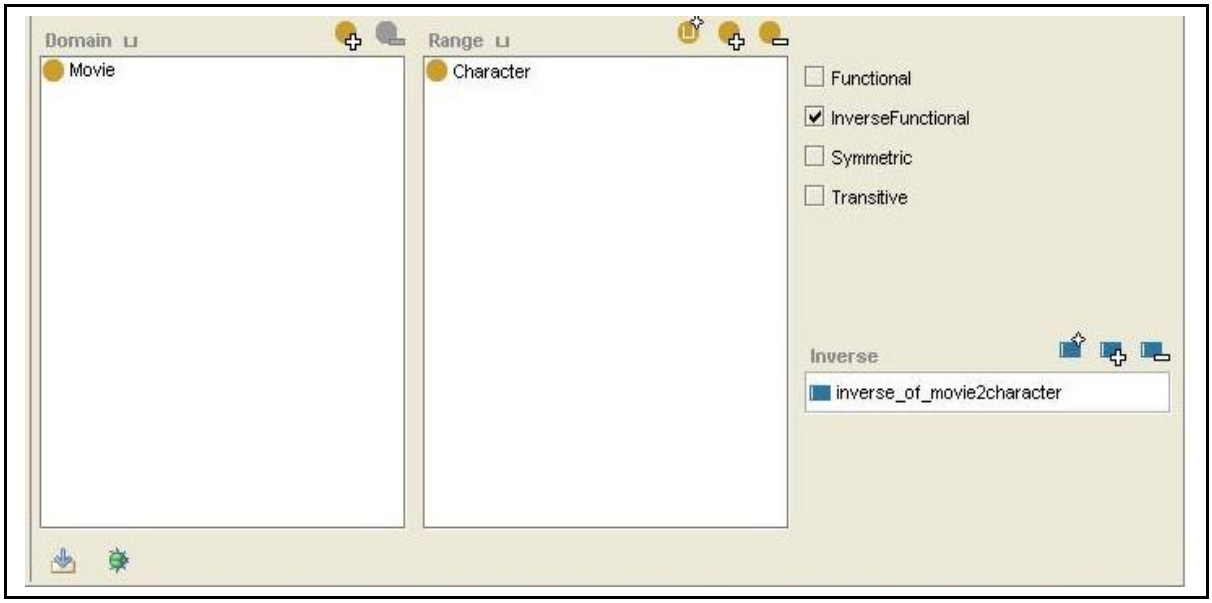
Nella scheda Property Browser è possibile definire i Concept Attribute (Datatype) e i Role (Object), ad esempio per esprimere:

δ (name) ⊑ PERSON
funct (name)
ρ (name) ⊑ xsd: string



Viene quindi definito il dominio di name, ovvero Person, e viene specificata la funzionalità di tale attributo.

∃ (movie2character) ⊑ MOVIE
MOVIE ⊑ ∃ (movie2character)



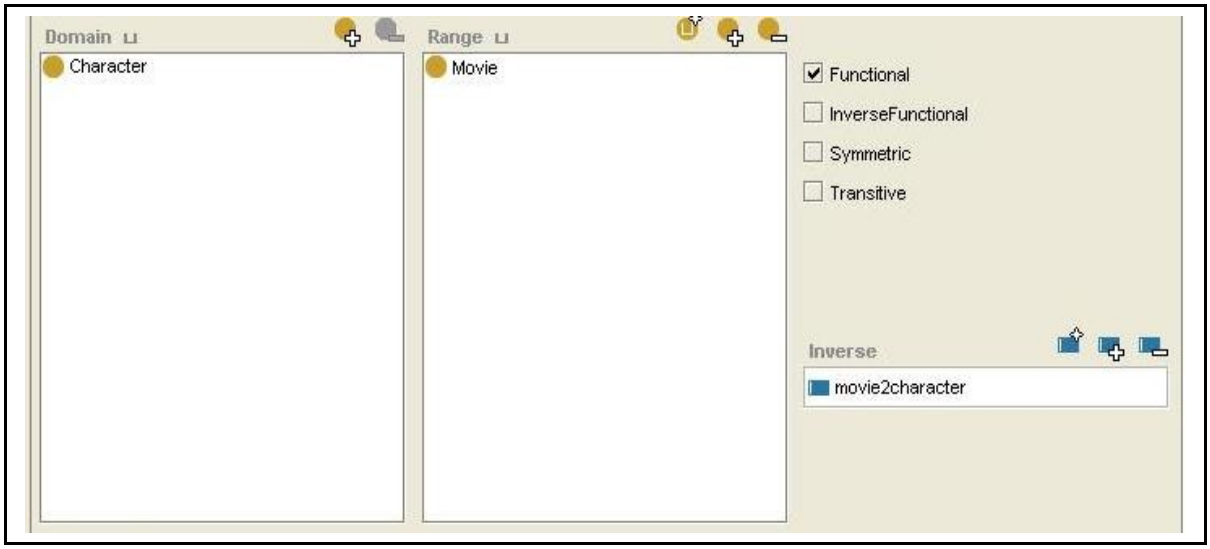
Questa schermata ci mostra come vengono definiti dominio e codominio di un Role. Il problema in Protégé è che non si possono fare asserzioni dirette Role inversi, ad esempio:

$\exists$ (movie2character)- $\sqsubseteq$ CHARACTER

Esiste però un modo per raggirare il problema: si crea un nuovo Role, inverso appunto,che presenta come dominio il codominio del Role di partenza, e come codominio il dominio di quest'ultimo.

Quindi:

$\exists$ (movie2character)- $\sqsubseteq$ CHARACTER
funct (movie2character)-



La scheda degli Object Property visualizza accanto ad ogni Role la presenza di un eventuale Inverse:

6.4 Interrogazioni e test

Configurazione hardware delle macchine di test:

1) Intel Pentium 4 3.2Ghz, 2GB RAM, 200GB hard disk SATA 7200RPM

2) Intel Pentium 3 800Mhz, 128Mb RAM, hard disk 4200RPM

3) Intel Celerom ULV 1.2GHz, 512MB RAM, Solid State Hard Drive

Configurazione software:

- DBMS mySQL Server 5.0.21
- Eclipse 3.2
- Protege 3.1
- OBDA Plugin 04-09-2008

I primi test sono stati effettuati sulle macchine meno potenti, riscontrando tempi di risposta molto elevati per le query più semplici, mentre le query più complesse che implicavano reasoning oppure elevata quantità di dati, non terminavano.

