Basi di Dati - Linguaggio SQL

Introduzione

SQL è il linguaggio più diffuso per le Basi di Dati relazionali.

SQL è un linguaggio **dichiarativo** basato sul **Calcolo relazionale su Ennuple** e **Algebra Relazionale.**

Il linguaggio comprende:

- **DML (Data Manipulation Language):**
  ricerche e/o modifiche interattive, come l'interrogazione o le query
- **DDL (Data Definition Language)**
  definizione e amministrazione delle Basi di Dati

DML:

1) Comando SELECT:

Il comando SELECT del linguaggio SQL mi **permette di effettuare delle interrogazioni in una tabella**.
Gli attributi sono la lista delle colonne da visualizzare, separate tra loro da una virgola. Per vederle tutte basta indicare l'asterisco *.

```
SELECT attributo1, attributo2 FROM tabelle
// In questo modo vado a selezionare gli attributi delle tabelle
```

```
SELECT * FROM tabelle
// Così vado a selezionare tutti gli attributi delle tabelle senza condizioni
```

2) Comando FROM:

Indica la tabella o le tabelle in cui deve operare il comando SELECT.

3) Comando WHERE:

Indica la condizione logica con cui vengon filtrate le colonne. Tra parentesi quadre [] si può indicare un parametro da chiedere prima della selezione. Nel caso vi siano due o più tabelle selezionate le si può mettere in relazione indicando in un'eguaglianza i campi voluti.

Ad esempio, data la seguente tabella denominata 'studenti' strutturata in tre campi o colonne (nome, cognome e classe) e contenente 5 record (righe):

Tabella Studenti

nome	cognome	classe
Mario	Rossi	3
Giuseppe	Bianchi	3
Luca	Verdi	2
Mario	Russo	2
Paolo	Rossi	1

record

campi

```
SELECT *
FROM Studenti
WHERE classe='3'
// In questo modo andiamo a selezionare tutti gli studenti che hanno classe = 3
```

Mario	Rossi	3
Giuseppe	Bianchi	3

Questo sarà il risultato ottenuto.

Qualificazione: notazione con il punto e alias:

Per alias si intende l'associazione delle relazioni ad un identificatore.

Essenzialmente viene utilizzato se si sta operando su più copie della stessa relazione.

Es: Generare una tabella che contenga cognomi e matricole degli studenti e dei loro tutor:

```
SELECT s.Cognome, s.Matricola, t.Cognome, t.Matricola
FROM Studenti s, Studenti t
WHERE s.Tutor = t.Matricola
```

La clausola **AS** in SQL serve per rinominare un attributo, quindi assegnarli un alias:

```
SELECT north_east_user_subscriptions AS ne_subs // Rinominato a ne_subs
FROM users
WHERE ne_subs > 5;

// Questo alias value per la durata della query
```

Clausola DISTINCT:

La clausola **Distinct** serve a non ripetere nei risultati della SELECT quelli con lo stesso valore.
Permette di eliminare le ripetizioni dei dati uguali in una query. Si tratta di una clausola **e** non di un comando **SQL**.

```
SELECT DISTINCT City
FROM Station
WHERE ID%2=0 // Oppure MOD(ID,2)=0

// In questo caso vado a selezionare le città senza ripetizioni
```

Funzioni matematiche/aritmetiche:

Ovviamente SQL dispone di diverse funzioni che permettono di fare operazioni aritmetiche, come ad esempio la funzione SUM, MIN, MAX, COUNT, AVG.

```
// In questo modo andiamo a contare ciò che si trova all'interno di Qualcosa:
SELECT COUNT(*)
FROM Qualcosa

// Supponiamo di dover sommare tutti gli attributi Sium della tabella Qualcosa:
```

```
SELECT SUM(Sium)
FROM Qualcosa

// Troviamo anno di nascita minimo, massimo e medio degli Studenti:
SELECT MIN(Nascita), MAX(Nascita), AVG(Nascita)
FROM Studenti

// C'è anche la funzione per controllare il resto di una divisione:
MOD(Sium,2) // -> ovviamente va utilizzata in un certo modo, come nello scorso
            // paragrafo
```

Operazione JOIN:

In SQL l’operazione JOIN combina le righe di due o più tabelle in base ai valori contenuti nelle colonne.

