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

Confronto tra TypeQL e SQL

A. Messina, U. Maniscalco, P. Storniolo

Rapporto Tecnico N.:
RT-ICAR-PA-2021-03 **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

Confronto tra TypeQL e SQL

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

Rapporto Tecnico N.:
RT-ICAR-PA-2021-03 **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

Sin dagli anni ’70, SQL è stato lo standard di fatto in termini di linguaggio per interagire con i database. Essendo un linguaggio dichiarativo, è abbastanza semplice scrivere query e costruire potenti applicazioni. I database relazionali, però, entrano in difficoltà quando hanno a che fare con dati complessi ed interconnessi. Con questo tipo di dati, infatti, con SQL sorgono problemi soprattutto nella modellazione e nell’interrogazione dei dati.

TypeQL è il linguaggio di query utilizzato in TypeDB. Proprio come SQL è il linguaggio di query standard nei database relazionali, TypeQL è il linguaggio di query di TypeDB. Sia SQL che TypeQL sono linguaggi di query dichiarativi che astraggono le operazioni di livello inferiore.

Entrambi sono:

- linguaggi che cercano di essere leggibili e comprensibili;
- linguaggi che cercano di consentire di porre domande a un livello più alto;
- linguaggi grazie ai quali il sistema capisce come eseguire operazioni di basso livello.

In termini pratici, ciò significa che tali linguaggi diventano accessibili a gruppi di persone che altrimenti non avrebbero potuto accedervi. In questo documento vengono esaminati concetti comuni specifici e ci si concentrerà poi sul confronto e sull'esplorazione delle differenze tra i due linguaggi.

1.1 Le origini di SQL e del Modello Relazionale

Nel 1970, fu pubblicato un articolo del matematico Edgar Codd, noto come "Ted", nel quale vennero introdotti due linguaggi: un'algebra relazionale e un calcolo relazionale per esprimere query estremamente complesse. Quando uscirono, questi erano considerati uno strano tipo di notazione matematica. Per trasformare queste idee in un sistema di gestione di database, Ted creò un gruppo di ricerca chiamato System R, con sede presso le strutture di ricerca IBM a San Jose.

A quei tempi, i database erano principalmente basati su modelli di navigazione, di rete e gerarchici, per i quali era necessario conoscere a priori il livello dei dati fisici prima di poter scrivere un piano di navigazione per descrivere le query. Ted, visto la complessità insita in tutto ciò, volle rendere più semplice la scrittura di query di database.

Tuttavia, poiché le idee di Ted erano basate sulla notazione matematica e sul simbolismo matematico, erano difficili da capire e non molto accessibili alla maggior

parte delle persone, quindi due membri di System R affrontarono questo problema creando un semplice linguaggio di query: *SEQL*. Poiché questo nuovo linguaggio era basato esclusivamente su parole inglesi, fu un punto di svolta perché rese molto più semplice per le persone comprendere le idee di Ted.

Alla fine degli anni '70, i database relazionali erano cresciuti in popolarità e il mondo era arrivato ad accettare quanto SQL e il modello relazionale fossero superiori ai suoi predecessori. La storia da allora è ben nota: i database relazionali sono diventati lo standard per la creazione di software proprio quando il mondo veniva coinvolto nella rivoluzione digitale.

1.2 L'essenza di SQL e di TypeQL

Per comprendere TypeQL, è utile esaminare le idee sottostanti che hanno creato SQL, poiché sono concettualmente strettamente correlate. L'essenza di TypeQL e SQL può essere riassunta nel modo seguente:

- Un linguaggio che può essere letto e compreso intuitivamente. Un linguaggio soddisfa questi criteri quando appare semplice, gestibile e presenta un grado di somiglianza con il testo naturale.
- Un linguaggio che consente di fare interrogazioni ad un livello più alto. Qui si fa riferimento a un linguaggio che consente all'utente di descrivere operazioni a un nuovo e più alto livello semantico.
- Un linguaggio con il quale il sistema capisce come eseguire operazioni di baso livello. Poiché l'utente descrive le operazioni ad alto livello, il sistema si occupa delle operazioni senza che l'utente debba pensarci.

In tal senso, sia SQL che TypeQL sono linguaggi che astraggono operazioni di basso livello. In termini pratici, ciò significa che tali linguaggi diventano accessibili a gruppi di persone che altrimenti non avrebbero potuto accedervi. Ciò significa che sono abilitati a creare valore, mentre coloro che potrebbero già utilizzarli ora possono fare le cose molto più velocemente. Una cosa simile si può dire di Python, ad esempio, un linguaggio di programmazione di alto livello che ha permesso a milioni di programmatori di creare software senza doversi preoccupare di operazioni di livello inferiore grazie alle astrazioni.

2 Modellazione e Definizione degli schemi

2.1 Introduzione

Innanzitutto, vediamo come mettere a confronto SQL e TypeQL in termini di modellazione dei dati. Utilizziamo il diagramma entità-relazioni (diagramma ER) in quanto è lo strumento di modellazione più comune in uso. Un modello di base è composto dai tipi di entità e dalle relazioni che possono esistere tra di essi (modello concettuale). Di seguito è riportato un esempio di diagramma ER.

