Basi di dati Appunti
-
secondo parziale

Dimitri

A.A. 2024/2025

Indice

Chapter 1

Algebra Relazionale

1.1 Introduzione

L'algebra relazionale è un linguaggio procedurale formale di tipo algebrico i cui operandi sono relazioni. Questo linguaggio non è usato nelle implementazioni dei vari DBMS ma definisce in maniera semplice tutte le operazioni tipiche dei diversi linguaggi di interrogazione. Da un punto di vista didattico l'algebra relazionale è utile perché essendo svincolata dai "dettagli implementativi" dell'SQL (o di altri linguaggi), permette di comprendere rapidamente la tecnica d'uso dei linguaggi di interrogazione per basi di dati relazionali.

1.2 Operatori

1.2.1 Introduzione

In Algebra relazionale è possibile classificare i diversi operatori in base alla derivabilità oppure in modo funzionale. Questi operatori possono essere binari o unari, e tramite questi ultimi vengono descritte le procedure di interrogazione.
Nella classificazione in base alla derivabilità si distiguono 6 operatori di base e diversi derivati:
Operatori di Base:

- Selezione
- Proiezione
- Ridenominazione
- Unione
- Differenza
- Prodotto cartesiano

Operatori derivati:

- Intersezione
- Join

1.2.2 Selezione σ

La selezione è un'operazione unaria. Seleziona le tuple di una relazione che soddisfano un predicato producendo un sottoinsieme delle stesse, (ossia seleziona le righe della tabella che rispettano la condizione data, ricordiamo che con tuple si intendono le righe della tabella mentre con attributi le singole colonne).
Lo schema della relazione risultato è lo stesso di quella di origine ed il predicato è costituito dal nome di un attributo, da un operatore, e da un altro argomento che può essere un attributo o un valore costante.

$$\sigma_{\text{predicato}}(relazione)$$

La cardinalità di questa operazione è la seguente:

$$0 \leq |\sigma_{\text{F}}(R)| \leq |R|$$

1.2.3 Proiezione Π

La proiezione è un'operazione unaria. Data una relazione, la sua proiezione su un dato insieme di attributi è costituita dalla tabella generata dagli attributi specificati, contenente tutte le tuple della tabella di partenza, (ossia estrae tutte le colonne corrispondenti alla lista di attributi specificati).

$$\Pi_{\text{lista attributi}}(relazione)$$

La cardinalità di questa operazione è la seguente:

$$min(|R|, 1) \leq \Pi_{\text{y}}(R) \leq |R|$$

1.2.4 Ridenominazione ρ

A volte in preparazione all'esecuzione di una interrogazione o in seguito ad una sua esecuzione si ha bisogno di rinominare gli attributi di una relazione. A tal fine l'operatore di ridenominazione che permette di ottenere una nuova tabella con i nuovi nomi per gli attributi modificati e che ha le stesse tuple della tabella originale.

$$\rho_{\text{vecchio nome} \rightarrow \text{nuovo nome}}(relazione)$$

1.2.5 Unione $\cup$

L'unione fra due tabelle è rappresentata da una tabella costituita dall'unione "matematica" delle due tabelle, dove quindi sono presenti le tuple della prima tabella e quelle della seconda. Affinché l'unione abbia senso, è necessario che:

- le due tabelle abbiano lo stesso numero di attributi;
- i tipi degli attributi corrispondenti siano uguali;

Se il numero degli attributi delle due relazioni (tabelle) non è uguale, si genera un errore.
La cardinalità di questa operazione è la seguente:

$$max(|R1|, |R2|) \leq |R1 \cup R2| \leq |R1| + |R2|$$

1.2.6 Differenza -

La differenza fra due tabelle A e B è una tabella che contiene le tuple che sono presenti in A ma non in B. Come nel caso dedll'unione, questa operazione può essere eseguita solo se le relazioni hanno lo stesso grado (numero di colonne) e gli attributi sono compatibili.
La cardinalità di questa operazione è la seguente:

$$0 \leq |R1 - R2| \leq |R1|$$

1.2.7 Prodotto Cartesiano $\times$

Il prodotto cartesiano fra due tabelle è una tabella con schema la somma degli schemi, se due attributi sono uguali questi sono ripetuti le tuple della tabella sono il risultato del prodotto cartesiano dei suoi elementi, ossia da tutte le coppie possibili composte dagli elementi appartenenti alle due relazioni.

$$R1 \times R2$$

La cardinalità di questa operazione è la seguente (ricordiamo che se il Join è completo il limite inferiore diventa $max(|R1|, |R2|)$):

$$|R1 \times R2| = |R1| \cdot |R2|$$

1.2.8 Intersezione $\cap$

Il risultato dell'operazione di intersezione tra due relazioni è una relazione contenente le tuple che appartengono ad entrambe le relazioni, anche in questo caso valgono le stesse condizioni di unione e differenza per la validità dell'operazione.

$$R1 \cap R2$$

La cardinalità di questa operazione è la seguente:

$$0 \leq |R1 \cap R2| \leq min(|R1|, |R2|)$$

1.3 Join

1.3.1 Introduzione

E' l'operatore più caratteristico dell'algebra relazionale, in quanto è quello che permette di correlare dati contenuti in relazioni diverse confrontando i valori comuni contenuti in esse.
Esistono diverse varianti di tale operatore comunque riconducibili l'una con l'altra:

- Join Naturale
- Join Esterni
- Theta Join ed Equi Join

1.3.2 Join Naturale $\bowtie$

Il Join naturale è un operatore binario che correla dati in relazioni diverse sulla base dei valori uguali in attributi con lo stesso nome. La relazione risultante è una tabella che ha come attributi l'unione degli attributi delle tabelle iniziali e contiene solamente le tuple che hanno valori uguali negli attributi in comune.
Quando non si hanno attributi comuni il join naturale diventa un prodotto cartesiano, perché genera tutte le possibili coppie.

