Bibbia SQL

Anno 2022/23

A cura di:

Erica Corda

Matteo Dessì

Leonardo Dessì

Simone Giuffrida

SOMMARIO

Introduzione

Il comando SQL fondamentale per la definizione dei dati è l'istruzione CREATE, che può essere usata per creare schemi, tabelle, domini e altri oggetti. SQL usa i termini **tabella, riga e colonna** in luogo, rispettivamente, dei termini **relazione, tupla e attributo** del modello relazionale.

I tipi di dato che un attributo può assumere possono essere:

- Tipi predefiniti:
  - → tipi **numerici** (esatti e approssimati):
    - INTEGER: numero intero costituito al massimo da 9 cifre.
    - SMALLINT: numero intero per definizione più piccolo di quello esprimibile tramite il tipo INTEGER.
    - FLOAT: espresso singolarmente indica un numero in virgola mobile; FLOAT(n): indica invece un numero in virgola mobile della lunghezza di n bit.
    - REAL: è anch'esso un numero in virgola mobile ma espresso in teoria con maggiore precisione rispetto a FLOAT.
    - DOUBLE PRECISION: è un numero in virgola mobile in pratica equivalente al tipo REAL;
  - → tipi **carattere** (singoli caratteri o stringhe):
    - CHAR: indica un singolo carattere; CHAR(n) indica invece una stringa della lunghezza fissa di n caratteri.
    - VARCHAR(n): indica una stringa di lunghezza variabile composta al massimo da n caratteri;
  - → tipi **temporali** (data, ora, intervalli di tempo):
    - DATE: indica una data intera completa di giorno, mese ed anno.
    - TIMESTAMP: indica un'informazione contenente data e ora.
    - TIME: indica un orario completo di ore, minuti ed secondi.
    - INTERVAL: indica un intervallo di tempo;
  - → tipi **booleani**:
    - BOOLEAN: rappresenta i valori booleani TRUE, FALSE, UNKNOWN (logica a tre valori). Il valore UNKNOWN è stato introdotto in SQL per la gestione dei confronti con NULL.
- Tipi user-defined: creazioni di tipi personalizzati.

Quando si assegna ad un attributo il valore NULL questo permette di rappresentare **l'assenza di informazione**, ma non è un valore, si usa per attributi non applicabili o dal valore non noto.

- → Quando un valore NULL è coinvolto in un **confronto**, il risultato è UNKNOWN;
- → Quando un valore NULL è coinvolto in **un'operazione algebrica**, il risultato è NULL;
- → Ciascun valore NULL è considerato, in generale, differente da ogni altro NULL.

A partire dai domini predefiniti, è possibile definire nuovi domini da associare agli attributi di uno schema:

```
CREATE DOMAIN NomeDominio AS TipoDiDato [ ValoreDiDefault ] [ Vincoli ]
```

Dove **TipoDiDato** è un **dominio elementare** cioè che è un tipo predefinito o definito dall'utente in precedenza.

ESEMPIO:

```
CREATE DOMAIN Voto AS SMALLINT DEFAULT NULL CHECK ( value >=18 AND value <= 30
                                    )

            CREATE DOMAIN AnniCorso AS SMALLINT DEFAULT 3 NOT NULL
```

SQL consente la definizione di uno schema di basi di dati come collezione di oggetti (domini, tabelle, viste, etc):

```
CREATE SCHEMA [ NomeSchema ] [ AUTHORIZATION NomeProprietario ]
{ DefElementoSchema }
```

ESEMPIO:

```
CREATE SCHEMA AZIENDA AUTHORIZATION JSMITH
```

Se **AUTHORIZATION NomeProprietario** viene omesso, si assume come **proprietario l'utente che ha lanciato il comando.**

Se **NomeSchema** viene omesso, si assume come **nome dello schema il nome del proprietario.**

Definizione delle tabelle

Per definire una tabella si utilizza questa sintassi:

```
CREATE TABLE NomeTabella
( NomeColonna Dominio [ ValoreDiDefault ]
[ VincoliColonna ],
.....,
NomeColonna Dominio [ ValoreDiDefault ] [ VincoliColonna ],
[ AltriVincoli ]
)
```

ESEMPIO:

```
CREATE TABLE Impiegato
( Matricola CHAR(6) PRIMARY KEY,
Nome VARCHAR(20) NOT NULL,
Cognome VARCHAR(20) NOT NULL,
Dipartimento VARCHAR(15),
Stipendio DECIMAL(10,2) DEFAULT 0,
FOREIGN KEY(Dipartimento)
REFERENCES Dipartimento(NomeDip),
UNIQUE (Cognome, Nome)
)
```

Solitamente lo **schema** SQL in cui vengono dichiarate le tabelle è specificato implicitamente dall'ambiente in cui sono eseguite le istruzioni CREATE TABLE. Si può **esplicitamente associare il nome dello schema** ai nomi delle relazioni:

```
CREATE TABLE NomeSchema.NomeTabella
```

Per specificare dei valori di default si utilizza questa sintassi:

```
DEFAULT < GenericoValore | USER | NULL >
```

Dove:

- **GenericoValore** rappresenta un valore compatibile con il dominio (una costante o il risultato di un'espressione),
- L'opzione **USER** impone come default l'identificativo dell'utente che esegue il comando di aggiornamento;
- L'opzione **NULL** corrisponde al valore di default di base.

ESEMPIO:

```
Nome VARCHAR(20) DEFAULT USER
```

Vincoli di integrità

I vincoli di integrità sono dei vincoli che possono essere specificati in SQL come parte della creazione di una tabella, e possono essere:

- Vincoli **intra-relazionali** (specificano condizioni sui valori di una singola tabella:):
  - → vincoli di (super)chiave;
  - → vincoli di dominio;
  - → vincoli su singole tuple.
- Vincoli **inter-relazionali**: vincoli di integrità referenziale.

