# Fondamenti Di Algebra Relazionale e SQL

Simone Elia
Ingegneria Informatica
Università degli Studi di Bologna


# Contents

**NOTE DELL'AUTORE** **2**

**1 ALGEBRA RELAZIONALE** **3**
1.1 Introduzione . . . 3
1.2 Notazione . . . 3

**2 TABELLE** **4**
2.1 ATTRIBUTI . . . 4
2.1.1 Vincoli sul tipo . . . 4
2.1.2 Impostare le chiavi . . . 4
2.1.3 Ulteriori controlli . . . 5
2.2 ESEMPIO DI TABELLA . . . 6

**3 ISTRUZIONI** **7**
3.1 SELECT . . . 7
3.1.1 Esempi SELECT . . . 7
3.1.2 Esempi SELECT ... WHERE . . . 9
3.1.3 Esempi altre funzioni utili . . . 10
3.1.4 Join . . . 11
3.1.5 Union, Intersect, Except . . . 12
3.2 MODIFICHE DATI . . . 13
3.2.1 Insert . . . 13
3.2.2 Delete . . . 14
3.2.3 Update . . . 14
3.3 GRUPPI . . . 15
3.3.1 Funzioni aggregate . . . 15
3.3.2 Raggruppamenti . . . 16
3.4 SUBQUERY . . . 18
3.5 VISTE . . . 21

**GLOSSARIO** **22**

## NOTE DELL'AUTORE

*Carissimi,*
*il seguente documento è stato creato da me, Simone Elia, per aiutarmi a superare l'esame di* Sistemi Informativi *del corso di ingegneria informatica presso l'*Università degli Studi di Bologna *nell'anno accademico 2022/2023.*

*Ho intenzione di condividere e pubblicare il suddetto in maniera totalmente volontaria e gratuita perché spero possa tornare utile ai miei compagni di corso, ai futuri studenti di ingegneria o semplicemente a chi sarà intenzionato ad imparare i fondamenti di* `SQL` *in maniera rapida e indipendente.*

*Se così dovesse essere, e quindi il mio lavoro vi aiutasse in maniera più o meno significativa al raggiungimento dei vostri obbiettivi, vi lascio la possibilità, ovviamente volontaria, di regalarmi qualche spicciolo, per offrirmi semplicemente un caffè o per dimostrarmi un simbolico apprezzamento.*

*Vi ringrazio in anticipo,*
*Simone Elia*

paypal.me/simoneeelia

# 1 ALGEBRA RELAZIONALE

## 1.1 Introduzione

Il modello **relazionale** è un modello utilizzato per mettere in relazione dei dati, inventato per sostituire il modello gerarchico e quello reticolare, entrambi basati sui puntatori e non sui valori stessi.

In matematica la **relazione** è un insieme di *n*-ple, ordinate, su domini non necessariamente distinti, ma nel caso di domini ripetuti l'interpretazione si complica, poiché la posizione è l'unico modo per associare ai dati la corretta interpretazione. Per ovviare a questo problema il modello relazionale associa ad ogni occorrenza del dominio un nome **univoco**, detto **attributo**, il vincolo posizionale perde quindi d'importanza e viene resa più facile una **rappresentazione tabellare**.

Nella rappresentazione tabellare si chiamano quindi **attributi** i nomi delle colonne, ogni riga della colonna viene denominata **tupla**, l'insieme delle tuple è detto **istanza**. Alla tabella (**relazione**), viene dato un nome, il nome della relazione e l'insieme di attributi costituiscono lo **schema di una relazione**.

Infine l'insieme delle **relazioni** costituisce il **DataBase**.

## 1.2 Notazione

- A o {A} rappresenta un attributo.
- X (o **Y**) rappresenta l'insieme degli attributi di una relazione: X = {$A_1$,$A_2$, ..., $A_n$}
- t rappresenta una tupla
- t[A] o t.A rappresenta il valore della tupla t sull'attributo A
- t[$A_1$, $A_2$] rappresenta il valore della tupla t su {$A_1$, $A_2$}
- R rappresenta il nome di una relazione
- r rappresenta l'istanza della relazione
- R(X) rappresenta lo schema di una realazione su X
- **R** reppresenta il DataBase: **R** = {$R_1(X_1)$, $R_2(X_2)$, ..., $R_m(X_m)$}
- **r** rappresenta l'istanza di un DB: **r** = $r_1$, $r_2$, ..., $r_m$