Attribuendo quindi la causa di tale lentezza alle macchine 2 e 3, abbiamo effettuato gli stessi test sulla macchina 1, ben più potente delle altre, riscontrando un miglioramento notevole per le query semplici, mentre le altre restavano ancora problematiche.

A questo punto abbiamo iniziato ad ipotizzare che il problema non fosse legato alla potenza della macchina di test, bensì alla complessità delle query sql generate dal reasoner e dalla particolare configurazione dei dati.

6.4.1 Consistency Check

Il consistency check sull'ontologia impiega un tempo elevatissimo mandando in stallo il processo server di mySQL.
Per verificare la funzionalità del solo Concept Attribute *name* di Person viene generata la seguente query SQL (le altre funzionalità sono state eliminate):

```
SELECT ALL alias_0.term2 FROM ((SELECT ALL
CONCAT('getPersonObject(',name,',', surname, ')') AS term1,name AS
term2 FROM (SELECT name,surname FROM composer) DummyTable) UNION ALL
(SELECT ALL CONCAT('getPersonObject(',name,',', surname, ')') AS
term1,name AS term2 FROM (SELECT name,surname FROM designer)
DummyTable) UNION ALL (SELECT ALL CONCAT('getPersonObject(',name,',',
surname, ')') AS term1,name AS term2 FROM (SELECT name,surname,sex
FROM actor) DummyTable) UNION ALL (SELECT ALL
CONCAT('getPersonObject(',name,',', surname, ')') AS term1,name AS
term2 FROM (SELECT name,surname FROM director) DummyTable) UNION ALL
(SELECT ALL CONCAT('getPersonObject(',name,',', surname, ')') AS
term1,name AS term2 FROM (SELECT name,surname FROM proddesigner)
DummyTable) UNION ALL (SELECT ALL CONCAT('getPersonObject(',name,',',
surname, ')') AS term1,name AS term2 FROM (SELECT name,surname FROM
cinematographer) DummyTable) UNION ALL (SELECT ALL
CONCAT('getPersonObject(',name,',', surname, ')') AS term1,name AS
term2 FROM (SELECT name,surname FROM writer) DummyTable) UNION ALL
(SELECT ALL CONCAT('getPersonObject(',name,',', surname, ')') AS
term1,name AS term2 FROM (SELECT name,surname FROM editor) DummyTable)
UNION ALL (SELECT ALL CONCAT('getPersonObject(',name,',', surname,
')') AS term1,name AS term2 FROM (SELECT name,surname FROM producer)
DummyTable)) alias_0 , ((SELECT ALL
CONCAT('getPersonObject(',name,',', surname, ')') AS term1,name AS
term2 FROM (SELECT name,surname FROM composer) DummyTable) UNION ALL
(SELECT ALL CONCAT('getPersonObject(',name,',', surname, ')') AS
term1,name AS term2 FROM (SELECT name,surname FROM designer)
DummyTable) UNION ALL (SELECT ALL CONCAT('getPersonObject(',name,',',
surname, ')') AS term1,name AS term2 FROM (SELECT name,surname,sex
FROM actor) DummyTable) UNION ALL (SELECT ALL
CONCAT('getPersonObject(',name,',', surname, ')') AS term1,name AS
term2 FROM (SELECT name,surname FROM director) DummyTable) UNION ALL
(SELECT ALL CONCAT('getPersonObject(',name,',', surname, ')') AS
term1,name AS term2 FROM (SELECT name,surname FROM proddesigner)
DummyTable) UNION ALL (SELECT ALL CONCAT('getPersonObject(',name,',',
surname, ')') AS term1,name AS term2 FROM (SELECT name,surname FROM
cinematographer) DummyTable) UNION ALL (SELECT ALL
CONCAT('getPersonObject(',name,',', surname, ')') AS term1,name AS
term2 FROM (SELECT name,surname FROM writer) DummyTable) UNION ALL
(SELECT ALL CONCAT('getPersonObject(',name,',', surname, ')') AS
term1,name AS term2 FROM (SELECT name,surname FROM editor) DummyTable)
UNION ALL (SELECT ALL CONCAT('getPersonObject(',name,',', surname,
```

```
')') AS term1,name AS term2 FROM (SELECT name,surname FROM producer)
DummyTable)) alias_1 WHERE alias_0.term2<>alias_1.term2  AND
alias_0.term1=alias_1.term1
```

Da questo momento in poi, per motivi di efficienza, per proseguire il testing su Mastro abbiamo chiamato la funzione `expandAndEvaluateWithoutConsistencyCheck(ucq)` mentre in Protege sono state eliminate dalla TBox tutte le asserzioni di funzionalità (funct).

6.4.2 CQ q(x,y): movie2character(x,y)

Conjunctive Query eseguita su MastroI:

q(x,y): movie2character(x,y)

con il seguente mapping:

movie2character(x,y) ∧ Movie(x) ∧ Character(y)
←
S**ELECT** movie, role **FROM** actor2movie

x → mov(movie)
y → char(role)

Tale query non viene espansa dal reasoner quindi è di complessità minima, tuttavia la tabella del DB actor2movie è di grandezza considerevole (5.438.771 tuple, 130MB dati + 85MB indice).

La query SQL generata è:

```
select distinct alias_0.term1, alias_0.term2 from (

select distinct concat('mov(',movie,')') as term1,
concat('char(',role,')') as term2 from actor2movie) alias_0
```

Per escludere qualsiasi problema dovuto al reasoner oppure ai driver JDBC abbiamo eseguito la stessa query direttamente tramite l'interfaccia di mySQL Server. Dopo alcune ore di elaborazione e 20GB di file temporanei scritti su disco abbiamo bloccato la query.

Per individuare il problema sono state testate alcune query create a partire dalla precedente apportando alcune modifiche.

Eliminazione dell'operatore DISTINCT:

```
select alias_0.term1, alias_0.term2 from (

select concat('mov(',movie,')') as term1, concat('char(',role,')') as
term2 from actor2movie) alias_0
```

La query senza distinct termina restituendo il risultato corretto in soli 297 secondi (testata sulla macchina 1)

In seguito abbiamo riscritto la query in modo più pulito, eliminando l'annidamento:

```
select concat('mov(',movie,')') as term1, concat('char(',role,')') as
term2 from actor2movie
```

Termina in 100 secondi circa. Comunque l'annidamento non comporta peggioramenti sensibili nell'efficienza.

Modifica a DIG-Mastro:
in seguito a questo risultato empirico è stata apportata una modifica al DIG Server in modo che Mastro nell'unfolding della query non inserisca più l'operatore SQL DISTINCT.
Tutte le query testate da questo momento in poi terminano in tempi accettabili.

