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

Comparazione tra TypeDB e i Property Graph Database

A. Messina, U. Maniscalco, P. Storniolo

Rapporto Tecnico N.:
RT-ICAR-PA-2021-05 **Giugno 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

Comparazione tra TypeDB e i Property Graph Database

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

Rapporto Tecnico N.:
RT-ICAR-PA-2021-05 **Giugno 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 4

2 LE SFIDE PER LAVORARE CON UN DATABASE A GRAFO 5

2.1 Introduzione 5

2.2 Modellazione di dati altamente interconnessi 5

2.3 Mantenere la coerenza dei dati 6

2.4 Scrivere appropriatamente le query 6

3 MODELLAZIONE E DEFINIZIONE DELLO SCHEMA 8

3.1 Filosofia di modellazione dei property graph 8

3.2 Modellazione a livello concettuale 8

3.3 Modellazione iperrelazionale 9

3.4 Relazioni N-arie e ternarie 11

3.5 Relazioni nidificate 12

4 LETTURA DEI DATI 14

4.1 Introduzione 14

4.2 Query di esempio 14

5 REASONING AUTOMATICO 16

5.1 Reasoning basato sul tipo 16

5.2 Reasoning basato su regole 18

6 CONCLUSIONI 24

1 Introduzione

I database a grafo sono diventati tecnologie tradizionali e stanno diventando sempre più preziosi per le organizzazioni di qualsiasi settore. Sono più flessibili dei database relazionali tradizionali in quanto ci consentono di sfruttare le relazioni nei nostri dati in un modo che i database relazionali non possono fare. In un momento in cui le organizzazioni cercano di ottenere il massimo dai propri dati, questo crea opportunità per qualsiasi organizzazione.

Tuttavia, lo sviluppo con database a grafo porta a molte sfide, in particolare, tra gli altri, quando si tratta di modellare i dati e mantenerne la coerenza.

Nei capitoli seguenti, vedremo come TypeDB si confronta con i database a grafo, in particolare i labelled property graphs, e come TypeDB affronta queste sfide. Sebbene entrambe le tecnologie condividano alcune somiglianze, sono fondamentalmente diverse. Vedremo come leggere e scrivere dati, come modellare domini complessi e vedremo anche la capacità di TypeDB di eseguire operazioni di reasoning automatico.

Le principali differenze tra TypeDB e i database a grafo possono essere riassunte come segue:

1. TypeDB fornisce uno schema a livello di concetto con un sistema di tipizzazione che implementa completamente il modello Entity-Relationship (ER). Invece, i database a grafo utilizzano vertici e archi senza vincoli di integrità imposti da uno schema.
2. TypeDB contiene un reasoner automatico integrato, mentre i database a grafo non forniscono capacità di reasoning native.
3. TypeDB è un livello di astrazione su un grafico. TypeDB sfrutta un database grafico dietro le quinte per creare un'astrazione di livello superiore, con il risultato che entrambe le tecnologie funzionano a diversi livelli di astrazione

Sono disponibili diverse tecnologie per i grafi. Alcuni di questi sono basati su RDF e SPARQL, altri sono linguaggi di query imperativi basati sui path come Gremlin. Il più popolare, tuttavia, è Cypher, che è cresciuto fino a diventare il linguaggio di query del database a grafo più adottato per i property graph. Per questo motivo, in questo confronto ci concentreremo solo su Cypher e sui labelled property graph.

2 Le sfide per lavorare con un database a grafo

2.1 Introduzione

Come suggerisce il nome, i database a grafo si basano sulla teoria dei grafi come concetto matematico per formare il modello di dati.

Considerano la connessione come un first-class citizen, il che li rende, rispetto ai database relazionali, particolarmente efficienti a rappresentare i dati connessi. Questo perché, nei database relazionali, le relazioni tra le entità non sono rese semanticamente esplicite nel modello, mentre i database a grafo memorizzano tali connessioni direttamente tra gli elementi.

Un database a grafo è caratterizzato dalle seguenti proprietà:

- Il modello di dati contiene nodi, relazioni, proprietà ed etichette
- I nodi possono avere proprietà
- I nodi possono avere tag con una o più etichette
- Le relazioni sono dirette e collegano i nodi per creare una struttura nel grafico
- Le relazioni possono avere proprietà

Rispetto ai database tradizionali, questa struttura dati è più adatta a lavorare con dati altamente interconnessi grazie alla elevata flessibilità ed espressività.

Tuttavia, lo sviluppo su un database a grafo richiede la comprensione di molti dettagli di implementazione di basso livello del graph computing. Come utente di un database a grafo, dobbiamo superare molte sfide preliminari e passare attraverso una ripida curva di apprendimento prima di poter utilizzare la tecnologia in modo ottimale. Questo crea un'alta barriera all'ingresso.

Nei paragrafi seguenti vengono evidenziate le principali difficoltà quando si ha a che fare con i database a grafo.

2.2 Modellazione di dati altamente interconnessi

A causa dell'alto livello di espressività fornito dai database a grafo, la modellazione di domini complessi su un grafo non è facile ed è equivalente alla conoscenza della modellazione, ovvero all'ingegneria dell'ontologia.

Per modellare una struttura a grafo sono necessari ingegneri specializzati. Questo approccio, tuttavia, non è scalabile per un'adozione diffusa.

Invece, ciò che serve è un sistema che permetta a qualsiasi ingegnere di modellare facilmente il proprio dominio su un grafo, senza dover essere esperto nell'ingegneria dell'ontologia o essere un esperto nella struttura dei dati del grafo sottostante.

