# ALGEBRA RELAZIONALE

**Prodotto cartesiano:**
$T_1, T_2$ sono insiemi;
$T_1 \times T_2$ è un prodotto cartesiano tra $T_1$ e $T_2$ ed è l'insieme delle coppie ordinate $(v_1, v_2)$ tale che $v_1 \in T_1$ , $v_2 \in T_1$ .

Es:
$T_1 = \{1,2,4\}$ $T_2 = \{a, b\}$
$T_1 \times T_2 = \{(1, a), (1, b), (2, a), (2, b), (4, a), (4, b)\}$
Sono tutte le possibili coppie in cui il primo elemento appartiene a $T_1$ mentre il secondo a $T_2$
Gli insiemi si possono rappresentare come tabelle.
Si può fare il prodotto cartesiano tra più insiemi: $T_1 \times T_2 \times \ldots \times T_n$ .

**Relazione (matematica):**
Una relazione sugli insiemi $T_1, T_2$ è un sottoinsieme di $T_1 \times T_2$ .
Oppure, una relazione è un'insieme di n-uple ordinate $(v_1, v_2, \ldots, v_n)$ con $v_1 \in D_1, \ldots, v_n \in D_n$ .
Ogni n-upla stabilisce un legame fra i dati che contiene.

- Le n-uple (righe) non sono ordinate in modo definito.
- Non vi sono n-uple uguali fra loro.

Es:
$T_1 \times T_2 = \{(1, a), (1, b), (2, a), (2, b), (4, a), (4, b)\}$
Relazione su $T_1 \times T_2 = \{(1, a), (1, b), (4, a)\}$

| | | | |
|---|---|---|---|
| Juventus | Lazio | 3 | 1 |
| Lazio | Milan | 2 | 0 |
| Juventus | Roma | 1 | 2 |
| Roma | Milan | 0 | 1 |

Ogni n-ulpa (riga) stabilisce un legame fra i dati del tipo:
"Juventus Vs Lazio → Ha vinto la Juventus 3 a 1"

**Record (righe):**
Una relazione è sostanzialmente un insieme di record omogenei (definiti sugli stessi campi).
Ad ogni campo (colonna) è associato un nome, detto **attributo**, che descrive il ruolo che il campo ha nel dominio stesso.

**Tupla:**
$t[\text{NomeColonna1}, \text{NomeColonna2}]$ è una tupla su due attributi (una riga formata dai due elementi appartenenti alle colonne indicate).

**Unione:** ∪
T1 ∪ T2
- Unisce le due tabelle, includendo una sola volta eventuali righe presenti in entrambe le tabelle.
**Intersezione:** ∩
T1 ∩ T2
- Unisce le due tabelle, tenendo solo le righe presenti in entrambe le tabelle.
**Differenza:** –
T1 – T2
- Prende T1 e ne elimina le righe presenti anche nella tabella T2.
Attenzione!
Queste tre operazioni hanno senso solo se gli attributi di T1 e T2 hanno tutti rispettivamente lo stesso nome.

**Ridenominazione: REN_{n1 ← n2} (T2)**
- Sostituisco il nome (n2) dell'attributo di T2, con il nome n1
**Es 1):**
Tabella: Paternità {padre, figlio, …}
REN_{genitore ← padre} (Tabella)
Tabella: Paternità {genitore, figlio, …}
**Es 2):**
Tabelle: Impiegati {Cognome, Ufficio, Stipendio}
Operai {Cognome, Fabbrica, Salario}
REN_{Sede, Retribuzione ← Ufficio, Stipendio} (Impiegati) ∪
REN_{Sede, Retribuzione ← Fabbrica, Salario} (Operai)
Tabelle: Impiegati {Cognome, Sede, Retribuzione}
Operai {Cognome, Sede, Retribuzione }
- Con la virgola, ho potuto concatenare delle ridenominazioni.
- Senza le ridenominazioni, l'unione fra queste due tabelle non avrebbe avuto senso.