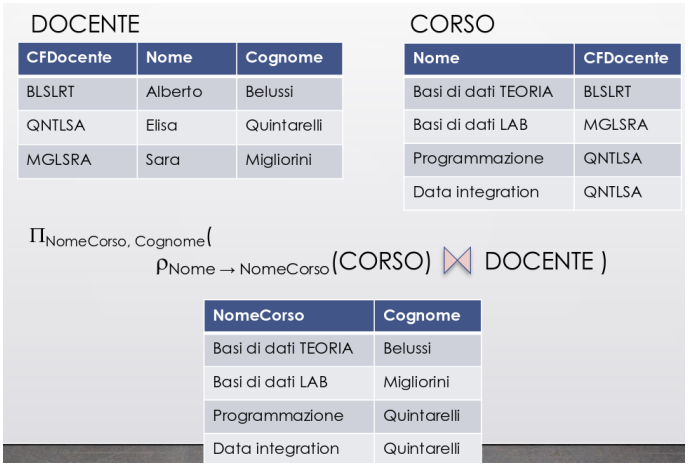
Esempio. Prendiamo lo schema:

```
DOCENTE(CFDocente, Nome, Cognome)
CORSO(Nome, CFDocente)
```

Si richiede di produrre l'insieme dei corsi riportando: nome del corso e cognome del docente.

$$\Pi_{\text{NomeCorso, Cognome}}(\rho_{\text{Nome} \to \text{NomeCorso}}(CORSO) \bowtie DOCENTE)$$

Graficamente si ottiene:



Il Join può essere ottenuto tramite un prodotto cartesiano e una selezione imponendo l'uguaglianza su tutti gli attributi in comune:

$$\sigma_{\text{NomeCorso} = \text{Nome}}(DOCENTE \times CORSO)$$

Il Join si dice completo se ogni tupla della relazione A contribuisce a generare almeno una tupla della relazione risultato, altrimenti si dice incompleto e le tuple che non contribuiscono al risultato si chiamano **dangling tuples**.

La cardinalità del Join Naturale è, in generale:

$$0 \leq |R1 \bowtie R2| \leq |R1| \cdot |R2|$$

Se il join è completo la cardinalità diventa:

$$max(|R1|, |R2|) \leq |R1 \bowtie R2| \leq |R1| \cdot |R2|$$

Se $X_1 \cap X_2$ è una superchiave per $R2$:

$$0 \leq |R1 \bowtie R2| \leq |R1|$$

Il caso più classico che ricounduce a questa situazione è quando si fa l'uguaglianza tra una *chiave* e una *chiave esportata*, ovvero abbiamo esportato la chiave di $R2$ su $R1$. Se prendo una tupla di $R1$ che ha tra i suoi attributi una *superchiave* di $R2$, si avrà al massimo una tupla di $R2$ che va in combinazione con la tupla di partenza di $R1$, perciò, *al massimo* la cardinalità sarà quella di $R1$. Se $X_1 \cap X_2$ è una superchiave di $R2$ ed esiste un *vincolo di integrità referenziale* tra $X_1 \cap X_2$, o una parte di $R1$, e $R2$: $|R1 \bowtie R2| = |R1|$.

1.3.3 Theta join ed equi-join

Nel theta join il predicato di join viene esplicitato, è indipendente dallo schema. Date le relazioni $R1$ e $R2$, rispettivamente di schema X_1 e X_2, la **precondizione** al theta join è che gli schemi di $R1$ e $R2$ devono essere *disgiunti*: $X_1 \cap X_2 = \varnothing$.

$$R1 \bowtie_\theta R2 = \sigma_\theta(R1 \bowtie R2)$$

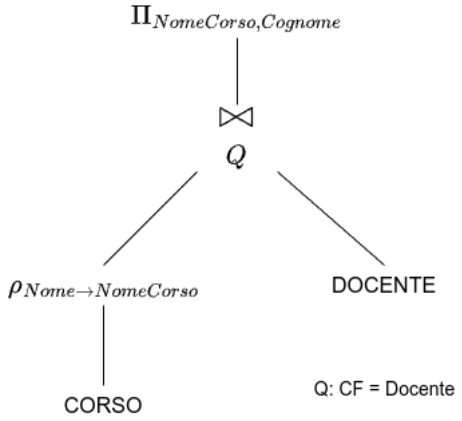
Dove il join naturale all'interno della selezione funge da *prodotto cartesiano* e θ è un predicato conforme alla sintassi prevista per l'operatore di selezione.

Esempio

`DOCENTE(CF, Nome, Cognome)`
`CORSO(Nome, Docente)`
Si richiede di produrre l'elenco di tutti i corsi riportando: nome del corso e cognome del docente.

$$\Pi_{\text{NomeCorso, Cognome}}(\rho_{\text{Nome} \to \text{NomeCorso}}(CORSO) \overset{\bowtie}{\underset{\text{CF = Docente}}{}} DOCENTE)$$

Graficamente:



Equi-join

Il theta join si dice *equi-join* se la condizione θ è una congiunzione di uguaglianze:

$$R1 \overset{\bowtie}{\underset{A_1 = B_1 \wedge \cdots \wedge A_n = B_n}{}} R2$$

Si noti che **non esiste** un operatore misto che applica la logica del join naturale per gli attributi comuni e aggiunge la condizione specificata esplicitamente attraverso il predicato θ.

Proprietà dell'equi-join

Date due relazioni $R1$ e $R2$ di schema X_1 e X_2, con $X_1 \cap X_2 = \{C_1, \ldots, C_m\}$ con $m > 0$, vale la seguente equivalenza:

$$R1 \bowtie R2 = \Pi_{X_1 \cup X_2}\left(R1 \overset{\bowtie}{\underset{C_1 = C'_1 \wedge \cdots \wedge C_m = C'_m}{}} \rho_{C_1,\ldots,C_m \to C'_1,\ldots,C'_m}(R2)\right)$$

In questo caso si hanno due relazioni che hanno attributi comuni ($\{C_1, \ldots, C_m\}$) e quindi si devono ridenominare gli attributi di una delle due relaizoni, nell'esempio quelli di $R2$. A questo punto gli schemi sono disgiunti e la condizione del theta-join sarà una congiunzione di uguaglianze: $C_1 = C'_1 \wedge \cdots \wedge C_m = C'_m$. Gli attributi che si ottengono nel risultato del theta-join sono la somma degli attributi di $R1$ e di $R2$, cosa che non avviene nel join naturale. Per questo motivo si proietta su $\Pi_{X_1 \cup X_2}$, in questo modo l'unione considera gli attributi di tutte e due le relazioni ma esclude i $C'_1, \ldots, C'_m$ generati nell'espressione.