2.3 Mantenere la coerenza dei dati

È essenziale che i dati che entrano nel database siano conformi al modello, ad esempio lo schema.

I database a grafo, come altri database NoSQL, delegano l'adesione ad uno schema a livello di applicazione.

Di conseguenza, è molto difficile fornire un sistema sufficientemente generico da garantire la coerenza dei dati rispetto al modello ma che mantenga il più alto livello di espressività possibile.

Non esistono dati senza uno schema, almeno se si desidera ricavarne un valore significativo: è definito esplicitamente (come nei database relazionali) o implicitamente a livello di utente.

Dato il grado di complessità dei dati altamente interconnessi acquisiti dai grafi, la mancanza di coerenza diventa un ostacolo significativo all'adozione di database a grafo.

2.4 Scrivere appropriatamente le query

Scrivere le giuste query che interrogheranno il database a grafo ha le sue sfide. È necessario essere espliciti nel definire il percorso da attraversare tra le istanze di dati. Dato che il modello di dati scelto governa i percorsi tra le tue istanze di dati, si devono progettare query specifiche per il modo in cui si è definito il modello.

Ciò che rende tale attività particolarmente impegnativa è che si potrebbe non aver modellato i dati nel modello più generico, coerente e concettualmente corretto (ad esempio, a volte si è definita una relazione come un nodo, altre volte come un arco).

Di conseguenza, le query potrebbero non raggiungere i dati corretti. Ogni interrogazione che si vuol porre al grafo necessita di una query personalizzata, scritta

in base al modello di dominio personalizzato, il che potrebbe non fornire il percorso ottimale per la query. Pertanto, ad esempio, si potrebbe non essere in grado di astrarre la query in funzioni che possano prendere l'input dell'utente come argomento e riutilizzare tali funzioni in più casi d'uso.

La scrittura di query di grafi è un compito impegnativo in quanto è necessario comprendere gli algoritmi dei grafi e le strutture di dati. Con l'ulteriore sfida di non essere in grado di astrarre e riutilizzare le query al punto in cui ci si può concentrare solo sul dominio del problema, l'adozione di database di grafici diventa molto difficile.

Per affrontare queste sfide, TypeDB astrae i bassi livelli di un database a grafio implementando un sistema di tipi. Ciò consente di astrarre i dettagli di implementazione di basso livello del grafo e consente di adottare la tecnologia senza una curva di apprendimento particolarmente ripida.

3 Modellazione e definizione dello schema

3.1 Filosofia di modellazione dei property graph

Il modello a grafi è costituito principalmente da nodi e relazioni che hanno nomi chiamati etichette. Le relazioni sono dirette e visualizzate come punte di freccia. Entrambi hanno proprietà (sotto forma di coppie chiave/valore) e sono chiamati nomi di proprietà.

Poiché le relazioni sono dirette, hanno sempre un nodo iniziale e un nodo finale. Per dare un'etichetta a una relazione si usa il concetto di “verbo”. Ad esempio, possiamo dire che un uomo è "sposato con" una donna e creare la relazione MARRIED_TO per rappresentare quel matrimonio.

Tuttavia, il modello a grafi non trova diretta corrispondenza in un modello concettuale o diagramma ER, poiché i nodi e gli archi non sempre vengono mappati direttamente a entità e relazioni. I database a grafo inoltre non offrono supporto nativo per concetti come, ad esempio, relazioni ternarie, relazioni n-arie o ruoli. Pertanto, per implementare un modello a grafi, dobbiamo prima passare attraverso un processo di normalizzazione e mappare il nostro modello concettuale (diagramma ER) al modello grafico.

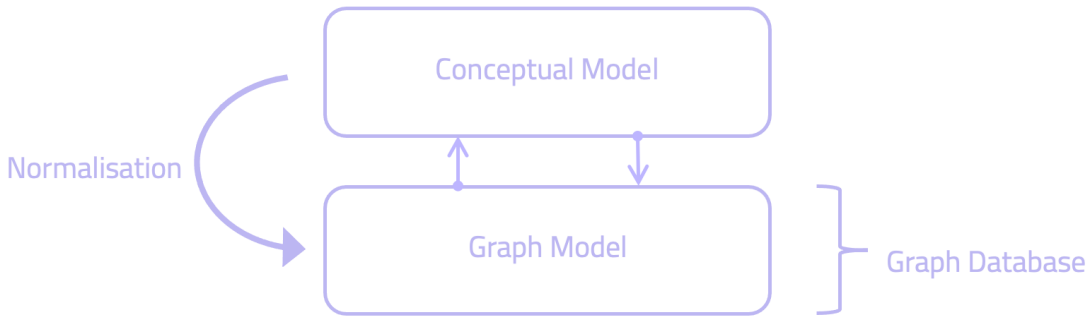


Figura 1 - In un database a grafo dobbiamo mappare il nostro modello concettuale al modello a grafo

3.2 Modellazione a livello concettuale

TypeDB fornisce uno schema a livello di concetto che implementa completamente il modello Entity-Relationship (ER). Lo schema di TypeDB è un sistema di tipi che implementa i principi della rappresentazione della conoscenza e del reasoning.

A differenza di un database a grafo, ciò significa che possiamo mappare qualsiasi diagramma ER direttamente implementandolo in TypeQL, evitando la necessità di passare attraverso un processo di normalizzazione. In TypeDB, creiamo una mappatura diretta del diagramma ER con entità, relazioni, attributi e ruoli su come lo implementiamo successivamente nel codice. Modellando a livello concettuale utilizzando un sistema di tipi, evitiamo la necessità di passare attraverso un processo di normalizzazione che sarebbe altrimenti richiesto in un database a grafo.