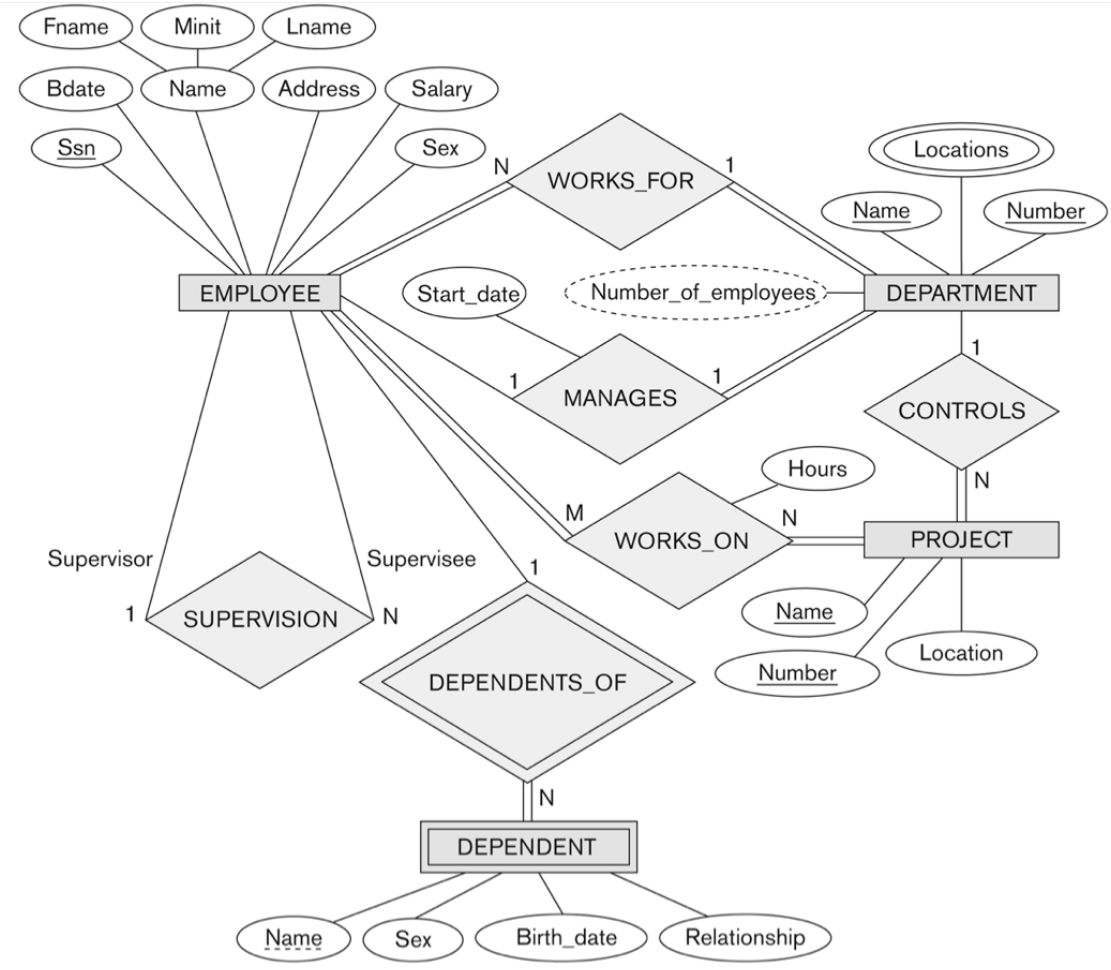

Figura 1 - Esempio di diagramma ER. I quadrati sono le entità, i rombi sono le relazioni, le ellissi sono gli attributi

2.2 Modellazione in SQL

Se vogliamo implementare questo modello in un database relazionale, passiamo prima attraverso un processo di normalizzazione. Iniziamo dalla prima forma normale (1NF) e, cercando cose come le dipendenze funzionali e le dipendenze

transitive, alla fine arriviamo alla nostra terza forma normale desiderata (3NF).