**Selezione: SEL_{condizione booleana} (tabella)**
- Restituisce la tabella con le sole righe che soddisfano la condizione.
**Es:**
Impiegati che guadagnano più di 50 € e lavorano a Milano.
Tabella: Impiegati {Cognome, Filiale, Stipendio}
SEL_{Stipendio > 50 AND Filiale = 'Milano'} (Impiegati)

**Proiezione: PROJ_{ListaAttributi} (tabella)**
- Estrae dalla tabella gli attributi indicati.
**Es:**
Tabella: Impiegati {Cognome, Filiale, Stipendio}
PROJ_{Cognome, Filiale} (Impiegati)
Tabella: Impiegati {Cognome, Filiale}

Combinazione PROJ e SEL
PROJ_{ListaAttributi} ( SEL_{condizione booleana} ) (tabella)
- Anche senza le parentesi:
PROJ_{ListaAttributi} SEL_{condizione booleana} (tabella)

**Join:  T1  JOIN_{Condizione} T2**
- Crea una nuova tabella che ha come attributi l'unione degli attributi di T1 e T2 e come righe quelle che soddisfano la condizione.

Tabelle:  Tab1  {Numero, Voto}
  Tab2  {Numero, Candidato}
 Tab1  JOIN_{Tab1.Numero = Tab2.Numero}  Tab2
Tabella risultante:  Tab3  {Numero, Candidato, Voto}

**Join Naturale:  T1  JOIN_NAT T2**
- E' come il join normale che però confronta in automatico gli attributi che T1 e T2 hanno in comune.

Tabelle:  Tab1  {Numero, Voto}
  Tab2  {Numero, Candidato}
 Tab1  JOIN_NAT  Tab2
Che è equivalente a:
 Tab1  JOIN_{Tab1.Numero = Tab2.Numero}  Tab2
Tabella risultante:  Tab3  {Numero, Candidato, Voto}

**Left Join:  T1 LEFT_JOIN_{Condizione} T2**
- E' una join che tiene tutti i valori della tabella T1, completando i valori mancanti di T2 con "null". Vale rispettivamente per il right join e il full join.

**Definire pezzi di codice**

```
pezzo_di_codice( attributi ) :=
   proj_{…} (…) …
```

**Fasi generiche dello svolgimento e Consigli**

1) verificare in quali tabelle sono contenuti i dati che mi servono
2) togliere con la PROJ tutti quegli attributi che non mi serviranno
2) ridurre con la SEL le righe delle tabelle con la condizione (prima di metterle in JOIN).
3) faccio il JOIN se necessario (da cui poi posso ulteriormente togliere colonne con un'altra PROJ su tutto).

N.B.
- Potrei dover usare l'Unione prima del JOIN (se le tabelle non hanno stessi attributi allora prima dell'Unione devo fare REN degli attributi o togliere degli attributi/colonne con PROJ).
- La negazione è una differenza (costruttori che fanno laptot ma non pc = costruttori di laptop - costruttori di pc)
- computer con velocità massima: tutti i computer – i computer che hanno almeno un computer più veloce di loro.

# SQL

```
CREATE TABLE voti (
        studente integer not null references studenti(matricola),
        materia varchar(50) not null references materie(codice),
        tipo varchar(7) not nu1l,
        voto numeric(3,1) not null,
        foreign key(tipo,voto) references esami(tipologia,valutazione),

        data_voto date not null,
        check (voto between 0.0 and 10.01),

        check (tipo = 'scritto' or tipo = 'orale'),
        //oppure
        check (tipo in ('scritto', 'orale')),

        primary key(studente, materia, data_voto)
        //oppure
        unique(studente, materia, data_voto)
);
```