1.3.4 Join esterni

Consentono di ottenere nel risultato del join tutte le tuple, anche le **dangling tuples**, di una o di entrambe le relazioni coinvolte nel join, eventualmente estese con valori nulli:

- **left join**: $r_1 \bowtie_{LEFT} r_2$;
- **right join**: $r_1 \bowtie_{RIGHT} r_2$;
- **full join**: $r_1 \bowtie_{FULL} r_2$;

1.4 Ottimizzazione

1.4.1 Valori nulli

E' opportuno estendere l'algebra relazionale affinché possa manipolare anche relazioni che contengono *valori nulli* (NULL). Le operazioni che devono essere raffinate per gestire relazioni che contengono valori nulli sono in particolare **selezione** e **join naturale**.
Le altre operazioni riportano semplicemente nelle tuple del risultato il valore nullo presente sulle tuple di input.

Selezione

Nel caso della **selezione** abbiamo che la selezione applicata a:

- $A\theta B$ risulta essere falsa (quindi non seleziona quella determinata tupla) se uno dei due attributi A o B è NULL.
- $A\theta const$ risulta essere falsa se l'attributo A è NULL.

Inoltre, si aggiungono le seguenti condizioni atomiche:

- `A IS NULL`: è una condizione atomica che viene valutata sulla tupla t. È **vero** se $t[A]$ *contiene* il valore NULL, altrimenti è **falso**.
- `A IS NOT NULL`: è una condizione atomica che viene valutata sulla tupla t. È **vero** se $t[A]$ <u>non</u> *contiene* il valore NULL, altrimenti è **falso**.

Join naturale

La condizione di uguaglianza sugli attributi comuni alle due relazioni è **falsa** sulle tuple t_1 e t_2 se almeno uno degli attributi comuni di t_1 o di t_2 è NULL. Inoltre, il confronto particolare $NULL = NULL$ è **sempre valutato FALSO**.

1.4.2 Ottimizzatore

Ogni espressione DML[1] ricevuta dal DBMS viene sottoposta ad un processo di elaborazione, tra cui, anche uno di ottimizzazione. L'ottimizzatore genera un'espressione equivalente all'interrogazione di input e di costo inferiore. quest'ultimo viene valutato in termini di **dimensione dei risultati intermedi**. L'ottimizzatore esegue **trasformazioni di equivalenza** allo scopo di riddurre la dimensione dei risultati intermedi.

Equivalenza tra espressioni algebriche

Equivalenza dipendente dallo schema. Dato uno schema R: $E_1 \equiv E_2$ se $E_1(r) = E_2(r)$ per ogni istanza di r di schema R.
Esempio:

$$\Pi_{AB}(R_1) \bowtie \Pi_{AC}(R_2) \equiv_R \Pi_{ABC}(R_1 \bowtie R_2)$$

Con $R = \{R_1(A, B, D), R_2(A, C, E)\}$.

La condizione generale che lo schema deve soddisfare è che l'**unico attributo comune tra R_1 e R_2 sia A**.

[1] Solitamente specificata in linguaggio dichiarativo

Equivalenza assoluta. È indipendente dallo schema.

$E_1 \equiv E_2$ se $E_1 \equiv_R E_2$ per ogni schema R compatibile con E_1 ed E_2.

Esempio:

$$\Pi_{AB}(\sigma_{A>0}(R_1)) \equiv \sigma_{A>0}(\Pi_{AB}(R1))$$

Trasformazioni di equivalenza

Le principali trasformazioni sono quattro. Consideriamo, per gli esempi, il seguente schema relazionale:
`TRENO(NumTreno, OrarioPart, Cat, Dest, OrarioArr)`
`FERMATA(NumTreno, Stazione, Orario)`

Sia E un'espressione di schema X, si definiscono le seguenti trasformazioni di equivalenza:

- **Atomizzazione delle selezioni:** una congiunzione di selezioni può essere sostituita da una sequenza di selezioni atomiche.

  $$\sigma_{\mathrm{F1} \wedge \mathrm{F2}}(E) \equiv \sigma_{\mathrm{F1}}(\sigma_{\mathrm{F2}}(E))$$

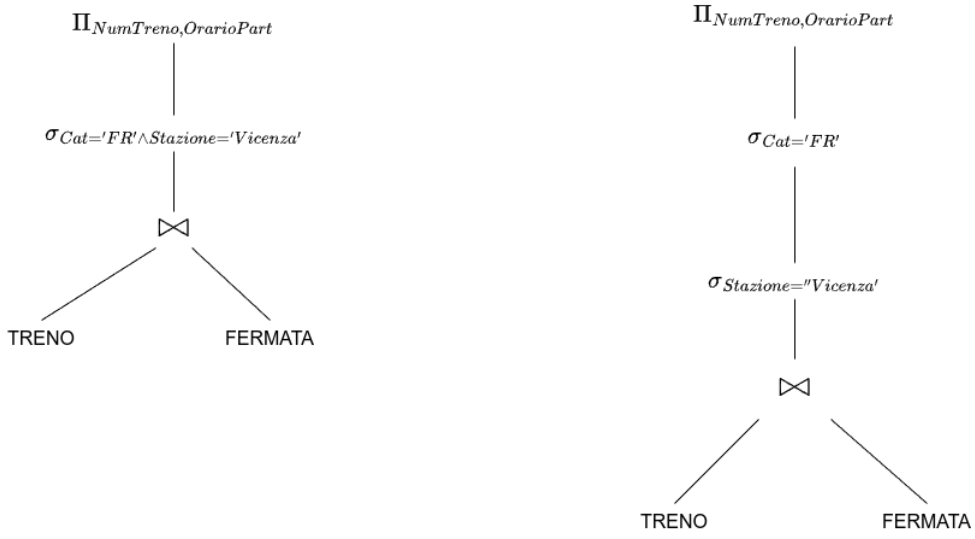
  È propedeutica ad altre trasformazioni. Non ottimizza se non è seguita da altre trasformazioni.
  Esempio:

  $$\Pi_{\mathrm{NumTreno,\ OrarioPart}}(\sigma_{\mathrm{Cat='FR'} \wedge \mathrm{Stazione='Vicenza'}}(TRENO \bowtie FERMATA)$$

  $$\Rightarrow$$

  $$\Pi_{\mathrm{NumTreno,\ OrarioPart}}(\sigma_{\mathrm{Cat='FR'}}(\sigma_{\mathrm{Stazione='Vicenza'}}(TRENO \bowtie FERMATA)))$$

  Graficamente:



- **Idempotenza delle proiezioni:** una proiezione può essere trasformata in una sequenza di proiezioni che eliminano i vari attributi in varie fasi.

  $$\Pi_{\mathrm{A}}(E) \equiv \Pi_{\mathrm{A}}(\Pi_{\mathrm{A,B}}(E)) \text{ dove } B \subseteq X$$

  È propedeutica ad altre trasformazioni. Non ottimizza se non è seguita da altre trasformazioni.