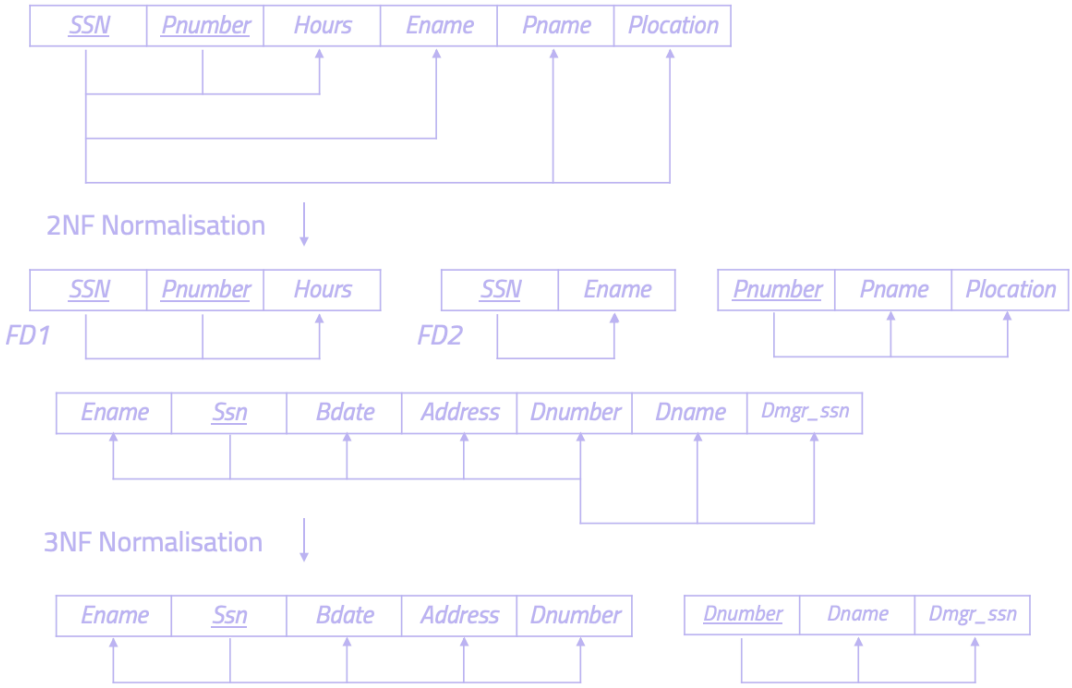

Figura 2 - Normalizzazione da 1NF a 3NF

Dopo questo processo di normalizzazione, arriviamo al nostro modello logico in 3NF che possiamo implementare in un database relazionale. Siamo passati dal nostro modello concettuale (diagramma ER) al modello logico (3NF), senza mai dover scendere al livello fisico del database. Questa è stata proprio la svolta che il modello relazionale ci ha portato, astraendo il livello fisico. La chiamiamo **indipendenza fisica dei dati**.

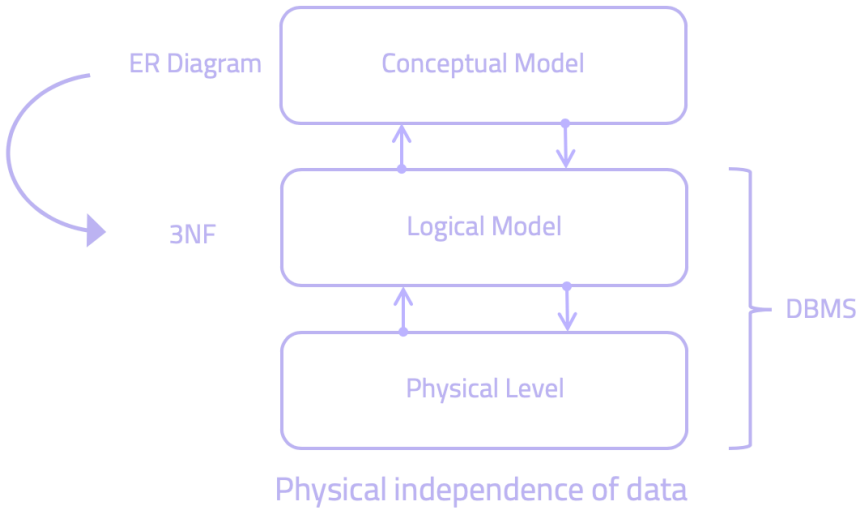

Figura 3 - SQL ci dà l'indipendenza fisica dei dati

2.3 Modellazione in TypeQL

Vediamo ora come questo si confronta con TypeQL. Possiamo mappare qualsiasi diagramma ER direttamente in TypeQL, il che significa che non abbiamo bisogno di passare attraverso un processo di normalizzazione. TypeQL ci consente di creare una mappatura diretta del diagramma ER con entità, relazioni, attributi e ruoli in modo coerente a come verrà implementato successivamente nel codice. Questo è diverso da SQL, dove dobbiamo imporre una struttura tabulare sul nostro modello come livello logico (come descritto sopra).

Ciò significa che eliminiamo completamente il processo di normalizzazione richiesto in SQL e continuiamo a lavorare sul modello concettuale. In altre parole, TypeQL astrae sia il modello logico che quello fisico. In questo senso, laddove SQL ci ha dato l'indipendenza fisica dei dati, **TypeQL ci dà l'indipendenza logica dei dati**.

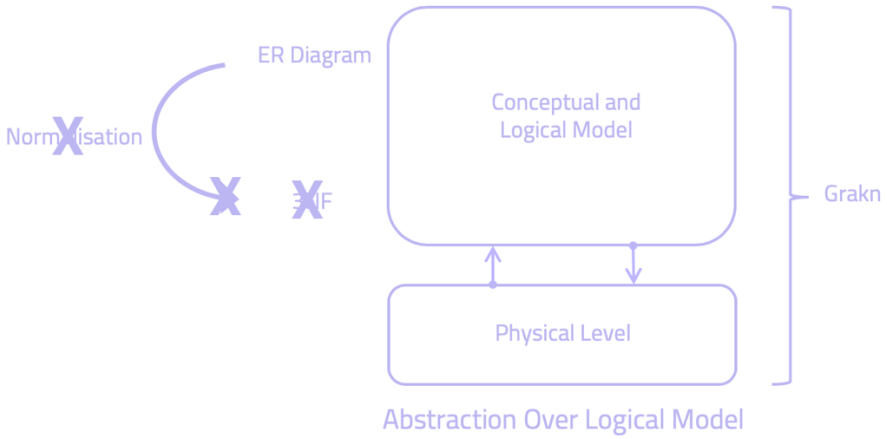

Figura 4 - Modellando a livello concettuale, TypeQL ci fornisce un'astrazione sul modello logico

2.4 Definizione di schemi in SQL e TypeQL

Consideriamo adesso alcuni dati reali. Chiunque abbia studiato SQL probabilmente ha familiarità con il set di dati Northwind. Esso contiene dati di vendita per Northwind Traders, una società fittizia di esportazione e importazione di specialità alimentari.