6.4.3 Queries SPARQL

Riportiamo in seguito alcune delle queries SPQRQL testate su Protege

Nomi e Cognomi di tutte le persone:

```
SELECT  $y $z  WHERE {$x rdf:type 'Person'.
                $x :name $y.
                $x :surname $z
                }
```

Nomi e cognomi di tutti gli Actor:

```
SELECT  $y $z  WHERE {$x rdf:type 'Actor'.
                $x :name $y.
                $x :surname $z
                }
```

Identificatori dei film e relativi character:

```
SELECT  $x $y  WHERE {$x: 'movie2character' $y
                }
```

Titolo del film 140438 e i personaggi del film (220sec):

```
SELECT $x $y WHERE { 'getMovieObject(140438)' :movieTitle $x.
                        'getMovieObject(140438)' :movie2character $y
                }
```

Unfolding della query precendente:

```
SELECT ALL alias_1.term2, alias_0.term2 FROM (SELECT ALL
CONCAT('getMovieObject(',movie, ')') AS term1,CONCAT('getCharacterObject(',role,
')') AS term2 FROM (SELECT movie,role FROM actor2movie) DummyTable) alias_0 ,
((SELECT ALL CONCAT('getMovieObject(',id, ')') AS term1,title AS term2 FROM
(SELECT
movies.id,movies.title,movies.year,movies.genre,movies.mpaa,movies.runtimes,movies.
colortype,movies.sound FROM movies where title like '%(TV)%') DummyTable)
UNION ALL (SELECT ALL CONCAT('getMovieObject(',id, ')') AS term1,title AS
term2 FROM (SELECT
movies.id,movies.title,movies.year,movies.genre,movies.mpaa,movies.runtimes,movies.
colortype,movies.sound FROM movies) DummyTable)) alias_1 WHERE
alias_0.term1='getMovieObject(140438)'  AND
alias_1.term1='getMovieObject(140438)'
```

6.4.4 Queries EQL

Tutte le persone:

```
SELECT person.x, person.y,person.z
FROM sparqltable( SELECT $x $y $z
                  WHERE {    $x rdf:type 'Person'.
                             $x name $y.
                             $x surname $z }) person;
```

Tutti i film che NON sono serie-tv:

```
SELECT movies.x
FROM sparqltable( SELECT $x
                        WHERE {$x rdf:type 'Movie'}) movies
WHERE movies.x NOT IN (
                  SELECT series.y
                  FROM sparqltable ( SELECT $y
                                     WHERE { $y rdf:type 'TVSerie'}) series );
```

Appendice A: IMDB TBox

Versione1.5 – 11/04/08

PERSON ⊑ δ (name)
δ (name) ⊑ PERSON
funct (name)
PERSON ⊑ δ (surname)
δ (surname) ⊑ PERSON
funct (surname)
PERSON ⊑ δ (birthDate)
δ (birthDate) ⊑ PERSON
funct (birthDate)
PERSON ⊑ δ (birthName)
δ (birthName) ⊑ PERSON
funct (birthName)
δ (trademark) ⊑ PERSON
funct (trademark)
δ (trivia) ⊑ PERSON
funct (trivia)
δ (personalQuotes) ⊑ PERSON
funct (personalQuotes)
δ (miniBiography) ⊑ PERSON
funct (miniBiography)
δ (height) ⊑ PERSON
funct (height)
δ (nick) ⊑ PERSON
funct (nick)
δ (lastAppearance) ⊑ PERSON
funct (lastAppearance)
δ (alternateNames) ⊑ PERSON
δ (otherWorks) ⊑ PERSON

∃ (cityOfBirth) ⊑ PERSON
∃ (cityOfBirth)- ⊑ CITY
PERSON ⊑ ∃ (cityOfBirth)
funct (cityOfBirth)

∃ (salary) ⊑ PERSON
∃ (salary)- ⊑ MOVIE
salary ⊑ δ (money)

δ (money) ⊑ salary
funct (money)
∃ δ (money) ⊑ PERSON
∃ δ (money)- ⊑ MOVIE

∃ (songWrittenBy) ⊑ PERSON
∃ (songWrittenBy)- ⊑ SONG
SONG ⊑ ∃ (songWrittenBy)-
funct (songWrittenBy)-

∃ (songPerformedBy) ⊑ PERSON
∃ (songPerformedBy)- ⊑ SONG
SONG ⊑ ∃ (songPerformedBy)-
funct (songPerformedBy)-

∃ (songSingedBy) ⊑ PERSON
∃ (songSingedBy)- ⊑ SONG
SONG ⊑ ∃ (songSingedBy)-
funct (songSingedBy)-

∃ (directedBy) ⊑ PERSON
∃ (directedBy)- ⊑ MOVIE
MOVIE ⊑ ∃ (directedBy)-

∃ (writtenBy) ⊑ PERSON
∃ (writtenBy)- ⊑ MOVIE
MOVIE ⊑ ∃ (writtenBy)-
δ (role1) ⊑ writtenBy
funct (role1)
∃ δ (role1) ⊑ PERSON
∃ δ (role1)- ⊑ MOVIE

∃ (producedBy) ⊑ PERSON
∃ (producedBy)- ⊑ MOVIE
MOVIE ⊑ ∃ (producedBy)-
δ (role2) ⊑ producedBy
funct (role2)
∃ δ (role2) ⊑ PERSON
∃ δ (role2)- ⊑ MOVIE

∃ (editedBy) ⊑ PERSON
∃ (editedBy)- ⊑ MOVIE
MOVIE ⊑ ∃ (editedBy)-

∃ (composedBy) ⊑ PERSON
∃ (composedBy)- ⊑ MOVIE

MOVIE ⊑ ∃ (composedBy)-

∃ (casting) ⊑ PERSON
∃ (casting)- ⊑ MOVIE
MOVIE ⊑ ∃ (casting)-

∃ (cinematography) ⊑ PERSON
∃ (cinematography)- ⊑ MOVIE
MOVIE ⊑ ∃ (cinematography)-

∃ (customeDesign) ⊑ PERSON
∃ (customeDesign)- ⊑ MOVIE
MOVIE ⊑ ∃ (customeDesign)-

∃ (makeupDept) ⊑ PERSON
∃ (makeupDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (makeupDept)-
δ (role3) ⊑ makeupDept
funct (role3)
∃ δ (role3) ⊑ PERSON
∃ δ (role3)- ⊑ MOVIE