Siano E_1 ed E_2 espressioni di schema X_1 e X_2, si definiscono le seguenti trasformazioni di equivalenza:

- **Anticipazione della selezione rispetto al join:** questa espressione vale solo se F conivolge **solo** gli attributi di E_2.

$$\sigma_F(E_1 \bowtie E_2) \equiv_R E_1 \bowtie \sigma_F(E_2)$$

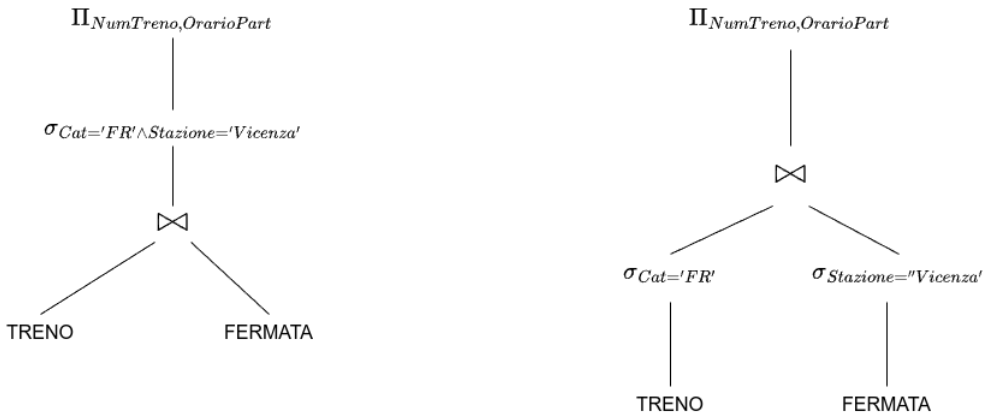
Esempio:

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'} \wedge \text{Stazione='Vicenza'}}(TRENO \bowtie FERMATA)$$

$$\Rightarrow$$

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'}}(TRENO)) \bowtie \sigma_{\text{Stazione='Vicenza'}}(FERMATA))$$

Graficamente:



- **Anticipazione della proiezione rispetto al Join:** vale solo se Y sono attributi di B e I suoi attributi sono coinvolti nel join.

$$\Pi_{\text{Y}}(A \bowtie B) \equiv A \bowtie (\Pi_{\text{Y}}(B))$$

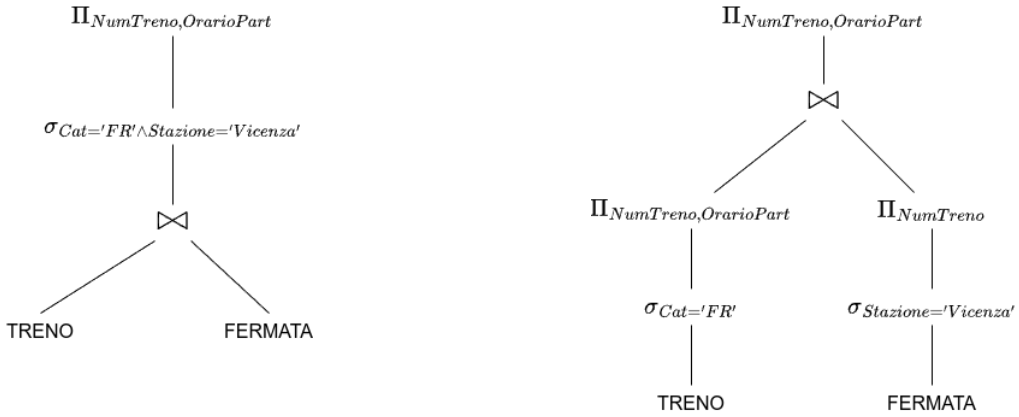
Esempio:

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'} \wedge \text{Stazione='Vicenza'}}(TRENO \bowtie FERMATA)$$

$$\Rightarrow$$

$$\Pi_{\text{NumTreno, OrarioPart}}(\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'}}(TRENO)) \bowtie \Pi_{\text{NumTreno}}(\sigma_{\text{Stazione='Vicenza'}}(FERMATA))$$

Graficamente:



- Esistono poi altre otimizzazioni minori come l'inglobamento di una selezione in un join (da eseguire solo se non è possibile anticipare la selezione), in questo caso la condizione di selezione sarà assorbita nel **theta-join**.

$$\sigma_{\mathrm{F}}(A \bowtie B) \equiv A \bowtie_{\mathrm{F}} B$$

Infine è bene ricordare anche le trasformazioni con gli operatori insiemistici:

$$\sigma_{\mathrm{F} \vee \mathrm{G}}(A) \equiv \sigma_{\mathrm{F}}(A) \cup \sigma_{\mathrm{G}}(A)$$
$$\sigma_{\mathrm{F} \wedge \mathrm{G}}(A) \equiv \sigma_{\mathrm{F}}(A) \cap \sigma_{\mathrm{G}}(A)$$

Chapter 2

Calcolo Relazionale

È un linguaggio di interrogazione dichiarativo che specifica le proprietà del risultato dell'interrogazione. Esistono due versione del *calcolo relazionale*:

- sui domini[1];
- **sulle tuple** (è il fondamento teorico del linguaggio **SQL**).

2.1 Calcolo relazionale sulle tuple

Nel calcolo relazionale sulle tuple l'interrogazione viene specificata attraverso una *formula logica con variabili libere*. La formula viene interpretata sul **contenuto della base di dati** e le variabili assumono come valori le tuple delle relazioni.

Il calcolo relazionale sulle tuple definisce la semantica del linguaggio SQL *semplice*[2], includendo le interrogazioni nidificate.

2.1.1 Formule logiche con variabili libere

Una variabile si dice **libera** in una forma logica F se non compare in un *quantificatore* (esistenziale $\exists$ o universale $\forall$) di F.

Esempi:

- $F_1(x) \equiv x > 3$ x è libera.
- $F_2(x, y) \equiv x > 3 \wedge x < 6 \wedge y = 10$ x, y sono variabili libere.
- $F_3(x) \equiv x > 0 \wedge \exists y(y < 0)$ x è libera, y è legata.
- $F_4(x) \equiv x > 3 \wedge \exists y(y < x)$ x è libera, y è legata.
- $F_5(z) \equiv z < 10 \wedge \forall y(y < z)$ z è libera, y è legata.

