Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

TypeDB e gli standard per il Semantic Web

A. Messina, U. Maniscalco, P. Storniolo

Rapporto Tecnico N.:
RT-ICAR-PA-2021-04 **Maggio 2021**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sede di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sede di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

TypeDB e gli standard per il Semantic Web

A. Messina[1], U. Maniscalco[1], P. Storniolo[1]

Rapporto Tecnico N.:
RT-ICAR-PA-2021-04 **Maggio 2021**

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Palermo, Via Ugo La Malfa n. 153, 90146 Palermo.

Sommario

1 Introduzione

In questo lavoro, metteremo a confronto TypeDB con gli standard per il Semantic Web, concentrandoci in particolare su RDF, XML, RDFS, OWL, SPARQL e SHACL.

Esistono alcune somiglianze chiave tra questi due insiemi di tecnologie, principalmente perché entrambi sono radicati nel campo dell'IA simbolica, della rappresentazione della conoscenza e del reasoning automatico. Queste somiglianze includono:

1. Entrambi consentono agli sviluppatori di rappresentare ed eseguire query su set di dati complessi ed eterogenei.
2. Entrambi danno la possibilità di aggiungere semantica a set di dati complessi.
3. Entrambi consentono all'utente di eseguire ragionamenti deduttivi automatizzati su grandi quantità di dati.

Tuttavia, esistono differenze fondamentali tra queste tecnologie, poiché sono state progettate per diversi tipi di applicazioni. Nello specifico, il Semantic Web è stato ideato, appunto, per il Web, con dati incompleti provenienti da molte fonti, dove chiunque può contribuire alla definizione e mappatura tra le fonti di informazione. TypeDB, al contrario, non è stato creato per condividere dati sul web, ma per funzionare come database transazionale per organizzazioni “chiuse”. Per questo motivo, confrontare le due tecnologie, a volte, potrebbe essere fuorviante.

Queste differenze possono essere riassunte nel modo seguente:

1. Rispetto al Semantic Web, TypeDB riduce la complessità mantenendo un alto grado di espressività. Con TypeDB, evitiamo di dover apprendere diversi i standard del Semantic Web, ciascuno con alti livelli di complessità. Ciò consente di essere produttivi più velocemente.
2. TypeDB fornisce un'astrazione di livello superiore per lavorare con dati complessi rispetto agli standard Semantic Web. Con RDF modelliamo il mondo in triple, che è un modello di dati di livello inferiore rispetto allo schema a livello di concetto di relazione tra entità di TypeDB. Modellazione e query per relazioni di ordine superiore e dati complessi sono nativi in TypeDB.
3. Gli standard Semantic Web sono costruiti per il Web, TypeDB funziona per sistemi chiusi con dati privati. I primi sono stati progettati per funzionare con i dati collegati su un Web aperto con dati incompleti, mentre TypeDB funziona come un tradizionale sistema di gestione di database in un ambiente chiuso.

In questo lavoro metteremo in evidenza che ci sono forti sovrapposizioni nel modo in cui entrambe le tecnologie offrono strumenti per la rappresentazione della conoscenza e il reasoning automatico e illustreremo i concetti più importanti ad alto livello senza entrare troppo nei dettagli. L'obiettivo è aiutare gli utenti con un background RDF/OWL a familiarizzare con TypeDB.

1.1 Lo stack per il Semantic Web

Il Semantic Web si è diffuso alla fine degli anni '90 per estendere l'architettura esistente del web con uno strato di semantica formale. Consiste in una serie di standard che insieme formano il cosiddetto Stack del Semantic Web. Le tecnologie che verranno trattate includono: XML, RDF, RDFS, OWL, SPARQL e SHACL.

- RDF è uno standard per lo scambio di dati sul web e ha XML come suo strumento di serializzazione.
- RDFS fornisce uno schema e alcuni costrutti ontologici di base.
- OWL migliora ulteriormente questo aspetto con i costrutti della logica descrittiva. SPARQL è il linguaggio per interrogare e inserire dati RDF.
- SHACL fornisce una serie di vincoli di verifica per convalidare logicamente i dati.

Oltre a questi standard, ci sono diverse librerie e implementazioni tra cui un utente deve scegliere per utilizzarli effettivamente nella pratica. Ad esempio, esistono diverse librerie che consentono all'utente di utilizzare RDF o SHACL (per Java è possibile scegliere tra TopBraid e Apache Jena), ma tutte si discostano leggermente dagli standard presentando delle sfumature individuali.

Tuttavia, nonostante l'abbondanza di materiale didattico disponibile, il Semantic Web non ha ottenuto una vera e propria adozione di massa al di fuori del mondo accademico. A causa del gran numero di tecnologie che un utente deve apprendere, insieme alla loro complessità intrinseca, un utente impiega molto tempo ad istruirsi sul Semantic Web prima di iniziare. La barriera all'ingresso è alta e ciò rende difficile per la maggior parte degli sviluppatori iniziare.

TypeDB, invece, fornisce all'utente una sola tecnologia che può sostituire molti degli standard del Semantic Web. Ciò significa, ad esempio, che un utente che crea un'applicazione non ha bisogno di istruirsi su quale tipo di reasoning, su quale sistema di verifica o su quale linguaggio di query utilizzare. Con TypeQL, tutto questo avviene all'interno della stessa tecnologia che l'utente deve imparare una sola volta.

TypeDB funziona a un livello superiore ed è più facile da imparare, riducendo la barriera all'ingresso, consentendo a milioni di sviluppatori di accedere a

tecnologie semantiche che prima erano inaccessibili. In TypeDB, la facilità d'uso è un primo principio.

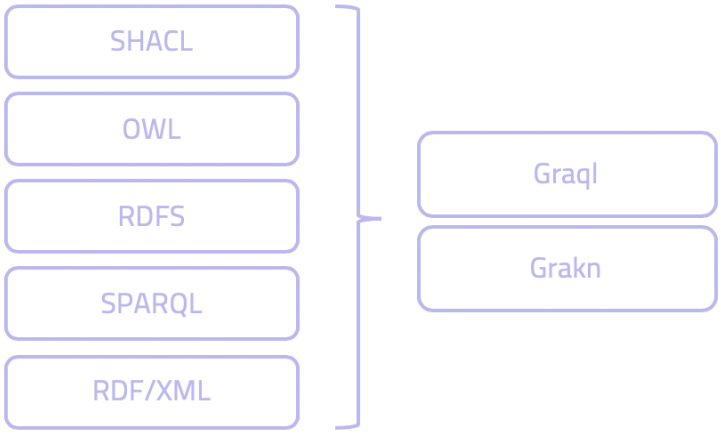

Figura 1 - Invece di RDF, SPARQL, RDFS, OWL e SHACL, utilizzeremo solo TypeDB e TypeQL.