3.3 Modellazione iperrelazionale

Un componente centrale del sistema di tipi di TypeDB è la sua capacità di rappresentare iperrelazioni e iperentità (in TypeDB, entità, relazioni e attributi sono costrutti di modellazione di prima classe). Mentre in un grafo binario una relazione o un arco è solo una coppia di vertici, un'iperrelazione (o iperarco) è un insieme di vertici.

Ciò ci consente di modellare in modo nativo concetti come relazioni n-arie, relazioni nidificate o relazioni ristrette alla cardinalità. Usando questi costrutti, possiamo facilmente creare modelli di conoscenza complessi che possono evolversi in modo flessibile. Le iper-entità sono definite come entità con più istanze per un tipo di attributo, cosa non possibile in un database a grafo.

Il formalismo dell'ipergrafo di TypeDB si basa su tre premesse:

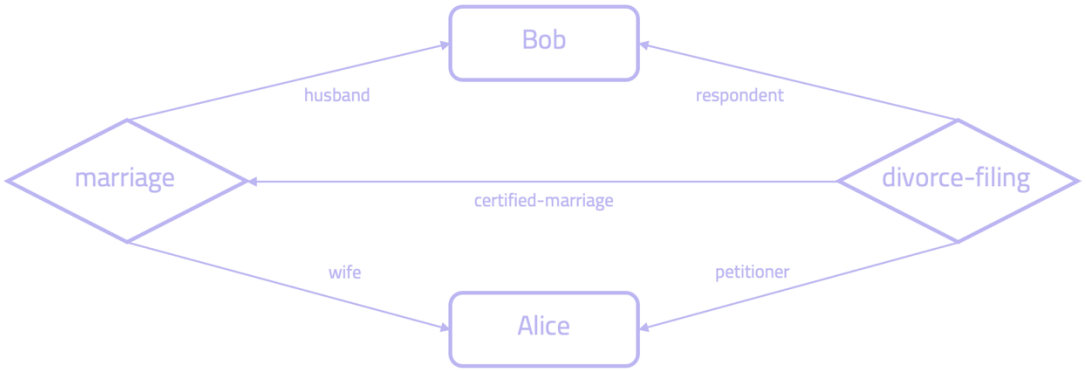
1. Un ipergrafo è costituito da un insieme non vuoto di vertici e un insieme di iperarchi
2. Un iperarco è un insieme finito di vertici (distinguibili da ruoli specifici che giocano in quell'iperarco)
3. Un iperarco è anche un vertice stesso e può essere collegato da altri iperarchi

Poiché i database property graph non consentono alle relazioni di connettere più di due nodi, non sono in grado di rappresentare da soli le iperrelazioni. Tuttavia, ci sono alcuni modi per aggirare questo problema aggiungendo una chiave esterna ai nodi che prendono parte a quell'iperrelazione o rappresentando l'iperrelazione come un nodo (reificazione). Poiché le chiavi esterne non si sposano bene con l’approccio a grafi, la reificazione è spesso l'approccio migliore.

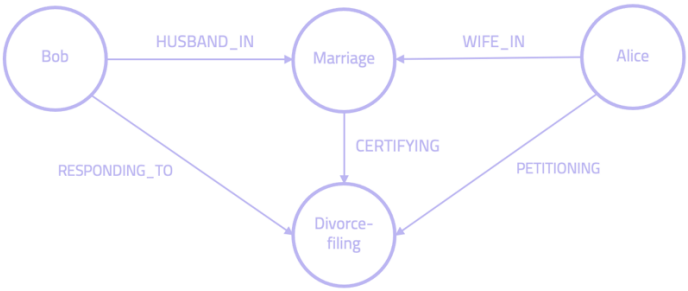
Tuttavia, la reificazione del grafico dovrebbe essere evitata in quanto può comportare il refactoring dell'intero grafico e la rottura del modello di dati. Richiederebbe anche modifiche a qualsiasi query e codice dell'applicazione che potrebbe produrre o utilizzare tali dati.

Nell'immagine sottostante, vediamo un esempio di due iperrelazioni modellate in TypeDB:

- *marriage*: descrive una relazione matrimoniale binaria tra Bob e Alice che interpretano rispettivamente i ruoli di *husband* e *wife* e la relazione matrimoniale che svolge il ruolo *certified-marriage* nella relazione *divorce-filing*
- *divorce-filing*: descrive una relazione ternaria di domanda di divorzio che coinvolge i tre attori con ruoli *certified-marriage*, *petitioner* e *respondent*



In questo esempio, vediamo che un'iperrelazione potrebbe essere considerata una raccolta di coppie di ruoli e interpreti di ruoli di cardinalità arbitraria. Poiché le iperrelazioni non possono essere rappresentate nativamente come relazioni binarie, l'esempio precedente potrebbe essere modellato in un database a grafo in questo modo:



Per rappresentare queste iperrelazioni in un modello a grafo con proprietà, dobbiamo reificare le due iperrelazioni (*marriage* e *divorce-filing*) e

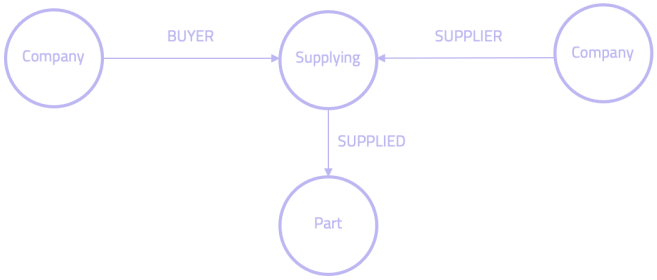
rappresentarle come nodi. Tuttavia, come accennato in precedenza, questo approccio è altamente indesiderabile poiché finiamo per modellare quattro nodi che rappresentano entità e relazioni. Abbiamo quindi compromesso il modello. Ecco perché la modellazione nativa nelle iperrelazioni offre un formalismo di rappresentazione dei dati più naturale e diretto, consentendo la modellazione a livello concettuale utilizzando diagrammi entità-relazione.