L'**interpretazione** è la scelta degli insiemi da cui prelevare i valori da sostituire alle variabili. Ad esempio, posso segliere di interpretare le formule nell'insieme degli interi ($\mathbb{Z}$) o nell'insieme dei naturali ($\mathbb{N}$).

Esempio:

$$F_3(x) \equiv x > 0 \wedge \exists y(y < 0)$$

$F_3(5)$ è vera in $\mathbb{N}$? $5 > 0 \rightarrow$`True`, $\exists y(y < 0) \rightarrow$`False`. `True` $\wedge$ `False` = `False`, perciò la formula è falsa. $F_3(5)$ è invece vera nell'insieme dei numeri interi $\mathbb{Z}$. Possiamo dunque notare che al variare dell'insieme di interpretazione, cambia il risultato.

[1] Non presente nel programma.
[2] Senza operatori aggregati e senza join esterni.

2.1.2 Sintassi del calcolo relazionale sulle tuple

Le espressioni del calcolo relazionale sulle tuple hanno la seguente forma:

$$\{\texttt{T}|\texttt{L}|\texttt{F}\}$$

- **Target list** `T` → corrisponde alla `SELECT` dell'SQL. Definisce lo *schema della relaziona risultato* e come le tuple risultato si ottengono dalle variabili libere.
- **Range list** `L` → corrisponde a `FROM` dell'SQL. Definisce le *variabili libere* e le collega alle relazioni della base di dati coinvolte nell'interrogazione.
- **Formula** `F` → corrisponde a `WHERE` dell'SQL. Specifica la condizione che deve essere soddisfatta dalle variabili libere, vale a dire dalle tuple coinvolte nell'interrogazione.

Target list

`T` è una lista di elementi separati da virgole $e_1, \ldots, e_n$ dove ogni elemento e_i può essere:

- `Y: x. (Z)`, dove `Y` e `Z` sono sequenze di attributi con la stessa cardinalità e `x` è una *variabile libera*.
- `x. (Z)`, dove `Z` è una sequenza di attributi e `x` è una *variabile libera* (equivale a `Z: x. (Z)`).
- `x. *`, dove `x` è una *variabile libera*; in questo caso gli attributi sono esattamente tutti quelli della relazione associata alla variabile `x` nella **range list** `L`.

Range list

`L` è una lista di elementi separati da virgole $r_1, \ldots, r_m$ dove ogni elemento r_i è strutturato come $\texttt{x}_\texttt{j}\texttt{(R)}$, in cui `R` rappresenta una relazione, o tabella, della base di dati. (La variabile x_j prenderà come valori le tuple della tabella R quando si va a valutare la formula). Il numero degli elementi **deve corrispondere** al numero di variabili libere.

Formula

`F` può essere

- una **formula atomica**, del tipo $\texttt{x.A}\ \theta\ \texttt{c}$ oppure $\texttt{x}_1\texttt{.A}_1\ \theta\ \texttt{x}_2\texttt{.A}_2$, dove $\texttt{x, x}_1\texttt{, x}_2$ sono variabili, $\texttt{A, A}_1\texttt{, A}_2$ sono attributi, `c` è una costante e $\theta \in \{=, \neq, >, <, \leq, \geq\}$. Nelle formule atomiche *tutte le variabili* sono *LIBERE*.
- oppure una **formula non atomica**:
  - se f_1 e f_2 sono formule, allora anche $f_1 \wedge f_2$, $f_1 \vee f_2$, $\neg f_1$, e $\neg f_2$ lo sono.
  - Se f è una formula, allora anche $\forall x(R)(f)$ e $\exists x(R)(f)$ lo sono.

  Nelle formule $\forall x(f)$ e $\exists x(f)$, x è una variabile *LEGATA*, mentre tutte le altre variabili sono libere o legate se lo sono in f. Inoltre, le congiunzioni, disgiunzioni e la negazione non cambiano l'insieme delle variabili *LIBERE* o *LEGATE* di una formula.

2.1.3 Semantica: interpretazione delle formule

Le formule vengono interpretate sull'istanza corrente della

- base di dati $db = \{r_1, \ldots, r_n\}$
- di schema $S = \{R_1(X_1), \ldots, R_n(X_n)\}$

Ogni formula contiene un certo numero di variabili libere:

- $f(x_1, \ldots, x_2)$

Data una ennupla di tuple $t_1, \ldots, t_n$, tale ennupla rende vera la formula se, quando sostituisco le tuple $t_1, \ldots, t_n$ alle variabili libere $x_1, \ldots, x_n$, tale sostituzione soddisfa la formula.

Formule atomiche

- $\texttt{x}_1\texttt{.A}_1 \ \theta \ \texttt{x}_2\texttt{.A}_2$ è vera sulle tuple (t_1, t_2), se il confronto $\texttt{t}_1\texttt{[A}_1\texttt{]} \ \theta \ \texttt{t}_2\texttt{[A}_2\texttt{]}$ è soddisfatto.
- $\texttt{x}_1\texttt{.A}_1 \ \theta \ \texttt{c}$ è vera sulla tupla t_1, se il confronto $\texttt{t}_1\texttt{[A}_1\texttt{]} \ \theta \ \texttt{c}$ è soddisfatto.

Formule non atomiche

Le formule $f_1 \wedge f_2$, $f_1 \vee f_2$, $\neg f_1$, $\neg f_2$ sono vere secondo le usuali definizioni dei connettivi logici (tabelle di verità).

Formule con quantificatori

- $\exists x(R_i)(f)$ con variabili libere $(x_1, \ldots, x_q)$ è vera sulle tuple $(t_1, \ldots, t_q)$, se **esiste almeno** una tupla $t \in r_1$[3] tale che f è vera quando si sostituiscono le tuple $(t, t_1, \ldots, t_q)$ alle variabili $(x, x_1, \ldots, x_q)$.
- $\forall x(R_i)(f)$ con variabili libere $(x_1, \ldots, x_q)$ è vera sulle tuple $(t_1, \ldots, t_q)$, se **per ogni** tupla $t \in r_i$, f è vera quando si sostituiscono le tuple $(t, t_1, \ldots, t_q)$ alle variabili $(x, x_1, \ldots, x_q)$.

