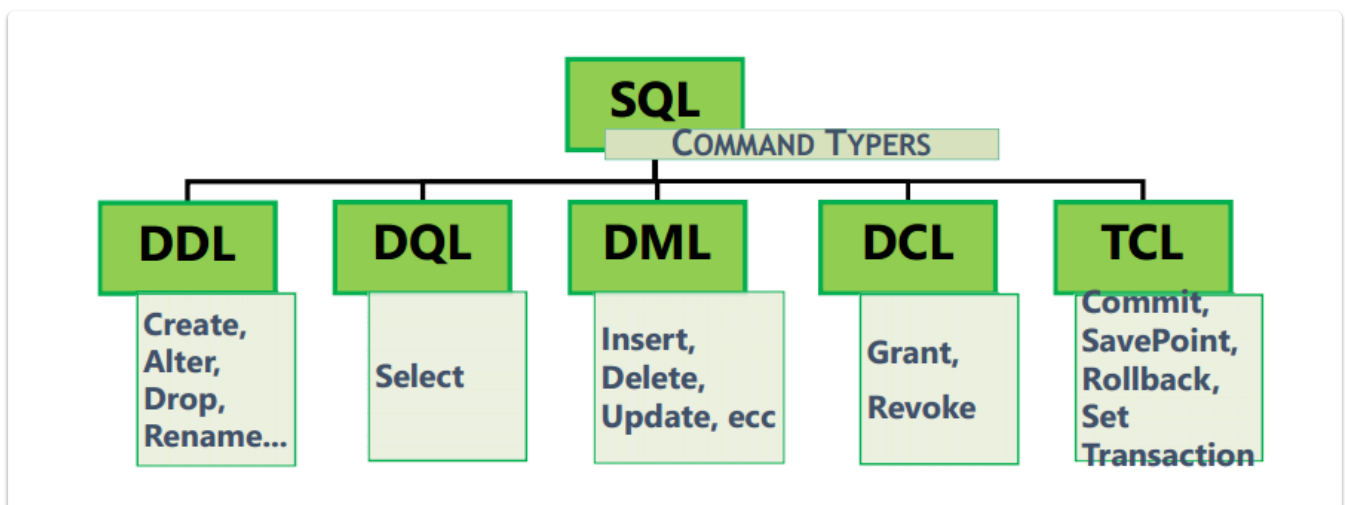


1 Introduzione al linguaggio

Linguaggio **standard** e universale per la gestione dei database relazionali.

È un linguaggio **dichiarativo** quindi l'utente non specifica un insieme di operazioni da compiere per ottenere un risultato ma descrive le proprietà logiche delle informazioni desiderate, lasciando al DBMS il compito di determinare il modo di efficiente per ottenerle. Il linguaggio svolge diversi compiti unificati:

1. **DDL** (*Data Definition Language*): permette di creare, modificare o eliminare oggetti dello schema del database;
2. **DML** (*Data Manipulation Language*): consente di inserire, modificare, eliminare o leggere i dati all'interno di tabelle;
3. **DCL** (*Data Control Language*): consente di fornire o revocare agli utenti i permessi necessari per utilizzare i comandi DML e DDL
4. **DQL** (*Data Query Language*): dedicato alla creazione di query e all'estrazione di informazioni dal DB
5. **TCL** (*Transaction Control Language*): gestisce le transazioni



2 Caratteristiche generali e sintassi

- **Case-Insensitive**: il linguaggio è case-insensitive per quanto riguarda le parole chiave e gli identificatori non quotati (`SELECT` è equivalente a `select`). Gli identificatori quotati (`NomeTabella`) e i valori stringa sono case-sensitive. Per evitare ambiguità si usano sempre nomi in minuscolo
- **Terminazione dei comandi**: le istruzioni SQL terminano con `;`
- **Token**: I comandi SQL sono composti da "token" come parole chiave (es. `SELECT`), identificatori (nomi di tabelle, colonne), letterali (costanti numeriche o stringhe) e caratteri speciali (es. `%`, `_` per `LIKE`, `*` per `SELECT ALL`)
- **Commenti**: `--` per singola riga e `/* ... */` per commenti multi-riga
- **Esecuzione "Tutto o Niente"**: quando si esegue un comando SQL, o tutte le sue operazioni vanno a buon fine o nessuna di esse viene applicata

2.1 Domini elementari

Tipo di Dato SQL Standard	Descrizione
<code>CHAR [(n)]</code>	Stringa di caratteri a lunghezza fissa. Se <code>n</code> è omissso, si assume la lunghezza di 1 carattere.