Pensiam per esempio di avere due tabelle. Da queste due tabelle devo ricavare una tabella in cui per ciascuna persona compaia il reddito e il nome del padre.

Persone

Nome	Reddito
Mario	25000
Giovanni	15000
Paolo	30000
Giuseppe	20000
Maria	35000

Paternità

Padre	Figlio
Giovanni	Maria
Mario	Paolo
Mario	Giuseppe

```
SELECT Nome, Reddito, Padre
FROM Persone JOIN Paternità
ON Nome = Figlio // Con ON vado a dare una condizione su cui effettuare il JOIN

// Praticamente sono andato a unire li dove il nome delle persone corrisponde
// al nome del figlio nella tabella Paternità.
```

E questa è la tabella generata:

JOIN Nome=Figlio

Nome	Reddito	Padre
Paolo	30000	Mario
Giuseppe	20000	Mario
Maria	35000	Giovanni

Diversi tipi di JOIN:

Ovviamente l'operazione JOIN può essere fatta in modo diverso a seconda di ciò che viene richiesto. C'è il NATURAL JOIN, CROSS JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN:

```
// Il cross join viene utilizzato per fare una combinazione di tutte le righe
// della prima tabella con tutte le righe della seconda:
// Quindi se ho nella prima tabella A,B e nella seconda a,b otterrò:
// (A,a),(A,b),(B,a),(B,b).

// Ora utilizzeremo la consegna dello scorso paragrafo per fare esempi:

// Natural Join:
SELECT * FROM Persone NATURAL JOIN Paternità;
// Nel caso di join naturale non va specificata la condizione di join, il
// natural join infatti verifica automaticamente se esistono colonne con lo
// stesso nome nelle due tabelle.

// Left Join:
SELECT Nome, Reddito, Padre
FROM Persone LEFT JOIN Paternità
ON Nome = Figlio
// Voglio mostrare il nome, il reddito e, se presente, il nome del padre della
// persona. Quindi per non escludere le righe della prima tabella senza
// corrispondenza nella seconda utilizzo il LEFT JOIN. Ottenendo di fatto le
// corrispondenze e le non corrispondenze.

// Right Join:
// Mantiene tutte le righe della seconda tabella a cui aggiunge le righe della
// prima che soddisfano la condizione di join.

// Full Join:
// E' la combinazione del LEFT e RIGHT OUTER JOIN. Mantiene tutte le tuple di
// entrambe le tabelle, estendendole con valori nulli se necessario.
```

LEFT JOIN Nome=Figlio

Nome	Reddito	Padre
Mario	25000	
Giovanni	15000	
Paolo	30000	Mario
Giuseppe	20000	Mario
Maria	35000	Giovanni

Ecco la tabella generata dal Left Join.

Clausola ORDER BY:

In SQL è possibile ordinare i dati in uscita aggiungendo la clausola Order By.

Questa clausola compie un ordinamento crescente (ASC) o decrescente (DESC) delle righe della tabella in base al valore contenuto nell'attributo.

```
// Ordinamento per età crescente degli alunni:
SELECT *
FROM alunni
ORDER BY età ASC
```

Operatori insiemistici:

SQL comprende operatore **insiemistici** per combinare i risultati di tabelle con colonne di ugual nome e ugual tipo. Questi operatori sono UNION, INTERSECT e EXCEPT.

- **Union**: Produce l'unione dei dati presenti in due colonne (attributi) di una o più tabelle, senza i valori duplicati.
- **Intersect**: Produce l'intersezione dei dati presenti in due colonne (attributi) di una o più tabelle, ossia i valori presenti in entrambe le colonne, escludendo i valori duplicati.
- **Except**: Calcola la differenza dei valori tra due colonne (attributi) in due o più colonne, ossia i valori della prima colonna senza quelli della seconda, senza i valori duplicati.