In breve, TypeDB è un database logico distribuito sotto forma di grafo della conoscenza che implementa uno schema a livello di concetto. Questo sistema di rappresentazione della conoscenza viene quindi interpretato da un motore di reasoning automatico che esegue un ragionamento deduttivo automatizzato durante il runtime della query. L'interrogazione, lo schema e il reasoning avvengono tutti attraverso il linguaggio di query di TypeDB, TypeQL.

Le basi formali dello schema a livello di concetto di TypeDB sono fornite dall'ipergrafo, che svolge lo stesso ruolo del modello relazionale per i database relazionali, i grafi diretti nel Web semantico e i grafi di proprietà per i database a grafo. Gli ipergrafi generalizzano la nozione comune di cosa sia un arco. Nei grafici RDF e di proprietà, un arco è solo una coppia di vertici. Invece, un ipergrafo è un insieme di vertici, che può essere ulteriormente strutturato. I vantaggi rispetto a un modello a grafo diretto includono:

- Il meccanismo naturale di raggruppamento di informazioni rilevanti in uno stile relazionale, che è in gran parte perso nei grafici diretti.
- Gestione uniforme di tutte le relazioni n-arie, a differenza dei grafi orientati in cui le relazioni con più di due attori di ruolo richiedono un cambiamento radicale nell'approccio di modellazione (n-arie).
- Un modo naturale di esprimere informazioni di ordine superiore (relazioni tra relazioni, nidificazione delle informazioni), che nei grafi orientati richiedono tecniche di modellazione dedicate (es. reificazione).

2 RDF

2.1 Triple RDF

RDF è un sistema per modellare i dati e distribuirli sul web. È costituito da un multigrafo etichettato e diretto con vertici e archi etichettati. Questi sono costituiti da URI (cose), letterali (valori di dati) e nodi vuoti (nodi fittizi).

RDF memorizza le triple in forma *soggetto-predicato-oggetto*, come dichiarazioni atomiche dichiarate una per una. Di seguito è riportato un esempio in notazione XML di una persona "Peter Parker" che conosce un'altra persona "Aunt May":

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
 <foaf:Person>
   <foaf:name>Peter Parker</foaf:name>
   <foaf:knows>
       <foaf:Person>
         <foaf:name>Aunt May</foaf:name>
       </foaf:Person>
   </foaf:knows>
 </foaf:Person>
</rdf:RDF>
```

Come si vede, RDF guadagna in flessibilità a scapito della compattezza. Ciò significa che può essere espressivo, pur essendo estremamente granulare. Ogni singola relazione tra un punto dati viene dichiarata in modo esplicito: una relazione esiste o meno. Ciò semplifica l'unione di dati provenienti da fonti diverse rispetto ai database relazionali tradizionali. Le triple di cui sopra compongono questo grafo:

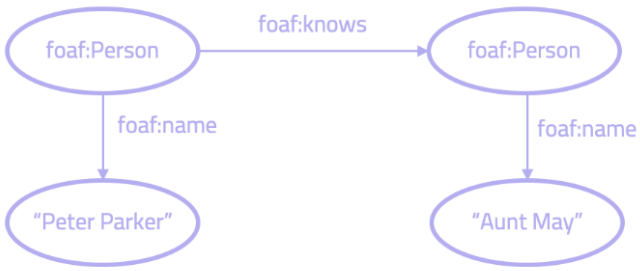

Figura 2 - Rappresentazione grafica delle triple RDF

TypeDB non funziona con le triple. Al contrario, espone un modello a livello di concetto entità-relazione. Quindi, invece di modellare in forma soggetto-predicato-oggetto, TypeDB rappresenta i nostri dati a un livello superiore, con

entità, relazioni, ruoli e attributi. Per l'esempio sopra riportato, diremmo che ci sono due entità persona, che hanno attributi di nome e sono correlate attraverso una relazione conosciuta:

```
$p isa person, has name "Peter Parker";
$p2 isa person, has name "Aunt May";
($p, $p2) isa knows;
```

Figura 3 - Modello TypeDB con due entità ("Peter Parker" e "Aunt May") e una relazione ("knows")

2.2 Iper-Relazioni

Come già accennato, il modello di dati di TypeDB si basa su ipergrafi. Mentre in RDF un arco è solo una coppia di vertici, un iper-arco è un insieme di vertici. Il fondamento formale del modello di dati di TypeDB si basa su tre premesse:

Un ipergrafo è costituito da un insieme non vuoto di vertici e un insieme di iper-archi

Un iper-arco è un insieme finito di vertici (distinguibili da ruoli specifici che giocano in quell'iper-arco)

Un iper-arco è anche un vertice stesso e può essere collegato da altri iper-archi

Nota: sebbene TypeDB sfrutti gli iper-archi, TypeDB in realtà non espone archi o iper-archi. Funziona invece con le relazioni, o iper-relazioni. Di seguito è riportata una figura che illustra come funziona. L'esempio mostra due iper-relazioni:

- *marriage*, che descrive una relazione matrimoniale binaria tra *Bob* e *Alice* che interpretano rispettivamente i ruoli di *husband* e *wife*
- *divorce-filling* che descrive una relazione ternaria di domanda di divorzio che coinvolge tre attori nei ruoli di *certified marriage*, *petitioner* e *respondent*

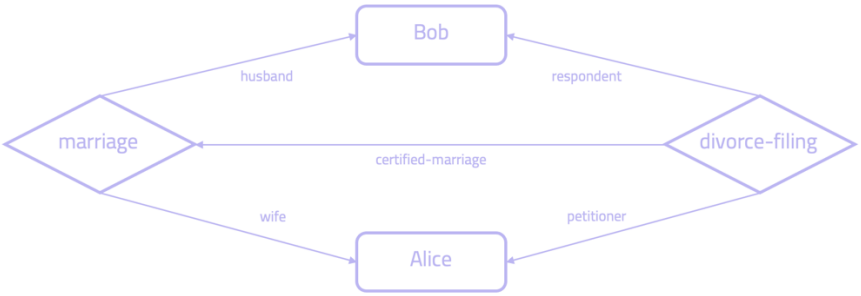

Figura 4 - Esempio di iper-relazione in TypeDB

Una iper-relazione può essere vista semplicemente come una raccolta di coppie di ruoli e attori di cardinalità arbitraria. Poiché le iper-relazioni non possono essere rappresentate in modo nativo in un grafo diretto etichettato, l'esempio di cui sopra in notazione RDF verrà espresso nel seguente modo:

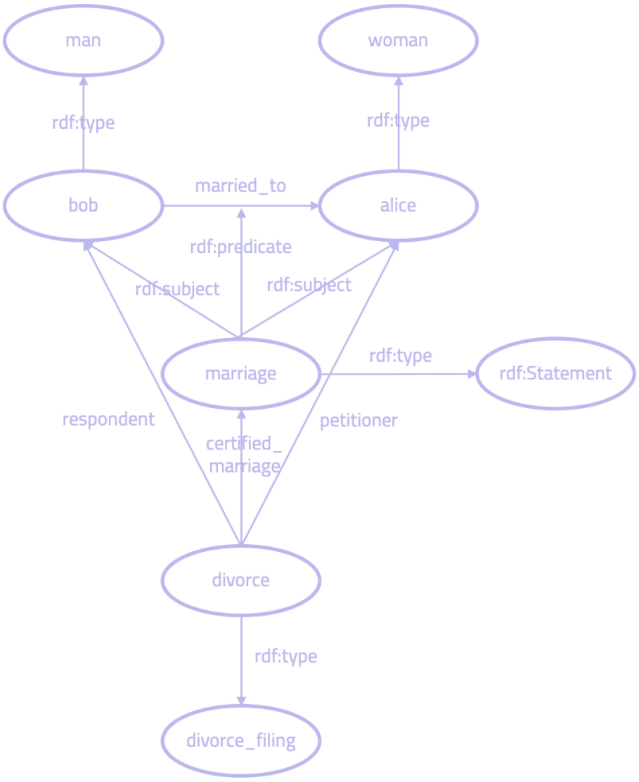

Figura 5 - Rappresentazione delle iper-relazioni in RDF

Come mostrato, ogni iper-relazione in TypeDB può essere mappata nel corrispondente grafo diretto del modello RDF. Ad esempio, in questo modello, anche i tipi di entità e relazione sono codificati esplicitamente come risorse RDF nello stile RDF. Pertanto, le iper-relazioni possono essere implementate su un

triple store RDF. Pertanto, in termini di modellazione, le iper-relazioni offrono un formalismo di rappresentazione dei dati molto naturale e diretto, consentendo la modellazione a livello concettuale utilizzando diagrammi entità-relazione.

La differenza, tuttavia, è che in TypeDB le iper-relazioni diventano costrutti di modellazione di prima classe. Questo è importante perché in uno scenario reale, quando il modello concettuale completo non è completamente previsto all'inizio, l'effettivo risultato della modellazione può creare molta complessità non necessaria.

Inoltre, la modellazione delle iper-relazioni in modo nativo, rispetto agli archi diretti binari, porta a miglioramenti nella pianificazione e nell'ottimizzazione delle query, poiché i dati raggruppati nella stessa struttura "contenitori" vengono spesso recuperati in raggruppamenti simili da utenti e applicazioni. Riconoscendo la struttura di questi prima dell'interrogazione, il processo di recupero può essere pianificato ed eseguito in modo più ottimale.

2.3 Namespace

A causa della natura pubblica del Web, RDF utilizza i namespace per interpretare e identificare diverse ontologie, che di solito vengono fornite all'inizio di un documento per renderlo più leggibile.

Ogni risorsa in RDF ha un identificatore univoco. Il tag rdf:RDF gli dice che è un documento RDF:

```
rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Poiché TypeDB non opera sul Web, non sono necessari gli URI. Di converso, i database sono logicamente separati all'interno di un'istanza TypeDB quindi, a differenza dei namespace RDF, questi non possono comunicare tra loro.

2.4 Serializzazione

Ci sono molti modi per esprimere RDF in forma testuale. Un modo comune è rappresentare le triple in formato XML (come raccomandato dal W3C):

```
<http://example.org/#spiderman>
  <http://www.perceive.net/schemas/relationship/enemyOf>
```

```
    <http://example.org/#green-goblin> .
<http://example.org/#green-goblin>
  <http://www.perceive.net/schemas/relationship/enemyOf>
    <http://example.org/#spiderman> .
```

In TypeDB, non è necessario utilizzare le serializzazioni e utilizziamo TypeQL. L'esempio può essere rappresentato così:

```
$g isa person, has name "Green Goblin";
$s isa person, has name "Spiderman";
(enemy: $g, enemy: $s) isa enemyship;
```

Avremo due entità *person*, con il nome di attributo "Green Goblin" e "Spiderman", collegate attraverso una relazione di tipo *enemyship* e dove entrambe svolgono il ruolo *enemy*.

2.5 Relazioni di ordine superiore

Data la forma *soggetto/predicato/oggetto* di una tripla, la modellazione in RDF può soffrire di limitazioni quando si deve rappresentare una relazione di ordine superiore. Ad esempio, prendiamo questa tripla:

```
lit:HarryPotter bio:author lit:JKRowling .
```

La tripa afferma che JK Rowling ha scritto Harry Potter. Tuttavia, potremmo voler qualificare questa affermazione dicendo che JK Rowling ha scritto Harry Potter nel 2000. In RDF, per fare questo dovremmo passare attraverso un processo chiamato *reificazione*, creando una tripla per statement, dove il soggetto della tripla sarebbe lo stesso nodo:

```
bio:n1 bio:author lit:JKRowling .
bio:n1 bio:title "Harry Potter" .
bio:n1 bio:publicationDate 2000 .
```

In TypeDB, dato il suo schema a livello di concetto, la necessità di reificazione non esiste e possiamo rappresentare relazioni di ordine superiore in modo nativo. “JK Rowling ha scritto Harry Potter” si sarebbe espresso così:

```
$a isa person, has name "JK Rowling";
$b isa book, has name "Harry Potter";
(author: $a, publication: $b) isa authorship;
```

Quindi, se dire che JK Rowling ha scritto Harry Potter nel 2000, aggiungeremmo semplicemente un attributo alla relazione:

```
$a isa person, has name "JK Rowling";
$b isa book, has name "Harry Potter";
(author: $a, publication: $b) isa authorship, has date 2000;
```

2.6 Nodi vuoti

A volte in RDF non vogliamo dare un URI o un letterale. In questi casi abbiamo a che fare con nodi vuoti, che sono risorse anonime senza un'identificazione Web. Un esempio è l'affermazione secondo cui Harry Potter è stato ispirato da un uomo che vive in Inghilterra:

```
lit: HarryPotter bio:name lit:"Harry Potter" .
lit:HarryPotter lit:hasInspiration [a :Man;
                                                bio:livesIn geo:England] .
```

Poiché TypeDB non è pensato per il Web, l'idea di un nodo vuoto non si traduce direttamente in TypeDB. Mentre in RDF usiamo un nodo vuoto per indicare l'esistenza di una cosa per la quale non abbiamo un URI, ci sono molti modi in cui ciò potrebbe essere fatto in TypeDB. Se, come nell'esempio sopra, stiamo usando un nodo vuoto per indicare che non sappiamo nient'altro di quell'uomo, oltre al fatto che vive in Inghilterra, lo rappresentiamo come segue:

```
$b isa book, has name "Harry Potter";
$m isa man; ($b, $m) isa inspiration;
$l isa location, has name "England";
($m, $l) isa lives-in;
```

Notiamo che la variabile *$m* è assegnata all'entità di tipo *man*, per la quale non vengono fornite ulteriori informazioni, se non che è collegata all'entità di tipo *location* con nome “England”, tramite una relazione di tipo *lives-in*.