# 2 TABELLE

Per creare una tabella xxx:

```
CREATE TABLE xxx(
    ...
)
```

Per distruggerla:

```
DROP TABLE xxx
```

## 2.1 ATTRIBUTI

Per aggiungere attributi

```
CREATE TABLE xxx(
    ATTRIBUTO1 ...,
    ATTRIBUTO2 ...,
    ATTRIBUTO3 ...
)
```

### 2.1.1 Vincoli sul tipo

Il primo attributo deve essere un vincolo sul tipo

- Intero: `int`
- Decimale con x cifre e y cifre decimali: `dec x,y`
- Carattere: `char`
- Stringa di esattamente N caratteri: `char(N)`
- Stringa di massimo N caratteri: `varchar(N)`
- Data: `date`

### 2.1.2 Impostare le chiavi

Le chiavi devono contenere valori **non nulli**.
Per creare una chiave composta da un solo attributo:

```
ATTRIBUTO1 int NOT NULL UNIQUE
```

Per creare foreign key (attributi che sono chiavi in altri schemi):

```
    ATTRIBUTO1 int NOT NULL REFERENCES xxy(ATTRIBUTOx)
```

Per creare chiavi primarie composte da più attributi:

```
    PRIMARY KEY (ATTRIBUTOx, ATTRIBUTOy, ...)
```

Per creare *foreign key* con ulteriori specifiche:

```
    FOREIGN KEY ATTRIBUTOx REFERENCES yyy
        ON DELETE ...
        ON UPDATE ...
```

le specifiche possono essere:

- `NO ACTION` utilizzato per `ON DELETE` e `ON UPDATE` cancellazioni o modifiche non permesse.
- `CASCADE` utilizzato per `ON DELETE` elimina a cascata tutte le tuple che referenziano una tupla cancellata.
- `SET NULL` utilizzato per `ON DELETE`, la foreign key viene resa `NULL`, se permesso.

#### 2.1.3 Ulteriori controlli

Per attuare ulteriori controlli su degli attributi si usa il `CHECK`, seguito da un' espressione booleana:

```
    ATTRIBUTO1 int CHECK (ATTRIBUTO1 >= 10)
                         AND (ATTRIBUTO1 <= 100)
```

Se si vogliono inserire vincoli che riguardano attributi:

```
    CONSTRAINT nomeControllo CHECK (ATTRIBUTO1 >= ATTRIBUTO2)
                                AND (ATTRIBUTO3 IS NULL)
```

**N.B.** eCONSTRAINT si può utilizzare anche come semplice check sulla stessa riga di un attributo, è utile perchè assegna un nome a un controllo.

## 2.2 ESEMPIO DI TABELLA

```
CREATE TABLE esempioA(
    ATT1 int NOT NULL UNIQUE,
    ATT2 char(2) CONSTRAINT controlloAtt2 CHECK (ATT2 = 'SI')
                                              OR (ATT2 = 'NO')
);

CREATE TABLE esempioA(
    ATTa  int NOT NULL FOREIGN KEY esempioA(ATT1),
    ATTb  varchar(12) NOT NULL,
    ATTc  varchar(12),
    PRIMARY KEY (ATTa, ATTb),
    CONSTRAINT controlloEsempio CHECK (ATTc IS NULL)
                                    OR (ATTc < ATTb)
);
```

# 3 ISTRUZIONI

Sono fornite una serie di istruzioni DML (Data Manipulation Lenguage) per compiere operazioni su dati e tabelle. In particolare

- `SELECT`: Per interrogare il DB (query) e visualizzare solo dati filtrati.
- `INSERT`: Per inserire nuove tuple nel DB.
- `DELETE`: Per eliminare tuple nel DB.
- `UPDATE`: Per aggiornare tuple nel DB

## 3.1 SELECT

L' istruzione `SELECT` è spesso accompaganta da ulteriri specifiche:

```
SELECT ...
FROM ...
WHERE ...
```

Il `SELECT` indica **cosa** si vuole come risultato,
il `FROM` indica **da dove** si vuole prendere i dati,
il `WHERE` indica **quali condizioni** devono essere rispettate.
Inoltre è possibile utilizzare un ” se si vogliono tenere tutti i dati estratti.