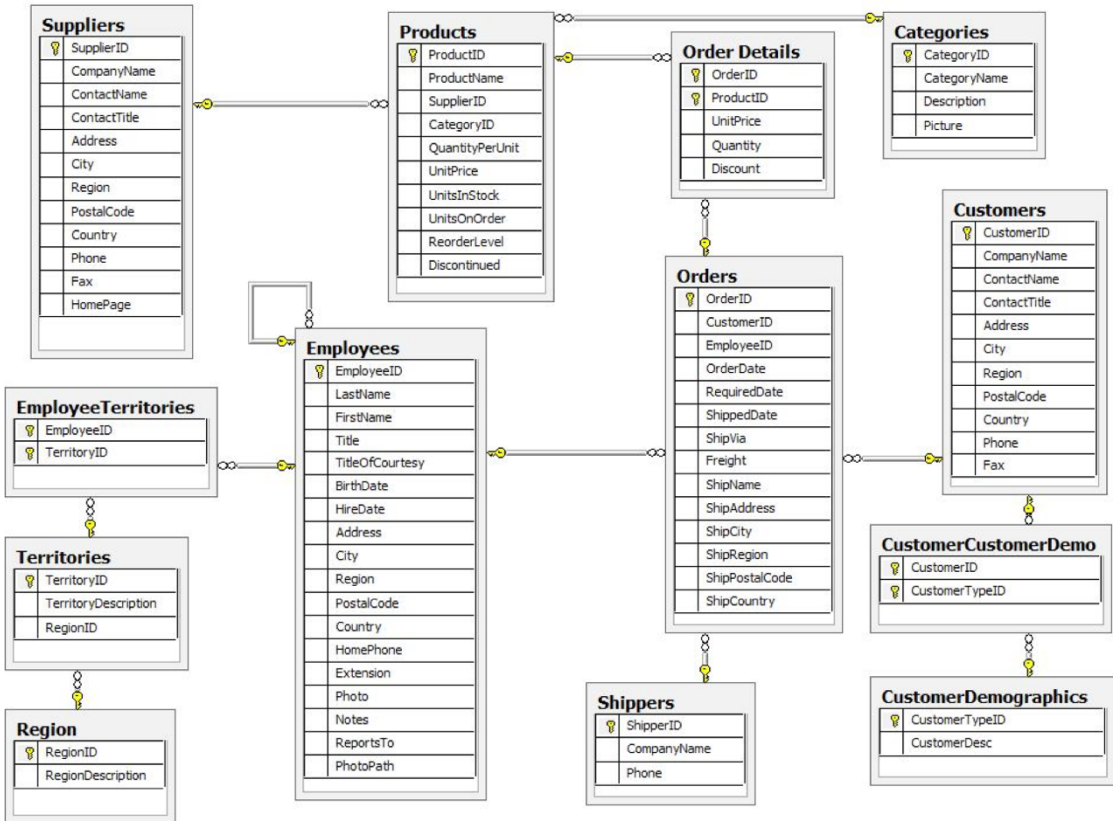

Figura 5 - Lo schema Northwind

Come possiamo definire la tabella **products** mostrata sopra in TypeQL e SQL? Di seguito vediamo la sintassi TypeQL che definisce l'entità **prodotto** e la **relazione** corrispondente. Vediamo anche quali sarebbero le istruzioni SQL che creano la nuova tabella e gli attributi corrispondenti.

TypeQL

```
define
product sub entity,
key product-id,
has product-name,
has quantity-per-unit,
plays assignment:assigned;
product-id sub attribute, value double;
product-name sub attribute, value string;
quantity-per-unit sub attribute, value double;
assignment sub relation,
relates category,
relates assigned;
```

SQL

```
CREATE TABLE products (
product_id smallint NOT NULL PRIMARY KEY,
product_name character varying(40) NOT NULL,
category_id smallint,
quantity_per_unit character varying(20),
FOREIGN KEY (category_id) REFERENCES categories
);
```

Alcuni punti importanti:

- Possiamo vedere che la tabella SQL ha tre attributi, ognuno con il proprio tipo di valore, che possiamo definire anche in TypeQL. Uno di questi attributi è una chiave primaria, che definiamo in TypeQL utilizzando la parola chiave *key*.
- Nell'istruzione SQL c'è anche una chiave esterna, che a seconda del nostro modello, modelliamo in TypeQL come relazione. Lo facciamo collegando l'entità *product* alla relazione *assignment* utilizzando il ruolo *assigned*.
- In TypeQL non esiste il concetto di valori *null*. Se un concetto non ha un attributo, in realtà non lo ha. Questo perché in un contesto grafico un attributo nullo viene semplicemente omesso dal grafico.
- Infine, un punto importante è che nel modello TypeQL, gli attributi sono first-class citizen, a differenza che in SQL.

Riassumendo:

- La modellazione di un diagramma ER in SQL comporta un processo di normalizzazione da 1NF a 3NF
- Il diagramma ER viene mappato in mado naturale su TypeQL e non è necessario eseguire alcun tipo di normalizzazione.

3 Lettura e Scrittura dei Dati

3.1 Inserimento dei dati

Vediamo come scriviamo e leggiamo i dati utilizzando operatori relazionali. Innanzitutto, utilizzando lo schema Northwind, inseriamo i seguenti dati: un nuovo *prodotto* con nome *Chocolate*, id prodotto *12*, quantità per unità *421* e nome categoria *Confections*.