∃ (productionDept) ⊑ PERSON
∃ (productionDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (productionDept)-
δ (role4) ⊑ productionDept
funct (role4)
∃ δ (role4) ⊑ PERSON
∃ δ (role4)- ⊑ MOVIE

∃ (setDecoration) ⊑ PERSON
∃ (setDecoration)- ⊑ MOVIE
MOVIE ⊑ ∃ (setDecoration)-

∃ (artDirection) ⊑ PERSON
∃ (artDirection)- ⊑ MOVIE
MOVIE ⊑ ∃ (artDirection)-

∃ (productionDesign) ⊑ PERSON
∃ (productionDesign)- ⊑ MOVIE
MOVIE ⊑ ∃ (productionDesign)-

∃ (composer) ⊑ PERSON
∃ (composer)- ⊑ MOVIE
MOVIE ⊑ ∃ (composer)-

∃ (artDept) ⊑ PERSON

∃ (artDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (artDept)-
δ (role5) ⊑ artDept
funct (role5)
∃ δ (role5) ⊑ PERSON
∃ δ (role5)- ⊑ MOVIE

∃ (soundDept) ⊑ PERSON
∃ (soundDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (soundDept)-
δ (role6) ⊑ soundDept
funct (role6)
∃ δ (role6) ⊑ PERSON
∃ δ (role6)- ⊑ MOVIE

∃ (musicDept) ⊑ PERSON
∃ (musicDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (musicDept)-
δ (role7) ⊑ musicDept
funct (role7)
∃ δ (role7) ⊑ PERSON
∃ δ (role7)- ⊑ MOVIE

∃ (editorialDept) ⊑ PERSON
∃ (editorialDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (editorialDept)-
δ (role8) ⊑ editorialDept
funct (role8)
∃ δ (role8) ⊑ PERSON
∃ δ (role8)- ⊑ MOVIE

∃ (castingDept) ⊑ PERSON
∃ (castingDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (castingDept)-
δ (role9) ⊑ castingDept
funct (role9)
∃ δ (role9) ⊑ PERSON
∃ δ (role9)- ⊑ MOVIE

∃ (costumeAndWardrobeDept) ⊑ PERSON
∃ (costumeAndWardrobeDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (costumeAndWardrobeDept)-
δ (role10) ⊑ costumeAndWardrobeDept

funct (role10)
∃ δ (role10) ⊑ PERSON
∃ δ (role10)- ⊑ MOVIE

∃ (animationDept) ⊑ PERSON
∃ (animationDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (animationDept)-
δ (role11) ⊑ animationDept
funct (role11)
∃ δ (role11) ⊑ PERSON
∃ δ (role11)- ⊑ MOVIE

∃ (cameraAndElecticalDept) ⊑ PERSON
∃ (cameraAndElecticalDept)- ⊑ MOVIE
MOVIE ⊑ ∃ (cameraAndElecticalDept)-
δ (role12) ⊑ cameraAndElecticalDept
funct (role12)
∃ δ (role12) ⊑ PERSON
∃ δ (role12)- ⊑ MOVIE

∃ (stuntman) ⊑ PERSON
∃ (stuntman)- ⊑ MOVIE
MOVIE ⊑ ∃ (stuntman)-
δ (role13) ⊑ stuntman
funct (role13)
∃ δ (role13) ⊑ PERSON
∃ δ (role13)- ⊑ MOVIE

∃ (specialEffect) ⊑ PERSON
∃ (specialEffect)- ⊑ MOVIE
MOVIE ⊑ ∃ (specialEffect)-
δ (role14) ⊑ specialEffect
funct (role14)
∃ δ (role14) ⊑ PERSON
∃ δ (role14)- ⊑ MOVIE

∃ (visualEffect) ⊑ PERSON
∃ (visualEffect)- ⊑ MOVIE
MOVIE ⊑ ∃ (visualEffect)-
δ (role15) ⊑ visualEffect
funct (role15)
∃ δ (role15) ⊑ PERSON
∃ δ (role15)- ⊑ MOVIE

∃ (assistantDirector) ⊑ PERSON
∃ (assistantDirector)- ⊑ MOVIE
MOVIE ⊑ ∃ (assistantDirector)-

δ (role16) ⊑ assistantDirector
funct (role16)
∃ δ (role16) ⊑ PERSON
∃ δ (role16)- ⊑ MOVIE

∃ (otherCrew) ⊑ PERSON
∃ (otherCrew)- ⊑ MOVIE
MOVIE ⊑ ∃ (otherCrew)-
δ (role17) ⊑ otherCrew
funct (role17)
∃ δ (role17) ⊑ PERSON
∃ δ (role17)- ⊑ MOVIE

∃ (award2person)- ⊑ PERSON
∃ (award2person) ⊑ AWARD
AWARD ⊑ ∃ (award2person)
funct (award2person)

∃ (host)- ⊑ PERSON
∃ (host) ⊑ CEREMONY
CEREMONY ⊑ ∃ (host)
funct (host)

MOVIE ⊑ δ (movieTitle)
δ (movieTitle) ⊑ MOVIE
funct (movieTitle)
MOVIE ⊑ δ (year)
δ (year) ⊑ MOVIE
funct (year)
MOVIE ⊑ δ (genre)
δ (genre) ⊑ MOVIE
MOVIE ⊑ δ (tagline)
δ (tagline) ⊑ MOVIE
funct (tagline)
MOVIE ⊑ δ (plotOutline)
δ (plotOutline) ⊑ MOVIE
funct (plotOutline)
MOVIE ⊑ δ (plotSynopsis)
δ (plotSynopsis) ⊑ MOVIE
funct (plotSynopsis)
δ (plotKeyword) ⊑ MOVIE

δ (mpaa) ⊑ MOVIE
funct (mpaa)
MOVIE ⊑ δ (runTime)
δ (runTime) ⊑ MOVIE
funct (runTime)
δ (color) ⊑ MOVIE
funct (color)
δ (camera) ⊑ MOVIE
δ (laboratory) ⊑ MOVIE
funct (laboratory)
MOVIE ⊑ δ (language)
δ (language) ⊑ MOVIE
funct (language)
δ (parentsGuide) ⊑ MOVIE
funct (parentsGuide)
δ (negativeFormat) ⊑ MOVIE
funct (negativeFormat)
δ (printedFormat) ⊑ MOVIE
funct (printedFormat)
δ (aspectRatio) ⊑ MOVIE
funct (aspectRatio)
δ (soundMix) ⊑ MOVIE
δ (certificate) ⊑ MOVIE
δ (movieTrivia) ⊑ MOVIE
funct (movieTrivia)
δ (tv) ⊑ MOVIE
funct (tv)
δ (video) ⊑ MOVIE
funct (video)
δ (goofs) ⊑ MOVIE
funct (goofs)
δ (quotes) ⊑ MOVIE
funct (quotes)
δ (cinematographicProcess) ⊑ MOVIE