Tipo di Dato SQL Standard	Descrizione
CHARACTER [(n)]	Equivalente a CHAR [(n)] .
VARCHAR (n)	Stringa di caratteri a lunghezza variabile. n è obbligatorio e specifica la lunghezza massima.
CHARACTER VARYING (n)	Equivalente a VARCHAR (n) .
NUMERIC [(p [, s])]	Valore numerico esatto con parte frazionaria opzionale. p indica la precisione, s la scala. Esempio: numeric(3,1) → da -99.9 a +99.9.
DECIMAL [(p [, s])]	Equivalente a NUMERIC [(p [, s])] .
INTEGER	Per numeri interi.
SMALLINT	Per numeri interi di dimensioni più piccole.
REAL	Per numeri in virgola mobile a precisione singola.
DOUBLE PRECISION	Per numeri in virgola mobile a precisione doppia.
FLOAT [(p)]	Per numeri in virgola mobile, con p che indica la precisione binaria minima richiesta (opzionale).
DATE	Contiene valori di data nel formato aaaa-mm-gg .
TIME [(p)]	Contiene valori di tempo nel formato hh:mm:ss . p (opzionale) indica la precisione per i secondi frazionari.
TIMESTAMP [(p)]	Contiene data e ora nel formato AAAAMMGGhhmmss . p (opzionale) indica la precisione per i secondi frazionari.
INTERVAL t1 [to t2]	rappresenta una durata di tempo (es. giorni, mesi, ore). t1 e t2 indicano le unità temporali comprese nell'intervallo.
BIT [(n)]	Sequenza fissa di n bit. Se n è omissso, si assume la lunghezza di 1 bit. Valori come B'1001' sono ammessi.
BLOB	Binary Large Object: usato per dati binari (immagini, audio, ecc.). Introdotto nello standard SQL:1999.
CLOB	Character Large Object: usato per grandi quantità di testo. Introdotto nello standard SQL:1999.
BOOLEAN	Per valori booleani TRUE , FALSE e NULL . Introdotto nello standard SQL:1999.
BIGINT	Rappresenta numeri interi con ampia gamma, tipicamente a 64 bit. È utilizzato quando i valori interi superano il limite di INTEGER , ad esempio per identificatori univoci o contatori molto grandi.

2.2 DDL - Data Definition Language

2.2.1 Creazione CREATE

```
CREATE SCHEMA [NomeSchema] [[authorization] Autorizzazione] {DefElementoSchema}
CREATE TABLE NomeTabella (NomeAttributo Dominio [ValoreDefault] [Vincoli]...)
CREATE DOMAIN NomeDominio as TipoDiDato [ValoreDefault] [Vincolo]
```

1. AUTHORIZATION Autorizzazione : opzionale, specifica il proprietario dello schema (in PostgreSQL, è l'utente creatore per default)
2. {DefElementoSchema} : è possibile creare direttamente oggetti (es. tabelle, domini) all'interno dello schema

☰ Example

💡 Esempio completo di creazione schema, dominio e tabella in SQL

```
-- 1. Creazione dello schema
CREATE SCHEMA universita;

-- 2. Creazione di un dominio per età valida
CREATE DOMAIN universita.EtaValida AS INT
    DEFAULT 18
    CHECK (VALUE >= 17 AND VALUE <= 120);

-- 3. Creazione di una tabella nello schema "universita"
CREATE TABLE universita.studenti (
    matricola SERIAL PRIMARY KEY,
    nome TEXT NOT NULL,
    cognome TEXT NOT NULL,
    email TEXT UNIQUE,
    eta universita.EtaValida, -- usa il dominio creato
    corso_di_laurea TEXT DEFAULT 'Ingegneria'
);
```