- **CREATE TABLE**: crea una nuova tabella
- tipi di dato (domini):
    - char(d): può contenere una stringa di caratteri a lunghezza fissa, pari a d
    - varchar(d): può contenere una stringa di caratteri a lunghezza variabile, pari a d
    - integer: valore numerico intero
    - float: valore con virgola
    - date: contiene la data ma può contenere anche ore, minuti e secondi
        - current_date -- data di "oggi"
        - 'today'::date
        - 'yesterday'::date
        - 'tomorrow'::date
        - La sottrazione tra due date restituisce il numero di giorni.
    - time: usato per ore, minuti, secondi (12:00:00 è mezzogiorno).
    - numeric(p,s): p = numero totale di cifre decimali usate.
        s = numero di cifre decimali usate a destra della virgola (s = 0 si ha un intero).
    - text: sequenza di caratteri "senza limite" esplicito (c'è e dipende dalla versione di postgres).
    - bytea: array di byte
    - boolean: 1 byte; si preferisce scrivere TRUE e FALSE.
    - point: (x,y) indica un punto
    - line: retta indicata con due punti
    - box: indicato con due punti (diagonale del rettangolo)
- **not null**: attributo che non può essere vuoto
- **references Tab1(attributo_di_Tab1)**: vincolo di integrità applicato all'attributo che farà riferimento ad un attributo di un'altra tabella (utile per fare le JOIN).
- **foreign key**(tipo,voto) references esami(tipologia,valutazione): ho creato una chiave su una coppia di attributi e poi ho applicato su di essa un vincolo di integrità verso una coppia di attributi di un'altra tabella.

- **check (vincolo)**: applica un vincolo (condizione che viene controllata ogni volta che inserisco dati nella1 tabella). Il vincolo può essere composto anche da una SELECT.
- tipo **in** ('scritto', 'orale'): tipo deve avere un valore (in) tra quelli dell'insieme ('scritto', 'orale').
- **primary key / unique (lista_attributi)**: la sequenza di valori indicata dalla lista degli attributi, non può ripetersi in modo uguale all'interno della tabella.
Se la lista di attributi contenesse solo studente, allora potrei scrivere all'inizio:

studente integer … primary key,

**Operatori e funzioni su tipi di dato**:

- Applicando il diviso a due interi, tronco il risultato; se almeno uno dei due elementi è decimale allora si fa la divisione decimale.
- Radice quadrata: |/
- Radice cubica: ||/
- String || String: restituisce una stringa che è la concatenazione delle due string.
- Lower(string): tutto in minuscolo
- Upper(string): tutto in maiuscolo
- Substring(string from int for int): sottostringa

**ALTER TABLE:**

- ALTER TABLE table-name ADD column-name datatype;

Aggiungere colonna.

- ALTER TABLE table-name DROP COLUMN column-name;

Togliere colonna.

- ALTER TABLE table-name MODIFY COLUMN column-name datatype;

Modificare il datatype di una colonna.

- ALTER TABLE table-name MODIFY column-name datatype NOT NULL;

Aggiunge il vincolo not null ad una colonna.

- ALTER TABLE table-name ADD UNIQUE(column1, column2...);

Aggiunge un vincolo di integrità alla tabella.

- ALTER TABLE table-name ADD CHECK (condition);

Aggiunge un vincolo check alla tabella.

- ALTER TABLE table-name ADD PRIMARY KEY (column1, column2...);

Aggiungo una chiave primaria.

- ALTER TABLE table-name ADD CONSTRAINT specific-name …;

Al posto dei "…" si può mettere "unique", "check", "primary key".
"specific-name" identifica ciò che aggiungo nei "…"

- ALTER TABLE table-name DROP CONSTRAINT specific-name;

Toglie ciò che ho aggiunto con il nome "specific-name" .

**SELECT:**

- Select generica:

```
SELECT  ListaAttributi
FROM  ListaTabelle
[ WHERE  Condizione ]
```

Es:

```
SELECT  nome, cognome
```

FROM studenti
WHERE eta > 18

- Select con abbreviazioni:

SELECT s.nome as nome, s.cognome as cognome, d.nome as nomeD
FROM studenti as s, docenti as d
WHERE nome = "Luca" AND nomeD = "Rossi"

**as** : rinomina dei termini.

Nel FROM, posso anche evitare di scrivere gli as, ottenendo:

FROM studenti s, docenti d

- Select senza proiezione:

SELECT *
FROM studenti

***** : Stampa tutti gli attributi della tabella risultate (studenti).

- Espressioni nella target list:

SELECT Reddito/**2** as redditoSemestrale
FROM Persone

**DISTINCT**:

SELECT DISTINCT nome
FROM studenti

Nella colonna di nomi risultante, esclude i nomi ripetuti.