∃ (releasedIn) ⊑ MOVIE
∃ (releasedIn)- ⊑ COUNTRY
MOVIE ⊑ ∃ (releasedIn)
releasedIn ⊑ δ (releaseDate)
δ (releaseDate) ⊑ releasedIn
funct (releaseDate)
∃ δ (releaseDate) ⊑ MOVIE
∃ δ (releaseDate)- ⊑ COUNTRY
releasedIn ⊑ δ (akaname)
δ (akaname) ⊑ releasedIn
funct (akaname)
∃ δ (akaname) ⊑ MOVIE

∃ δ (akaname)- ⊑ COUNTRY

∃ (producedIn) ⊑ MOVIE
∃ (producedIn)- ⊑ COUNTRY
MOVIE ⊑ ∃ (producedIn)
funct(producedIn)

∃ (filmedIn) ⊑ MOVIE
∃ (filmedIn)- ⊑ LOCATION
MOVIE ⊑ ∃ (filmedIn)

∃ (followedBy) ⊑ MOVIE
∃ (followedBy)- ⊑ MOVIE
funct (followedBy)
funct (followedBy)-

∃ (remakeOf) ⊑ MOVIE
∃ (remakeOf)- ⊑ MOVIE
funct (remakeOf)
funct (remakeOf)-

∃ (spinOff) ⊑ MOVIE
∃ (spinOff)- ⊑ MOVIE
funct (spinOff)

∃ (references) ⊑ MOVIE
∃ (references)- ⊑ MOVIE

∃ (referenced) ⊑ MOVIE
∃ (referenced)- ⊑ MOVIE

∃ (features) ⊑ MOVIE
∃ (features)- ⊑ MOVIE

∃ (featuredIn) ⊑ MOVIE
∃ (featuredIn)- ⊑ MOVIE

∃ (soundtrack) ⊑ MOVIE
∃ (soundtrack)- ⊑ SONG

∃ (movie2character) ⊑ MOVIE
∃ (movie2character)- ⊑ CHARACTER
MOVIE ⊑ ∃ (movie2character)
CHARACTER ⊑ ∃ (movie2character)-
funct (movie2character)-

∃ (productionCompany) ⊑ MOVIE
∃ (productionCompany)- ⊑ COMPANY
MOVIE ⊑ ∃ (productionCompany)
δ (role18) ⊑ productionCompany
funct (role18)
∃ δ (role18) ⊑ MOVIE
∃ δ (role18)- ⊑ COMPANY

∃ (distributionCompany) ⊑ MOVIE
∃ (distributionCompany)- ⊑ COMPANY
MOVIE ⊑ ∃ (distributionCompany)
distributionCompany ⊑ δ (distributionYear)
δ (distributionYear) ⊑ distributionCompany
funct (distributionYear)
∃ δ (distributionYear) ⊑ MOVIE
∃ δ (distributionYear)- ⊑ COMPANY
distributionCompany ⊑ δ (type)
δ (type) ⊑ distributionCompany
funct (type)
∃ δ (type) ⊑ MOVIE
∃ δ (type)- ⊑ COMPANY

∃ (sfxCompany) ⊑ MOVIE
∃ (sfxCompany)- ⊑ COMPANY
MOVIE ⊑ ∃ (sfxCompany)
δ (role19) ⊑ sfxCompany
funct (role19)
∃ δ (role19) ⊑ MOVIE
∃ δ (role19)- ⊑ COMPANY

∃ (otherCompany) ⊑ MOVIE
∃ (otherCompany)- ⊑ COMPANY
MOVIE ⊑ ∃ (otherCompany)
δ (role20) ⊑ otherCompany
funct (role20)
∃ δ (role20) ⊑ MOVIE
∃ δ (role20)- ⊑ COMPANY

∃ (award2movie)- ⊑ MOVIE
∃ (award2movie) ⊑ AWARD
AWARD ⊑ ∃ (award2movie)
funct (award2movie)

TVSERIE ⊑ MOVIE
TVSERIE ⊑ δ (seasonNo)

δ (seasonNo) ⊑ TVSERIE
(funct seasonNo)
TVSERIE ⊑ δ (episodeNo)
δ (episodeNo) ⊑ TVSERIE
(funct episodeNo)
TVSERIE ⊑ δ (episodeTitle)
δ (episodeTitle) ⊑ TVSERIE
(funct episodeTitle)
TVSERIE ⊑ δ (resume)
δ (resume) ⊑ TVSERIE
(funct resume)
TVSERIE ⊑ δ (originalAirDate)
δ (originalAirDate) ⊑ TVSERIE
(funct originalAirDate)

LOCATION ⊑ δ (address)
δ (address) ⊑ LOCATION
funct (address)
LOCATION ⊑ δ (pointOfInterest)
δ (pointOfInterest) ⊑ LOCATION
funct (pointOfInterest)

∃ (loc2city) ⊑ LOCATION
∃ (loc2city)- ⊑ CITY
LOCATION ⊑ ∃ (loc2city)
funct (loc2city)

∃ (located)- ⊑ LOCATION
∃ (located) ⊑ CEREMONY
CEREMONY ⊑ ∃ (located)
funct (located)

CITY ⊑ δ (cityName)
δ (cityName) ⊑ CITY
funct (cityName)
CITY ⊑ δ (region)
δ (region) ⊑ CITY
funct (region)
CITY ⊑ δ (state)
δ (state) ⊑ CITY
funct (state)
∃ (city2country) ⊑ CITY
∃ (city2country)- ⊑ COUNTRY
CITY ⊑ ∃ (city2country)

funct (city2country)

COUNTRY ⊑ δ (countryName)
δ (countryName) ⊑ COUNTRY
funct (countryName)

COMPANY ⊑ δ (companyName)
δ (companyName) ⊑ COMPANY
funct (companyName)

SONG ⊑ δ (songTitle)
δ (songTitle) ⊑ SONG
funct (songTitle)
δ (copyrightInfo) ⊑ SONG
funct (copyrightInfo)

ACTOR ⊑ PERSON
ACTOR ⊑ δ (sex)
δ (sex) ⊑ ACTOR
funct (sex)
∃ (actor2character) ⊑ ACTOR
∃ (actor2character)- ⊑ CHARACTER
ACTOR ⊑ ∃ (actor2character)
CHARACTER ⊑ ∃ (actor2character)-
funct (actor2character)-