In SQL, eseguiamo due query. Prima recuperiamo l'id di *Confections*, poi inseriamo la nuova riga.

```
SELECT category.categoryID
FROM category
WHERE category.CategoryName = "Confections";

INSERT INTO products
VALUES (12, "Chocolate", 42, 421)
```

In TypeQL facciamo qualcosa di diverso. Per prima cosa abbiniamo la categoria *Confections*, assegniamo il risultato alla variabile *$c* e poi inseriamo i nuovi dati.

```
match
$c isa category,
  has name "Confections";

insert
$p isa product,
  has product-id 12,
  has product-name "Chocolate",
  has quantity-per-unit 421;
```

3.2 Lettura dei dati

Il costrutto operativo di base in SQL è l'espressione **SELECT - WHERE - FROM**, utilizzata per derivare nuove tabelle da quelle esistenti:

```
SELECT column1, column2, …
FROM table_name
WHERE condition;
```

L'espressione equivalente in TypeQL è l'espressione **match-get**:

```
match
$a isa thing,
  has attribute $v;
get $a, $v;
```

Vediamo adesso alcuni degli operatori più comunemente usati in SQL e come appaiono in TypeQL. In tal modo, vedremo come pensare a una query concettualmente, invece di guardare il codice effettivo.

La differenza tra una query TypeQL e una query SQL è che in TypeQL si pensa alle query a livello concettuale; allo stesso modo in cui pensiamo semanticamente a una domanda, piuttosto che imporre su di essa una struttura tabellare.

Nelle figure sottostanti, a sinistra, viene mostrata la rappresentazione tabellare della query in SQL, e a destra la rappresentazione concettuale della query in TypeQL (i quadrati sono entità, i rombi sono relazioni e i cerchi sono attributi).

3.2.1 Proiezione

Questo operatore SQL restituisce una tabella che contiene tutte le righe che rimangono dopo la rimozione di colonne specifiche. In TypeQL, chiediamo un'entità con valori di attributi specifici.

Un esempio è:

Restituisci tutti gli ID prodotto e i relativi prezzi unitari.

Figura 6 – Tabella proiezione (a sinistra) e operazione equivalente in TypeQL (a destra).

3.2.2 Restrizione

Questo operatore SQL ci fornisce una tabella con righe da una tabella specificata a una condizione specifica. In TypeQL, chiediamo un'entità e i suoi attributi filtrati da un attributo con valori specifici.

Ad esempio:

Restituisci gli id prodotto e i nomi dei prodotti, per i prodotti con un prezzo unitario superiore a 12,5.

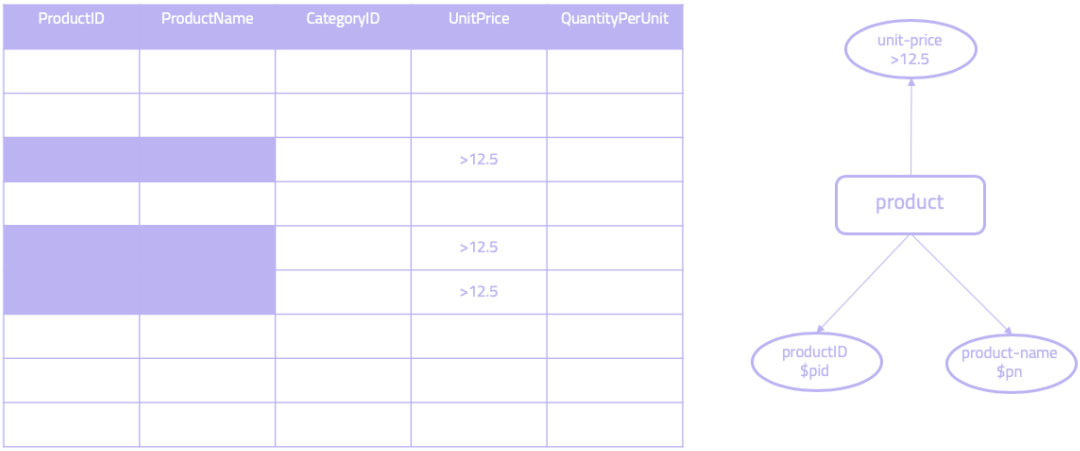

Figura 7 – Tabella restrizione (a sinistra) e operazione equivalente in TypeQL (a destra).

3.2.3 Unione

Questo operatore SQL restituisce le righe delle tabelle che appaiono in una o entrambe le due tabelle specificate. In TypeQL, chiediamo entità e i loro attributi che sono collegati a una o due altre entità attraverso una relazione.

Ad esempio:

Ottieni tutte le diverse città in cui si trovano fornitori e clienti.

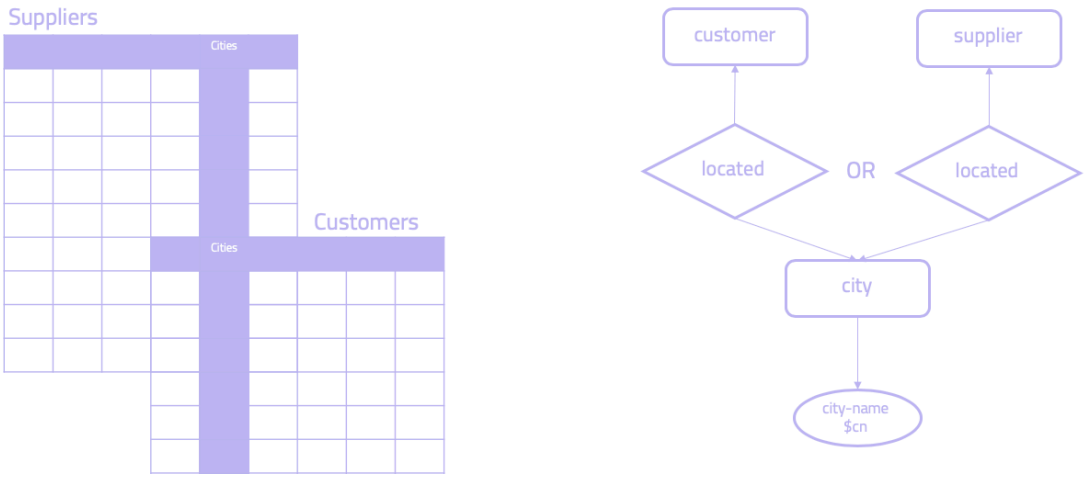

Figura 8 - Tabella unione (a sinistra) e operazione equivalente in TypeQL (a destra).