2.1.4 Semantica di una interrogazione

Interpretazone di un'espressione del calcolo come interrogazione. Struttura:

$$Q = \{Y_1 : x_1.(Z_1), \ldots, Y_k : x_k.(Z_k) | x_1(R_1), \ldots, x_n(R_n) | f(x_1, \ldots, x_n)\}$$

La valutazione dell'interrogazione Q produce una relazione risultato R dove:

- Lo schema di R è $Y = Y_1 \cup \cdots \cup Y_k$
- R contiene tutte le tuple t su Y tali che esiste una ennupla di tuple $(t_1, \ldots, t_n) \in R_1 \times \cdots \times R_n$, che generano t e che sostituite alle variabili libere $(x_1, \ldots, x_n)$ rendono vera f.

2.1.5 Osservazioni

Il calcolo relazionale sulle tuple **non è equivalente all'algebra relazionale** e non è equivalente al calcolo relazionale sui domini, in quanto l'unione di due relazioni non è rappresentabile nel calcolo relazionale sulle tuple.

È invece possibile rappresentare nel calcolo relazionale sulle tuple sia l'intersezione che la differenza tra due relazioni.

2.1.6 Operatori insiemistici nel calcolo relazionale sulle tuple

Intersezione

L'intersezione tra due relazioni R_1 e R_2 di schema $X = (A_1, \ldots, A_n)$ si ottiene:

$$R_1 \cap R_2 \equiv \{x. * | x(R_1), y_2(R_2) | x.A_1 = y.A_1 \wedge \cdots \wedge x.A_n = y.A_n\}$$

Possiamo ottenere lo stesso risultato utilizzando il quantificatore esistenziale ($\exists$):

$$R_1 \cap R_2 \equiv \{x. * | x(R_1) | \exists y(R_2)(x.A_1 = y.A_1 \wedge \cdots \wedge x.A_n = y.A_n)\}$$

Differenza

La differenza tra due relazioni R_1 e R_2 di schema $X = (A_1, \ldots, A_n)$ si scrive:

$$R_1 - R_2 \equiv \{x. * | x(R_1) | \neg\exists y(R_2)(x.A_1 = y.A_1 \wedge \cdots \wedge x.A_n = y.A_n)\}$$

Unione

L'unione tra due relazioni R_1 e R_2 di schema $X = (A_1, \ldots, A_n)$ **non è esprimibile** in quanto **non è possibile associare una variabile libera a più relazioni della base di dati**.

Notare che $\{x_1.*, x_2. * | x_1(R_1), x_2(R_2) | true\}$ genera il *prodotto cartesiano* e non l'unione.

[3] Istanza di R_i

2.1.7 Esercizi

Schema:
TRENO(Num, Cat, Part, Arr, Dest)
FERMATA(Treno, Staz, Orario)

Q_0: Trovare il numero e l'ora di partenza dei treni Freccia Rossa con destinazione Milano Centrale.

$$Q_0 : \{t.(\texttt{Num, Part})|t(\texttt{TRENO})|t.\texttt{Cat='FR'} \wedge t.\texttt{Dest='Milano Centrale'}\}$$

Q_1: Trovare l'elenco di tutte le fermate dei treni regionali, riportando nel risultato il numero del treno e la stazione di fermata.

$$Q_1 : \{f.(\texttt{Treno, Staz})|t(\texttt{TRENO}), f(\texttt{FERMATA})|t.\texttt{Cat='REG'} \wedge t.\texttt{Num} = f.\texttt{Treno}\}$$

Q_2: Trovare l'elenco di tutti i treni Freccia Rossa che fermano a Venezia Mestre riportando: il numero del treno, l'orario di partenza e l'orario di arrivo.

$$Q_2 : \{t.(\texttt{Num, Part}), f.(\texttt{Orario})|t(\texttt{TRENO}), f(\texttt{FERMATA})|t.\texttt{Num} = f.\texttt{Treno} \wedge f.\texttt{Staz='Venezia Mestre'} \wedge t.\texttt{Cat='FR'}\}$$

Q_3: Trovare la destinazione e l'orario di partenza dei treni che fermano a Padova.

$$Q_3 : \{t.(\texttt{Dest, Part})|t(\texttt{TRENO}), f(\texttt{FERMATA})|t.\texttt{Num} = f.\texttt{Treno} \wedge f.\texttt{Staz='Padova'}\}$$

oppure

$$Q_3 : \{t.(\texttt{Dest, Part})|t(\texttt{TRENO})|\exists f(\texttt{FERMATA})(t.\texttt{Num} = f.\texttt{Treno} \wedge f.\texttt{Staz='Padova'})\}$$

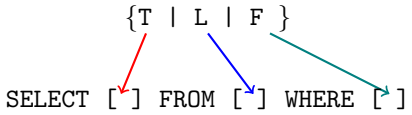
Q_4: Trovare la destinazione e l'orario di partenza dei treni che non fermano a Padova. In questo caso non possiamo mettere FERMATA nella target list. Questa interrogazione non può essere ottenuta attraverso un join, ma si ottiene con una differenza.

$$Q_4 : \{t.(\texttt{Dest, Part})|t(\texttt{TRENO})|\neg\exists f(\texttt{FERMATA})(t.\texttt{Num} = f.\texttt{Treno} \wedge f.\texttt{Staz='Padova'})\}$$

2.1.8 Calcolo relazionale sulle tuple

Il calcolo relazionale sulle tuple è equivalente al linguaggio di interrogazione adottato dallo standard SQL2 se consideriamo la forma base dell'interrogazione SQL senza operatori aggregati e raggrupamenti.

L'espressione del calcolo { T | L | F} è in diretta corrispondenza con la forma base delle interrogazioni SQL.



Uso dei quantificatori

Nelle formule del calcolo è possibile esprimere condizioni attravarso l'uso di quantificatori ($\exists$ e $\forall$).

Le espressioni che possono essere specificate attraverso i quantificatori corrispondono in SQL a clausole WHERE nelle quali compaiono **predicati complessi che richiedono interrogazioni nidificate**.