3.4 Relazioni N-arie e ternarie

Il sistema a tipi di TypeDB consente di modellare relazioni n-arie e ternarie. L'esempio seguente modella una relazione ternaria tra un fornitore, un acquirente e una parte. Possono essere collegati attraverso un'unica relazione che chiamiamo *supplying*.

Per prima cosa, diamo un'occhiata a come lo modelleremmo in un database a grafo. Poiché non possiamo rappresentare nativamente le tre entità in una relazione, possiamo memorizzare chiavi esterne come proprietà su ciascun nodo o, più preferibilmente, creare un nodo aggiuntivo (intermedio) per supportarlo, in modo simile a come abbiamo reificato il grafico nell'esempio sopra.

Ecco come creeremmo un nodo *supplying* (intermediario) che collegherebbe tutti e tre gli altri nodi:




```
(:Company)-[:SUPPLIER]->(:Supplying)
(:Company)-[:BUYER]->(:Supplying)
(:Part)<-[:SUPPLIED]-(:Supplying)
```

In TypeDB, invece di creare un nodo intermedio, creiamo una relazione *supplying* che si riferisce contemporaneamente al fornitore, all'acquirente e alla parte che viene fornita:

```
$supplier isa company; $part isa part; $buyer isa company;
(supplier: $supplier, supplied: $part, buyer: $buyer) isa supplying;
```

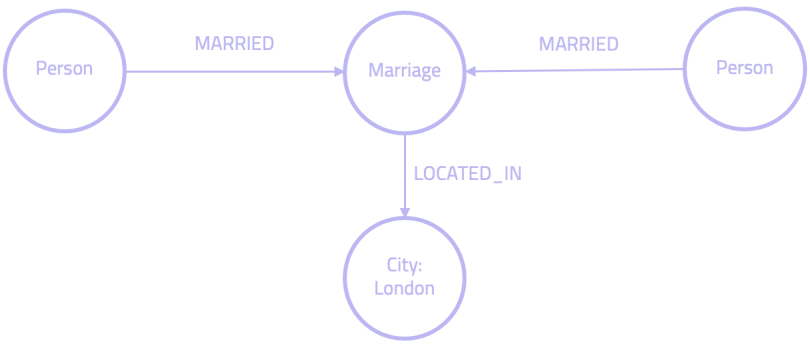
Lo schema in TypeDB sarebbe quindi il seguente (notare come la relazione *supplying* si riferisce a tre ruoli):

```
define
company sub entity,
plays supplying:supplier,
plays supplying:buyer;

part sub entity,
plays supplying:supplied;

supplying sub relation,
relates supplier,
relates buyer,
relates supplied;
```

3.5 Relazioni nidificate

In una relazione annidata, vogliamo che una relazione svolga un ruolo in un'altra relazione. Ad esempio, potremmo aver modellato un matrimonio come una relazione e vogliamo localizzare questo evento a Londra attraverso una relazione *located-in*. Per fare ciò sarebbe necessario collegare la relazione *marriage* attraverso una relazione di tipo *located* all'entità *London*.

In un database a grafo, non possiamo creare relazioni annidate. Invece, possiamo modellare un matrimonio tra due persone con una relazione *MARRIED_TO*. Tuttavia, connettere quell'arco a un nodo *city* diventa impossibile, poiché non possiamo avere una relazione connessa a un'altra relazione. Dovremmo per forza reificare il modello e trasformare l'arco *MARRIED_TO* in un nodo *marriage*, così da poter connettere quel nodo al nodo *city* attraverso l'arco *LOCATED_IN*.




```
(:Person)-[:MARRIED]->(marriage:Marriage)<-[:MARRIED]-(:Person)
(marriage)-[:LOCATED_IN]->(:City {name:"London"})
```

Il sistema a tipi di TypeDB supporta in modo nativo le relazioni nidificate come costrutti di modellazione. Per il modello suddetto, creeremmo una relazione *located* che collega la relazione *marriage* con la città "*London*":

```
$london isa city, has name "London";
$marriage (spouse: $person1, spouse: $person2) isa marriage;
($marriage, $london) isa located;
```

Lo schema sarebbe il seguente:

```
define
city sub entity,
plays locating:location;

person sub entity,
plays marriage:spouse;

marriage sub relation,
relates spouse,
plays locating:located;

locating sub relation,
relates located,
relates location;
```

4 Lettura dei dati

4.1 Introduzione

Per recuperare i dati in Cypher, descriviamo un modello grafico che inizia con un nodo associato che indica dove dovrebbe iniziare l'attraversamento. Possiamo aggiungere filtri e utilizzare un'istruzione *WHERE* per eseguire il pattern matching.