3.2.4 Intersezione

Questo operatore SQL restituisce una tabella con tutte le righe che appaiono in entrambe le due tabelle specificate. In TypeQL, chiediamo un'entità collegata a due entità specifiche tramite una relazione.

Ad esempio:

Ottieni tutte le città in cui si trovano contemporaneamente fornitori e clienti.

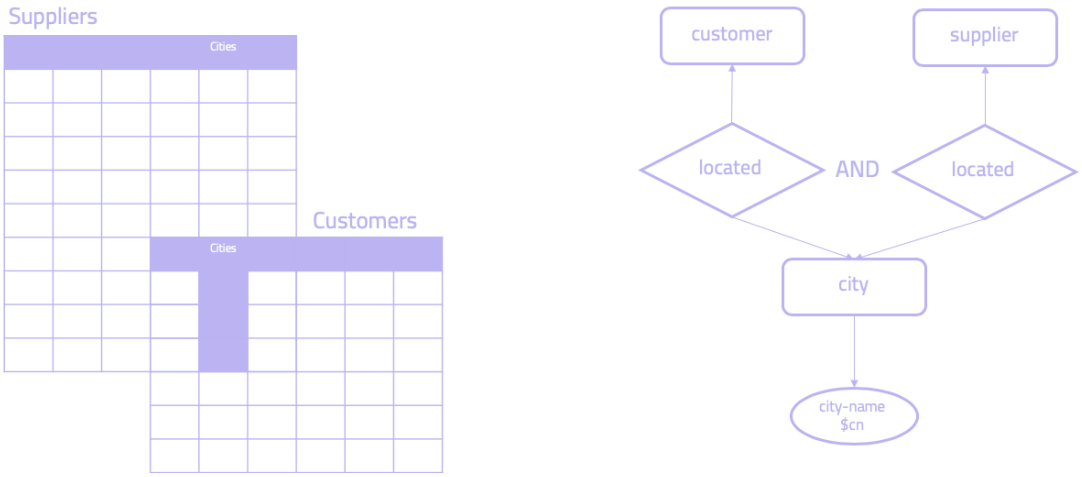

Figura 9 - Tabella intersezione (a sinistra) e operazione equivalente in TypeQL (a destra).

3.2.5 Join

È l'operatore SQL più famoso: una join restituisce una tabella contenente tutte le possibili righe che sono una combinazione di due righe, una da ciascuna delle due tabelle specificate, in modo tale che due righe che contribuiscono a una determinata riga di risultati abbiano valori comuni per gli attributi comuni delle due tabelle. In TypeQL, chiediamo le entità e i loro attributi che sono collegati tramite una relazione specifica. Ciò significa che non abbiamo bisogno di alcuna tabella di join, sia nel caso di relazioni 1-1, che relazioni 1-molti o relazioni molti-molti. Il concetto non è più necessario in TypeDB.

Ad esempio:

Ottieni tutte le diverse città in cui si trovano fornitori e clienti.

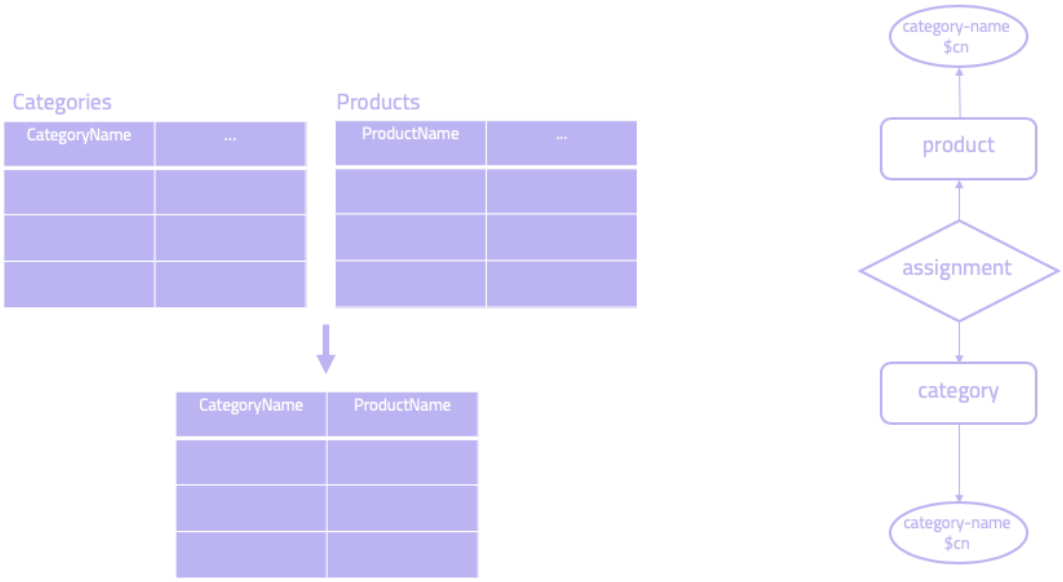

Figura 10 - Tabella join (a sinistra) e l'operazione equivalente in TypeQL (a destra).