Esempio: trovare nome, cognome e matricola degli studenti di Venezia e di quelli che hanno preso più di 28 a qualche esame:

```
SELECT Nome, Cognome, Matricola
FROM Studenti
WHERE Provincia = 'VE'
UNION
SELECT Nome, Cognome, Matricola
FROM Studenti JOIN Esami ON Matricola=Candidato
WHERE Voto>28;
```

Ora qualche esempio dell’utilizzo di UNION, INTERSECT e EXCEPT:

```
// UNION: voglio raggruppare tutti i nomi degli studenti e dei tutor
SELECT Nome, Cognome
FROM Studenti
UNION
SELECT Nome, Cognome
FROM TUTOR

// INTERSECT: voglio trovare tra studenti e tutor la gente con lo stesso nome
SELECT Nome
FROM Studenti
INTERSECT
SELECT Nome
FROM Tutor

// EXCEPT: immaginiamo di avere due colonne:
// La Colonna_1 contiene (A,B,E,F)
// La Colonna_2 contiene (A,B,C,D)
SELECT Colonna_1
FROM Tabella
EXCEPT
SELECT Colonna_2
FROM Tabella
// Il risultato di questa operazione sarà una tabella con (E,F), ovvero gli
// unici elementi che non sono anche nella seconda colonna.
```

Valori NULL:

È possibile che alcune query richiedano di trovare qualcosa con valore nullo, quindi per esempio:

```
// Trovare gli studenti che non hanno tutor:
SELECT *
FROM Studenti
WHERE Tutor IS NULL

// L'espressione sopra quindi ritornerà i valori che soddisfano la condizione:
WHERE Tutor = NULL
```

Altri operatori per i valori NULL possono essere:

- Espresione1 **IS NOT DISTINCT FROM** Espressione2
  è vero se i due valori sono diversi, o uno solo sei due è NULL;
  è falso quando i due valori sono uguali anche nel caso in cui sono entrambi uguali a NULL.
- **COALESCE**(Espressione_1, … , Espressione_n)
  Viene usato per trasformare un valore NULL in un valore non nullo. Valuta le espressioni in sequenza, da sinistra verso destra. Viene restituito il primo valore trovato diverso da NULL. L'operatire ritorna NULL se tutte le espressioni hanno valore NULL.

ps: è possibile specificare il simbolo diverso in sql con <>. Stessa cosa che scrivere != in C/C++.

Operatore BETWEEN:

Viene utilizzata per verificare valori che vanno da una certa soglia a un'altra:

```
SELECT *
FROM Studenti
WHERE Matricola BETWEEN 12000 AND 7200;
```

Operatore LIKE:

Operatore utilizzato per il pattern matching:

```
WHERE CustomerName LIKE 'a%'
--Finds any values that start with "a"
WHERE CustomerName LIKE '%a'
--Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'
--Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'
--Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a__%'
--Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'
--Finds any values that start with "a" and ends with "o"
```

SELECT annidate:

È possibile specificare select **annidate**, inserendo nel campo WHERE una condizione che usa una select (che a sua volta può contenere sottoselect, e così via…).

Grazie alle select annidate si può:

- Eseguire **confronti con l'insieme** di valori ritornati dalla sottoselect (sia quando questo è un singoletto, sia quando contiene più elementi).
- Verificare la **presenza/assenza** di valori dati nell'insieme ritornato dalla sottoselect.
- Verificare se l'insieme di valori ritornato dalla sottoselect è o meno **vuoto**.

```
// Esempio: Trovare gli studenti che vivono nella stessa provincia dello
// studente con matricola 892300, escluso lo studente stesso.

SELECT *
FROM Studenti
WHERE (Matricola <> '892300') AND
      Provincia = (SELECT Provincia
                   FROM Studenti
                   WHERE Matricola = '892300')
```

Quantificazione esistenziale EXISTS:

La condizione EXISTS può essere usata nel WHERE per verificare che una tabella (o una colonna o quello che è) non sia vuota.

```
// Esempio: La query studenti con almeno un voto > 27
SELECT *
FROM Studenti
WHERE EXISTS (SELECT *
              FROM Esami e
              WHERE e.Candidato = s.Matricola
                    AND e.Voto > 27
```

Quantificazione esistenziale ANY:

In SQL l'operatore ANY è soddisfatto se almeno un confronto è vero. Si usa nelle select nidificate.