2.2.2 Modifica ALTER

```
ALTER TABLE NomeTabella <
    ALTER COLUMN NomeAttributo <SET DEFAULT NuovoDefault | DROP DEFAULT> |
    -- Modifica di una colonna esistente:
    -- SET DEFAULT imposta un nuovo valore di default per la colonna
    -- DROP DEFAULT rimuove il valore di default assegnato

    ADD CONSTRAINT DefVincolo |
    -- Aggiunge un vincolo alla tabella (es. CHECK, UNIQUE, FOREIGN KEY)

    DROP CONSTRAINT NomeVincolo |
    -- Rimuove un vincolo esistente, identificato dal nome

    ADD COLUMN DefAttributo |
    -- Aggiunge una nuova colonna (attributo) alla tabella

    DROP COLUMN NomeAttributo
    -- Elimina una colonna esistente dalla tabella
>

ALTER DOMAIN NomeDomain <
    SET DEFAULT ValoreDefault |
    -- Imposta un valore di default per il dominio (tipo personalizzato)

    DROP DEFAULT |
    -- Rimuove il valore di default dal dominio








    ADD CONSTRAINT DefVincolo |
    -- Aggiunge un vincolo (tipicamente CHECK) al dominio

    DROP CONSTRAINT NomeVincolo
```

```
-- Rimuove un vincolo precedentemente definito sul dominio  
>
```

Example

Esempio completo di comandi ALTER in SQL

```
--  Aggiunta di una nuova colonna alla tabella "studenti"  
ALTER TABLE universita.studenti  
ADD COLUMN matricolato BOOLEAN DEFAULT TRUE;  
  
--  Modifica del tipo della colonna "corso_di_laurea"  
ALTER TABLE universita.studenti  
ALTER COLUMN corso_di_laurea TYPE VARCHAR(100);  
  
--  Impostazione NOT NULL sulla colonna "email"  
ALTER TABLE universita.studenti  
ALTER COLUMN email SET NOT NULL;  
  
--  Rimozione del vincolo NOT NULL dalla colonna "email"  
ALTER TABLE universita.studenti  
ALTER COLUMN email DROP NOT NULL;  
  
--  Modifica del valore di default del dominio EtaValida  
ALTER DOMAIN universita.EtaValida  
SET DEFAULT 21;  
  
--  Rimozione del valore di default dal dominio  
ALTER DOMAIN universita.EtaValida  
DROP DEFAULT;  
  
--  Aggiunta di un vincolo CHECK al dominio  
ALTER DOMAIN universita.EtaValida  
ADD CONSTRAINT eta_non_minorenne CHECK (VALUE >= 18);  
  
--  Rimozione del vincolo CHECK dal dominio  
ALTER DOMAIN universita.EtaValida  
DROP CONSTRAINT eta_non_minorenne;
```

2.2.3 Cancellazione DROP

```
DROP <SCHEMA | DOMAIN | TABLE | VIEW | ASSERTION> NomeElemento [restrict | cascade]
```

1. **restrict**: **impedisce l'eliminazione** se nel database esistono altri oggetti che dipendono da esso. È il comportamento di default qualora non venga specificato nulla.
2. **cascade**: forza l'eliminazione dell'oggetto specificato e di tutti gli oggetti che dipendono da esso, ricorsivamente. Il sistema eliminerà automaticamente l'oggetto target e qualsiasi altro oggetto direttamente o indirettamente dipendente da esso, senza generare errori

2.2.4 Schema

Comando DDL	Scopo Principale	Sintassi / Dettagli Chiave	Note / Esempi
CREATE SCHEMA	Crea uno schema logico contenente oggetti (tabelle, tipi, ecc.)	CREATE SCHEMA nome_schema [AUTHORIZATION nome_utente]	In PostgreSQL, se omissso, AUTHORIZATION coincide col nome dell'utente.
ALTER SCHEMA	Modifica uno schema esistente		Sintassi non sempre standardizzata, dipende dal DBMS.
DROP SCHEMA	Elimina uno schema e tutti gli oggetti in esso contenuti	DROP SCHEMA nome_schema [CASCADE/RESTRICT]	CASCADE elimina anche gli oggetti contenuti, RESTRICT blocca se presenti oggetti.