### 3.1.1 Esempi SELECT

Per considerare tutti i dati riportati nella `TABLE` 'Corsi':

```
SELECT *
FROM Corsi
```

è possibile però considerarne solo alcuni:

```
SELECT CodCorso, Titolo, Anno
FROM Corsi
```

o solamente uno:

```
SELECT Titolo
FROM Corsi
```

tuttavia questa soluzione non elimina i duplicati. Quindi è necessario utilizzare la keyword `DISTINCT`:

```
SELECT DISTINCT Titolo
FROM Corsi
```

Inoltre è possibile visualizzare dati non presenti tra gli attributi iniziali:

```
SELECT Matricola, Voto/3
FROM Esami
```

Tuttavia questa soluzione ha due problemi: viene eseguita una divisione tra interi (il risultato è intero) e la colonna dei `voti/3` non ha titolo. Per risolvere il primo problema è possibile applicare un `casting`:

```
SELECT Matricola, CAST(Voto AS Dacimal (4,2))/3
FROM Esami
```

(la divisione poteva essere fatta anche dentro l'operazione di casting). Per assegnare un titolo si usa nuovamente la keyword `AS`:

```
SELECT Matricola, CAST(Voto AS Dacimal(4,2))/3 AS Decimi
FROM Esami
```

In questo modo il risultato ha due colonne, la prima composta dalle matricole degli studenti, la seconda, denominata `Decimi` contiene i voti divisi per 3 ed espressi con 2 cifre decimali.
NOTA: `AS` può essere usato anche per rinominare colonne già esistenti:

```
SELECT Matricola AS CodStudente
FROM Esami
```

Infine è possibile selezionare una colonna formata dall'unione di stringhe delle altre due:

```
SELECT Matricola,
    Nome CONCAT ' ' CONCAT Cognome AS NomeCognome
FROM Studenti
```

In questo modo tutti i nomi della colonna 'Nome' vengono concatenati prima a uno spazio vuoto e poi al rispettivo cognome. Un altro esempio più complesso può essere:

```
SELECT Matricola CONCAT ' ha preso '
        CONCAT Voto CONCAT ' in '
        CONCAT CodCorso
FROM Esami
```

### 3.1.2 Esempi SELECT ... WHERE

Per applicare delle selezioni e quindi filtrare solo determinate tuple è necessario utilizzare il `WHERE`:

```
SELECT Matricola, Voto, Lode
FROM Esami
WHERE CodCorso = 913
```

In questo modo vengono filtrate solo gli esami del corso 913.

Inoltre è possibile applicare più condizioni alla selezione utilizzando l'AND come nell'algebra booleana.

```
SELECT Matricola, Voto, Lode
FROM Esami
WHERE CodCorso = 913
AND Voto = 28
```

(analogamente è possibile utilizzare OR).
Per selezionare stringhe che rispettano un determinato pattern è possibile utilizzare la keyword LIKE:

```
SELECT *
FROM Studenti
WHERE Email LIKE '_b%.it'
```

Dove '_' indica un carattere arbitrario e '%' una stringa arbitraria (la query precedente filtra tutti gli studenti che hanno una mail che ha una 'b' come secondo carattere e termina con '.it').
Per controllare che un numero appartenga a un determinato intervallo si può usare BETWEEN:

```
SELECT *
FROM Esami
WHERE Voto BETWEEN 26 AND 29
```

rimangono poi la possibilità di usare >, <, >=, <=, = (uguale), <> (diverso). NOTA: i controlli sono validi per tutti i dati ordinabili (come ad esempio date)
La keyword IN viene utilizzata per controllare se un determinato dato è presente in una lista di valori predefiniti:

```
SELECT *
FROM Esami
WHERE CodCorso IN (483, 729)
```

alcune operazioni trascurano le tuple in cui un determinato valore è nullo. Per includere (o escludere) queste tuple è possibile utilizzare IS NULL:

```
SELECT *
FROM Studenti
WHERE DataNascita IS NULL
```

#### 3.1.3 Esempi altre funzioni utili

Per compiere il *sort* delle tuple secondo determinati attributi viene utilizzato ORDER BY, seguito dagli attributi secondo i quali si vuole ordinare (in ordine di

priorità), gli attributi possono essere accompagnati da `ASC` se devono essere in ordine crescente o `DESC` se in ordine decrescente.

```
SELECT *
FROM Esami
ORDER BY CodCorso, Voto DESC
```

Se vengono utilizzate più tabelle diventa necessario capire a quale tabella ci si sta riferndo, esiste quindi la notazione <nome tabella>.<nome colonna>:

```
SELECT Studenti.Cognome, Studenti.Nome, Esami.
FROM Esami, Studenti
WHERE Esami.Matricola = Studenti.Matricola
```

e per comodità si può usare la seguente notazione per abbreviare il nome delle tabelle:

```
SELECT S.Cognome,S.Nome,E.
FROM Esami E, Studenti S
WHERE E.Matricola = S.Matricola
```

equivalente a `FROM Esami AS E, Studenti AS S`.