3 SPARQL

3.1 Cos'è SPARQL

SPARQL è un linguaggio standardizzato W3C per interrogare informazioni da database che possono essere mappati su RDF. Simile a SQL, SPARQL consente di inserire e interrogare i dati. A differenza di SQL, le query non sono vincolate a un solo database e possono essere federate su più endpoint HTTP.

In quanto linguaggio di query di TypeDB, TypeQL è il linguaggio di query equivalente. Come in SPARQL, TypeQL consente di inserire e interrogare i dati. Tuttavia, dato che TypeQL non è costruito come un linguaggio Web aperto, non consente di eseguire query su più endpoint in modo nativo (questo può essere fatto con uno dei driver client di TypeDB). In quanto tale, TypeQL è più simile a SQL e ad altri sistemi di gestione di database tradizionali.

3.2 Inserimento dati con SPARQL

Per aggiungere dati nel grafo predefinito, questo frammento di codice descrive come vengono inserite due triple RDF con SPARQL:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA
{
  <http://example/book1> dc:title "A new book" ;
                         dc:creator "A.N.Other" .
}
```

In TypeQL, iniziamo con l'istruzione *insert* per dichiarare che i dati devono essere inseriti. La variabile *$b* è assegnata all'entità di tipo *book*, che ha *title* di valore "A new book" e *creator* "A.N.Other".

```
insert
$b isa book, has title "A new book", has creator "A.N.Other";
```

3.3 Interrogazioni con SPARQL

In SPARQL, prima dichiariamo gli endpoint da cui vogliamo recuperare i nostri dati e possiamo associarli a un certo PREFIX. La query effettiva inizia con SELECT

prima di indicare i dati che vogliamo restituire. Quindi, nella clausola WHERE, indichiamo il modello per il quale SPARQL troverà i dati che corrispondono. In questa query, cerchiamo tutte le persone che "Adam Smith" conosce utilizzando i namespace *foaf* e *vCard*:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT ?whom
WHERE {
        ?person rdf:type  foaf:Person .
        ?person vcard:family-name "Smith" .
        ?person vcard:given-name  "Adam" .
        ?person foaf:knows ?whom .
 }
```

In TypeQL, iniziamo con l'istruzione *match* per dichiarare che vogliamo recuperare dei dati. Cerchiamo un'entità di tipo *person* che ha cognome "Smith" e nome "Adam". Quindi, lo colleghiamo tramite una relazione di tipo *knows* a *$p2*. Poiché vogliamo sapere chi conosce "Adam Smith", vogliamo che venga restituito *$p2* dichiarato nell'istruzione *get*:

```
match $p isa person, has family-name "Smith", has given-name "Adam";
($p, $p2) isa knows;
get $p2;
```

Vediamo ora una query diversa: dammi il regista e i film in cui ha recitato James Dean, in cui anche una donna ha recitato un ruolo, e quella donna ha recitato in un film diretto da John Ford. Di seguito è riportato il codice SPARQL e la rappresentazione visiva di questa query di tipo traversal.

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

SELECT  ?director ?movie
WHERE{
?actor         rdf:type             foaf:Man ;
               movie:name          "James Dean" ;
               movie:playedIn       ?movie .
?actress       movie:playedIn       ?movie ;
               rdf:type             foaf:Woman ;
               movie:playedIn       ?anotherMovie .
?JohnFord      rdf:type             foaf:Man ;
               movie:name           "John Ford" .
?anotherMovie  movie:directedBy  ?JohnFord .
}
```

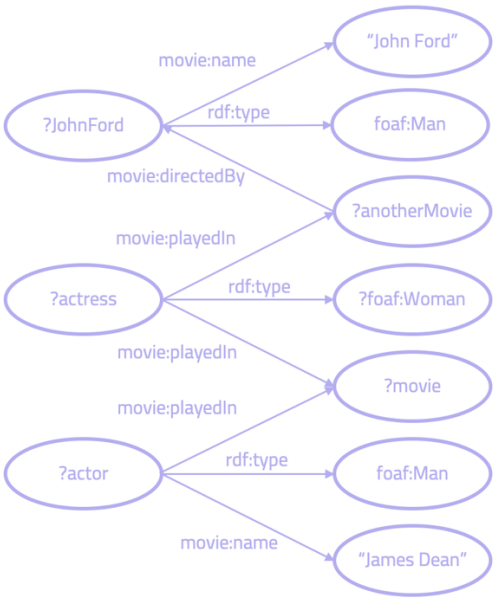

Figura 6 - Rappresentazione grafica della query di attraversamento Sparql.

In TypeDB, possiamo chiedere la stessa cosa nel modo seguente:

```
match
$p isa man, has name "James Dean";
$w isa woman;
(actor: $p, actress: $w, casted-movie: $m) isa casting;
(actress: $w, casted-movie: $m2) isa casting;
$d isa man, has name "John Ford";
($m2, $d) isa directorship; get $d, $m;
```

In sostanza, assegniamo l'entità di tipo *man* con attributo *name* di valore "James Dean" alla variabile *$p*. Diciamo poi che $w è una entità di tipo *woman*. Queste due entità sono collegate al film in una relazione a tre vie chiamata *casting*. La donna svolge anche un ruolo in un'altra relazione di casting, in cui l'entità del film è collegata a "John Ford" che si relaziona ad essa attraverso una relazione *directorship*.

Nell'esempio sopra, il casting di iper-relazione in TypeDB rappresenta le due proprietà *playedIn* in SPARQL. Tuttavia, in SPARQL possiamo avere solo due archi che collegano la donna e "James Dean" con il film, ma non tra di loro. Questo mostra come la modellazione in TypeDB sia fondamentalmente diversa da RDF data la sua capacità di modellare gli ipergrafi. TypeDB consente di rappresentare nativamente un numero N di attori in una relazione senza dover reificare il modello.

Schematicamente, ecco come viene rappresentata visivamente la query suddetta (notare il casting della relazione ternaria):

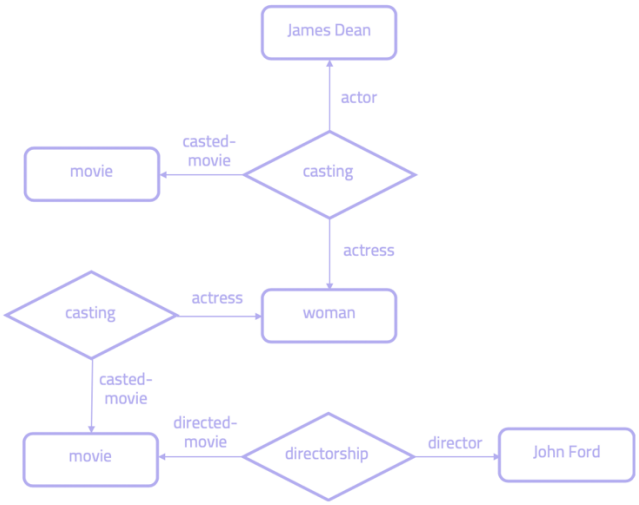

Figura 7 - Rappresentazione visiva della query in TypeDB

3.4 Negazione

In SPARQL, possiamo anche specificare nella nostra query che alcuni dati non sono presenti utilizzando la parola chiave *NOT EXISTS*. Questo trova un modello di grafo che corrisponde solo se quel sottografo non corrisponde. Nell'esempio seguente, cerchiamo attori che hanno recitato nel film Giant e che non sono ancora deceduti:

```
PREFIX movie: <http://example.com/moviedb/0.1/>

SELECT ?actor
WHERE {
         ?actor movie:playedIn movie:Giant .
         NOT EXISTS {?actor movie:diedOn ?deathdate .
}
```

Anche TypeDB supporta la negazione. Ciò viene ottenuto usando la parola chiave *not* seguita dal pattern da negare. L'esempio di sopra è rappresentato come segue:

```
match
$m isa movie, has name "Giant"; ($a, $m) isa played-in;
not {$a has death-date $dd;}; get $a;
```

In questa query, stiamo cercando una entità di tipo *movie* con nome "Giant", che è collegata a *$a*, un attore, attraverso una relazione di tipo *played-in*. Nella sottoquery *not* specifichiamo che *$a* non deve avere un attributo di tipo *death-date* con alcun valore. Infine prendiamo l'attore *$a*.

4 RDF Schema

4.1 Introduzione

Poiché RDF è solo un modello di scambio di dati, di per sé è "senza schema". Ecco perché RDF Schema (RDFS) è stato introdotto per estendere RDF con la semantica ontologica di base. Questi consentono, ad esempio, semplici gerarchie di tipi su dati RDF. In TypeDB, TypeQL viene utilizzato come linguaggio di schema.

4.2 Classi RDFS

RDFS estende il vocabolario RDF e permette di descrivere tassonomie di classi e proprietà. Una classe RDFS dichiara una risorsa RDFS come classe per altre risorse. Possiamo abbreviarlo usando *rdfs:Class*. Usando XML, la creazione di una classe *animal* con una sottoclasse *horse* sarebbe avverrebbe nel modo seguente:

```
<?xml version="1.0"?>

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xml:base="http://www.animals.fake/animals#">

<rdfs:Class rdf:ID="animal" />
<rdfs:Class rdf:ID="horse">
  <rdfs:subClassOf rdf:resource="#animal"/>
</rdfs:Class>
</rdf:RDF>
```

Per fare lo stesso in TypeDB, scriveremmo:

```
define
animal sub entity;
horse sub animal;
```

RDFS consente anche la sottotipizzazione delle proprietà:

```
<?xml version="1.0"?>

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xml:base="http://www.animals.fake/animals#">

<rdfs:Class rdf:ID="mammal" />

<rdfs:Class rdf:ID="human">
  <rdfs:subClassOf rdf:resource="#mammal"/>
</rdfs:Class>
```

```
<rdfs:Property rdf:ID="employment" />

<rdfs:Property rdf:ID="part-time-employment">
  <rdfs:subPropertyOf rdf:resource="#employment"/>
</rdfs:Property>

</rdf:RDF>
```

Che in TypeDB sarebbe simile a questo:

```
define
mammal sub entity;
human sub mammal;
employment sub relation;
part-time-employment sub employment;
```

Come mostrano gli esempi, RDFS descrive principalmente i costrutti per i tipi di oggetti (classi), che si ereditano l'uno dall'altro (sottoclassi), proprietà che descrivono gli oggetti (proprietà) ed ereditano anche l'uno dall'altro (sottoproprietà). Questo comportamento di sottotipizzazione può essere ottenuto con la parola chiave di TypeQL *sub*, che può essere utilizzata per creare gerarchie di tipi di qualsiasi cosa (entità, relazioni e attributi).

Tuttavia, creare una mappatura uno-a-uno tra una classe e un'entità in TypeDB o una proprietà con una relazione in TypeDB, nonostante le loro apparenti somiglianze, non dovrebbe essere sempre fatto. Questo perché il modello in RDF è costruito utilizzando un modello di dati di livello inferiore, lavorando in triple, mentre TypeDB consente di modellare a un livello superiore.

4.3 Ereditarietà multipla

Un'importante differenza di modellazione tra TypeDB e il Semantic Web riguarda l'ereditarietà multipla. In RDFS, una classe può avere tante superclassi quante sono denominate o dedotte logicamente.

Consideriamo il seguente esempio:

```
company      rdf:type  rdfs:Class
government   rdf:type  rdfs:Class

employer     rdf:type         rdfs:Class
employer     rdfs:subClassOf  company
employer     rdfs:subClassOf  government
```

Qui definiamo un datore di lavoro sia come classe *company* che come *government*. Tuttavia, sebbene ciò possa sembrare corretto, il problema è che spesso l'ereditarietà multipla, come concetto di modellazione, non viene utilizzata nel

modo corretto. L'ereditarietà multipla dovrebbe raggruppare le cose e non i "tipi" di sottoclassi, dove ogni tipo è una definizione di qualcos'altro. In altre parole, non vogliamo rappresentare istanze di dati. Questo è un errore comune.

Invece dell'ereditarietà multipla, TypeDB supporta l'ereditarietà di tipo singolo, dove dovremmo assegnare ruoli piuttosto che più classi. Un ruolo definisce il comportamento e l'aspetto di una cosa nel contesto di una relazione e possiamo assegnare più ruoli a una cosa (da notare che i ruoli vengono ereditati quando i tipi subclassano un altro).

Ad esempio, un governo può assumere una persona e un'azienda può assumere una persona. Si potrebbe quindi suggerire di creare una classe che erediti sia il governo che l'azienda che può assumere una persona e finire con una classe datori di lavoro sottoclasse di entrambe (come mostrato nell'esempio sopra).

Tuttavia, questo è un utilizzo forzato dell'ereditarietà. In questo caso, dovremmo creare un ruolo *employer*, che si riferisce a un rapporto di lavoro e contestualizza il modo in cui un'azienda o un governo è coinvolto in quel rapporto (interpretando il ruolo di datore di lavoro).