In TypeDB, la struttura di base per recuperare i dati è una query *match get*, in cui specifichiamo un modello di entità, relazioni, ruoli e attributi contro cui TypeDB recupererà i dati. La parola chiave *get* indica quali variabili specifiche vogliamo che vengano restituite, in modo simile all'istruzione *RETURN* in Cypher.

Diamo un'occhiata a due query di esempio e come si confrontano in Cypher e TypeQL.

4.2 Query di esempio

"*Chi sono gli amici comuni di Susan e Bob?*"

Vogliamo sapere quali siano tutti gli amici che sia Susan che Bob conoscono. In Cypher, possiamo scrivere come segue:

```
MATCH (bob:Person {name:"Bob"})-[:ISA_FRIEND_OF]->
(susan:Person {name:"Susan"})-[:ISA_FRIEND_OF]->(commonfriend:Person),
(bob)-[:ISA_FRIEND_OF]->(commonfriend)
RETURN commonfriend
```

Il modello descrive il percorso che collega Bob a Susan, attraverso la relazione *KNOWS*. Specifichiamo anche una relazione *KNOWS* tra Susan e Bob con un'altra persona. Quindi restituiamo quell'amico comune.

In TypeDB, la stessa query si presenta così (potremmo effettivamente ottimizzare questa query tramite il reasoner di TypeDB scrivendo una regola e deducendo gli amici comuni):

```
match $bob isa person, has name "Bob";
$susan isa person, has name "Susan"; $common-friend isa person;
($bob, $susan) isa friendship; ($susan, $common-friend) isa friendship;
($common-friend, $bob) isa friendship; get $common-friend;
```

In TypeDB, chiediamo le relazioni di tipo amicizia tra Bob e Susan, Susan e una

persona non definita, e Bob e una persona non definita. Notare che la relazione *ISA_FRIEND_OF* non si traduce direttamente in una relazione *friendship* in TypeDB. Il primo è un arco direzionale che rappresenta solo come una persona è amica di un'altra persona, ma non viceversa. Al contrario, la relazione *friendship* di TypeDB rappresenta che entrambe le persone svolgono il ruolo *friend*.

Vediamo un altro esempio:

"*Quali film sono usciti dopo il 2002 all'Odeon Cinema di Londra della Pixar?*"

In questa query, esaminiamo un modello più complesso. Cypher consente l'uso di una parola chiave *WHERE* che fornisce i criteri per filtrare i risultati della corrispondenza del modello. Nell'esempio seguente, otteniamo i titoli dei film che sono stati rilasciati dopo il 2002 in uno specifico cinema Odeon a Londra dalla Pixar:

```
MATCH (cinema:Cinema {name:'Odeon London'}),
 (london:City {name:'London'}),
 (pixar:Studio {name:'Pixar'}),
 (london) <-[:LOCATED_AT]-(cinema) <-[:RELEASED_AT]
  -(movie:Movie) - [:PRODUCED_BY] -> (pixar)
WHERE movie.year > 2002
RETURN movie.title
```

In TypeDB, la query sarebbe simile a questa:

```
match $cinema isa cinema, has name "Odeon London";
$london isa city, has name "London";
$studio isa studio, has name "Pixar";
$movie isa movie, has title $movie-title;
($london, $cinema) isa locating;
($cinema, $movie) isa release;
($movie, $studio) isa production, has year $year; $year > 2002;
get $movie-title;
```

In TypeDB, invece di usare *WHERE*, la corrispondenza dei modelli e il filtro possono essere eseguiti ovunque nella query. Il filtraggio di valori specifici può essere effettuato assegnando una variabile al valore di un attributo. Queste variabili possono quindi essere restituite all'utente aggiungendole nella clausola *get*. Nell'esempio sopra, chiediamo la variabile *$movie-title*, che è assegnata ai valori dell'attributo *title*, che è di proprietà dell'entità *film*.

5 Reasoning automatico

5.1 Reasoning basato sul tipo

Il sistema di tipi di TypeDB consente il reasoning basato sul tipo attraverso la modellazione di gerarchie di tipi in entità, attributi e relazioni. Una gerarchia di tipi per i veicoli in TypeDB potrebbe assomigliare a questa:

```
define

vehicle sub entity;
car sub vehicle;
sedan sub car;
coupe sub car
minivan sub car;
aircraft sub vehicle;
fixed-wing aircraft;
jet-aircraft sub aircraft;
truck sub vehicle;
garbage-truck sub truck;
heavy-truck sub truck;
```

Dato questo modello, se volessimo recuperare ogni singolo tipo di veicolo, piuttosto che specificare ogni singolo tipo uno per uno, possiamo semplicemente interrogare il tipo genitore *vehicle* e TypeDB, tramite il reasoning basato sul tipo, restituirà anche le istanze di tutti i sottotipi:

```
match $vehicle isa vehicle;
```

Sebbene i database a grafo non supportino le gerarchie di tipi e il ragionamento basato sui tipi, ci sono alcuni modi per aggirarlo. Ad esempio, se inseriamo un minivan, un coupé, un aereo a reazione e un camion della spazzatura, potremmo aggiungere i loro tipi principali come etichette aggiuntive a questi nodi. In TypeDB, ovviamente, non avremmo bisogno di specificare i loro tipi genitore poiché questi sarebbero dedotti dalla gerarchia dei tipi.