#### 3.1.4 Join

Per applicare dei *join* esistono diversi modi, primo tra tutti:

```
SELECT S.Cognome,S.Nome,E.*
FROM Esami E, Studenti S
WHERE E.Matricola = S.Matricola
```

in questo caso `FROM Esami E, Studenti S` compie il prodotto cartesiano delle due tabelle, mentre il `WHERE` è detto **predicato di join**.
Il concetto è estendibile a più tabelle:

```
/*I docenti dei corsi di cui lo studente
Giorgio Bianchi ha sostenuto l'esame*/

    SELECT C.Docente
    FROM Corsi C, Esami E, Studenti S
    WHERE C.CodCorso = E.CodCorso
    AND E.Matricola = S.Matricola
    AND S.Cognome = 'Bianchi'
    AND S.Nome = 'Giorgio'
```

Il *join* può essere esplicitato attraverso la funzione `... JOIN ... ON ()`:

```
    SELECT S., E.CodCorso, E.Voto, E.Lode
    FROM Studenti S JOIN Esami E ON (S.Matricola = E.Matricola)
    WHERE E.Voto > 26
```

**ATTENZIONE:** negli *outer join* prima si applicano tutti i predicati del join e poi si uniscono le tuple *dangling*:

```
    SELECT *
    FROM Studenti S LEFT JOIN Esami E ON
    (S.Matricola = E.Matricola) AND (E.Anno = 2)
    WHERE E.CodCorso IS NULL
-- Restituisce gli studenti senza esami nel secondo anno

    SELECT *
    FROM Studenti S LEFT JOIN Esami E ON
    (S.Matricola = E.Matricola)
    WHERE E.CodCorso IS NULL AND (E.Anno = 2)
-- Non restituisce nulla
```

#### 3.1.5 Union, Intersect, Except

Possono essere compiute operazioni insiemistiche tra più `SELECT` utilizzando `UNION` per l'unione di due insiemi, `INTERSECT` per l'intersezione e `EXCEPT` per le sottrazioni (il primo insieme senza gli elementi dell'intersezione).
Aggiungendo la *keyword* `ALL` non vengono eliminati i duplicati.

**ESEMPI:**

```
    SELECT A
    FROM R
    UNION
    SELECT C
    FROM S
-- Compie l'unione tra la colonna A della tabella R
-- e la colonna C della tabella S.
-- Il risultato sarà una colonna senza nome.
```

```
    SELECT A
    FROM R
    UNION
    SELECT C AS A
    FROM S
-- Compie l'unione tra la colonna A della tabella R
-- e la colonna C (rinominata A) della tabella S.
-- Il risultato sarà una colonna denominata A.
```

```
    SELECT B
    FROM R
    UNION ALL
    SELECT B
    FROM S
-- Compie l'unione tra la colonna B della tabella R
-- e la colonna B della tabella S.
-- Il risultato sarà una colonna denominata B che conterrà
-- duplicati.
```

## 3.2 MODIFICHE DATI

### 3.2.1 Insert

Per inserire nuove tuple in una tabelle si usa la *keyword* `INSERT INTO` accompagnata dalla *keyword* `VALUES`:

```
    INSERT INTO Corsi(Titolo,CodCorso,Docente,Anno)
    VALUES  ('StoriaAntica',456,'Grigi',3),
            ('StoriaModerna',457,'Gialli',2)
```

La lista degli attributi da inserire può essere omessa, nel caso gli attributi vengono considerati nell'ordine in cui sono scritti:

```
    INSERT INTO Corsi
    VALUES  (456,'StoriaAntica','Grigi',3),
            (450,'Storia Moderna', NULL, DEFAULT)
-- Inserimento di tuple senza specificare gli attributi inseriti
```

Inoltre è possibile specificare solo alcuni degli attributi da inserire, gli altri verranno sostituiti con il valore di *default* o il valore `NULL` (se l'attributo non ha default e non ammette valori nulli viene lanciato errore):

```
    INSERT INTO Corsi(CodCorso,Titolo)
    VALUES (456,‘StoriaAntica’)
-- ’CodCorso’ e ’Titolo’ vengono inseriti, gli altri
-- attributi della table ’corsi’ vengono inseriti come
-- DEFAULT o NULL
```