Esempio

Schema:
TRENO(Num, Cat, Part, Arr, Dest)
FERMATA(Treno, Staz, Orario)

Q_5: Trovare il numero e la categoria dei treni che fermano a Vicenza e fermano a Padova dopo le 20:30.

$$Q_5 : \{\texttt{Numero, Categoria} : t.(\texttt{Num, Cat}) | t(\texttt{TRENO}) | \exists f_1(\texttt{FERMATA})(t.\texttt{Num} = f_1.\texttt{Treno} \wedge f_1.\texttt{Staz='Vicenza'})$$
$$\wedge \exists f_2(\texttt{FERMATA})(t.\texttt{Num} = f_2.\texttt{Treno} \wedge f_2.\texttt{Staz='Padova'} \wedge f_2.\texttt{Orario > 20:30})\}$$

È possibile eliminare il quantificatore esistenziale nell'esempio Q_5, perché non è negato. In altre parole, questa interrogazione è possibile farla in *algebra relazionale* utilizzando due join. Alternativa:

$$Q_5 : \{\texttt{Numero, Categoria} : t.(\texttt{Num, Cat}) | t(\texttt{TRENO}), f_1(\texttt{FERMATA}), f_2(\texttt{FERMATA}) |$$
$$t.\texttt{Num} = f_1.\texttt{Treno} \wedge f_1.\texttt{Staz='Vicenza'} \wedge t.\texttt{Num} = f_2.\texttt{Treno} \wedge f_2.\texttt{Staz='Padova'} \wedge f_2.\texttt{Orario > 20:30}\}$$

2.1.9 Esercizi

Schema:
TRENO(Num, Cat, Part, Arr, Dest)
FERMATA(Treno, Staz, Orario)

Q_6: Trovare per ogni stazione il numero dell'ultimo treno che ferma in quella stazione.

$$Q_6 : \{f.(\texttt{Treno, Staz}) | f(\texttt{FERMATA}) | \neg \exists g(\texttt{FERMATA})(g.\texttt{Staz} = f.\texttt{Staz} \wedge g.\texttt{Orario} > f.\texttt{Orario})$$

Che equivale all'espressione con il quantificatore universale $\forall$:

$$Q_6 : \{f.(\texttt{Treno, Staz}) | f(\texttt{FERMATA}) | \forall g(\texttt{FERMATA})(g.\texttt{Staz} = f.\texttt{Staz} \Rightarrow g.\texttt{Orario} \leq f.\texttt{Orario})$$

Q_7: Trovare l'elenco di tutte le stazioni dove fermano tutti i treni con destinazione Venezia SL.

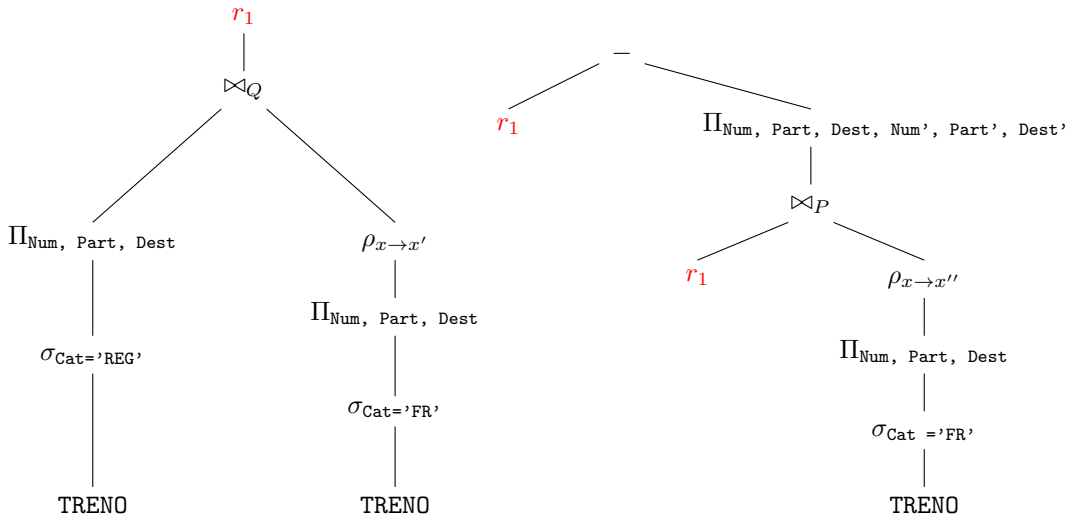
$$Q_7 : \{f.(\texttt{Staz}) | f(\texttt{FERMATA}) | \forall t(\texttt{TRENO})(t.\texttt{Dest='Venezia SL'} \Rightarrow \exists g(FERMATA)(t.\texttt{Num} = g.\texttt{Treno} \wedge f.\texttt{Staz} = g.\texttt{Staz}))$$

In questo caso non è possibile in alcun modo eliminare il quantificatore universale.

Q_8: Trovare per ogni treno regionale il treno freccia rossa immediatamente successivo con la stessa de stinazione, riportando nel risultato il numero e l'orario di partenza dei due treni insieme alla loro destinazione.

$$Q_8 : \{\texttt{Num\_REG, Part\_REG} : t.(\texttt{Num, Part}), \texttt{Num\_FR, Part\_FR} : p.(\texttt{Num, Part}) | t(\texttt{TRENO}), p(\texttt{TRENO}) |$$
$$t.\texttt{Cat='REG'} \wedge p.\texttt{Cat='FR'} \wedge t.\texttt{Dest} = p.\texttt{Dest} \wedge t.\texttt{Part} < p.\texttt{Part} \wedge \neg \exists q(\texttt{TRENO})(t.\texttt{Dest} = q.\texttt{Dest} \wedge q.\texttt{Cat='FR'} \wedge$$
$$t.\texttt{Part} < q.\texttt{Part} \wedge q.\texttt{Part} < p.\texttt{Part})$$

Possiamo scrivere questa interrogazione in Algebra:



Q: Dest = Dest' $\wedge$ Part < Part'

P: Dest = Dest'' $\wedge$ Part < Part'' $\wedge$ Part'' < Part'

Chapter 3

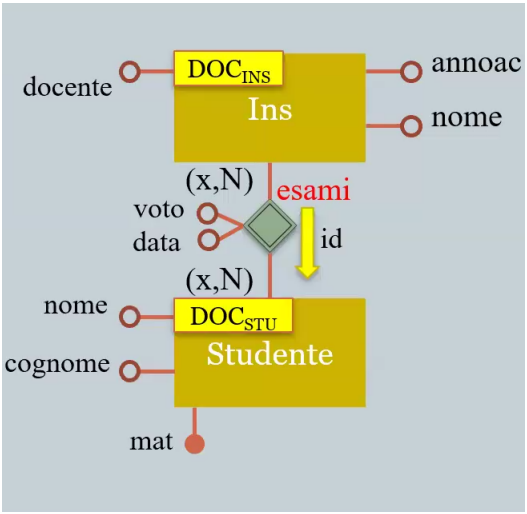
Interrogazioni nei sistemi document-based

3.1 Linguaggi di interrogazione nei sistemi NoSQL

Nei sistemi NoSQL spesso non sono disponibili veri e propri linguaggi di interrogazione come nei sistemi relazionali. Solo i sistemi **document-based** forniscono strumenti simili ad algebra e calcolo relazionale per l'accesso ai dati. Due esempi sono:

- **Couchbase**, che fornisce il linguaggio SQL++, simile all'SQL e dichiarativo.
- **MongoDB**, che fornisce una **serie di operatori** che consentono di scrivere complesse espressioni algebriche (pipeline) per estrarre dati dalle collezioni di documenti.