**LIKE**:

…
WHERE nome LIKE 'A_d%'

La condizione estrae i nomi che iniziano per "A" ed hanno come 3^ lettera la "d".

**_** : carattere qualunque.

**%** : "resto della parola" (LIKE '%tt%' → parole che contengono 'tt')

**SELECT come JOIN**:

Tabelle: professori {nome, cognome, materia, …}
materie {codice, nome, …}

SELECT materie.nome
FROM professori, materie
WHERE professori.materia = materie.codice AND professori.nome = 'Rossi'

Stampa il nome delle materie insegnate da Rossi.

Attenzione! Vi "deve" essere un vincolo di integrità (references) tra l'attributo "materia" della tabella "professori" e l'attributo "codice" nella tabella "materie".

**ORDER BY**:

SELECT nome …
ORDER BY nome

Dispone i nomi (record) della tabella risultante, in ordine alfabetico (va messo sempre alla fine).

**(NOT) EXISTS**:

```
SELECT *
FROM fornitori
WHERE EXISTS (    SELECT *
                  FROM ordini
                  WHERE fornitori.id = ordini.fornitori_id  );
```

- Restituisce tutti i record della tabella "fornitori" in cui vi è (esiste) almeno un record nella tabella degli "ordini" con lo stesso "fornitori_id" (cioè, restituisce tutti i fornitori che hanno fatto un ordine almeno una volta).
- Usando NOT EXISTS avrei ottenuto la tabella dei fornitori che non hanno mai fatto un ordine.

```
SELECT *      FROM T1
WHERE  T1.attributo1  NOT IN  T2    -- oppure IN
```

- T2 = tabella con 1 solo attributo; esclude le righe di T1 in cui l'attributo1 non compare in T2

Operatori aggregati:

Tabella: Studenti {nome, cognome, eta, altezza…}

```
SELECT  COUNT(*) / SUM(altezza) / AVG(altezza) / MAX(altezza) / MIN(altezza)
FROM Studenti
WHERE eta = 18
```

**COUNT**: conta il numero di righe della tabella risultante.
**SUM**: somma i valori della tabella (singola colonna) risultante.
**AVG**: media dei valori della tabella (singola colonna) risultante.
**MAX / MIN**: massimo / minimo tra i valori della tabella (singola colonna) risultante.
In tutti i casi, il risultato sarà un solo valore (e non una tabella).

**GROUP BY**:

Tabella: CarrieraScolastica {NomeStudente, Svolto, Materia, Voto, …}

```
SELECT NomeStudente, COUNT(*) as NumEsamiDati
FROM CarrieraScolastica cs
WHERE cs.Svolto = true
GROUP BY NomeStudente
```

- Restituisce una tabella con i nomi degli studenti affiancati dal numero degli esami che hanno svolto.
- Senza GROUP BY (e senza il COUNT) avrei ottenuto una colonna con i nomi degli studenti ripetuti più volte; il GROUP BY (con l'utilizzo di un operatore aggregato), mi permette di raggruppare i nomi ripetuti e svolgere operazioni.

**HAVING**:

Es del GROUP BY:

```
… SELECT NomeStudente, COUNT(*)
…
GROUP BY NomeStudente
HAVING  COUNT(*) > 10
```

- Restituisce una tabella con i nomi degli studenti che hanno fatto più di 10 esami (affiancati dal numero degli esami che hanno )
- HAVING permette di applicare condizioni sugli operatori aggregati.

**INSERT**:

```
INSERT INTO Tabella (Attributi) VALUES (Valori)
```

Oppure

```
INSERT INTO Tabella (Attributi) SELECT …
```

- Usato per inserire nuovi dati nella tabella.