Esempio: Scriviamo una query per trovare nome, cognome e residenza delle persone che abitano in Lombardia:

```
SELECT Nome, Cognome, Residenza
FROM Persone
WHERE Residenza = ANY (SELECT Città
                       FROM Italia
                       WHERE Regione = 'Lombardia')
```

L'abbinamento = ANY equivale a dire "trova almeno un confronto uguale".

Quantificazione esistenziale IN:

Nel linguaggio SQL l'operatore IN si usa per verificare l'appartenenza di un valore a un insieme. La sua negazione è l'operatore NOT IN.

Si tratta di un operatore logico relazione. L'operatore IN mi permette di specificare una lista di valori come clausole di ricerca in un'unica query.

```
// Sintassi:
SELECT [campi]
FROM [tabella]
WHERE [campo] IN ([clausole di ricerca])
```

Esempio: Se volessi interrogare un database per cercare tutte le persone che si chiamano Mario o Maria potrei scrivere nel sequente modo:

```
SELECT *
FROM Persone
WHERE Nome IN ('Mario','Maria')

// La stessa cosa può essere fatta anche senza IN:
SELECT *
FROM Persone
WHERE (Nome='Mario') OR (Nome='Maria')
```

Quantificazione universale:

In SQL non c’è l’operatore generale esplicito FOR ALL. Quindi si usa l’equivalenza logica con il NOT:

Esempio: Selezionare gli studenti che hanno preso solo 30:

```
SELECT *
FROM Studenti s
WHERE NOT EXISTS (SELECT *
                  FROM Esami e
                  WHERE e.Candidato = s.Matricola AND e.Voto <> 30)
```

Quindi invece di andare a dire: FOR ALL studenti che hanno preso 30, vado a dire gli studenti che non hanno preso 30 negato (<>).

Quantificazione universale ALL:

È disponibile un operatore duale rispetto ad ANY, ovvero ALL.

In SQL la clausola ALL è soddisfatta se tutti i confronti sono veri. Si utilizza nelle select nidificate.

Esempio: Pensiamo di avere due tabelle Persone e Italia. Scrivo una select nidificata per trovare le persone che non risiedono in Lombardia.

```
SELECT Nome, Cognome, Residenza
FROM Persone
WHERE Residenza <> ALL(SELECT Città
                       FROM Italia
                       WHERE Regione = 'Lombardia')
```

Clausola GROUP BY:

la clausola GROUP BY divide una tabella in gruppi in base al valore di uno o più attributi. Si utilizza nelle interrogazioni.

La clausola GROUP BY va indicata dopo FROM o WHERE se presente.

L'ordine degli attributi determina la gerarchia del raggruppamento nella tabella Sql.

> Esempio: Voglio calcolare la media dell'età degli studenti per ogni città di provenienza:

```
SELECT Città, AVG(Età)
FROM Studenti
GROUP BY Città
```

Esempio 2: Per includere una condizione di selezione aggiungo la clausola WHERE dopo FROM.

```
// Mettiamo di voler escludere Napoli:
SELECT Città, AVG(Età)
FROM Studenti
WHERE Città <> 'Napoli'
GROUP BY Città
```

Clausola HAVING:

Nel linguaggio SQL la clausola Having mi permette di aggiungere una condizione di tipo aggregato sui gruppi creati con GROUP BY.

```
// Sintassi:
SELECT ...
GROUP BY attributo
HAVING condizione
```

L'interrogazione SELECT visualizza soltanto i dati aggregati che soddisfano la condizione indicata in HAVING.