```
company sub entity,
    plays employment:employer;

government sub entity,
    plays employment:employer;

employment sub relation,
    relates employer;
```

4.4 rdfs:domain e rdfs:range

Due istanze comunemente usate di *rdf:property* sono *domain* e *range*, che vengono usati per affermare che rispettivamente i membri o i valori di una proprietà sono istanze di una o più classi. Di seguito è riportato un esempio di *rdfs:domain*:

```
:hasBrother rdfs:domain :Person
```

Qui, *rdfs:domain* assegna la classe *Person* al soggetto della proprietà *hasBrother*.

Questo è un esempio di *rdfs:range*:

```
:hasBrother rdfs:range :Male
```

In questo caso, *rdfs:range* assegna la classe *Male* all'oggetto della proprietà *hasBrother*.

In TypeDB, non esiste un'implementazione diretta di *range* e *domain*. Le inferenze di base tratte da esse sarebbero già rappresentate nativamente nel modello di dati TypeDB attraverso l'uso di ruoli, oppure possiamo creare regole per rappresentare la logica che vogliamo dedurre.

Tuttavia, si tenga presente che l'uso delle regole in TypeDB offre maggiore espressività nel consentirci di rappresentare il tipo di inferenze che vogliamo fare. In altre parole, la traduzione di *range* e *domain* in TypeDB dovrebbe essere eseguita caso per caso.

Nell'esempio sopra, *rdfs:domain* può essere tradotto in TypeDB dicendo che quando un'entità ha un tipo di attributo *published-date*, svolge il ruolo *published-book* in una relazione di tipo *publishing*. Rappresentando il tutto come regola TypeDB abbiamo:

```
rule publishing-rule:
when {
       $b has published-date $pd;
} then {
       (published-book: $b) is publishing;
};
```

L'esempio di *rdfs:range* può essere creato con la seguente regola TypeDB, che aggiunge l'attributo di tipo *gender* con valore "male", solo se una *person* svolge il ruolo di *brother* in qualsiasi relazione *siblingship*, dove il numero di altri fratelli è N.

```
rule gender-male:
when {
       $r (brother: $p) isa siblingship;
} then {
       $p has gender "male";
};
```

Vediamo anche un altro esempio. In un contesto marittimo, se disponiamo di una nave di classe *DepartingVessel*, che ha la proprietà *nextDeparture* specificata, potremmo affermare:

```
ship:Vessel rdf:type rdfs:Class .
ship:DepartingVessel rdf:type rdfs:Class .
ship:nextDeparture rdf:type rdf:Property .
ship:QEII a ship:Vessel .
ship:QEII ship:nextDeparture "Mar 4, 2010" .
```

Con il seguente *rdfs:Domain*, qualsiasi nave per la quale è specificato *nextDeparture*, verrà dedotta come membro della classe *DepartingVessel*. In questo esempio, ciò significa che a QEII è assegnata la classe *DepartingVessel*.

```
ship:nextDeparture rdfs:domain ship:DepartingVessel .
```

Per fare lo stesso in TypeDB, possiamo scrivere una regola che trovi tutte le entità con attributo *next-departure* e assegnarle a una relazione *departure* che assume il

ruolo *departing-vessel*.

```
rule departure-next:
when {
       $s has next-departure $nd;
} then {
       (departing-vessel: $s) isa departure;
};
```

Quindi, se vengono importati i seguenti dati:

```
$s isa vessel, has name "QEII", has next-departure "Mar 4, 2010";
```

TypeDB deduce che la nave QEII svolge il ruolo *departing-vessel* in una relazione di tipo *departure*, l'equivalente in questo caso della classe *nextDeparture*.

L'uso di *rdfs:domain* e *rdfs:range* sono utili nel contesto del web, dove i dati federati possono spesso essere trovati incompleti. Poiché TypeDB non vive sul web, la necessità di questi concetti è ridotta. Inoltre, la maggior parte di questi dati dedotti è già rappresentata in modo nativo nel modello concettuale di TypeDB. Molto di questo è dovuto al suo modello di livello superiore e all'uso di regole. Pertanto, mappare direttamente *rdfs:range* e *rdfs:domain* su un concetto in TypeDB è solitamente ingenuo e porta a ridondanze. Invece, la traduzione di questi concetti in TypeDB dovrebbe essere eseguita caso per caso utilizzando regole e ruoli.

5 OWL

5.1 OWL e TypeDB

OWL è una famiglia di linguaggio ontologico basato sulla logica descrittiva (DL) che aggiunge costrutti ontologici a RDFS per esprimere condizioni e derivare nuovi fatti. Per farne un uso significativo, OWL fornisce diverse alternative: *OWL QL*, *OWL RL*, *OWL DL*, e l'utente deve decidere quale si adatta meglio al proprio caso d'uso.

TypeDB, d'altra parte, viene fornito con le proprie capacità di inferenza native integrate. Questa è una distinzione importante, perché OWL presuppone una buona comprensione del campo della logica da parte dell'utente, mentre l'utilizzo di TypeDB non richiede che l'utente abbia studiato a fondo questo campo.

Il risultato è che OWL fatica a mantenere un equilibrio soddisfacente tra espressività e complessità. Giusto per ragionare su due semplici vincoli: "*ogni genitore ha un figlio e ogni bambino è una persona"* richiede l'uso di un reasoner OWL DL a tutti gli effetti. Inoltre, OWL non è adatto a ragionare con relazioni complesse. Le sue basi formali, basate su una proprietà del modello ad albero, lo rendono più adatto ai dati a forma di albero, ma scala male con dati più complessi.

OWL adotta ipotesi proprie del mondo aperto, invece delle ipotesi del mondo chiuso di TypeDB. Ciò significa che, in un esempio in cui OWL ha il vincolo "*ogni genitore deve avere almeno un figlio"*, se abbiamo una persona senza figli, questo è ancora coerente con il vincolo, poiché potremmo non sapere ancora dei figli di John. Tuttavia, con l'ipotesi del mondo chiuso di TypeDB, se non ci sono menzioni effettive dei figli di John, ciò significa che in realtà non ha figli e non è un genitore.

I presupposti del mondo aperto si prestano bene per il web, che include informazioni incomplete da più fonti, motivo per cui OWL fornisce molti concetti per gestire e affrontare questa incompletezza. Tuttavia, a causa di questo presupposto del mondo aperto, OWL rende difficile convalidare la consistenza dei dati. Ecco perché i database relazionali mantengono vincoli di schema per garantire la qualità dei dati. TypeDB combina entrambi gli stili di ragionamento: l'inferenza del mondo aperto in stile ontologico e lo schema come il controllo dei vincoli del mondo chiuso.

Premesso ciò, OWL ha una soglia di ingresso molto alta per i *non logici*. Poiché si basa sulla logica della descrizione, gli sviluppatori evitano di utilizzare OWL in quanto non è banale comprendere il linguaggio e il suo comportamento previsto. Per questo motivo, i formalismi di rappresentazione della conoscenza di TypeDB

rimangono leggeri, fornendo capacità semantiche a un pubblico molto più ampio di quello di OWL. In altre parole, TypeDB è più semplice da usare rispetto a OWL.

Esaminiamo ora alcuni assiomi comuni in OWL comparandoli con TypeDB. Il seguente non è un elenco esaustivo e viene fornito per aiutare l'utente a capire come pensare di migrare a TypeDB.

5.2 Restrictions

Una funzionalità chiave di OWL è definire classi di restrizione (*owl:Restriction*). Queste classi senza nome sono definite in base alle restrizioni dei valori per determinate proprietà della classe. OWL consente di modellare situazioni in cui alcuni membri di una classe devono avere determinate proprietà.

Le restrizioni consentono di distinguere tra qualcosa che si applica a tutti i membri di una classe. Una restrizione è definita fornendo una descrizione che limiti il tipo di cose che si possono dire su un membro di quella classe.

Un esempio è la restrizione *AllValuesFrom*, che afferma che nel contesto di una classe specifica, l'intervallo di una proprietà dovrebbe essere sempre una classe specifica. Ad esempio, se *AllValuesFrom* è stato applicato per una classe *Person*, e questo è applicato all'intervallo di *hasParent* con *Person*, allora questi possono avere solo delle *Person* genitori, mentre un *Animal* non può avere un genitore *Person*. L'esempio seguente mostra anche che solo un *Animal* può avere dei genitori *Animal*.