Esempi:

```
INSERT INTO Persone VALUES ('Mario',25,182)
INSERT INTO Persone(Nome, Eta, Altezza) VALUES ('Rossi',25,180)
INSERT INTO Persone(Nome, Altezza) VALUES ('Luca',170)
INSERT INTO Persone (Nome) SELECT s.Nome
                           FROM Studenti s
                           WHERE s.Nome NOT IN ( SELECT p.Nome
                                                 FROM Persone p)
```

Info:

- L'ordine e il numero degli attributi è significativo.
- Se non si indica la lista di attributi, si prendono tutti nell'ordine che hanno.
- Se la lista dei valori da inserire non contiene tutti i valori necessari, negli attributi mancanti si mette valore nullo (o valore di default).

**DELETE**:

```
DELETE FROM Tabella
[ WHERE Condizione ]
```

- Usato per cancellare dati dalla Tabella (senza WHERE si sottintende: "WHERE true").

Es:

```
DELETE FROM  Uomini
WHERE  Eta < 18
```

**UPDATE**:

```
UPDATE Tabella
SET attributo1 = valore1, attributo2 = valore2, …
WHERE Condizione
```

- Usato per modificare record della Tabella.

Esempi:

```
UPDATE Persone
   SET Altezza = 175
   WHERE Nome = 'Rossi'
UPDATE Persone
   SET Reddito = Reddito * 1.1
   WHERE Eta < 30
```

**JOIN:**

```
SELECT *
FROM Tabella1 JOIN Tabella2
     ON Tabella1.id = Tabella2.id
```

- Si fa così anche per il left join (il natural join non ha la condizione ON).

**VISTE**:

```
CREATE OR REPLACE VIEW NomeVista ( ListaAttributi ) as  SELECT…
```

Es:

```
CREATE VIEW StudentiMaggiorenni (NomeS, CognomeS, EtaS) as
        SELECT Nome, Cognome, Eta
        FROM Studenti
        WHERE Eta >= 18
```

- Creo una vista che può essere utilizzata come se fosse una tabella, quindi potrò scrivere:

```
SELECT *
FROM StudentiMaggiorenni
```

che stamperà una tabella con attributi "NomeS, CognomeS, EtaS".

UPDATE su VISTE:

- Es usato precedentemente:

```
CREATE VIEW StudentiMaggiorenni …
WHERE Eta >= 18
with check option
```

**with check option**: permette di considerare la condizione del WHERE come un vincolo, quindi non potrò fare un UPDATE di uno studente con età <18 in "StudentiMaggiorenni".

**ALTRE specifiche**

*Numero professori che iniziano con una certa lettera:*

```
SELECT substring( cognome from 1 for 1 ) as iniziale_cognome, count(*)
FROM organico
GROUP BY substring( cognome from 1 for 1 )
ORDER BY iniziale_cognome  -- qui ho potuto usare "iniziale_nome", nel group by no
```

*Spezzare un nome (come "Cesare Mario Antonio"), cioè, disaggregare:*

```
SELECT unnest(string_to_array(nome, ' '))
FROM organico
ORDER BY nome_singolo
-- "alessandra," la virgola viene inclusa nel nome.
```

*Tutti gli aeroporti che posso raggiungere partendo da Roma*

```
WITH RECURSIVE raggiungibile(origine, destinazione) as (
        SELECT partenza, arrivo from voli
                UNION distinct  -- distinct di default
        SELECT origine, arrivo
        FROM raggiungibile, voli
        WHERE destinazione = partenza
)
SELECT * FROM raggiungibile WHERE origine = 'Roma'
-- se mettevo all al posto di distinct, va in loop e si blocca
```

```
        INSERTI INTO musei
        VALUES ('Magnani Rocca', 'Parma');
        INSERTI INTO musei
        VALUES ('Maria Luigia', 'Parma');
```
Posso riscriverlo come:
```
        INSERTI INTO musei
        VALUES ('Magnani Rocca', 'Parma'), ('Maria Luigia', 'Parma');
-- non è standard ma funziona
```