Sistemi document-store



DOC_Ins

```
{
 _id: autogen
 nome: String
 annoac: String
 docente: String
}
```

DOC_Stu

```
{
 _id: autogen
 nome: String
 cognome: String
 esami: [{Ins_id: integer
                  voto: integer
                  data: date},
                  ...]
}
```

3.1.1 Approccio MongoDB

Buona parte del linguaggio di interrogazione di MongoDB è realizzato attraverso il metodo `find`. Interrogazione sull'esempio sopra riportato:

```
db.studenti.insertMany([
{_id: ''VR00010'', nome: ''Mario'', cognome: ''Rossi''
 esami: [{ins: ''Basi di dati'', voto: 22, data: ''1/7/2016'' },
         {ins: ''Algebra'', voto: 26, data: ''5/7/2015'' }] }
{_id: ''VR00011'', nome: ''Maria'', cognome: ''Bianchi''
 esami: [{ins:' ''Algebra'', voto: 30, data: ''1/7/2016'' }]}
]);
```

Interrogazioni semplici

Il sistema nasce inizialmente per fare solo questo tipo di interrogazioni. Le interrogazioni semplici sono quelle *monocollezione*.

- Trovare tutti gli studenti:

  ```
  db.studenti.find( {} )
  ```

- Trovare tutti gli studenti di cognome *Rossi*:

  ```
  db.studenti.find( {cognome: ''Rossi''} )
  ```

- Trovare tutti gli studenti di cognome "Rossi" e nome "Mario":

  ```
  db.studenti.find( {cognome: ''Rossi'', nome: ''Mario''} )
  ```

- Trovare tutti gli studenti di cognome "Rossi" o "Bianchi":

  ```
  db.studenti.find( {cognome: {$in:[''Rossi'', ''Bianchi'']} } )
  ```

Operatori logici e di confronto

- `$and`: unisce due clausole con un and logico e restituisce tutti i documenti che rispettano entrambe le clausole.

  ```
  {$and: [{Esami.voto: {$gte: 21}}, {Esami.voto: {$lte: 26}} ]}
  ```

- `$not`: inverte l'effetto dell'espressione di interrogazione e restituisce i documenti che **non rispettano** la clausola.
- `$nor`: unisce le clausole con un nor logico e restituisce i documenti che **non rispettano** entrambe le clausole.
- `$or`: unisce le clausole con un or logico e restituisce i documenti che rispettano almeno una delle due clausole.

  ```
  {$or: [{Cognome: ''Rossi''}, {Cognome: ''Bianchi''}]}
  ```

- `$eq`: restituisce i valori uguali ad un valore specificato.
- `$gt`/`$lt`: restituisce i valori maggiori/minori di un valore specificato.
- `$gte`/`$lte`: restituisce i valori maggiori o uguali/minori o uguali di un valore specificato.
- `$in`: corrispondenza con qualsiasi valore all'interno di un dato array.
- `$ne`: corrispondenza con tutti i valori diversi da un valore specificato.
- `$nin`: corrispondenza con nessuno dei valori specificati in un array.

Interrogazioni su dati incapsulati

- Trovare tutti gli studenti che hanno registrato *almeno un esame* con voto 30:

```
db.studenti.fint( {esami.voto: 30} )
```

- Trovare tutti gli studenti che hanno registrato *almeno un esame* con voto maggiore di 22:

```
db.studenti.find( {esami.voto: { $gt: 22 } } )
```

Interrogazioni con proiezione

La proiezione permette di definire qual è la struttura del documento risultato.
Esempi:

- Trovare il cognome degli studenti di nome "Mario":

```
db.studenti.find( {nome: ''Mario''}, {cognome: 1} )
```

- Trovare il cognome degli studenti di nome "Mario" escludendo il campo `_id`:

```
db.studenti.find( {nome: ''Mario''}, {cognome: 1, _id: 0} )
```

- Trovare la matricola e i voti di tutti gli studenti:

```
db.studenti.find( { }, {_id: 1, esami.voto: 1} )
```

Interrogazioni con join

Per eseguire un join tra due collezioni di documenti è necessario usare l'operazione `$lookup` di `aggregate`. Questo tipo di interrogazioni non sono consigliate dal sistema.

Esempio:

```
STUDENTI: {_id: "VR00010",
 nome: "Mario",
 cognome: "Rossi" }
CORSI: {_id: "ins01",
 nome: "Basi di dati",
 annoac: "2016/2017",
 docente: "Belussi" }
{_id: "ins02",
 nome: "Algebra",
 annoac: "2015/2016",
 docente: "Gregorio"}
ESAMI: {_id: "01",
stud_id: "VR00010"
 ins_id: "ins01",
 voto: 22,
 data: "1/7/2016" }
{_id: "02", stud_id: "VR00010",
 ins_id: "ins02",
 voto: 26,
 data: "5/7/2015" }
```

```
db.STUDENTI.aggregate([
  { $lookup:
        { from: ESAMI,
          localField: _id,
          foreignField: stud_id,
          as: ''esami_fatti''
        }
  }
])
```

Si vuole, per esempio, incapsulare dentro a **STUDENTI** i suoi esami. Non si generano le coppie, ma una **nuova proprietà** dentro **STUDENTI** che conterrà tutte le istanze di **ESAMI** che si agganciano a quello studente.
Risultato:

```
{ _id: ''VR00010'',
 nome: ''Mario'',
 cognome: ''Rossi'',
 ''esami_fatti'': [
       {_id: ''01'', stud_id: ''VR00010'',
        ins_id: ''ins01'', voto: 22,
        data: ''1/7/2016'' },
   {_id: ''02'', stud_id: ''VR00010'',
    ins_id: ''ins02'', voto: 26,
    data: ''5/7/2015'' } ]
}
```