```
:Person
  a owl:Class ;
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty :hasParent ;
      owl:allValuesFrom :Person
   ] .

:Animal
  a owl:Class ;
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty :hasParent ;
      owl:allValuesFrom :Animal
   ] .
```

Per ottenere la stessa cosa in TypeDB, la restrizione è rappresentata nella definizione dello schema. Vengono creati le entità di tipo *person* e *animal* e due relazioni con restrizioni: *person-parentship* e *animal-parentship*. La prima si riferisce solo a *person* mentre la secondo si riferisce ad *animal*.

```
define
person sub entity,
plays person-parentship:child,
plays person-parentship:parent;

animal sub entity,
plays animal-parentship:child,
plays animal-parentship:parent;

parentship sub relation, abstract;

person-parentship sub parentship,
relates child,
relates parent;

animal-parentship sub parentship,
relates child,
relates parent;
```

5.3 Proprietà transitive

Un'inferenza comune in OWL è la transitività. Una relazione R si dice *transitiva* quando, se A è connesso a B con R(a,b) e B è connesso a C con R(b,c), ciò implica che A è connesso a C con R(a,c). A questo scopo viene utilizzato il costrutto *owl:TransitiveProperty*.

L'esempio seguente consente di dedurre che Kings Cross si trova in Europa.

```
:isLocated rdf:type owl:TransitiveProperty.

:KingsCross isLocated :London.
:London isLocated :UK.
:UK isLocated :Europe.
```

In TypeDB, avremmo bisogno solo di una regola per rappresentare la transitività:

```
rule transitive-location:
when {
       $r1 (located: $a, locating: $b) isa location;
       $r2 (located: $b, locating: $c) isa location;
} then {
       (located: $a, locating: $c) isa location;
};
```

Una volta definita la regola, carichiamo i dati seguenti:

```
insert

$a isa city, has name "London";
$b isa country, has name "Uk";
$c isa continent, has name "Europe";
(located: $a, locating: $b); (located: $b, locating: $c);
```

Se poi chiediamo:

```
match
$b isa continent, has name "Europe";
(located: $a, locating: $b); get $a;
```

ciò produrrebbe non solo il paese "UK", che ha una relazione diretta con il continente "Europ", ma anche la città "London", poiché questa sarebbe dedotta come avente una relazione transitiva con il continente "Europe".

5.4 Proprietà equivalenti

OWL fornisce anche un costrutto per modellare proprietà equivalenti. Ciò indica che due proprietà hanno la stessa estensione di proprietà. Un esempio:

```
:borrows owl:equivalentProperty :checkedOut .
```

Questo può essere rappresentato con una regola in TypeDB, con il quale si può dedurre una nuova relazione *checked-out* come la seguente:

```
rule borrowing-checked-out-equivalent:
when {
    (borrower: $x, borrowing: $y) isa borrowing;
} then {
    (checking-out: $x, checked-out: $y) isa checked-out;
};
```

5.5 Proprietà simmetriche

Una relazione simmetrica rappresenta una relazione che ha la propria proprietà inversa. Ad esempio, se Susan è correlata a Bob tramite una proprietà *hasSibling*, è possibile dedurre che anche Bob è correlato a Susan tramite una proprietà *hasSibling*.

```
:hasSibling rdf:type owl:SymmetricProperty .
```

In TypeDB, tale simmetria può essere modellata semplicemente ripetendo i ruoli in una o più relazioni:

```
(sibling: $p, sibling: $p2) isa siblingship;
```

In altre parole, ciò che necessita di un costrutto esplicito in OWL è nativo del modello di TypeDB, ovvero la regola di simmetria non richiede di essere resa esplicita. Tuttavia, è qui che diventa difficile confrontare OWL con TypeDB, poiché entrambi servono casi d'uso diversi. In particolare, dati i presupposti del mondo

chiuso di TypeDB, molti dei problemi in OWL semplicemente non esistono.

5.6 Proprietà funzionali

Di seguito vediamo come possiamo modellare alcuni altri costrutti OWL. Il costrutto *owl:FunctionalProperty* è rappresentato in questo modo:

```
hasFather rdf:type owl:FunctionalProperty .
```

In TypeDB è possibile utilizzare una regola:

```
rule same-fatherhood:
when {
       (father: $x, child: $ y) isa fatherhood;
       (father: $d, child: $ y) isa fatherhood;
} then {
       (father: $x, father: $y) isa same-father;
};
```

5.7 IntersectionOf

Vediamo un utilizzo di *owl:intersectionOf* con questo esempio:

```
:Mother rdfs:subClassOf [ owl:interesctionOf ( :Female :Parent ) ]
```

In tal modo, si assegna la classe *:Mother* se la risorsa è sia *:Female* che *:Parent*. In TypeDB possiamo scegliere di utilizzare una regola, contenente una condizione congiuntiva per rappresentare questa casistica:

```
rule parenthood-motherhood:
when {
       $p isa person, has gender "female";
       (mother: $p) isa motherhood;
} then {
       (parent: $p) isa parenthood;
};
```

5.8 UnionOf

Consideriamo il seguente esempio di *owl:unionOf*:

```
:Person owl:equivalentClass [ owl:unionOf (: Woman :Man ) ]
```

Si deduce la classe *:Person* se la risorsa è di classe *:Woman* o *:Man*. TypeDB non ha un equivalente diretto con *owl:equivalentClass*, ma c'è una certa copertura offerta dalle regole. Poiché *sub* è un equivalente di *subClassOf*, quanto sopra può essere ottenuto in TypeDB con:

```
define
person sub entity;
man sub person;
woman sub person;
```

In questo caso, potremmo anche creare un'istanza di una persona che non è né una donna né un uomo. Se lo si vuole evitare, possiamo aggiungere la parola chiave *abstract* e rendere astratta la classe *person*, il che renderebbe le definizioni equivalenti.

5.9 HasValue

Con la restrizione *owl:hasValue* si può affermare che i vini rossi dovrebbero avere il colore "red" come valore della loro proprietà *color*:

```
:RedWine
  a owl:Class ;
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty :color ;
      owl:hasValue "red"
    ] .
```

In TypeDB, è possibile utilizzare una regola per rappresentare questa circostanza:

```
rule red-wine-color:
when {
       $w isa red-wine;
} then {
       $w has color "red";
};
```

5.10 hasSelf

Con la restrizione *owl:hasSelf* si può affermare, ad esempio, che chi è un narcisista ama se stesso.

```
:Narcissist rdfs:subClassOf
       [ owl:hasSelf true ; owl:onProperty :loves ]
```

Questo potrebbe essere rappresentato in TypeDB nel modo seguente:

```
rule love-narcissist:
when {
       $n isa narcissist;
} then {
       (loving: $n) isa loves;
};
```

Come accennato prima, TypeDB non è creato per gli stessi casi d'uso per cui è stato creato OWL. Pertanto, non è possibile effettuare una mappatura diretta uno a uno. Quando si lavora con più classi, ad esempio, è necessario prendere in considerazione le decisioni di modellazione: quali diventano entità e quali ruoli, in TypeDB.

5.11 Verifica con SHACL

Tradizionalmente, RDF non garantisce che i dati importati aderiscano a una serie di condizioni, ad esempio uno schema. A questo serve lo standard SHACL, che verifica se l'RDF è logicamente coerente con uno schema prima che possa essere eseguito il commit. Esistono anche altri modi per verificare la verifica logica con implementazioni leggermente diverse.

Come SHACL, anche TypeDB verifica la convalida ed evidenzia i problemi relativi alla qualità dei dati. Ma mentre SHACL è solo uno schema di convalida, TypeDB implementa uno schema semantico.

Utilizzando uno schema di convalida, se l'origine dati da inserire contiene violazioni dello schema, la transazione avrà esito negativo. Quindi guarderemmo la fonte, la sistemeremmo e importeremmo di nuovo.

Con uno schema semantico, caricheremmo i dati, li convalideremmo e li contrassegneremmo per le violazioni. Quindi possiamo gestirli online mentre vengono caricati.

In questo senso, uno schema semantico ti offre un'ulteriore garanzia che tutti i dati inseriti saranno conformi al tuo schema. Ciò significa che tutti i dati nel database sono coerenti rispetto allo schema che è stato definito.

Questo frammento di codice in SHACL mostra come i dati devono rispettare determinate restrizioni:

```
:Person a sh:NodeShape, rdfs:Class ;
       sh:property [
       sh:path schema:worksFor ;
       sh:node :Company ;
] .

:Company a sh:Shape ;
       sh:property [
       sh:path     schema:name ;
       sh:datatype xsd:string;
] .
```

In TypeDB, questa convalida avviene nello schema in linguaggio TypeQL. Un'entità *Person* è definita solo quando svolge il ruolo *employee* in una relazione *employment*, che è correlata a un'entità *company* attraverso il ruolo *employer* e include un attributo di tipo *nome* e valore *string*.

```
define
person sub entity,
       plays employment:employee;
company sub entity,
       has name,
       plays employment:employer;
employment sub relation,
       relates employer,
       relates employee;
name sub attribute, value string;
```

6 Conclusioni

In conclusione, possiamo affermare che:

1. Rispetto al Web semantico, TypeDB riduce la complessità mantenendo un alto grado di espressività. Con TypeDB, evitiamo di dover imparare diversi standard per il Semantic Web, ognuno con alti livelli di complessità. Questo riduce la barriera all'ingresso.
2. TypeDB fornisce un'astrazione di livello superiore per lavorare con dati complessi rispetto agli standard web semantici. Con RDF modelliamo il mondo in triple, che è un modello di dati di livello inferiore rispetto allo schema a livello di concetto di relazione tra entità di TypeDB. Modellazione e query per relazioni di ordine superiore e dati complessi sono nativi in TypeDB.
3. Gli standard per il Semantic Web sono costruiti esclusivamente per il Web, mentre TypeDB funziona per sistemi chiusi. I primi sono stati progettato per funzionare con i dati collegati su un Web aperto con dati incompleti, mentre TypeDB funziona come un tradizionale sistema di gestione di database in un ambiente a mondo chiuso.

TypeDB ci fornisce un linguaggio che ci fornisce un modello a livello di concetto, un sistema di tipi, un linguaggio di query, un motore di reasoning e il controllo dello schema.

Per fare lo stesso con gli standard del Semantic Web sono necessari più standard con le rispettive implementazioni, ognuna con le proprie complessità intrinseche. In particolare, OWL è estremamente ricco di funzionalità, il che ha comportato un alto grado di complessità rendendolo inadatto alla maggior parte delle applicazioni software. Invece, TypeDB fornisce un equilibrio adeguato tra complessità ed espressività quando si tratta di lavorare con la rappresentazione della conoscenza e il reasoning automatico.