Esempio: Ho la tabella Studenti, voglio selezionare le città in cui la media dell'età degli studenti è inferiore a 21 anni:

```
SELECT Città, AVG(Età)
FROM Studenti
GROUP BY Città
HAVING AVG(Età)<21
```

Comando INSERT:

Nel linguaggio SQL l'inserimento dei dati in una tabella si compie tramite l'istruzione INSERT INTO.

La tabella deve essere già stata creata nel database.

Gli attributi e i valori devono essere separati tra loro da una virgola dentro le rispettive liste.

Esempio:

Persone

Nome	Cognome	Età	Residenza
Mario	Rossi	21	Roma
Giovanni	Bianchi	22	Milano
Paolo	Rossi	19	Roma
Giuseppe	Verdi	23	Bergamo
Antonio	Ferrari	19	Napoli

Per aggiungere un nuovo record nel database scrivo:

```
INSERT INTO Persone
VALUES ('Ciano','Trevisan',78,'Venessia')

// Oppure:
INSERT INTO Persone(Età,Nome,Cognome,Residenza)
VALUES (78,'Ciano','Trevisan','Venessia')

// È anche possibile insierire solo determinati valori, ma ciò renderà NULL
// quelli non inseriti:
INSERT INTO Persone(Nome,Cognome)
VALUES ('Ciano','Trevisan')
```

Per aggiungere nella tabella dei dati presi da un’altra tabella:
Esempio:

```
INSERT INTO Persone(SELECT Nome, Cognome, Età, Città
                    FROM Studenti)
```

Comando DELETE:

Per cancellare un record o un insieme di records con il linguaggio SQL uso l'istruzione DELETE FROM. La sintassi è la seguente:

```
DELETE FROM Tabella
WHERE colonna = valore
```

Esempio: Per eliminare dalla tabella gli alunni con undici anni scrivo:

```
DELETE FROM Alunni
WHERE Età = 11
```

Per svuotare la tabella senza eliminarla dal database utilizzo il comando TRUNCATE:

```
TRUNCATE TABLE NomeTabella

// Equivale a scrivere:

DELETE FROM NomeTabella
```

Se invece vuoi distruggere la tabella, cancellarla proprio dall’esistenza si fa DROP TABLE nometabella.

Comando UPDATE:

I dati nella tabella SQL possono essere modificati tramite il comando UPDATE.

Questo comando mi consente di modificare il valore dei record (righe) in uno o più campi (colonne) della tabella.

Nella clausola SET è indicato l'attributo e il valore da registrare dopo il simbolo uguale.

Esempio:

```
UPDATE Persone
SET regione = 'Lazio'
WHERE residenza = "Roma"
```

Esempio: Ho la tabella persone. Per aumentare di un anno il campo Età di tutte le righe scrivo il comando UPDATE senza condizioni:

```
UPDATE Persone
SET Età = Età + 1
```

Esempio: Devo scrivere "Firenze" in tutte le righe in cui il cognome è "Rossi":

```
UPDATE Persone
SET Residenza = 'Firenze'
WHERE Cognome = 'Rossi'
```

Per modificare una singola riga devo indicare una condizione in grado di identificarla in modo univoco.

L’ideale per fare ciò è avere una Primary Key, ovvero una chiave che identifica un attributi come unico.