Infine è possibile utilizzare come argomento dell’`INSERT` una *query*:

```
    INSERT INTO StudentiSenzaEmail(Matr,Cog,Nom)
    SELECT Matricola,Cognome,Nome
    FROM Studenti
    WHERE Email IS NULL
```

#### 3.2.2 Delete

L’istruzione `DELETE` elimina le tuple di una tabella che rispettano una determinata caratteristica:

```
    DELETE
    FROM Corsi
    WHERE Docente = ‘Biondi’
-- elimina i corsi di Biondi
```

o tutte le tuple:

```
    DELETE
    FROM Corsi
-- elimina tutte le tuple
```

#### 3.2.3 Update

L’istruzione `UPDATE` è seguita da `SET` e aggiorna le tuple nella maniera definita dal `SET`:

```
    UPDATE Corsi
    SET Docente = ‘Bianchi’,
        Anno = 2
    WHERE Docente = ‘Biondi’
-- I corsi con docente ’Biondi’ ora hanno docente ’Bianchi’
-- e sono del secondo anno
```

```
    UPDATE Dipendenti
    SET Stipendio = 1.1Stipendio
    WHERE Ruolo = ‘Programmatore’
-- Lo stipendio dei dipendenti programmatori viene
-- aumentato del 10%
```

## 3.3 GRUPPI

### 3.3.1 Funzioni aggregate

Spesso è utile compiere delle operazioni non su una singola tupla, ma su un gruppo:

- `MIN`: minimo.
- `MAX`: massimo.
- `SUM`: somma.
- `AVG`: media aritmetica.
- `COUNT`: contatore.

Le funzioni aggregate vengono utilizzate nella `SELECT`.
`SUM` somma tra loro tutti gli elementi di una colonna:

```
    SELECT SUM(Stipendio) AS ToTStipS01
    FROM Imp
    WHERE Sede = ‘S01’
-- Somma tutti gli stipendi della Sede ’S01’
```

```
    SELECT SUM(Stipendio*12) AS ToTStipAnnuiS01
    FROM Imp
    WHERE Sede = ‘S01’
-- Moltiplica x12 tutti gli stipendi della Sede ’S01’ e
-- li somma tra loro
```

```
    SELECT SUM(DISTINCT Stipendio)
    FROM Imp
    WHERE Sede = ‘S01’
-- Somma tutti gli stipendi presi una sola volta.
```

`COUNT` conta le righe:

```
    SELECT COUNT(*) AS NumImpS01
    FROM Imp
    WHERE Sede = 'S01'
-- Conta il numero di tuple con sede 's01'
```

```
    SELECT COUNT(Stipendio) AS NumStipS01
    FROM Imp
    WHERE Sede = 'S01'
-- Conta il numero di tuple con sede 's01' e lo stipendio
-- diverso da NULL
```

`AVG` fa una media aritmetica di un attributo:

```
    SELECT AVG(Stipendio) AS AvgStip
    FROM Imp
-- Fa una media di tutti gli stipendi
```

```
    SELECT AVG(CAST(Stipendio AS Decimal(6,2))) AS AvgStip
    FROM Imp
-- fa una media di tutti gli stipendi e la mostra con 2 decimali
```

`MIN` e `MAX` restituiscono il valore più piccolo e più grande di un attributo:

```
    SELECT MAX(Stipendio), MIN(Stipendio)
    FROM Imp
```

### 3.3.2 Raggruppamenti

**ATTENZIONE!** è errato fare la select di un attributo e di una funzione aggregata contemporaneamente:

```
    SELECT MAX(Stipendio), Nome
    FROM  Imp
-- NO!
```

questo perchè le funzioni aggregate restituiscono **un singolo valore**.

Per risolvere questo problema è necessario fare dei raggruppamenti attraverso la *keyword* `GROUP BY`:

```
    SELECT Sede, COUNT(*) AS NumProg
    FROM Imp
    WHERE Ruolo = 'Programmatore'
    GROUP BY Sede
-- Viene selezionata una tabella di due colonne, la prima
-- con il codice della sede, la seconda con il numero di
-- programmatori presenti nella rispettiva sede.
```