CHARACTER ⊑ δ (characterName)
δ (characterName) ⊑ CHARACTER
funct (characterName)
CHARACTER ⊑ δ (characterSurname)
δ (characterSurname) ⊑ CHARACTER
funct (characterSurname)

AWARD ⊑ δ (category)
δ (category) ⊑ AWARD
funct (category)
AWARD ⊑ δ (result)
δ (result) ⊑ AWARD
funct (result)
AWARD ⊑ δ (motivation)
δ (motivation) ⊑ AWARD
funct (motivation)

∃ (award2ceremony) ⊑ AWARD
∃ (award2ceremony)- ⊑ CEREMONY
AWARD ⊑ ∃ (award2ceremony)
CEREMONY ⊑ ∃ (award2ceremony)-
funct (award2ceremony)

CEREMONY ⊑ δ (ceremonyDate)
δ (ceremonyDate) ⊑ CEREMONY
funct (ceremonyDate)
CEREMONY ⊑ δ (awardType)
δ (awardType) ⊑ CEREMONY
funct (awardType)
CEREMONY ⊑ δ (academy)
δ (academy) ⊑ CEREMONY
funct (academy)

ρ (name) ⊑ xsd: string
ρ (surname) ⊑ xsd: string
ρ (birthDate) ⊑ xsd: date
ρ (birthName) ⊑ xsd: string
ρ (trademark) ⊑ xsd: string
ρ (trivia) ⊑ xsd: string
ρ (movieTrivia) ⊑ xsd: string
ρ (personalQuotes) ⊑ xsd: string
ρ (miniBiography) ⊑ xsd: string
ρ (height) ⊑ xsd: string
ρ (nick) ⊑ xsd: string
ρ (lastAppearance) ⊑ xsd: string
ρ (alternateNames) ⊑ xsd: string
ρ (otherWorks) ⊑ xsd: string
ρ (money) ⊑ xsd: string
ρ (year) ⊑ xsd: int
ρ (genre) ⊑ xsd: string
ρ (tagline) ⊑ xsd: string
ρ (plotOutline) ⊑ xsd: string
ρ (plotSynopsis) ⊑ xsd: string
ρ (plotKeyword) ⊑ xsd: string
ρ (mpaa) ⊑ xsd: string
ρ (runTime) ⊑ xsd: int
ρ (color) ⊑ xsd: string

ρ (camera) ⊑ xsd: string
ρ (laboratory) ⊑ xsd: string
ρ (language) ⊑ xsd: string
ρ (parentsGuide) ⊑ xsd: string
ρ (negativeFormat) ⊑ xsd: string
ρ (printedFormat) ⊑ xsd: string
ρ (aspectRatio) ⊑ xsd: string
ρ (soundMix) ⊑ xsd: string
ρ (certificate) ⊑ xsd: string
ρ (tv) ⊑ xsd: string
ρ (video) ⊑ xsd: string
ρ (goofs) ⊑ xsd: string
ρ (quotes) ⊑ xsd: string
ρ (cinematographicProcess) ⊑ xsd: string
ρ (seasonNo) ⊑ xsd: int
ρ (episodeNo) ⊑ xsd: int
ρ (episodeTitle) ⊑ xsd: string
ρ (resume) ⊑ xsd: string
ρ (originalAirDate) ⊑ xsd: date
ρ (address) ⊑ xsd: string
ρ (pointOfInterest) ⊑ xsd: string
ρ (region) ⊑ xsd: string
ρ (state) ⊑ xsd: string
ρ (movieTitle) ⊑ xsd: string
ρ (songTitle) ⊑ xsd: string
ρ (copyrightInfo) ⊑ xsd: string
ρ (sex) ⊑ xsd: string
ρ (category) ⊑ xsd: string
ρ (result) ⊑ xsd: string
ρ (motivation) ⊑ xsd: string
ρ (awardType) ⊑ xsd: string
ρ (academy) ⊑ xsd: string
ρ (type) ⊑ xsd: string
ρ (akaname) ⊑ xsd: string
ρ (distributionYear) ⊑ xsd: int
ρ (releaseDate) ⊑ xsd: date
ρ (ceremonyDate) ⊑ xsd: date
ρ (cityName) ⊑ xsd: string
ρ (countryName) ⊑ xsd: string
ρ (companyName) ⊑ xsd: string
ρ (characterName) ⊑ xsd: string
ρ (characterSurname) ⊑ xsd: string
ρ (role1) ⊑ xsd: string
ρ (role2) ⊑ xsd: string
ρ (role3) ⊑ xsd: string
ρ (role4) ⊑ xsd: string
ρ (role5) ⊑ xsd: string

ρ (role6) ⊑ xsd: string
ρ (role7) ⊑ xsd: string
ρ (role8) ⊑ xsd: string
ρ (role9) ⊑ xsd: string
ρ (role10) ⊑ xsd: string
ρ (role11) ⊑ xsd: string
ρ (role12) ⊑ xsd: string
ρ (role13) ⊑ xsd: string
ρ (role14) ⊑ xsd: string
ρ (role15) ⊑ xsd: string
ρ (role16) ⊑ xsd: string
ρ (role17) ⊑ xsd: string
ρ (role18) ⊑ xsd: string
ρ (role19) ⊑ xsd: string
ρ (role20) ⊑ xsd: string

Appendice B: IMDB Mappings

ACTOR

1)
Actor(x) $\wedge$ name(x,y) $\wedge$ surname(x,z) $\wedge$ sex(x,w)
$\leftarrow$
SELECT name, surname, sex **FROM** actor

x $\rightarrow$ per(name, surname)
y $\rightarrow$ name xsd:String
z $\rightarrow$ surname xsd: String
z $\rightarrow$ sex xsd: Char

PERSON

2)
Person(x) $\wedge$ name(x,y) $\wedge$ surname(x,z)
$\leftarrow$
SELECT name, surname **FROM** cinematographer

x $\rightarrow$ per(name,surname)
y $\rightarrow$ name xsd:String
z $\rightarrow$ surname xsd: String

3)
Person(x) $\wedge$ name(x,y) $\wedge$ surname(x,z)
$\leftarrow$
SELECT name, surname **FROM** composer

x $\rightarrow$ per(name,surname)
y $\rightarrow$ name xsd:String
z $\rightarrow$ surname xsd: String