Cypher

```
CREATE (n:vehicle:car:minivan {name:'Maruti Suzuki Ciaz'})
CREATE (n:vehicle:car:coupe {name:'Audi A5'})
CREATE (n:vehicle:aircraft {name:'Concept Airplane'})
CREATE (n:vehicle:aircraft:jet_aircraft {name:'Boeing 747'})
CREATE (n:vehicle:garbage_truck {name:'Siku 1890 Super Bin Lorry'})
```

TypeQL

```
insert
$suzuki isa minivan, has name "Maruti Suzuki Ciaz";
$audi isa coupe, has name "Audi A5";
$aircraft isa aircraft, has name "Concept Airplane";
$boeing isa jet-aircraft, has name "Boeing 747";
$siku isa garbage-truck, has name "Siku Super Bin";
```

Se poi volessimo ottenere tutti i tipi di veicoli, possiamo semplicemente scrivere quanto segue in Cypher e TypeQL:

Cypher

```
MATCH (vehicle:Vehicle) RETURN
```

TypeQL

```
match $vehicle isa vehicle;
```

Tuttavia, questo approccio in Cypher di utilizzare più etichette diventa problematico in quanto non c'è modo di convalidare i nostri dati, quindi la query diventa pericolosa in quanto non avremmo alcuna garanzia che a tutti i sottotipi di veicoli sia stata assegnata l'etichetta corretta. Per ottenere questa certezza, dovremmo dichiarare esplicitamente tutte le etichette dei nodi che vogliamo che vengano restituite, rendendo la query molto più lunga e prolissa. In sostanza, siamo costretti a decidere tra una query lunga con risultati certi o una breve senza.

Cypher

```
MATCH (vehicle:Vehicle) RETURN vehicle UNION
MATCH (vehicle:Car) RETURN vehicle UNION
MATCH (vehicle:Sedan) RETURN vehicle UNION
MATCH (vehicle:Coupe) RETURN vehicle UNION
MATCH (vehicle:Minivan) RETURN vehicle UNION
MATCH (vehicle:Aircraft) RETURN vehicle UNION
MATCH (vehicle:Fixed_wing) RETURN vehicle UNION
MATCH (vehicle:Jet_aircraft) RETURN vehicle UNION
MATCH (vehicle:Truck) RETURN vehicle UNION
MATCH (vehicle:Garbage_truck) RETURN vehicle UNION
MATCH (vehicle:Heavy_truck) RETURN vehicle
```

TypeQL

```
match $vehicle isa vehicle;
```

Oltre alle entità, TypeDB consente anche il reasoning basato sul tipo per gli attributi e le relazioni. Ad esempio, possiamo modellare una gerarchia *employment* con due sottotipi: *part-time-employment* e *full-time-employment*. Lo schema sarebbe il seguente:

```
define
employment sub entity;
part-time-employment sub employment;
full-time-employment sub employment;
```

Poiché le relazioni in un database a grafo non possono avere più etichette, lo stesso approccio dell'utilizzo di più etichette usato sopra per i nodi non funzionerebbe per le relazioni. Invece, per questo esempio di gerarchia, per recuperare tutti i tipi di occupazione, dovremmo specificare tutte le etichette nella gerarchia. In TypeDB, chiederemmo solo la relazione genitore:

Cypher

```
MATCH
(organisation)-[employs]->(employees)
WHERE
employs:PART_TIME_EMPLOYED
OR
employs:FULL_TIME_EMPLOYED
OR
employs:EMPLOYED
RETURN employees
```

TypeQL

```
match
(employee: $person) isa employment;
get $person;
```

5.2 Reasoning basato su regole

In TypeDB, possiamo creare regole per astrarre e modularizzare la nostra logica di business ed eseguire ragionamenti automatizzati. I grafi con proprietà non supportano le regole. Le regole possono inferire qualsiasi tipo di concetto, ad esempio entità, relazioni o attributi.

Ad esempio, potremmo creare una regola che deduce la relazione tra fratelli, se due persone condividono lo stesso genitore. Se la relazione dedotta è chiamata *siblingship*, la regola sarebbe la seguente:

```
rule sibling-if-share-same-parent:
when {
  (parent: $parent, child: $child1) isa parenthood;
  (parent: $parent, child: $child2) isa parenthood;
} then {
  (sibling: $child1, sibling: $child2) isa siblingship;
};
```

Dopo aver definito questa regola, TypeDB dedurrebbe che i due figli sono fratelli se le condizioni nella regola sono soddisfatte dai dati precedentemente importati. Per chiamare l'inferenza, interroghiamo semplicemente per la relazione *siblingship* in una query *match*:

```
match (sibling: $sibling) isa siblingship; get $sibling;
```

Per ottenere la stessa risposta in un database a grafo, dovremmo scrivere manualmente la query con percorsi diversi che rappresentano le condizioni della regola (invece di interrogare direttamente la relazione di fratellanza). In questo esempio, la query Cypher potrebbe essere simile alla seguente, dove, oltre a interrogare direttamente i fratelli, chiediamo anche a tutte le persone che

condividono lo stesso genitore (che in TypeDB sopra abbiamo modellato come regola):

```
MATCH
(sibling1:Person)-[:SIBLING_OF]->(sibling2:Person)
OR
(sibling1:Person)<-[:PARENT_OF]-(:person)-[:PARENT_OF]->(sibling3:Person)
RETURN sibling1, sibling2, sibling3
```

Un esempio più complesso di reasoning automatico si ha quando i concetti dedotti dipendono da più regole (regole di concatenamento). Nell'esempio seguente, vogliamo recuperare tutte le persone che sono cugine l'una dell'altra, e per le quali sono state importate solo relazioni *parenthood* attraverso tre generazioni. Definite opportunamente le regole, saremmo in grado estrarre le relazioni *cousinship* semplicemente così:

```
match (cousin: $cousin) isa cousinship;
get $cousin;
```