**ESEMPI:**

```
    SELECT I.Ruolo, AVG(I.Stipendio) AS AvgStip
    FROM Imp I JOIN Sedi S ON (I.Sede = S.Sede)
    WHERE S.Citta = 'Milano'
    GROUP BY I.Ruolo
-- Restituisce per ogni ruolo lo stipendio medio a Milano.
```

```
    SELECT I.Sede, AVG(I.Stipendio) AS AvgStip
    FROM Imp I JOIN Sedi S ON (I.Sede = S.Sede)
    WHERE S.Citta = 'Milano'
    GROUP BY I.Sede
-- Restituisce per ogni sede a Milano lo stipendio medio.
```

```
    SELECT I.Sede, I.Ruolo, AVG(I.Stipendio)
    FROM Imp I JOIN Sedi S ON (I.Sede = S.Sede)
    WHERE S.Citta = 'Milano'
    GROUP BY I.Sede, I.Ruolo
-- Restituisce lo stipendio medio per ogni ruolo e sede a Milano.
```

Mediante la *keyword* `HAVING` è possibile selezionare dei gruppi sulla base di loro proprietà:

```
    SELECT Sede, COUNT(*) AS NumImp
    FROM Imp
    GROUP BY Sede
    HAVING COUNT(*) > 2
-- Raggruppa le sedi con almeno 2 impiegati
```

```
    SELECT Sede, COUNT(*) AS NumImp
    FROM Imp
    GROUP BY Sede
    HAVING MAX(Stipendio) <= 2000
-- Raggruppa le sedi in cui lo stipendio massimo non supera 2000
```

**ESEMPI:**

```
/* Per ogni sede di Bologna in cui il numero di impiegati è
almeno 3, si vuole conoscere il valor medio degli stipendi,
ordinando il risultato per valori decrescenti di stipendio
medio e quindi per sede */
    SELECT I.Sede, AVG(Stipendio) AS AvgStipendio
    FROM Imp I, Sedi S
    WHERE I.Sede = S.Sede
    AND S.Citta = 'Bologna'
    GROUP BY I.Sede
    HAVING COUNT(*) >= 3
    ORDER BY AvgStipendio DESC, Sede
```

### 3.4 SUBQUERY

A volte è utile **innestare diverse *query*** per controllare, attraverso la *keyword* `IN`, se un valore rispetta o no determinate condizioni:

```
-- impiegati delle sedi di Milano
    SELECT CodImp
    FROM Imp
    WHERE Sede IN (SELECT Sede
                FROM Sedi
                WHERE Citta = 'Milano')
```

che equivale:

```
-- impiegati delle sedi di Milano
    SELECT CodImp
    FROM Imp
    WHERE Sede IN ('S01','S03')
```

ma in particolare ciò permette di avere un confronto direttamente tra tuple:

```
    SELECT CodImp
    FROM Imp
    WHERE Sede <> 'S01'
    AND (Ruolo, Stipendio) IN
            (SELECT Ruolo, Stipendio
            FROM Imp
            WHERE Sede = 'S01')
```

Si possono usare gli operatori =, <,... se e solo se la *subquery* **non restituisce più di una tupla**:

```
    SELECT CodImp
    FROM Imp
    WHERE Stipendio  = (SELECT MIN(Stipendio)
                        FROM Imp)
-- impiegati con stipendio minimo
```

Inoltre esistono due *keyword*:

- ANY, la relazione vale per almeno uno dei valori
- ALL, la relazione vale per almeno tutti i valori

Ad esempio:

```
    SELECT Responsabile
    FROM Sedi
    WHERE Sede = ANY (SELECT Sede
                      FROM Imp
                      WHERE Stipendio > 1500)
-- Controlla che la sede sia tra le sedi con almeno un impiegato
-- con lo Stipendio maggiore di 1500
```

**N.B.** =ANY equivale a IN
Oppure:

```
    SELECT CodImp
    FROM Imp
    WHERE Stipendio <= ALL (SELECT Stipendio
                            FROM Imp)
-- impiegati con stipendio minimo
```

Ovviamente è possibile innestare più *subquery*:

```
    SELECT CodImp
    FROM Imp
    WHERE Sede IN (SELECT Sede
                FROM Sedi
                WHERE Citta NOT IN (SELECT Citta
                        FROM Prog
                        WHERE CodProg = 'P02'))
-- Sedi in città in cui non c'è il progetto 'p02'
```

**ATTENZIONE!** la precedente *query* **non** è equivalente a:

```
    WHERE Sede IN (SELECT Sede
            FROM Sedi, Prog
            WHERE Sedi.Citta <> Prog.Citta
            AND Prog.CodProg = 'P02')
-- Sedi non in città in cui tutti i progetti sono 'p02'
```

Mediante `EXISTS` è possibile verificare se una *subquery* restituisce almeno una tupla:

```
    SELECT Sede
    FROM Sedi S
    WHERE EXISTS (SELECT *
            FROM Imp
            WHERE Ruolo = 'Programmatore')
```

mentre con `NOT EXISTS` il predicato è vero so la *non restituisce alcuna tupla*.