4)
Person(x) $\wedge$ name(x,y) $\wedge$ surname(x,z)
$\leftarrow$
SELECT name, surname **FROM** designer

x → per(name,surname)
y → name xsd:String
z → surname xsd: String

5)
Person(x) ∧ name(x,y) ∧ surname(x,z)
←
SELECT name, surname **FROM** director

x → per(name,surname)
y → name xsd:String
z → surname xsd: String

6)
Person(x) ∧ name(x,y) ∧ surname(x,z)
←
SELECT name, surname **FROM** editor

x → per(name,surname)
y → name xsd:String
z → surname xsd: String

7)
Person(x) ∧ name(x,y) ∧ surname(x,z)
←
SELECT name, surname **FROM** proddesigner

x → per(name,surname)
y → name xsd:String
z → surname xsd: String

8)
Person(x) ∧ name(x,y) ∧ surname(x,z)
←
SELECT name, surname **FROM** producer

x → per(name,surname)
y → name xsd:String

z → surname xsd: String

9)
Person(x) ∧ name(x,y) ∧ surname(x,z)
←
SELECT name, surname **FROM** writer

x → per(name,surname)
y → name xsd:String
z → surname xsd: String

10)
Person(x) ∧ alternateNames(x,y)
←
SELECT actor, akaname **FROM** actorsaka
x → per(actor)
y → akaname xsd:String

MOVIE

11)
Movie(x) ∧ movieTitle(x,y) ∧ year(x,z) ∧ genre(x,w) ∧ mpaa(x,a)∧
runTime(x,b) ∧ color(x,c) ∧ soundMix(x,d)
←
SELECT id, title, year, genre, mpaa, runtimes, colortype, sound
FROM movies

x → mov(ID)
y → title xsd:String
z → year xsd:int
w → genre xsd:String
a → mpaa xsd:String
b → runtimes xsd:String
c → colortype xsd:String
d → sound xsd:String

11bis TVSERIE)
TvSerie(x) ∧ movieTitle(x,y) ∧ episodeTitle(x,e) ∧ year(x,z) ∧
genre(x,w) ∧ mpaa(x,a)∧ runTime(x,b) ∧ color(x,c) ∧ soundMix(x,d)

←
SELECT id, SUBSTRING(SUBSTRING_INDEX(title, ':', 1),5) as titolo_serie, SUBSTRING_INDEX(title, ':', -1) as titolo_episodio year, genre, mpaa, runtimes, colortype, sound **FROM** movies
WHERE title like '%(TV)%'

x → mov(ID)
y → titolo_serie xsd:String
e → titolo_episodio xsd:String
z → year xsd:int
w → genre xsd:String
a → mpaa xsd:String
b → runtimes xsd:String
c → colortype xsd:String
d → sound xsd:String

12)
Movie(x) ∧ certificate(x,y)
←
SELECT certificate.movie, concat(certificate.country, ' : ', certificate.type) AS CERT
FROM certificate

x →mov(movie)
y→cert xsd:String

13)
Movie(x) ∧ plotOutline(x,y)
←
SELECT movies.id, plot.plot
FROM movies, plot2movie, plot
WHERE plot2movie.movie=movies.id AND plot2movie.plot=plot.id

x → mov(movies.id)
y → plot.plot xsd:String

14)
Movie(x) ∧ plotKeyword(x,y)
←
SELECT movies.id, key2movie.keyword
FROM movies, key2movie
WHERE key2movie.movie=movies.id

x → mov(movies.id)
y → key2movie.keyword xsd:String

15)
Movie(x) ∧ language(x,y)
←
SELECT movies.id, lang2movie.language
FROM movies, lang2movie
WHERE lang2movie.movie=movies.id

x → mov(movies.id)
y → lang2movie.language xsd:String

16)
Movie(x) ∧ tagline(x,y)
←
SELECT movies.id, tag.tag
FROM movies, tag2movie, tag
WHERE tag2movie.movie=movies.id AND tag2movie.tag=tag.id

x → mov(movies.id)
y → tag.tag xsd:String

17)
Movie(x) ∧ movieTrivia(x,y)
←

SELECT trivia2movies.movie,trivia.trivia **FROM** trivia,trivia2movies
WHERE trivia.id=trivia2movies.trivia

x →mov(trivia2movies.movie)
y→trivia.trivia xsd:String

18)
Movie(x) ∧ goofs(x,y)
←
SELECT goof2movie.movie, goof.goof **FROM** goof, goof2movie
WHERE goof2movie.goof=goof.id

x →mov(goof2movie.movie)
y→goof.goof xsd:String

CITY

19)
City(x) ∧ cityName(x,y)
←
SELECT city **FROM** location

x → city(city)
y → city xsd:String
COUNTRY

20)
Country(x) ∧ countryName **(x,y)**
←
SELECT name FROM country

x → country(name)
y → name xsd:String

COMPANY

21)
Company(x) ∧ companyName(x,y)
←
SELECT name **FROM** sfxcompany

x → company(name)
y → name xsd:String

22)
Company(x) ∧ companyName(x,y)
←
SELECT name **FROM** distributor

x → company(name)
y → name xsd:String

23)
Company(x) ∧ companyName(x,y)
←
SELECT name **FROM** prodcompany

x → company(name)
y → name xsd:String

SONG

24)
Song(x) ∧ songTitle(x,y)
←
SELECT name **FROM** soundtrack

x → song(name)
y → name xsd:String

SONGWRITTENBY

26)
songWrittenBy(x,y) ∧ Person(x) ∧ Song(y)
←
SELECT writer, name **FROM** soundtrack

x → per(writer)
y → song(name)

SONGPERFORMEDBY (ipotizziamo che il performer è il composer)
27)
songPerformedBy(x,y) ∧ Person(x) ∧ Song(y)
←
SELECT composer, name **FROM** soundtrack

x → per(composer)
y → song(name)

SONGSINGEDBY (ipotizzo che il singer è il musician)

28)
songSingedBy(x,y) ∧ Person(x) ∧ Song(y)
←
SELECT musician, name **FROM** soundtrack

x → per(musician)
y → song(name)

DIRECTEDBY

29)
directedBy(x,y) ∧ Person(x) ∧ Movie(y)
←
SELECT movie, directorN, directorS **FROM** directedBy

x → per(directorN, directorS)
y → mov(movie)

WRITTENBY

30)
writtenBy(x,y) ∧ Person(x) ∧ Movie(y)
←
SELECT movie, writerN, writerS **FROM** writerBy

x → per(writerN, writerS)
y → mov(movie)

PRODUCEDBY

31)
producedBy(x,y) ∧ Person(x) ∧ Movie(y)
←
SELECT movie, producerN, producerS **FROM** prodBy

x → per(producerN, producerS)
y → mov(movie)