I vincoli intra-relazioni possono essere: (vengono inseriti subito dopo la definizione di un attributo)

- NOT NULL: indica che un attributo non può assumere valori nulli;
- UNIQUE: definisce una (super)chiave:
  - → due righe della tabella non possono avere gli stessi valori sull'attributo (o l'insieme di attributi) cui si applica il vincolo;
  - → viene fatta eccezione per il valore NULL;
- PRIMARY KEY: definisce la chiave primaria, gli attributi su cui è definito il vincolo PRIMARY KEY non possono assumere il valore nullo;
- CHECK (Condizione): esprime un generico vincolo tramite un'espressione che deve essere vera per tutte le tuple della tabella.

Il vincolo **UNIQUE** può essere definito in due modi:

1) Si fa seguire la specifica dell'attributo dalla parola chiave, solo quando il vincolo si applica ad un singolo attributo;
2) Si usa il costrutto **`UNIQUE (ListaAttributi)`** dopo aver definito tutti gli attributi della tabella.

```
Matricola CHAR(6) UNIQUE


CREATE TABLE Impiegato ( Matricola CHAR(6) PRIMARY KEY, Nome VARCHAR(20) NOT
    NULL, Cognome VARCHAR(20) NOT NULL, ....., UNIQUE (Cognome, Nome) )
```

Il vincolo **PRIMARY KEY,** come il vincolo UNIQUE, può essere definito:

1) direttamente su un singolo attributo;
2) specificando la lista di attributi a cui si applica il vincolo.

```
Matricola CHAR(6) PRIMARY KEY


Nome VARCHAR(20) , Cognome VARCHAR(20), .....,
PRIMARY KEY (Cognome, Nome)
```

In una tabella è possibile **specificare più chiavi UNIQUE ma una sola PRIMARY KEY.**

I vincoli CHECK che **coinvolgono un singolo attributo** possono essere specificati subito dopo la definizione dell'attributo:

```
Voto INTEGER CHECK ( Voto >= 18 AND Voto <= 30 )
```

I vincoli CHECK che **coinvolgono più attributi** devono essere specificati dopo aver definito tutti gli attributi:

```
CREATE TABLE Impiegato ( Matricola CHAR(6) PRIMARY KEY, ..........., Stipendio
DECIMAL(10,2) DEFAULT 0, Premio DECIMAL(10,2) DEFAULT 0,
CHECK (Stipendio > Premio) )
```

È' possibile assegnare un nome ai vincoli usando la parola chiave CONSTRAINT, ed è possibile aggiungere/rimuovere constraint alle tabelle:

```
CONSTRAINT StipOk CHECK (Stipendio > Premio)
```

I vincoli inter-relazionali coinvolgono più tabelle, i più diffusi e significativi sono i **vincoli di integrità referenziale**. Vengono definiti in SQL tramite i costrutti: FOREIGN KEY (chiave esterna) e REFERENCES.

Dopo aver specificato tutti gli attributi della tabella, si dichiarano le chiavi esterne come segue:

```
FOREIGN KEY(ListaAttributi) REFERENCES TabellaRiferita [
(ListaAttributiRiferiti) ]
```

Il vincolo impone che, per ogni tupla della tabella corrente (o interna), i valori assunti sugli attributi specificati (ListaAttributi), se diversi dal valore NULL, **siano presenti in una tupla della tabella riferita (o esterna) tra i valori dei corrispondenti attributi** (ListaAttributiRiferiti).

ESEMPIO:

```
CREATE TABLE Impiegato ( Nome VARCHAR(20) NOT NULL, Cognome VARCHAR(20) NOT
NULL, ……,
FOREIGN KEY(Nome, Cognome) REFERENCES Anagrafica(Nome, Cognome) )
```

L'insieme degli attributi riferiti **deve essere soggetto ad un vincolo di chiave** (UNIQUE o PRIMARY KEY). Non è necessario specificare la lista degli attributi riferiti, se questa coincide con la chiave primaria della tabella esterna.

- Se la chiave esterna è costituita da **un solo attributo**, il vincolo può essere definito subito dopo la specifica dell'attributo;
- Se la chiave esterna è costituita da **un più di attributo,** il vincolo deve essere definito alla fine, dopo la lista degli attributi.

```
Dipartimento VARCHAR (15) REFERENCES Dipartimento(NomeDip)


CREATE TABLE Infrazioni (
Codice CHAR(5) PRIMARY KEY,
Data DATE NOT NULL,
Vigile INTEGER NOT NULL REFERENCES Vigili(Matricola),
Provincia CHAR(2),
Numero CHAR(6),
FOREIGN KEY(Provincia, Numero) REFERENCES Auto(Provincia, Numero)
)
```

Violazione dei vincoli

Generalmente, quando il sistema **rileva una violazione di un vincolo di integrità**, il comando di aggiornamento viene rifiutato e viene segnalato l'errore all'utente. In particolare, **i vincoli di integrità referenziale possono essere violati in caso di**:

- inserimento o modifica di una tupla nella tabella referente;
- cancellazione o modifica di una tupla nella tabella riferita.