4 Modellazione ad alto livello in TypeQL

4.1 Introduzione

Rispetto a SQL, TypeQL ci consente di modellare a un livello di astrazione più elevato. Ciò significa che quando pensiamo alle query, in realtà non dovremmo pensare in termini di operatori relazionali, come abbiamo appena fatto nella sezione precedente. Invece, dovremmo ripensare al nostro modello e sfruttare l'espressività di TypeQL. Quindi, rivediamo il set di dati Northwind e pensiamo a come possiamo modellarlo in modo diverso. In particolare, diamo un'occhiata a come utilizziamo l'ipergrafo e le capacità di reasoning di TypeDB.

4.2 Modellazione in TypeQL

Se prendiamo le tabelle *suppliers*, *products*, *employees*, *orders* e *customers* dal set di dati Northwind, possiamo modellarlo in TypeQL creando le seguenti entità:

1. **Product**: Associamo questa entità direttamente alla tabella *products* (usando invece il termine singolare).
2. **Order**: mappa direttamente alla tabella *orders*.
3. **Employee**: anche questo viene mappato direttamente alla tabella *employee*.
4. **Company**: poiché sia la tabella dei fornitori che quella dei clienti si riferiscono alle aziende, decidiamo di modellarle come un unico tipo di entità. Possiamo definire un'azienda come un fornitore o un cliente con ruoli (vedi sotto).

Creiamo quindi le seguenti relazioni e i ruoli corrispondenti:

1. **Sale**: possiamo modellare questa relazione come ternaria, in quanto avviene una vendita tra un *Order* (che svolge il ruolo *order*), *Company* (che svolge il ruolo *customer*) e un Dipendente (che svolge il ruolo *seller*).
2. **Stocking**: Questa relazione si riferisce ad un *Product* (che svolge il ruolo *stock*) che viene immagazzinato da un'altra *Company*, che svolge il ruolo *supplier*.
3. **Containing**: Definiamo questa relazione tra l'entità *Order* (che svolge il ruolo *order*) e *Product* (ruolo *contained*), poiché un ordine può contenere più prodotti. Aggiungiamo anche due attributi a questa relazione: *quantity* e *unit-price*.

Questo è ciò che abbiamo appena modellato:

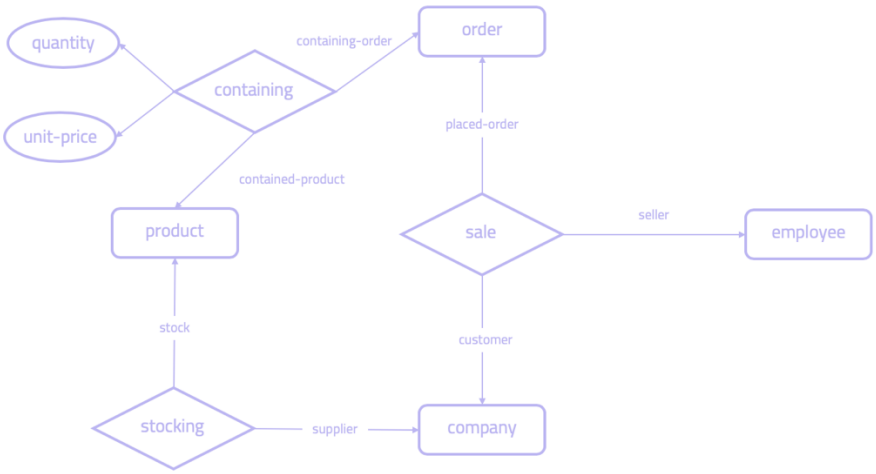

Figura 11 - Il modello del set di dati Northwind in TypeQL

Vediamo come appare in TypeQL:

```
define

employee sub entity,
  plays sale:seller;

company sub entity,
  plays stocking:supplier,
  plays sale:customer;

order sub entity,
  plays sale:order,
  plays containing:order;

product sub entity,
  plays containing:contained,
  plays stocking:stock;

sale sub relation,
  relates order,
  relates seller,
  relates customer;

containing sub relation,
  has quantity,
  has unit-price,
  relates contained,
  relates order;

stocking sub relation,
  relates stock,
  relates supplier;

 quantity sub attribute, value double;
 unit-price sub attribute, value double;
```

4.3 Query di attraversamento

Scrivere una query di attraversamento in SQL significa sfruttare l'operatore JOIN. Di seguito è riportato un confronto tra SQL e TypeQL per la seguente domanda:

Restituisci tutti gli ID dei dipendenti che hanno venduto a un cliente con sede a Londra e che ha la demografia del cliente "x"

SQL

```
SELECT Employees.EmployeeID
FROM Employees
INNER JOIN Orders ON
Employees.EmployeeID = Orders.EmployeeID
INNER JOIN Customers ON
Orders.CustomerID = Customers.CustomerID
AND Customers.City = 'London'
INNER JOIN CustomerCustomerDemo ON
Customers.CustomerID = CustomerCustomerDemo.CustomerID
INNER JOIN CustomerDemographic ON
CustomerCustomerDemo.CustomerTypeId = CustomerDemographic.CustomerTypeId
AND CustomerCustomerDemo.CustomerDesc = 'x'
```

TypeQL

```
match
$s isa employee, has employee-id $eid;
$c isa company, has city "London";
$r1 ($s, $c) isa sale;
$cd isa customer-demographic, has customer-description "x";
$r2 ($c, $cd);
get $eid;
```