La logica alla base dell'inferenza della relazione *cousinship* sarebbe la seguente:

- Quando la persona A ha un genitore B
- Se quel genitore B ha un genitore C
- Se quel genitore C ha un figlio D che non è un fratello di B
- E se quel bambino D ha un bambino E
- Allora la persona A e il bambino E sono cugini

Poiché non possiamo semplicemente rappresentare questa logica in una regola in un database a grafo, dobbiamo scriverla esplicitamente nella query stessa. Per questo esempio, avremmo:

```
MATCH
(parent:Person)-[PARENT_OF)->(child1:Person)
-[PARENT_OF]->(child4:Person),
(parent)-[PARENT_OF)->(child2:Person)
-[PARENT_OF]->(child3:Person)
RETURN child4, child3
```

Per rappresentare questa logica in TypeDB, possiamo creare due regole separate. La prima regola deduce la relazione di cugino. Questa regola dipende quindi dalla seconda regola, che deduce una relazione zio o zia.

Nella prima regola, deduciamo la relazione *cousinship*. La logica è la seguente:

- Quando il bambino A ha un genitore B
- E il genitore B ha un fratello C
- Se quel fratello C ha un figlio D
- Allora A e D saranno cugini

```
rule an-aunts-child-is-a-cousin:
when {
    $a isa person;
    $b isa person;
    $c isa person;
    $1 (parent: $a, child: $b) isa parenthood;
    $2 (uncle-aunt: $a, nephew-niece: $c) isa uncle-auntship;
} then {
    (cousin: $b, cousin: $c) isa cousinship;
};
```

Una delle condizioni di questa regola è la relazione *uncle-auntship*, che sarebbe una relazione dedotta e dipenderebbe dalla nostra seconda regola, che si basa sulla seguente logica:

- Quando il genitore A ha un figlio B
- Se quel genitore A ha un fratello C
- Allora B e C saranno in una relazione uncle-auntship

```
rule uncle-aunt-between-child-and-parent-sibling:
when {
    $a isa person;
    $b isa person;
    $c isa person;
    $1 (parent: $b, child: $a) isa parenthood;
    $2 (sibling: $b, sibling: $c) isa siblingship;
} then {
    (nephew-niece: $a, uncle-aunt: $c) isa uncle-auntship;
};
```

Le regole in TypeDB possono essere utilizzate anche per archiviare logiche di business più complesse. Ad esempio, supponiamo di voler eseguire una query per tutte le pianificazioni che si sovrappongono in una rete logistica. In Neo4j possiamo scrivere questa query così:

```
MATCH
(sched1: Schedule), (sched2: Schedule)
WHERE
sched1.end > sched2.start
sched1.end <= sched2.end
RETURN
```

Piuttosto che rappresentare la logica che determina se le pianificazioni si sovrappongono in una query, con TypeDB possiamo rappresentarlo in una regola. In questo modo, possiamo semplicemente interrogare direttamente tutte le pianificazioni che partecipano a una relazione *overlaps* per ottenere tutte le pianificazioni sovrapposte:

```
match $schedule isa schedule; ($schedule) isa overlaps; get $schedule;
```

La regola che deduce la relazione *overlaps* ha la forma seguente:

```
rule overlapping-schedules:
when {
  $schedule1 isa schedule, has end $1End;
  $schedule2 isa schedule, has start $2Start, has end $2End;
  $1End > $2Start;
  $1End <= $2End;
} then {
  ($schedule1, $schedule2) isa overlaps;
);
```

Le regole in TypeDB funzionano particolarmente bene quando abbiamo bisogno di dedurre connessioni tra dati altrimenti non connessi. Ad esempio, supponiamo di voler trovare tutte le malattie per le quali una determinata persona ha un fattore di rischio. Nel prossimo esempio, vogliamo dedurre quelle risposte usando la seguente logica:

- Se qualcuno consuma più di 10 unità di alcol a settimana, rischia di avere il diabete di tipo II e l'ipoglicemia
- Se al genitore di qualcuno è stato diagnosticato il diabete II e/o l'artrite, allora anche lui rischia di avere quelle malattie
- Se qualcuno consuma più di 12 sigarette a settimana, rischia di avere sclerosi multipla, cancro ai polmoni, ipertensione, sclerosi multipla, malattie polmonari ostruttive croniche e/o malattie cardiache, colesterolo alto

Quindi, per esempio, se John Doe soddisfa queste condizioni, vogliamo che ci venga restituito un elenco di malattie per le quali è a rischio, come nel modo seguente:

```
Diabetes Type II
Hypoglycemia
Arthritis
High Blood Pressure
Multiple Sclerosis
Chronic Obstructive Pulmonary Heart Disease
```