In quest'ultimo caso (e solo in esso!), è possibile scegliere quale reazione adottare in caso di violazione. La **politica di reazione viene specificata immediatamente dopo il vincolo**, con la seguente sintassi:

```
ON < DELETE | UPDATE >
< CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION >
```

Le politiche di reazione sono:

- CASCADE: in caso di modifica di una tupla nella tabella riferita (**ON UPDATE**), i nuovi valori vengono riportati sulle corrispondenti tuple della tabella referente. In caso di cancellazione (**ON DELETE**), le corrispondenti tuple della tabella referente vengono a loro volta cancellate;
- SET NULL: agli attributi referenti viene assegnato il valore NULL al posto del valore modificato (**ON UPDATE**) , ovvero cancellato (**ON DELETE**) nella tabella riferita;
- SET DEFAULT: agli attributi referenti viene assegnato il valore di default al posto del valore modificato (**ON UPDATE**) ovvero cancellato (**ON DELETE**) nella tabella riferita;
- RESTRICT: l'azione di modifica/cancellazione **non viene consentita**. Il controllo viene effettuato prima dell'esecuzione dell'azione;
- NO ACTION: simile a RESTRICT, l'azione di modifica/cancellazione non viene consentita, ma il controllo viene fatto dopo aver tentato di eseguire l'azione. È l'opzione di default.

```
CREATE TABLE Impiegato (
Matricola CHAR(6) PRIMARY KEY,
Nome VARCHAR(20) NOT NULL,
Cognome VARCHAR(20) NOT NULL,
Dipartimento VARCHAR(15) REFERENCES Dipartimento(NomeDip),
FOREIGN KEY(Nome,Cognome) REFERENCES Anagrafica(Nome,Cognome) ON DELETE
RESTRICT ON UPDATE CASCADE )
```

```
CREATE TABLE Relatore (
CodStudente Char(6) PRIMARY KEY,
CodDocente Char(6) NOT NULL,
FOREIGN KEY (CodStudente) REFERENCES Studente
ON DELETE CASCADE
ON UPDATE CASCADE,
FOREIGN KEY (CodDocente) REFERENCES Docente
ON DELETE RESTRICT
ON UPDATE CASCADE )
```

Modifica degli schemi

Per modificare gli schemi esistono due comandi:

- DROP: permette di rimuovere componenti di una base di dati;
- ALTER: permette di modificare domini e schemi di tabelle.

La **sintassi** del comando DROP è la seguente:

```
DROP < SCHEMA | DOMAIN | TABLE | VIEW | ASSERTION >
NomeElemento
[ RESTRICT | CASCADE ]
```

L'opzione RESTRINCT fa in modo che il comando non venga eseguito se l'oggetto specificato non è vuoto o se ci sono altri elementi che dipendono da esso (è **l'opzione di default**).

```
DROP TABLE Dipartimento RESTRICT
```

La tabella non viene rimossa se possiede delle righe o se compare nella definizione di qualche altra tabella o vista.

L'opzione CASCADE fa in modo che l'oggetto specificato venga comunque rimosso, assieme agli elementi in esso contenuti o da esso dipendenti.

```
DROP SCHEMA AZIENDA CASCADE
```

Lo schema viene eliminato con tutte le tabelle e gli oggetti che ne fanno parte.

La **sintassi** del comando ALTER per la **modifica di una tabella** è la seguente:

```
ALTER TABLE NomeTabella OpzioniDiModifica
```

Quando invece si vuole **modificare una colonna di una tabella** si utilizza:

```
ALTER TABLE NomeTabella
ALTER [ COLUMN ] NomeAttributo
< SET DEFAULT NuovoDefault | DROP DEFAULT >
```

ESEMPIO:

```
ALTER TABLE Impiegato ALTER Stipendio DROP DEFAULT
ALTER TABLE Progetto ALTER Budget SET DEFAULT 1000
```

Il commando ALTER può essere utilizzato anche per **aggiungere una colonna**:

```
ALTER TABLE NomeTabella ADD [ COLUMN ] DefAttributo
ALTER TABLE Dipartimento ADD COLUMN NumUffici NUMERIC(4)
```

Quando si aggiunge una colonna, **è necessario immettere un valore per la nuova colonna in ciascuna tupla della tabella**. Ciò può essere fatto specificando un **valore di default** o utilizzando il comando **UPDATE**.

Se non viene specificato un valore di default, la nuova colonna avrà il **valore NULL** in tutte le tuple della tabella (in questo caso non è consentito il vincolo NOT NULL).

Il commando ALTER può essere utilizzato anche per **rimuovere una colonna**:

```
ALTER TABLE NomeTabella
DROP [ COLUMN ] NomeAttributo [ RESTRICT | CASCADE ]
```

Quando si elimina una colonna ci sono due opzioni di eliminazione:

1) CASCADE: tutti i vincoli e le viste che si riferiscono alla colonna vengono automaticamente eliminati dallo schema;
2) RESTRICT: il comando ha successo solo se non vi sono elementi dello schema che fanno riferimento alla colonna.

Infine, il comando ALTER può aggiungere/rimuovere dei vincoli:

```
ALTER TABLE Esame ADD CONSTRAINT VotoOk CHECK (Voto >= 18 AND Voto <=30)

ALTER TABLE Impiegato DROP CONSTRAINT StipOk CASCADE
```

Quando si definisce un nuovo vincolo, questo deve essere soddisfatto dai dati già presenti, se l'istanza contiene delle violazioni per il nuovo vincolo, **l'inserimento viene rifiutato.**

Anche per i vincoli, le opzioni di eliminazione RESTRICT e CASCADE hanno lo stesso significato visto in precedenza.

Perché un **vincolo possa essere rimosso**, ad esso **deve essere stato associato un nome** quando è stato specificato.

La selezione

L'istruzione SELECT ha questa sintassi:

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

Dove **ListaAttributi** può essere uno o più attributi delle tabelle elencate (ovvero espressioni definite su tali attributi), **ListaTabelle** può essere un'unica tabella o più tabelle/viste e la **Condizione** è una combinazione booleana di predicati semplici.