Una *subquery* si dice **correlata** se fa riferimento a valori definiti nel blocco esterno:

```
    SELECT Sede
    FROM Sedi S
    WHERE EXISTS (SELECT *
            FROM Imp
            WHERE Ruolo = 'Programmatore'
            AND Sede = S.Sede)
-- sedi con almeno un programmatore
```

Spesso è possibile ricondurre forme correlate in forme 'piatte', senza quindi *query* innestate.

```
/* 'Sedi in cui sono presenti tutti i ruoli'
equivale a
'Sedi in cui non esiste un ruolo non presente' */

    SELECT Sede FROM Sedi S
    WHERE NOT EXISTS (SELECT *
                FROM Imp I1
                WHERE NOT EXISTS (SELECT *
                            FROM Imp I2
                            WHERE S.Sede = I2.Sede
                            AND I1.Ruolo = I2.Ruolo))
```

**N.B.** le *subquery* vengono utilizzate anche per `DELETE`, `UPDATE` o al momento dell'inizializzazione delle tabelle per i `CHECK`, come nei seguenti esempi:

```
    DELETE FROM Imp
    WHERE Sede IN (SELECT Sede
                FROM Sedi
                WHERE Citta = 'Bologna')
-- elimina gli impiegati di Bologna
```

```
    UPDATE Imp
    SET Stipendio = 1.1*Stipendio
    WHERE Sede IN (SELECT S.Sede
                FROM Sede S, Prog P
                WHERE S.Citta = P.Citta
                AND P.CodProg = 'P02')
-- Vengono aumentati del 10% gli stipendi nella stessa città
-- in cui si lavora al progetto 'P02'
```

```
    CHECK (2 <= (SELECT COUNT(*) FROM Imp I
                WHERE I.Sede = Sede -- correlazione
                AND I.Ruolo = 'Programmatore'))
-- Ogni sede deve avere almeno due programmatori
```

### 3.5 VISTE

Attraverso le viste è possibile creare delle 'tabelle virtuali'. La *keyword* `CREATE VIEW` permette di creare una nuova tabella dinamica partendo dal risultato delle *query*.

```
    CREATE VIEW ProgSedi(CodProg,CodSede)
    AS      SELECT P.CodProg, S.Sede
            FROM Prog P, Sedi S
            WHERE P.Citta = S.Citta
-- Genera una nuova tabella con due colonne
-- denominata 'ProgSedi'
```

Le viste possono essere a loro volta soggette a *query*:

```
    SELECT *
    FROM ProgSedi
    WHERE CodProg = 'P01'
```

## GLOSSARIO

Vengono riassunte in seguito tutte le *keywords* utilizzate nella dispensa:

`ALL`: utilizzato per compiere controlli su *subquery* che restituiscono più di un valore, il controllo è vero se è valido per tutti i valori restituiti (Esempio: `WHERE Stipendio <= ALL (SELECT Stipendio FROM Imp)`).

`AND`: utilizzato per unire più espressioni che richiedono di calcolare un valore booleano.

`ANY`: utilizzato per compiere controlli su *subquery* che restituiscono più di un valore, il controllo è vero se è valido per almeno uno dei valori restituiti (Esempio: `WHERE Sede = ANY (SELECT Sede FROM Imp WHERE Stipendio > 1500)`).

`AS`: utilizzato nella `SELECT` per assegnare un nome alle colonne o rinominarle, nel `CAST` per indicare come applicare il casting e nelle `CREATE VIEW`.

`ASC`: posto negli `ORDER BY` indica che l'attributo deve esserer in ordine ascendente (Esempio: `SELECT AVG(Stipendio) AS AvgStip`).

`AVG`: fa una media di tutti i valori di un attributo.

`BETWEEN`: utilizzato per controllare che un dato appartenga a un intervallo (Esempio: `WHERE Voto BETWEEN 26 AND 29`)

`CONCAT`: Utilizzato nella `SELECT` per concatenare tra loro stringhe di una colonna e rispettive stringhe di un'altra colonna, o semplicemente unire ogni elemento della colonna selezionata con una stringa fissata.

`COUNT`: conta le tuple selezionate, generali (*) o di un determinato attributo (Esempio: `SELECT COUNT(Stipendio) AS NumStipS01`).

`CREATE VIEW`: comando che crea delle tabelle virutali e dinamiche partendo dal risultato di una *query* (Esempio: `CREATE VIEW ProgSedi(CodProg,CodSede) AS SELECT ...`).