Per fare questa query in Cypher, dobbiamo rappresentare tutta la logica che determina i fattori di rischio di qualcuno nella query stessa:

```
MATCH (person:Person {name: "John2"})
OPTIONAL MATCH (person)-[consumes:CONSUMES]->(s:substance)
OPTIONAL  MATCH  (disease:Disease)<-[:HAS_DIAGNOSIS]-(:Person)<-[:IS_CHILD]-
(person)

WITH  COLLECT({units_per_week:consumes.units_per_week,  name:s.name})  AS
substance_consumptions,  COLLECT(disease) as  p_diseases

CREATE (td:TempDisease{diseases: []})
WITH substance_consumptions, td, p_diseases

FOREACH( sc in substance_consumptions |
  FOREACH( _ in case when sc.units_per_week > 10 and sc.name='Alcohol' then
[1] else [] end |
    SET td.diseases=td.diseases + "Diabetes Type II" )
  FOREACH( _ in case when sc.units_per_week > 10 and sc.name='Alcohol' then
[1] else [] end |
```

```
    SET td.diseases=td.diseases + "Hypoglycemia" )
  FOREACH( _ in case when sc.units_per_week > 12 and sc.name='Cigarettes' then
[1] else [] end |
    SET td.diseases=td.diseases + "Lung Cancer" )
  FOREACH( _ in case when sc.units_per_week > 12 and sc.name='Cigarettes' then
[1] else [] end |
    SET td.diseases=td.diseases + "High Cholesterol" )
  FOREACH( _ in case when sc.units_per_week > 12 and sc.name='Cigarettes' then
[1] else [] end |
    SET td.diseases=td.diseases + "High Blood Pressure" )
  FOREACH( _ in case when sc.units_per_week > 12 and sc.name='Cigarettes' then
[1] else [] end |
    SET td.diseases=td.diseases + "Multiple Sclerosis" )
  FOREACH( _ in case when sc.units_per_week > 12 and sc.name='Cigarettes' then
[1] else [] end |
    SET td.diseases=td.diseases + "Chronic Obstructive Pulmonary Disease" )
  FOREACH( _ in case when sc.units_per_week > 12 and sc.name='Cigarettes' then
[1] else [] end |
    SET td.diseases=td.diseases + "Heart Disease" )
)

FOREACH( disease IN p_diseases |
  FOREACH( _ in case when disease.name="Arthritis" then [1] else [] end |
    SET td.diseases=td.diseases + "Arthritis" )
  FOREACH( _ in case when disease.name="Diabetes Type II" then [1] else [] end
|
    SET td.diseases=td.diseases + "Diabetes Type II" )
)

WITH td, td.diseases as diseases

DELETE td

WITH diseases

UNWIND diseases AS disease
RETURN collect(DISTINCT(disease)) as diseases
```

In TypeDB, invece, scriviamo solo una query che recupera la relazione *risk-factor* per John Doe:

```
match $john isa person, has name "John Doe"; $disease isa disease;
($john, $disease) isa risk-factor; get $disease;
```

Questa relazione *risk-factor* è una relazione dedotta che contiene la logica necessaria per recuperare le malattie di cui qualcuno è a rischio. Li dividiamo in tre regole separate per modularizzare la logica:

```
rule alcohol-risk-of-diabetes:
when {
    $person isa person;
    $c(consumer: $person, consumed-substance: $substance) isa consumption, has
units-per-week $units;
    $units >= 10;
    $substance isa substance, has name "Alcohol";
    $disease isa disease, has name "Diabetes Type II";
    $disease2 isa disease, has name "Hypoglycemia";
} then {
  (person-at-risk:   $person,   risked-disease:   $disease,   risked-disease:
$disease2) isa alcohol-risk-factor;
};

rule heriditary-risk-of-diabetes:
when {
```

```
    $person isa person;
    $parent isa person;
    (parent: $parent, child: $person) isa parentship;
    (patient: $parent, diagnosed-disease: $disease) isa diagnosis;
    $disease isa disease, has name "Diabetes Type II";
    $disease2 isa disease, has name "Arthritis";
} then {
  (person-at-risk:  $person,  risked-disease:  $disease,  risked-disease:
$disease2) isa hereditary-risk-factor;
};

rule smoking-risk-of-multiple-sclerosis:
when {
    $person isa person;
    (consumer: $person, consumed-substance: $substance) isa consumption, has
units-per-week $units;
    $units >= 12;
    $substance isa substance, has name "Cigarettes";
    $disease isa disease, has name "Multiple Sclerosis";
    $disease2 isa disease, has name "Lung Cancer";
    $disease3 isa disease, has name "High Blood Pressure";
    $disease4 isa disease, has name "Multiple Sclerosis";
    $disease5 isa disease, has name "Chronic Obstructive Pulmonary Disease";
    $disease6 isa disease, has name "Heart Disease";
} then {
  (person-at-risk:  $person,  risked-disease:  $disease,  risked-disease:
$disease2, risked-disease: $disease3,
  risked-disease:  $disease4,  risked-disease:  $disease5,  risked-disease:
$disease6) isa smoking-risk-factor;
};
```

6 Conclusioni

In conclusione, abbiamo visto come:

1. Il sistema di tipi di TypeDB fornisce uno schema a livello di concetto che implementa completamente il modello Entity-Relationship (ER), mentre i database a grafo utilizzano vertici e archi senza vincoli di integrità imposti sotto forma di schema
2. TypeDB contiene un reasoner automatico integrato, i database a grafo non forniscono capacità di reasoning native
3. TypeDB è un livello di astrazione su un database a grafo. TypeDB sfrutta dietro le quinte un database a grafo per creare un'astrazione di livello superiore, con il risultato che entrambe le tecnologie funzionano a diversi livelli di astrazione

In sintesi, TypeDB 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 la verifica dello schema. Il risultato è un linguaggio di livello superiore che astrae le complessità di basso livello inerenti a un database a grafo.

Questo confronto ha avuto lo scopo di fornire somiglianze e differenze di alto livello tra entrambe le tecnologie, ma, ovviamente, c'è molto di più in TypeDB e nei database a grafo rispetto a quanto abbiamo cercato di mostrare qui.