La SELECT svolge il **prodotto cartesiano delle tabelle elencate**, e da quest'ultimo vengono selezionate le tuple che soddisfano la condizione espressa nella clausola WHERE e, per ciascuna tupla selezionata, si considerano solo gli attributi specificati in `ListaAttributi`.

ESEMPI:

```
SELECT Matricola, Cognome FROM Studenti WHERE Anno = 2

SELECT Codice FROM Corsi WHERE Docente = 'D4'
```

In **assenza di una condizione di selezione** (clausola WHERE) **tutte le tuple della tabella specificata nella clausola FROM contribuiscono al risultato.**

Per **specificare tutti gli attributi** di una tabella si utilizza il *:

```
SELECT * FROM Esami WHERE Voto>= 24 AND Voto<= 28
```

Come abbiamo detto prima la clausola WHERE ammette come argomento una combinazione booleana di predicati semplici, ciascuno predicato può contenere uno dei seguenti operatori:

- Operatori **relazionali**: dove l'espressione è costruita a partire dai valori degli attributi (nel caso più semplice è rappresentata dal nome di un attributo):
  - `=, <>, >, >=, <=;`
- Operatori di **range**: [NOT] BETWEEN, ci permette di trovare i valori compresi in un determinato intervallo:

```
SELECT * FROM Esami WHERE Voto BETWEEN 24 AND 28
```

- Operatori di **set:** [NOT] IN, ci permette di selezionare solo le tuple che appartengono al gruppo definito dopo IN.

```
SELECT * FROM Esami WHERE Voto IN (29, 30, 33)
```

- Operatori quantificati: ANY e ALL.

```
Esami con voto pari a 29, 30 o 33:
SELECT * FROM Esami WHERE Voto = ANY (29, 30, 33)
```

La sintassi di PostgreSQL richiede **array expression**: `ANY('{29,30,33}')`.

```
Esami con voto diverso da 29, 30 o 33:
SELECT * FROM esami WHERE Voto <> ALL (29, 30, 33)
```

- Operatore di confronto fra stringhe: [NOT] LIKE *stringa,* dove stringa può contenere:
  - _ → carattere arbitrario;
  - % → stringa arbitraria, anche vuota.

```
Studenti il cui nome inizia con 'A' e termina con 'o':
SELECT * FROM Studenti WHERE Nome LIKE 'A%o'
```

```
Studenti che hanno 'r' come terza lettera del cognome
SELECT * FROM Studenti WHERE Cognome LIKE '_ _ r%'
```

- Operatore di confront con NULL: IS [NOT] NULL

```
Studenti con l'attributo CittàResidenza non specificato:
SELECT * FROM Studenti WHERE CittàResidenza IS NULL
```

Due o più predicati possono essere combinati con gli operatori logici **AND** e **OR**. La sintassi non definisce una relazione di precedenza tra gli operatori AND e OR. Se l'interrogazione richiede l'uso di entrambi gli operatori, conviene quindi esplicitare l'ordine di valutazione mediante parentesi.

Eliminazione dei duplicati

SQL **non elimina automaticamente i duplicati principalmente per ragioni di efficienza** (l'eliminazione dei duplicati è un'operazione costosa), inoltre in alcuni casi l'utente può voler mantenere i duplicati nel risultato dell'interrogazione (specie quando si usano funzioni di aggregazione).

Eventuali duplicati possono essere eliminati dal risultato di un'interrogazione:

```
SELECT [ DISTINCT | ALL ] ..... FROM .....
```

Dove con DISTINCT si specifica che i duplicati vengono eliminati, e con ALL invece i duplicati vengono mantenuti (è l'opzione di default).

Ordinamento del risultato

SQL permette di specificare un ordinamento delle righe del risultato di un'interrogazione:

```
SELECT …… FROM …… WHERE …… ORDER BY ListaOrder
```

Dove `ListaOrder` è uno o più attributi specificati tramite il nome o la posizione nell'argomento della SELECT.

Le **righe vengono ordinate in base al primo attributo** di ListaOrder, a parità di valore di questo attributo, si considerano i valori degli attributi successivi, in sequenza. Per ciascun attributo di ordinamento, si possono specificare le opzioni:

- DESC: ordinamento decrescente;
- ASC: ordinamento crescente (opzione di default).

ESEMPIO:

Esami del corso 'C1' ordinati in senso decrescente rispetto al voto espresso in centesimi e, a parità di voto, in senso crescente rispetto alla matricola:

```
SELECT Studente, Data, (100*Voto)/30
FROM Esami WHERE Corso = 'C1'
ORDER BY 3 DESC, Studente
```

Dove il 3 indica il terzo argomento della SELECT.

Per utilizzare il DESC nel secondo argomento della ORDER BY si fa così:

```
SELECT codP,nome, categoria
FROM prodotti.inventario JOIN prodotti.prodotto ON prodotto = codp
ORDER BY categoria, prezzo DESC
```

Ridenominazione

È possibile assegnare un Alias agli attributi/espressioni specificati come argomento della SELECT:

```
SELECT Attributo1 AS A1 , ……, Attributok AS Ak FROM ……
```

ESEMPIO:

```
SELECT Cognome, Nome, CittàResidenza AS Città FROM Studenti WHERE Anno = 1
```

È' inoltre possibile assegnare un **Alias alle tabelle** specificate nella clausola FROM:

```
SELECT …… FROM Tabella1 [ AS ] T1 , ……, Tabellan [ AS ] Tn
```

ESEMPIO:

```
SELECT S.Cognome, S.Nome, FROM Studenti S WHERE S.Anno = 1
```

L'operatore punto si usa per **indicare la tabella da cui viene estratto un attributo**, permette di distinguere due o più attributi con lo stesso nome contenuti in tabelle diverse.