`DELETE`: elimina le tuple da una tabella, (la tabella è specificata con un `FROM`).

`DESC`: posto negli `ORDER BY` indica che l'attributo deve essere in ordine discendente.

`DINSTINCT`: utilizzato nella `SELECT`, elimina le righe duplicate.

`EXCEPT`: compie l'operazione insiemistica di sottrazione tra due `SELECT`, il risultato non contiene duplicati.

`EXCEPT ALL`: compie l'operazione insiemistica di sottrazione tra due `SELECT`, il risultato contiene duplicati.

`EXISTS`: utilizzato nelle *subquery*, verifica se il risultato della *subquery* restituisce almeno una tupla.

`FROM`: utilizzato nelle `SELECT` (o `DELETE`) per indicare le tabelle su cui compiere selezioni.

`GROUP BY`: utilizzato nelle `SELECT` per compiere raggruppamenti su cui applicare delle funzioni aggragate.

`HAVING`: utilizzato assieme a `GROUP BY`, aggiunge delle specifiche da rispettare (Esempio: `GROUP BY Sede HAVING COUNT(*) > 2`).

`IN`: utilizzato per controllare se un valore appartiene o no a un insieme (Esempio: `WHERE CodCorso IN (483, 729, 315)`).

`INSERT INTO`: utilizzato per aggiungere tuple ad una table, può essere seguito da `VALUES` o da una `SELECT`.

`INTERSECT`: compie l'operazione insiemistica d'intersezione tra due `SELECT`, il risultato non contiene duplicati.

`INTERSECT ALL`: compie l'operazione insiemistica d'intersezione tra due `SELECT`, il risultato contiene duplicati.

`IS NULL`: controlla se un determinato valore è nullo.

`JOIN`: utilizzato nel from per compiere un *join* tra più tabelle, può essere accompagnato da ulteriori specifiche:

- `INNER JOIN` - default
- `CROSS JOIN`
- `LEFT JOIN`
- `RIGHT JOIN`

- `FULL JOIN`

.

`LIKE`: utilizzato nelle espressioni booleane per controllare se le stringhe seguono un determinato pattern. Il pattern è definito da una stringa che contiene '_' al posto di un carattere arbitrario e '%' al posto di una stringa arbitraria. (Esempio: `WHERE Email LIKE '_b%.it'`)

`MAX`: restituisce il valore maggiore di un attributo (Esempio: `SELECT MAX(Stipendio)`).

`MIN`: restituisce il valore minore di un attributo (Esempio: `SELECT MIN(Stipendio)`).

`NOT EXISTS`: utilizzato nelle *subquery*, verifica se il risultato della *subquery* non restituisce tuple.

`NULL`: nessun valore, le *query* non considerano i valori `NULL` nè falsi nè veri.

`ON DELETE`: utilizzato durante la definizione delle `FOREIGN KEY`, prevede tre modalità:

- `ON DELETE NO ACTION`, cancellazione non permessa
- `ON DELETE CASCADE`, vengono eliminate le tuple referenziate
- `ON DELETE SET NULL`, vengono settate a `NULL` le foreign key referenziate

.

`ON UDATE`: utilizzato durante la definizione delle `FOREIGN KEY`, prevede tre modalità:

- `ON UPDATE NO ACTION`, modifica non permessa
- default, con DB2 le modifiche di una tupla ne aggiornano le modifiche delle foreign key referenziate.

.

`OR`: utilizzato per unire più espressioni che richiedono di calcolare un valore booleano.

`ORDER BY`: utilizzato nelle `SELECT` per indicare la chiave di ordinamento delle tuple.

`SELECT`: comando principale per compiere *query*, si consulti la dispensa per ulteriori informazioni.

`SET`: utilizzato negli `UPDATE` aggiorna le tuple, può essere implementato da un `WHERE`.

`SUM`: somma tutti i valori di un attributo (Esempio: `SELECT SUM(Stipendio) AS ToTStipS01`).

`UNION`: compie l'operazione insiemistica d'unione tra due `SELECT`, il risultato non contiene duplicati.

`UNION ALL`: compie l'operazione insiemistica d'unione tra due `SELECT`, il risultato contiene duplicati.

`UPDATE`: aggiorna le tuple secondo il `SET`

`VALUES`: utilizzato con `INSERT INTO`, specifica le tuple da inserire in una tabella.

`WHERE`: utilizzato per filtrare tuple di una tabella che rispettano determinate condizioni.