2.2.5 📄 Tabelle

Comando DDL	Scopo Principale	Sintassi / Dettagli Chiave	Note / Esempi
CREATE TABLE	Crea una nuova tabella	CREATE TABLE Nome (...)	Struttura fissa delle colonne, righe variabili.
ALTER TABLE	Modifica la struttura della tabella	ALTER TABLE ...	Aggiunge o rimuove colonne, cambia tipi, aggiunge vincoli.
DROP TABLE	Elimina una tabella	DROP TABLE [IF EXISTS] nome [CASCADE/RESTRICT]	Elimina anche indici, trigger, vincoli.

2.2.6 🏠 Domini

Comando DDL	Scopo Principale	Sintassi / Dettagli Chiave	Note / Esempi
CREATE DOMAIN	Definisce un tipo personalizzato	CREATE DOMAIN nome AS tipo [DEFAULT val] [CHECK (...)]	VALUE nella CHECK si riferisce al valore del dominio.
ALTER DOMAIN	Modifica un dominio esistente	ALTER DOMAIN nome ADD CONSTRAINT ... / DROP DEFAULT	Può fallire se dati esistenti non rispettano il nuovo vincolo.
DROP DOMAIN	Elimina un dominio	DROP DOMAIN nome [CASCADE/RESTRICT]	

2.2.7 🔒 Vincoli

Vincolo	Tipo	Descrizione	Esempio SQL
NOT NULL	Intrarelazionale	Impedisce valori nulli in una colonna	nome VARCHAR(50) NOT NULL
UNIQUE	Intrarelazionale	Impone valori unici nella colonna (ma consente NULL)	email VARCHAR(100) UNIQUE
PRIMARY KEY	Intrarelazionale	Identifica univocamente ogni riga; implica NOT NULL e UNIQUE	PRIMARY KEY (id)
CHECK	Intrarelazionale	Impone una condizione booleana sui valori. Si tratta di un vincolo generico.	CHECK (età >= 18)

Vincolo	Tipo	Descrizione	Esempio SQL
FOREIGN KEY	Interrelazionale	Collega una colonna a una colonna primaria di un'altra tabella	FOREIGN KEY (id cliente) REFERENCES clienti(id)
REFERENCES	Interrelazionale	Sintassi alternativa integrata nella definizione di colonna	id cliente INT REFERENCES clienti(id)

Example

```
CREATE TABLE clienti (
    id INT PRIMARY KEY,
    nome VARCHAR(50) NOT NULL
);

CREATE TABLE ordini (
    id INT PRIMARY KEY,
    data DATE NOT NULL,
    id_cliente INT,
    FOREIGN KEY (id_cliente) REFERENCES clienti(id)
);
```

A partire da SQL-2 è possibile introdurre **vincoli generici** mediante la keyword `CHECK`. Tramite tale keyword è comunque possibile ridefinire manualmente i vincoli predefiniti (non è una buona scelta).

Example

```
CREATE TABLE impiegato (
    matricola CHAR(6)
    CHECK (matricola IS NOT NULL AND
    1 = (SELECT COUNT(*)
    FROM impiegato I
    WHERE matricola = I.matricola
    )),
    cognome CHAR(20) CHECK (cognome IS NOT NULL),
    ...
);
```

Ogni vincolo di integrità introdotto tramite `CHECK` (o asserzione) è associato ad una politica di controllo che ne specifica il comportamento in termini di tempistiche di verifica:

- **Immediato** (`IMMEDIATE`): il vincolo è verificato immediatamente dopo ogni modifica;
- **Differito** (`DEFERRED`): il vincolo viene verificato al termine delle transizione;

Il controllo differito si utilizza in quei casi in cui non è possibile costruire uno stato consistente della base di dati con una sola modifica (per esempio se sono presenti vincoli di integrità incrociati).

```
SET CONSTRAINTS [NomeVincoli | ALL] IMMEDIATE | DEFERRED
```