Operatore JOIN

Negli esempi precedenti abbiamo visto come selezionare dati da una tabella vediamo ora come si possono combinare informazioni contenute in tabelle diverse. Per farlo si utilizza il prodotto cartesiano.

- **T1 x T2** → vengono combinate, in tutti i modi possibili, le tuple di T1 e quelle di T2

T1

A	B
a	2
b	4

T2

X	Y
2	e
3	f

T1 x T2 →

A	B	X	Y
a	2	2	e
a	2	3	f
b	4	2	e
b	4	3	f

In un'interrogazione SQL, il prodotto cartesiano si esprime riportando le tabelle interessate nella clausola FROM:

```
SELECT ..... FROM T1, T2
```

Di fatto, un prodotto cartesiano ha senso solo se è seguito da opportune **condizioni di selezione,** in questo caso prende il nome di theta-join. In un'interrogazione SQL, il join si esprime riportando:

- nella clausola FROM le tabelle interessate;
- nella clausola WHERE le condizioni di join.

ESEMPIO:

Cognome e nome degli studenti che hanno sostenuto l'esame del corso C3:

```
SELECT Cognome, Nome
FROM Studenti, Esami
WHERE Matricola = Studente AND Corso = 'C3'
```

È' anche possibile esprimere le **operazioni di join in modo esplicito nella clausola FROM:**

```
Tabella1 JOIN Tabella2 ON Condizione
```

ESEMPI:

```
SELECT Cognome, Nome
FROM Studenti JOIN Esami
ON Matricola = Studente WHERE Corso = 'C3'
```

Coppie di studenti residenti nella stessa città:

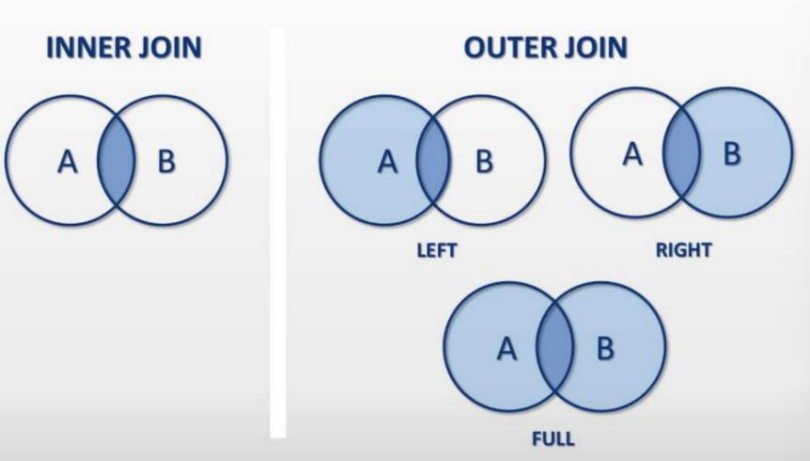
```
SELECT S1.Matricola, S2.Matricola
FROM Studenti S1 JOIN Studenti S2
ON (S1.CittàResid = S2.CittàResid
AND S1.Matricola <> S2.Matricola)
```

Con la stessa sintassi è possibile distinguere join interni ed esterni:

```
Tabella1 [ INNER | LEFT | RIGHT | FULL ] JOIN Tabella2 ON Condizione
```

Il **default è il join interno,** quindi la parola INNER può essere omessa.

Nell risultato dell'INNER JOIN **non si ha traccia delle tuple del primo operando che non hanno corrispondenza con il secondo operando** (e viceversa). È' possibile estendere grazie all'OUTER JOIN, con valori nulli, le tuple che verrebbero tagliate fuori da un join interno.



L'OUTER JOIN può essere:

- LEFT JOIN: mantiene tutte le tuple di T1, estendendole con valori nulli se necessario;
- RIGHT JOIN: mantiene tutte le tuple di T2, estendendole con valori nulli se necessario;
- FULL JOIN: mantiene tutte le tuple di entrambe le tabelle, estendendole con valori nulli se necessario.

Nella clausola FROM **è possibile esprimere più di un'operazione di join**, ciascuna operazione sarà caratterizzata da una specifica condizione di join (espressa nella clausola WHERE o, se si usa la sintassi esplicita per il join, nella clausola ON):

```
Matricola e cognome degli studenti che hanno sostenuto l'esame di Analisi:
SELECT Matricola, Cognome
FROM Studenti JOIN Esami ON Matricola = Studente
JOIN Corsi ON Corso = Codice
WHERE Corsi.Nome = 'Analisi'
```

Questo viene chiamato un join multiplo.

Un alltra variante di JOIN è quello naturale:

```
Tabella1 NATURAL JOIN Tabella2
```

Dove viene richiesta l'uguaglianza dei valori degli attributi che hanno lo stesso nome, oppure:

```
Tabella1 JOIN Tabella2 USING (…)
```

Dove viene richiesta l'uguaglianza dei valori degli attributi specificati nella clausola USING.

ESEMPI:

```
SELECT NomeDip FROM
Impiegati NATURAL JOIN Dipartimenti
WHERE Nome = 'Mario' AND Cognome = 'Rossi'

SELECT NomeDip FROM
Impiegati JOIN Dipartimenti USING (CodiceDip)
WHERE Nome = 'Mario' AND Cognome = 'Rossi'
```

Interrogazioni nidificate

Un'interrogazione viene detta nidificata (o innestata) se la sua **condizione è formulata usando il risultato di un'altra interrogazione** (sub-query).

```
attributo OP-REL (sub-query)
```

Esistono diversi operatori:

- Operatori **quantificati**:

- ANY: restituisce TRUE se almeno uno dei valori restituiti dalla sub-query rende vero il confronto

Cognome e nome degli studenti che hanno sostenuto l'esame del corso C3:

```
SELECT Cognome, Nome
FROM Studenti
WHERE Matricola = ANY (SELECT Studente
FROM Esami
WHERE Corso = 'C3')
```

L'interrogazione selezione le righe di Studenti per cui il valore dell'attributo Matricola è uguale ad almeno uno dei valori dell'attributo Studente nelle righe di Esami.