Esempio: Aggiornare lo stato civile di una persone segnata nella tabella con un id = 1111:

```
UPDATE Persone
SET StatoCivile = 'Morto'
WHERE (id = 1111)
```

Esercizi SQL:

1]

Trovare il nome e cognome degli atleti italiani o svedesi che hanno vinto una medaglia d’argento nella disciplina Short track.

```
SELECT a.Nazione, a.Nome, a.Cognome, m.Anno-a.AnnoNascita AS Età
FROM Atleti a NATURAL JOIN Medaglie m
WHERE m.Anno-a.AnnoNascita = (SELECT MIN(m1.Anno-a1.AnnoNascita)
                              FROM Atleti a1 NATURAL JOIN Medaglia m1
                              WHERE a1.Nazione = a.Nazione)
```

2]

Trovare il numero di medaglie vinte dall’Italia nell’anno 2016.

```
SELECT COUNT(*)
FROM Atleta a NATURAL JOIN Medaglie m
WHERE a.Nazione = ’Italia’ AND m.Anno=2016
```

3]

Trovare gli atleti che hanno vinto almeno una medaglia per tipo e restituire il nome e cognome, la nazione e il numero di medaglie totale vinte.

```
SELECT DISTINCT a.IdAtleta, a.Nome, a.Cognome, a.Nazione
FROM Atleti a JOIN Medaglie m1 USING (IdAtleta)
              JOIN Medaglie m2 USING (IdAtleta)
WHERE m1.Sport<>m2.Sport
```

4]

Per ogni atleta, trovare il numero di edizioni in cui ha vinto una medaglia nello Sport ”Sci di fondo” e restituire anche il nome e il cognome dell’atleta. Se l’atleta non ha vinto alcuna medaglia in tale sport, per quell’atleta si deve restituire 0.

```
SELECT a.Nome, a.Cognome, COUNT(DISTINCT m.Anno) AS NumEdizioni
FROM Atleta LEFT JOIN Medaglie m ON a.IdAtleta=m.IdAtleta AND m.Sport=’Sci’
WHERE
GROUP BY a.IdAtleta, a.Nome, a.Cognome
```

5]

Trovare gli atleti che hanno vinto meno di 5 medaglie e restituire il nome, cognome e la nazione di tali atleti.

```
SELECT
FROM Atleti a NATURAL LEFT JOIN Medaglie m
WHERE a.Nome, a.Cognome, a.Nazione
GROUP BY a.IdAtleta, a.Nome, a.Cognome, a.Nazione
HAVING COUNT(m.Codice) < 5
```

6]

Trovare nome e anno di nascita dei dipendenti del progetto Master.

```
SELECT d.Nome, d.AnnoNascita
FROM Dipendenti d NATURAL JOIN Staff s JOIN Progetti p USING(Cod)
WHERE p.Nome=’Master’
```

7]

Trovare nome e data degli esami per studenti che hanno superato l’esame di BD con 30.

```
SELECT Nome, Data
FROM Studenti JOIN Esami ON Matricola = Candidato
WHERE Materia = 'DB' AND Voto = 30
```

DDL:

SQL viene usato, oltre che per l’interrogazione delle query, anche per la definizione di basi di dati.

Creazione di uno schema:

Uno schema può essere creato con:

```
CREATE SCHEMA Università AUTHORIZATION rossi
// AUTHORIZATION mi dice chi è il proprietario dello schema
```

Uno schema può essere eliminato attraverso il costrutto drop:

```
DROP SCHEMA Nome [ CASCADE | RESTRICT ]
```

```
// Di default se non scriviamo nulla viene lasciato RESTRICT:
DROP SCHEMA NomeDelloSchema RESTRICT

// Per realizzare il dropschema forzato si una CASCADE:
DROP SCHEMA NomeDelloSchema CASCADE
```

Schemi:

Uno schema può contenere varie tabelle delle quali esistono più tipi:

- Tabelle base (base table)
  - ▼ i metadati appartengono allo schema
  - ▼ i dati fisicamente memorizzati
- Viste
  - ▼ i metadati sono presenti nello schema
  - ▼ i dati non sono fisicamente memorizzati

Creazione di una Tabella:

Una tabella base viene creata con il comando **CREATE TABLE**, è un insieme di colonne/attributi per ciascuna delle quali va specificato:

- nome
- tipo di dato, quindi i valori che possono essere assunti
  - ▼ predefinito
  - ▼ definito dall'utente (dominio), costruito con il comando **CREATE DOMAIN**

```
CREATE DOMAIN Voto AS SMALLINT
              CHECK (VALUE <= 30 AND VALUE >= 18)

// Come valore di valutazione posso assumere soltanto un intero
// compreso tra 18 e 30

// Creazioe di una tabella:
CREATE TABLE NomeTabella (
  colonna1 int,
  colonna2 varchar(255),
  colonna3 boolean
);
```

Tipi di dato predefiniti:

Ce ne sono tanti, i più importanti sono:

- tipi **interi:**
  - ▼ **INTEGER, SMALLINT…**
- **valori decimali:**
  - ▼ **NUMERIC(p,s)**
- **valori in virgola mobile:**
  - ▼ **REAL**
- **stringhe di bit:**
  - ▼ **BIT(x), BIT VARYING(x)**
- **booleani:**
  - ▼ **BOOLEAN**
- **stringhe** di caratteri:
  - ▼ **CHAR(x)** oppure **CHARACTER(x)**
  - ▼ **VARCHAR(x)** oppure **CHAR VARYING(x) / CHARACTER VARYING(x)**
- **date** e **ore:**
  - ▼ **DATE, TIME, TIMESTAMP**
- **intervalli temporali:**
  - ▼ **INTERVAL{YEAR, MONTH, DAY, HOUR, MINUTE, SECOND}**

Tipi di dato:

- **SERIAL** serve per scrivere ID sintetici, spesso è utile avere un attributo che genera identificatori. In questo caso SERIAL genera interi. Quando andiamo a definire una colonna SERIAL, creiamo una nuova tabella SEQUENZA, questa nuova tabella è costituita da un’unica riga e contiene il valore appena assegnato.

```
CREATE TABLE tablename (
  colonna1 SERIAL
```

```
);

//equivalente a:

CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
  colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
  ...);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

Vincoli di ennupla:

Per una colonna (di tabella) è possiblie specificare anche determinate cose, come:

- Un eventuale valore di default, con la clausola **DEFAULT**
- Un eventuale vincolo, tipo **NOT NULL**, **CHECK** (<condizione>)

```
CREATE TABLE Studenti (
  Nome varchar(255) NOT NULL,
  Cognome varchar(255) NOT NULL,
  Provincia char(6) DEFAULT 'VE',
  Voto integer CHECK (Voto >= 18 AND Voto <= 30)
);
```

Primary key e Foreign key:

Primary key designa un insieme di attributi come chiave primaria, che non può mai essere nulla ed è univoca per ogni record.

Foreign key designa un insieme di attributi come chiave esterna e un'eventuale azione da compiere (tipo ON UPDATE CASCADE).

Esempio:
• Atleti(IdAtleta, Nome, Cognome, Nazione*, Sesso, AnnoNascita)
  Nazione FK(Nazioni).
• Medaglie(Codice, Tipo, Sport, Disciplina, IdAtleta*, Anno)
  IdAtleta FK(Atleti).
• Nazioni(Nome, Estensione, NumAbitanti).

```
CREATE TABLE Atleti (
  IdAtleta varchar(255) PRIMARY KEY,
  Nome varchar(255),
```

```
  Cognome varchar(255),
  Nazione varchar(255) NOT NULL,
  Sesso varchar(255),
  AnnoNascita integer,
  FOREIGN KEY (Nazione) REFERENCES Nazioni(Nome)
                        ON UPDATE CASCADE
);

CREATE TABLE Medaglie (
  Codice integer PRIMARY KEY,
  Tipo varchar(255),
  Sport varchar(255),
  Disciplina varchar(255),
  IdAtleta varchar(255) NOT NULL,
  Anno integer,
  FOREIGN KEY (IdAtleta) REFERENCES Atleti(IdAtleta)
                         ON UPDATE CASCADE
);

CREATE TABLE Nazioni (
  Nome varchar(255) PRIMARY KEY,
  Estensione varchar(255),
  NumAbitanti integer
);
```