2.2.7.1 Asserzioni

Le asserzioni sono dei **vincoli** definiti direttamente sullo **schema** e che consentono quindi di esprimere condizioni che non sarebbero altrimenti definibili, come per esempio **vincoli che coinvolgono più tabelle**.

```
CREATE ASSERTION NomeAsserzione CHECK(condizione)
```

≡ Example

Imporre la presenza di almeno una riga

```
CREATE ASSERTION AlmenoUnImpiegato
CHECK(1 <= (SELECT COUNT(*) FROM IMPIEGATO))
```

2.2.7.2 Politiche di violazione dei vincoli (ON DELETE/ON UPDATE)

Per tutti i vincoli intra-relazionali, quando il sistema rileva una violazione, il comando viene semplicemente **rifiutato**. Per i vincoli di **integrità referenziale** invece, SQL consente di scegliere altre reazioni da adottare quando viene rilevata una violazione durante un comando di aggiornamento (ON UPDATE) o cancellazione (ON DELETE):

- CASCADE : **Propaga l'operazione** (elimina o aggiorna anche i record figli) verso la tabella interna:
 - **Es:** se elimino un cliente, vengono eliminati anche i suoi ordini.
- SET NULL : Imposta a NULL il valore nella tabella interna.
 - **Es:** se un cliente viene cancellato, `id_cliente` negli ordini diventa NULL .
- SET DEFAULT : Imposta il valore di **default** nella tabella interna.
- NO ACTION : **Impedisce l'operazione** se viola il vincolo (comportamento predefinito). L'azione di modifica/cancellazione non è consentita;
- RESTRICT : Come NO ACTION , ma **controllato immediatamente** (meno usato in SQL standard).

La politica di reazione si specifica subito dopo al vincolo:





≡ Example

```
CREATE TABLE ordini (
    id INT PRIMARY KEY,
    id_cliente INT,
    FOREIGN KEY (id_cliente) REFERENCES clienti(id) ON UPDATE CASCADE ON DELETE SET
    NULL
);
```

È possibile associare politiche diverse ai diversi eventi: è possibile per esempio usare una politica CASCADE per la modifica ed una politica SET NULL per le cancellazioni. Le reazioni alle violazioni possono generare una **reazione a catena**, qualora la tabella interna compaia a sua volta come tabella esterna in un altro vincolo di integrità.

2.2.7.3 Creazione tabelle con vincoli incrociati

La **creazione di due tabelle con vincoli di integrità referenziale incrociati** (cioè ognuna ha una FOREIGN KEY verso l'altra) in SQL è un problema **circolare** che va risolto con una **sequenza precisa di passi**.

Fase	Azione
 Crea tabelle	Senza vincoli FK incrociati
 Aggiungi vincoli	Dopo con <code>ALTER TABLE ... ADD CONSTRAINT ... FOREIGN KEY</code>
 Inserisci dati	In ordine strategico, con <code>NULL</code> temporanei se necessario
 Aggiorna dati	Per completare la referenza incrociata

```
-- CREAZIONE TABELLE SENZA VINCOLI
CREATE TABLE dipartimento (
    nome CHAR(4) PRIMARY KEY,
    citta VARCHAR(20),
    responsabile CHAR(16) -- FK da aggiungere dopo
);

CREATE TABLE impiegato (
    CF CHAR(16) PRIMARY KEY,
    cognome VARCHAR(20),
    nome VARCHAR(20),
    dipartimento CHAR(4) -- FK da aggiungere dopo
);
```

```
-- INSERIMENTO DEI VINCOLI
ALTER TABLE dipartimento
ADD CONSTRAINT fk_dip_responsabile
FOREIGN KEY (responsabile) REFERENCES impiegato(CF)
ON DELETE RESTRICT ON UPDATE RESTRICT;

ALTER TABLE impiegato
ADD CONSTRAINT fk_imp_dipartimento
FOREIGN KEY (dipartimento) REFERENCES dipartimento(nome)
ON DELETE RESTRICT ON UPDATE RESTRICT;
```

```
-- INSERIMENTO DEI VALORI ED AGGIORNAMENTO

-- 1. Inserisco un impiegato SENZA dipartimento
INSERT INTO impiegato (CF, cognome, nome, dipartimento)
VALUES ('AAAA1111BBBB2222', 'Rossi', 'Mario', NULL);

-- 2. Inserisco il dipartimento con quel responsabile
INSERT INTO dipartimento (nome, citta, responsabile)
VALUES ('INFO', 'Salerno', 'AAAA1111BBBB2222');

-- 3. Ora aggiorno l'impiegato per assegnargli il dipartimento
UPDATE impiegato
SET dipartimento = 'INFO'
WHERE CF = 'AAAA1111BBBB2222';
```