Questa stessa operazione era possibile anche con l'utilizzo del JOIN;

- ALL: restituisce TRUE solo se tutti i valori restituiti dalla sub-query rendono vero il confronto

Studenti con anno di corso più basso:

```
SELECT *
FROM Studenti
WHERE Anno <= ALL (SELECT Anno
FROM Studenti)
```

In questo caso il DBMS esegue la query interna trovando tutti gli anni degli studenti, poi esegue la SELECT esterna trovando le righe in cui l'anno è minore o uguale degli anni trovati dall'altra query.

Non è in generale corretto il confronto tra un attributo e il risultato di un'interrogazione, perché avremo il confronto fra un valore atomico ed un insieme di valori, per questo motivo occorre specificare dopo l'operazione (=,<, ..) ANY oppure ALL.

- Operatore di **SET**:

- IN: restituisce TRUE se il valore dell'attributo **è contenuto** nell'insieme di valori restituiti dalla sub-query, equivale a: **`attributo = ANY`** `(sub-query)` **:**

Cognome e nome degli studenti che hanno sostenuto l'esame del corso C3:

```
SELECT Cognome, Nome
FROM Studenti
WHERE Matricola IN (SELECT Studente
FROM Esami WHERE Corso = 'C3')
```

- NOT IN: restituisce TRUE se il valore dell'attributo **NON è contenuto** nell'insieme di valori restituiti dalla sub-query, equivale a: **`attributo <> ALL (sub-query)`**:

```
Cognome e nome degli studenti che NON hanno sostenuto alcun esame:

SELECT Cognome, Nome
FROM Studenti
WHERE Matricola NOT IN (SELECT Studente
FROM Esami)
```

  - Costruttore di tuple: permette di **confrontare il risultato della sub-query con una tupla** (anziché un singolo attributo), è rappresentato da una **coppia di parentesi tonde** che racchiudono la lista di attributi della tupla:

```
Studenti che hanno lo stesso nome e lo stesso cognome di un docente:

SELECT *
FROM Studenti
WHERE (Nome, Cognome) IN (SELECT Nome, Cognome
FROM Docenti)
```

  - Quantificatore **esistenziale**:
- EXISTS: ha valore TRUE se e solo se l'insieme di valori restituiti dalla sub-query è **NON vuoto**:

```
Cognome e nome degli studenti che hanno sostenuto l'esame del corso C3:

SELECT Cognome, Nome
FROM Studenti S
WHERE EXISTS (SELECT *
FROM Esami E
WHERE E.Corso = 'C3' AND E.Studente = S.Matricola)
```

- NOT EXISTS: ha valore TRUE se e solo se l'insieme di valori restituiti dalla sub-query **è vuoto**:

```
Cognome e nome degli studenti che NON hanno sostenuto l'esame del corso C3:
SELECT Cognome, Nome
FROM Studenti S
WHERE NOT EXISTS (SELECT *
FROM Esami E
WHERE E.Corso = 'C3' AND E.Studente = S.Matricola)
```

In generale, **una sub-query viene detta** correlata se la sua condizione è formulata usando relazioni e/o variabili (alias) definite nella query esterna.

Estrarre gli impiegati che hanno degli omonimi (stesso nome e cognome, ma diverso codice fiscale):

```
SELECT *
FROM Impiegato I (alias)
WHERE EXISTS (SELECT *
FROM Impiegato I1
WHERE I.Nome=I1.Nome and
I.Cognome = I1.Cognome and
I.CodFiscale <> I1.CodFiscale)
```

Si può osservare che l'interrogazione nidificata utilizza una variabile di range definita nell' interrogazione più esterna. Perciò, in questo caso, **per ogni riga esaminata nell'ambito dell' interrogazione esterna, si deve valutare l'interrogazione nidificata**. Nell'esempio vengono considerate una ad una le righe della variabile I; per ciascuna di queste righe, viene eseguita l'interrogazione nidificata che restituisce o meno l'insieme vuoto a seconda che vi siano o meno degli omonimi della persona considerata.

La **sub-query può fare riferimento a relazioni/variabili definite nella query esterna, ma non vale il viceversa**, ovvero una query non può fare riferimento a relazioni/variabili definite in blocchi più interni (o paralleli).

Quando una **stessa relazione compare sia nella query esterna che nella sub-query**, è indispensabile definire degli alias per distinguere le diverse occorrenze della relazione:

Per ogni corso, selezionare la matricola degli studenti che hanno superato l'esame col voto più alto:

```
SELECT E1.Corso, E1.Studente
FROM Esami E1
WHERE E1.Voto >= ALL (SELECT E2.Voto
FROM Esami E2
WHERE E2.Corso = E1.Corso)
```

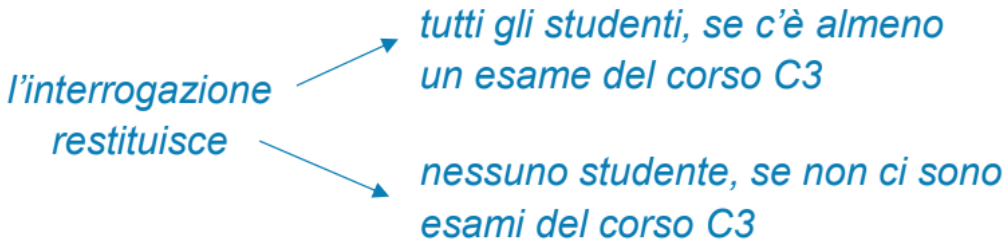
È' indispensabile **utilizzare sub-query correlate quando si formulano interrogazioni con il quantificatore esistenziale** [NOT] EXISTS.