PRODUCTIONDESIGN

32)
productionDesign(x,y) ∧ Person(x) ∧ Movie(y)
←
SELECT movie, proddesignerN, proddesignerS **FROM** proddesignby

x → per(proddesignerN, proddesignerS)
y → mov(movie)

EDITEDBY

33)
editedBy(x,y) ∧ Person(x) ∧ Movie(y)
←
SELECT movie, editorN,editorS **FROM** editby

x → per(editorN,editorS)
y → mov(movie)

COMPOSEDBY

34)
composedBy(x,y) ∧ Person(x) ∧ Movie(y)

←
SELECT movie, composerN, composerS **FROM** musicby

x → per(composerN, composerS)
y → mov(movie)

CINEMATOGRAPHY

35)
cinematography(x,y) ∧ Person(x) ∧ Movie(y)
←
SELECT movie, cinematographerN, cinematographerS **FROM** cinematographyby

x → per(cinematographerN, cinematographerS)
y → mov(movie)

COSTUMEDESIGN (Ipotizzo che il designer nel db è il costume designer)

36)
costumeDesign(x,y) ∧ Person(x) ∧ Movie(y)
←
SELECT movie, designerN, designerS **FROM** designer2movie

x → per(designerN,designerS)
y → mov(movie)

RELEASEIN

37)
releasedIn(x,y,z) ∧ Country(x) ∧ Movie(y) ∧ releaseDate(x,z)
←

SELECT movie, country, date **FROM** releasein

x → country(country)
y → mov(movie)
z → date xsd:date

FOLLOWEDBY

38)
followedBy(x,y) ∧ Movie(x) ∧ Movie(y)
←
SELECT id, movie
FROM movielinks, link, movies
WHERE movielinks.link=link.id AND movies.title=link.followed

x → mov(movie)
y → mov(id)

REMAKEOF

39)
remakeOf(x,y) ∧ Movie(x) ∧ Movie(y)
←
SELECT id, movie
FROM movielinks, link, movies
WHERE movielinks.link=link.id AND movies.title=link.remake

x → mov(movie)
y → mov(id)

SPINOFF

40)
spinOff(x,y) ∧ Movie(x) ∧ Movie(y)
←

SELECT id, movie
FROM movielinks, link, movies
WHERE movielinks.link=link.id AND movies.title=link.spin

x → mov(movie)
y → mov(id)

REFERENCED

41)
referenced(x,y) ∧ Movie(x) ∧ Movie(y)
←
SELECT id, movie
FROM movielinks, link, movies
WHERE movielinks.link=link.id AND movies.title=referenced_in

x → mov(movie)
y → mov(id)

REFERENCES

42)
references(x,y) ∧ Movie(x) ∧ Movie(y)
←
SELECT id, movie
FROM movielinks, link, movies
WHERE movielinks.link=link.id AND movies.title=link.ref

x → mov(movie)
y → mov(id)

FEATUREADIN

43)
featuredIn(x,y) ∧ Movie(x) ∧ Movie(y)
←
SELECT id, movie
FROM movielinks, link, movies

WHERE movielinks.link=link.id AND movies.title=link.featured_in

x → mov(movie)
y → mov(id)

SOUNDTRACK

44)
soundtrack(x,y) ∧ Movie(x) ∧ Song(y)
←
SELECT movie, soundtrack **FROM** tracklist

x → mov(movie)
y → song(soundtrack)

PRODUCTIONCOMPANY

45)
productionCompany(x,y) ∧ Movie(x) ∧ Company(y)
←
SELECT prodcompany, movie **FROM** prodcompany2movie

x → mov(movie)
y → company(prodcompany)

DISTRIBUTOR

46)
distributor(x,y) ∧ Movie(x) ∧ Company(y)
←
SELECT distributor, movie **FROM** distribuited

x → mov(movie)
y → company(distributor)

SFXCOMPANY

47)
sfxCompany(x,y) ∧ Movie(x) ∧ Company(y)
←
SELECT sfxcompany, movie **FROM** sfxby

x → mov(movie)
y → company(sfxcompany)

CITY2COUNTRY

48)
city2country(x,y) ∧ City(x) ∧ Country(y)
←
SELECT city, country **FROM** location

x → city(city)
y → country(country)

CHARACTER

49)
Character(x) ∧ characterName(x,y)
←
SELECT role **FROM** actor2movie

x → char(role)
y → role xsd:String

50)
movie2character(x,y) ∧ Movie(x) ∧ Character(y)
←
SELECT movie, role **FROM** actor2movie

x → mov(movie)
y → char(role)

51)
actor2character(x,y) $\wedge$ Actor(x) $\wedge$ Character(y)
$\leftarrow$
SELECT actor, role **FROM** actor2movie

x $\rightarrow$ per(actor)
y $\rightarrow$ char(role)

Bibliografia

1. Linking Data to Ontologies.
   Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Riccardo Rosati. Journal on Data Semantics, vol X, 133-173, Springer, 2008.
2. Ontology-based Database Access.
   Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati. Proc. of the 15th Italian Symposium on Advanced Database Systems (SEBD'07), pages 324-331, 2007.
3. Linking Data to Ontologies: The Description Logic DL-LiteA.
   Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati. Proc. of the OWL: Experiences and Directions 2006 (OWLED'06). 2006
4. Mastro-i: Efficient integration of relational data through DL ontologies.
   Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati. Proc. of the 2007 Description Logic Workshop (DL'07), pages 227-234, 2007.
5. Data Integration Through DL-LiteA Ontologies
   Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati, Marco Ruzzi. Invited paper at LNCS volume collecting selecte papers from Int. Workshop on Semantics in Data and Knowledge Bases (SDKB'08).
6. Realizing Ontology Based Data Access: A Plugin for Protégé.
   Mariano Rodriguez-Muro, Lina Lubyte and Diego Calvanese. Information Integration Methods, Architectures and Systems 2008 (IIMAS 08) workshop, attached to ICDE 2008. Cancún, Mexico. April, 2008
7. Ontology-based database access with DIG-Mastro and the OBDA Plugin for Protégé.
   Antonella Poggi, Mariano Rodriguez and Marco Ruzzi. OWL: Experiences and Directions 2008 (OWLED 2008 DC) workshop. Washington DC, USA. April, 2008.
8. Ontologie per accesso ai dati: tecniche per interrogazioni del primo ordine basate sulla chiusura dinamica della conoscenza.
   Tesi di laurea specialistica Emma Di Pasquale, relatore Giuseppe De Giacomo. 2007.