2.3 DML - Data Manipulation Language

Il DML fornisce i comandi per **inserire**, **modificare**, **eliminare** o leggere i dati all'interno delle tabelle di un database, assumendo che la struttura dei dati sia già stata definita tramite il Data Definition Language (DDL).

Comando DML	Scopo	Sintassi base	Note / Esempi
INSERT INTO	Inserisce una o più righe in una tabella	INSERT INTO tabella (col1, col2) VALUES (val1, val2)	Anche da query: INSERT INTO ... SELECT ... ; RETURNING mostra le righe inserite.
UPDATE	Modifica dati esistenti	UPDATE tabella SET col = valore WHERE condizione	Omesso WHERE = tutte le righe. Valori nuovi possono dipendere da quelli vecchi.
DELETE FROM	Elimina righe	DELETE FROM tabella WHERE condizione	Omesso WHERE = elimina tutte le righe. Attiva CASCADE se previsto da vincoli.

2.3.1 Inserimento

```
INSERT INTO NomeTabella [ListaAttributi] <VALUES (ListaValori) | SelectSQL>
```

- VALUES consente l'inserimento di singole righe, andando a specificare manualmente il valore di ogni attributo. Se non viene specificato alcun valore si assegna quello di default o in caso NULL.

Warning

- L'inserimento può anche essere effettuato a partire da una base di dati esistente, realizzando una nidificata.
- La corrispondenza tra gli attributi della tabella e i valori da inserire è data dall'ordine in cui compaiono i termini nella definizione della tabella.

Example

```
-- Inserimento con specifica delle colonne
INSERT INTO studenti (matricola, nome, eta)
VALUES (1001, 'Lucia', 21);

-- Inserimento in tutte le colonne (ordine esatto richiesto)
INSERT INTO studenti
VALUES (1002, 'Giorgio', 22, 'Matematica');

-- Inserimento tramite SELECT
INSERT INTO ProdottiMilanesi
(SELECT Codice, Descrizione
FROM Prodotto
WHERE LuogoProd = 'Milano')
```

2.3.2 Aggiornamento

```
UPDATE NomeTabella
SET Attributo = <Espressione | SelectSQL | NULL | DEFAULT>
{, Attributo = <Espressione | SelectSQL | NULL | DEFAULT>}
[WHERE Condizione]
```

Il comando consente di aggiornare uno o più attributi delle righe di `NomeTabella` che soddisfano l'eventuale Condizione .

2.3.3 🗑 Cancellazione

```
DELETE FROM NomeTabella [WHERE Condizione]
```

⚠ Warning

Se la condizione non é specificata, il comando cancella tutte le righe della tabella!

☰ Example

```
-- Eliminazione semplice
DELETE FROM Dipartimento
WHERE Nome = 'Produzione'

-- Eliminazione con nidificata
DELETE FROM Dipartimento WHERE Nome NOT IN (
    SELECT Dipart
    FROM Impiegato
)
```

⚠ Differenza tra DELETE e DROP

- `DELETE FROM Dipartimento` : elimina tutte le righe dalla tabella `Dipartimento` ma lo schema della base di dati rimane **immutato**. Il comando agisce solamente sulle istanze, la tabella rimane ma é vuota.
- `DROP TABLE Dipartimento CASCADE` : ha lo stesso effetto del comando precedente ma **modifica** anche lo **schema** della base di dati, eliminando non solo la tabella `Dipartimento` , ma anche tutte le viste e tabelle che fanno riferimento ad essa.

2.4 DQL - Data Query Language

Essendo `SQL` un linguaggio dichiarativo, le query vengono poste specificando l'obiettivo dell'interrogazione e non il modo in cui ottenerlo.