Studenti che hanno sostenuto l'esame del corso C3:

```
SELECT *
FROM Studenti S
WHERE EXISTS (SELECT *
FROM Esami E
WHERE E.Corso = 'C3' AND E.Studente = S.Matricola)
```

In assenza di correlazione:

```
SELECT *
FROM Studenti S
WHERE EXISTS (SELECT *
FROM Esami E
WHERE E.Corso = 'C3')
```



È' possibile formulare interrogazioni con:

- più livelli di nidificazione;
- più sub-query allo stesso livello.

Studenti che hanno sostenuto l'esame di un corso del docente D1:

```
SELECT * FROM Studenti WHERE Matricola IN
(SELECT Studente FROM Esami WHERE Corso IN
(SELECT Codice FROM Corsi WHERE Docente = 'D1') )
```

Studenti che hanno sostenuto l'esame del corso C3 e non hanno sostenuto l'esame del corso C4:

```
SELECT *
FROM Studenti
WHERE Matricola IN (SELECT Studente
FROM Esami
WHERE Corso = 'C3')
AND Matricola NOT IN (SELECT Studente
FROM Esami
WHERE Corso = 'C4')
```

Le query innestate formulate con gli operatori IN, ANY, EXISTS + sub-query correlata, **sono riducibili a query semplici equivalenti.**

Invece gli operatori NOT IN, ALL, NOT EXITS + sub-query correlate **non sono riducibili**.

Funzioni aggregate

SQL mette a disposizione una serie di funzioni per:

- contare le tuple che soddisfano una condizione (COUNT);
- elaborare i valori di un attributo (MAX, MIN, AVG, SUM)

Tali funzioni costituiscono una delle più importanti estensioni di SQL rispetto all'algebra relazionale.

La funzione COUNT ammette come argomenti:

- * conteggio del numero di righe;
- [ALL | DISTINCT]: con ALL (che è l'operatore di default) si fa il conteggio del numero di valori non nulli dell'attributo, con DISTINCT si fa il conteggio del numero di valori non nulli e distinti dell'attributo.

```
Numero di studenti memorizzati:
SELECT COUNT(*) FROM Studenti

Numero di studenti che hanno sostenuto almeno un esame:

SELECT COUNT (DISTINCT Studente) FROM Esami
```

Le funzioni SUM, AVG, MAX, MIN ammettono come argomento un attributo o un'espressione, eventualmente preceduti da:

- ALL: trascura i NULL (default);
- DISTINCT: trascura i NULL ed elimina i duplicati;

Con MAX e MIN l'uso di DISTINCT o ALL non ha effetto sul risultato.

In particolare:

- SUM e AVG ammettono come argomento **solo espressioni che rappresentano valori numerici** o intervalli di tempo;
- MAX e MIN richiedono semplicemente che **sull'argomento sia definito un ordinamento** (si possono applicare anche a stringhe).

```
Voto medio degli esami sostenuti dallo studente con matricola 'XXXXX':

SELECT AVG(Voto) FROM Esami WHERE Studente = 'XXXXX'

Stipendi minimo e massimo fra quelli di tutti gli impiegati:

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

Non **è lecita la presenza contemporanea, tra gli argomenti della SELECT, di nomi di attributi e funzioni aggregate** (eccetto che per le interrogazioni con la clausola GROUP BY). La seguente interrogazione, ad esempio, non è corretta:

```
SELECT Studente, MAX(Voto) FROM Esami
```

Raggruppamento

In un'interrogazione SQL, è **possibile raggruppare le tuple che possiedono gli stessi valori di determinati attributi**:

```
SELECT ..... FROM ..... WHERE ..... GROUP BY ListaGroup
```

Il risultato della SELECT sarà un unico record per ciascun gruppo.

Possono comparire come argomento della SELECT solo:

- uno o più attributi specificati in ListaGroup (che assumono lo stesso valore per ogni tupla del gruppo);
- **funzioni aggregate** (che vengono valutate sui gruppi e forniscono un unico valore per ciascuno di essi).

ESEMPI:

Considerati i noleggi effettuati, vogliamo ora ottenere il prezzo medio giornaliero e il prezzo giornaliero massimo **per ciascuna auto**:

Targa	PrezzoMedio	PrezzoMax
AA111BB	€ 6,375	€ 7
BB222CC	€ 5,00	€ 5
CC333DD	€ 27,50	€ 40
DD444EE	€ 8,00	€ 10
EE555FF	€ 5,25	€ 7

Cosa cambia rispetto agli esempi precedenti? Le due funzioni di aggregazione, AVG e MAX, vengono applicate più volte.

Perché visualizziamo la targa? Perché è il campo che ci permette di distinguere le varie auto e le funzioni di aggregazione vanno applicate a ciascuno dei gruppi di record con valore comune in questo campo.