4.4 Reasoning basato sul tipo

A differenza di SQL, in TypeQL abbiamo la capacità di aumentare l'espressività del nostro modello creando una gerarchia di tipi. Ad esempio, possiamo estendere il set di dati Northwind aggiungendo organizzazioni non profit, banche e aziende farmaceutiche. Concettualmente, questo appare come segue:

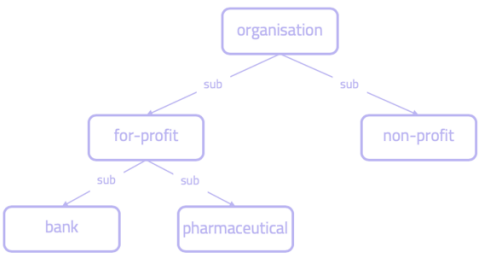

Figura 12 - Esempio di gerarchia dei tipi

In TypeQL la definiamo come segue:

```
define
organisation sub entity;

for-profit sub organisation;
non-profit sub organisation;

bank sub for-profit;
pharmaceutical sub for-profit;
```

4.5 Reasoning basato su regole

Con TypeQL, possiamo anche creare regole per astrarre e modularizzare la nostra logica di business. SQL non supporta le regole.

Per esempio, se sappiamo che la locazione x è contenuta in y, che a sua volta è contenuta in z, possiamo creare una regola che deduce ricorsivamente che x è contenuta anche in z. In TypeQL avremo qualcosa del tipo:

```
rule transitive-location:
when {
  (located: $x, locating: $y) isa locates;
  (located: $y, locating: $z) isa locates;
} then {
  (located: $x, locating: $z) isa locates;
};
```

Se osserviamo il nostro set di dati Northwind, possiamo vedere che esiste una transitività tra le tabelle Employees, Territories e Region, il che significa che possiamo sfruttare la regola in TypeQL di cui sopra. Per illustrare questo punto, consideriamo questa interrogazione:

Restituisci tutti gli id dei dipendenti che si trovano in una regione con descrizione della regione "x"

In SQL, scriveremo:

```
SELECT Employee.EmployeeID
FROM Employee
  INNER JOIN EmployeeTerritories AS Et ON
    Employee.EmployeeID = Et.EmployeeID
  INNER JOIN Territories AS Te ON
    Te.TerritoryID = Et.TerritoryID
  INNER JOIN Region AS Re ON
    Re.RegionID = Te.RegionID
    AND Re.RegionDescription = 'x'
```

Avendo definito in TypeQL la regola *transitive-location*, possiamo ora mettere in relazione direttamente il dipendente con la regione (togliendo la connessione da territorio a regione evitando di dover effettuare più join):

```
match
$e isa employee, has employee-id $eid;
$r isa region, has region-description "x";
$r1 (located: $e, locating: $r) isa locates;
get $eid;
```

Se incorporiamo anche la gerarchia dei tipi dell'organizzazione mostrata in precedenza, possiamo astrarre ancora più logica in TypeDB. Usiamo questa domanda come esempio:

Restituire gli ID per i clienti che sono aziende e enti di beneficenza, a cui i dipendenti hanno venduto e che si trovano in una regione con descrizione "x".

In SQL, scriviamo questa query in questo modo:

```
SELECT Companies.OrgID
FROM Companies
  INNER JOIN Orders ON
    Companies.OrgId = Orders.OrgId
  INNER JOIN Employees ON
    Orders.OrgId = Employees.OrgId
  INNER JOIN EmployeeTerritories AS Et ON
    Employees.EmployeeID = Et.EmployeeID
  INNER JOIN Territories AS Te ON
    Te.TerritoryID = Et.TerritoryID
  INNER JOIN Region AS Re ON
    Re.RegionID = Te.RegionID
    AND Re.RegionDescription = 'x'
UNION
SELECT Charities.OrgID
FROM Charities
  INNER JOIN Orders ON
    Charities.OrgId = Orders.OrgId
  INNER JOIN Employees ON
    Orders.OrgId = Employees.OrgId
  INNER JOIN EmployeeTerritories AS Et ON
    Employees.EmployeeID = Et.EmployeeID
  INNER JOIN Territories AS Te ON
    Te.TerritoryID = Et.TerritoryID
  INNER JOIN Region AS Re ON
    Re.RegionID = Te.RegionID
    AND Re.RegionDescription = 'x'
```

In TypeQL, scriviamo semplicemente:

```
match
$c isa organisation, has org-id $ci; # Organisation infers sub entities
companies and charities
$e isa employee;
$r isa region, has region-description "x";
$r1 (located: $e, locating: $r) isa locates;
$r2 ($c, $e) isa sale;
get $ci;
```

5 Conclusioni

In conclusione, abbiamo visto come:

TypeQL fornisce un'astrazione di livello superiore per lavorare con i dati. TypeQL semplifica la modellazione e l'interrogazione di dati complessi.

TypeQL ci consente di creare modelli concettuali che ci danno l'indipendenza fisica dei dati. Implementando uno schema a livello di concetto, TypeQL astrae il modello logico. Ciò significa che non abbiamo più bisogno di normalizzare i nostri dati.

Il motore di reasoning di TypeDB semplifica le nostre query. Utilizzando un reasoning automatizzato, TypeDB spinge verso il basso le operazioni di livello inferiore e ci consente di lavorare a un livello di astrazione più elevato.

Il motore di reasoning di TypeDB ci consente di astrarre la logica che altrimenti si verifica nella nostra query o nel livello dell'applicazione. L'inserimento di questa logica in TypeDB ci consente di scrivere query più semplici a un livello di espressività più elevato.