2.4.1 Ordine di esecuzione

1. `FROM` : prima clausola ad essere valutata. Il sistema identifica le tabelle specificate e crea un **prodotto cartesiano** (o una sua **derivazione** tramite `JOIN`).
 - `ON` (per `JOIN`) : Se sono presenti clausole `JOIN` (come `INNER JOIN` , `LEFT JOIN` , `RIGHT JOIN` , `FULL OUTER JOIN`), le condizioni specificate nella clausola `ON` vengono **valutate prima** dell'esecuzione del join stesso. Questo filtra le righe dal prodotto cartesiano o dalla tabella derivante dal join, mantenendo solo quelle che soddisfano la condizione di join.
2. `WHERE` : Dopo che il set di risultati é stato formato dalla clausola `FROM` (e `ON` per i join), la clausola `WHERE` **filtra** ulteriormente le righe. Le condizioni in `WHERE` sono applicate a **ciascuna riga risultante**, e solo le righe che soddisfano la condizione vengono mantenute. Queste condizioni vengono **valutate successivamente all'esecuzione del join**.

3. GROUP BY : **raggruppa le righe** che hanno gli stessi valori nelle colonne specificate. Prepara i dati per le funzioni di aggregazione;
4. HAVING : è simile alla WHERE ma **opera sui gruppi** creati dalla clausola GROUP BY . Filtra i gruppi in base a condizioni che spesso coinvolgono funzioni aggregate (es. COUNT , SUM , AVG , MIN , MAX). Un gruppo viene incluso nel risultato solo se soddisfa la condizione HAVING.
5. SELECT : clausola che **proietta gli attributi** desiderati. Dopo che le righe sono state filtrate e raggruppate, la clausola SELECT determina quali **colonne** (o **espressioni** calcolate) verranno **incluse nel set di risultati finale**. Qualsiasi funzione aggregata (COUNT , SUM , ecc.) viene calcolata in questa fase sui gruppi definiti da GROUP BY . L'uso di DISTINCT per eliminare le tuple duplicate viene applicato qui, dopo tutte le altre operazioni.
6. ORDER BY : Infine, la clausola ORDER BY ordina il set di risultati in base a una o più colonne, in ordine crescente (ASC) o decrescente (DESC). Questa è l'ultima operazione logica eseguita, determinando la presentazione finale dei dati.

2.4.2 Query Semplici

Il comando principale è SELECT .

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

1. SELECT (target list): Specifica gli attributi (colonne) che si desidera visualizzare nel risultato. È possibile anche utilizzare espressioni o funzioni aggregate (COUNT , MIN , MAX , AVG , SUM) in questa clausola. L'uso di DISTINCT evita la restituzione di tuple duplicate;
2. FROM : Indica le tabelle da cui recuperare i dati;
3. WHERE : Contiene le condizioni che devono essere soddisfatte dalle righe per essere incluse nel risultato. Le condizioni possono essere semplici confronti o espressioni booleane complesse, inclusi operatori come LIKE . Le query SELECT selezionano, tra le righe del prodotto cartesiano delle tabelle nella clausola FROM , quelle che soddisfano le condizioni WHERE

Più precisamente:

```
SELECT AttrEspr [[AS] Alias]{, AttrEspr [[AS] Alias]}  
from Tabella[[AS] Alias]{,Tabella [[AS] Alias]}
```

1. AttrEspr sta per "espressione di attributo, ovvero:
 - un nome di colonna (nome)
 - un'espressione (anno + 1 , COUNT(*) , UPPER(cognome))
2. [[AS] Alias] : è opzionale e serve a rinominare la colonna nel risultato. In molti DBMS non serve neanche specificare AS

Example

```
SELECT s.nome, c.nome AS corso  
FROM studenti AS s, corsi AS c  
WHERE s.corso_id = c.id;
```

L'interrogazione SQL seleziona, tra le righe che appartengono al prodotto cartesiano delle tabelle elencate nella clausola FROM , quelle che soddisfano le condizioni espresse nella clausola WHERE .