```
SELECT Dipart, Max(Età) FROM Impiegati GROUP BY Dipart
```

Codice	Nome	Età	Dipart
IMP1	Anna	29	DIP1
IMP2	Alice	43	DIP1
IMP3	Barbara	27	DIP2
IMP4	Bruno	23	DIP2
IPM5	Carlo	31	DIP2
IPM6	Camilla	44	DIP3

Dipart	Max(Età)
DIP1	43
DIP2	31
DIP3	44

```
SELECT Dipart, Max(Età) FROM Impiegati WHERE Età > 28 GROUP BY Dipartimento
```

Codice	Nome	Età	Dipart
IMP1	Anna	29	DIP1
IMP2	Alice	43	DIP1
IMP3	Barbara	27	DIP2
IMP4	Bruno	23	DIP2
IPM5	Carlo	31	DIP2
IPM6	Camilla	44	DIP3

Dipart	Max(Età)
DIP1	43
DIP2	31
DIP3	44

Selezionare il **voto massimo e minimo** ottenuto da ogni studente:

```
SELECT Studente, Max(Voto), Min(Voto) FROM Esami GROUP BY Studente
```

Per ogni **corso**, visualizzare il **codice**, il **nome** e il **numero di studenti** che hanno sostenuto il relativo esame (inclusi i corsi per cui non ci sono esami):

```
SELECT Codice, Nome, COUNT(Studente) FROM Corsi LEFT JOIN Esami ON Codice =
Corso GROUP BY Codice, Nome
```

I gruppi ottenuti tramite la clausola GROUP BY possono essere ulteriormente selezionati tramite la clausola HAVING: `GROUP BY ..... HAVING Condizione.` Solo i gruppi che soddisfano la condizione vengono inclusi nel risultato.

La condizione della clausola HAVING può essere formulata utilizzando:

- uno o più attributi specificati in ListaGroup;
- funzioni aggregate.

```
SELECT Dipart, Max(Età) FROM Impiegati WHERE Età > 28 GROUP BY Dipartimento
HAVING Count(*) > 1
```

Codice	Nome	Età	Dipart
IMP1	Anna	29	DIP1
IMP2	Alice	43	DIP1
IMP3	Barbara	27	DIP2
IMP4	Bruno	23	DIP2
IPM5	Carlo	31	DIP2
IPM6	Camilla	44	DIP3

Dipart	Max(Età)
DIP1	43

Numero di esami superati per ogni voto compreso tra 27 e 30:

```
SELECT Voto, COUNT(*) FROM Esami GROUP BY Voto HAVING Voto BETWEEN 27 AND 30
```

Media dei voti per ogni esame sostenuto da più di due studenti:

```
SELECT Corso, AVG(Voto) FROM Esami GROUP BY Corso HAVING COUNT (*) > 2
```

Matricola degli studenti che hanno sostenuto almeno due esami:

```
SELECT Studente FROM Esami GROUP BY Studente HAVING COUNT (Corso) >= 2
```

Matricola degli studenti che hanno sostenuto almeno due esami con lo stesso voto:

```
SELECT Studente FROM Esami GROUP BY Studente, Voto HAVING COUNT (Corso) >= 2
```

La condizione della clausola HAVING può essere formulata utilizzando il risultato di un'altra SELECT, questo porta quindi ad una **interrogazione nidificata**.

Matricola dello studente che ha sostenuto il maggior numero di esami:

```
SELECT Studente FROM Esami
GROUP BY Studente
HAVING COUNT(*) >= ALL (SELECT COUNT(*)
FROM Esami
GROUP BY Studente)
```

Viste

Una vista è una tabella virtuale, ciò vuol dire che non esiste necessariamente in forma fisica, ma viene definita, tramite un'interrogazione, a partire da altre tabelle (tabelle base o viste precedentemente definite).

```
CREATE VIEW NomeVista [ (ListaAttributi) ] AS SELECT .........
```

L'interrogazione non viene eseguita all'atto di definizione della vista, ma quando la vista viene effettivamente utilizzata.

ESEMPIO:

Vista contenente i dati degli studenti del primo e del secondo anno :

```
CREATE VIEW StudentiBiennio AS SELECT * FROM Studenti WHERE Anno <= 2
```

Vista contenente i dati degli impiegati che hanno uno stipendio inferiore a 1000 :

```
CREATE VIEW ImpiegatiPoveri (Matricola, Cognome, Nome, Stipendio)
AS SELECT Matricola, Cognome, Nome, Stipendio
FROM Impiegati
WHERE Stipendio < 1000
```

Le viste sono utili per:

- semplificare / modularizzare interrogazioni complesse;
- far fronte a modifiche dello schema che richiederebbero cambiamenti in applicazioni esterne;
- gestione del controllo degli accessi.

USO DELLE VISTE NELLE INTERROGAZIONI:

Determinare quale è il dipartimento che spende il massimo in stipendi:

Impiegati (Matricola, Nome, Cognome, Dipart, Ufficio, Stipendio)

Possibile Soluzione:

```
SELECT Dipart FROM Impiegati GROUP BY Dipart HAVING SUM(Stipendio) >= ALL
(SELECT SUM(Stipendio) FROM Impiegati GROUP BY Dipart)
```

Con le view:

```
CREATE VIEW BudgetStipendi (Dip, TotStipendi) AS SELECT Dipart, SUM(Stipendio)
FROM Impiegati GROUP BY Dipart
```

Dalla vista BudgetStipendi possiamo estrarre il dipartimento in cui la somma degli stipendi è più elevata:

```
SELECT Dip FROM BudgetStipendi WHERE TotStipendi = ( SELECT MAX(TotStipendi)
FROM BudgetStipendi )
```

ESEMPIO:

Determinare il numero medio di uffici per dipartimento, dobbiamo:

- contare il numero di uffici in ogni dipartimento;
- calcolare la media dei valori ottenuti.

```
CREATE VIEW DipartUffici (Dip, NroUffici) AS SELECT Dipart, COUNT (distinct
                    Ufficio) FROM Impiegati GROUP BY Dipart


                    SELECT AVG(NroUffici) FROM DipartUffici
```