2.4.2.1 Gestione dei duplicati

A differenza dell'algebra relazionale, dove le tabelle sono viste come relazioni matematiche, in SQL, le tabelle possono contenere tuple non uniche. La rimozione dei duplicati è un'operazione molto costosa e spesso non necessaria e per questo si è stabilito di permettere la presenza di duplicati in SQL.

La rimozione viene effettuata mediante la keyword DISTINCT, da porre subito dopo la SELECT:

Example

```
SELECT DISTINCT corso_di_laurea
FROM studenti;
```

2.4.3 JOIN interni ed esterni

Utilizzando la sintassi precedentemente presentata ed inserendo più tabelle all'interno della clausola WHERE, il DBMS effettua automaticamente il prodotto cartesiano tra le tabelle (Join Implicit).

Esiste tuttavia un modo più efficiente per combinare le tuple di tabelle differenti: il JOIN. Tale operatore consente di correlare dati presenti in tabelle diverse.

```
SELECT AttrEspr [[AS] Alias]{, AttrEspr [[AS] Alias]}
from Tabella[[AS] Alias]
{[TipoJoin] JOIN Tabella [[AS] Alias] ON CondizioneJoin}
[WHERE AltraCondizione]
```

1. { [TipoJoin] JOIN Tabella [[AS] Alias] ON CondizioneJoin } : specifica uno o più JOIN.
2. TipoJoin può essere:
 - LEFT [OUTER] : restituisce come risultato il join interno esteso con le righe della tabella di sinistra per le quali non esiste una corrispondente riga nella tabella di destra.
 - RIGHT [OUTER] : restituisce come risultato il join interno esteso con le righe della tabella di destra per le quali non esiste una corrispondente riga nella tabella di sinistra.
 - FULL [OUTER] : esegue il join interno esteso con le righe che fanno parte di una o entrambe le tabelle coinvolte;
 - INNER (Theta Join dell'algebra relazionale): restituisce solo le righe che hanno corrispondenza in entrambe le tabelle. Solitamente non si usa.
3. CondizioneJoin è la condizione di collegamento tra le tabelle

<u>Tipo di JOIN</u>	<u>Righe incluse nel risultato</u>	<u>Valori NULL per non corrispondenze?</u>
INNER JOIN	Solo le righe che hanno corrispondenze in entrambe le tabelle	✗ No – le righe senza corrispondenza vengono escluse
LEFT JOIN	Tutte le righe dalla tabella di sinistra e le corrispondenti della destra	✓ Sì – se non c'è corrispondenza, le colonne della tabella destra sono NULL
RIGHT JOIN	Tutte le righe dalla tabella di destra e le corrispondenti della sinistra	✓ Sì – se non c'è corrispondenza, le colonne della tabella sinistra sono NULL
FULL JOIN	Tutte le righe da entrambe le tabelle	✓ Sì – NULL su entrambi i lati dove non c'è corrispondenza

Example

```
-- INNER JOIN
SELECT s.nome, c.nome_corso
FROM studenti s
INNER JOIN corsi c ON s.corso_id = c.id;

-- LEFT JOIN
SELECT s.nome, c.nome_corso
FROM studenti s
LEFT JOIN corsi c ON s.corso_id = c.id;

-- RIGHT JOIN
SELECT s.nome, c.nome_corso
FROM studenti s
RIGHT JOIN corsi c ON s.corso_id = c.id;
```

2.4.4 Ordinamento

L'ordinamento può essere specificato mediante la clausola `ORDER BY` :

```
ORDER BY AttrDiOrdinamento [ASC|DESC]
        {,AttrDiOrdinamento [ASC|DESC]}
```

Example

```
SELECT * FROM AUTOMOBILE
ORDER BY Marca DESC, Modello
```

2.4.5 Operatori aggregati

Gli operatori aggregati (o funzioni aggregate) in SQL sono funzioni speciali che **operano su un insieme di valori** (un gruppo di righe o tutte le righe di una tabella) e restituiscono un singolo valore di riepilogo.

Operatore	Descrizione	Esempio SQL	Note sui NULL
<code>COUNT(*)</code>	Conta tutte le righe, inclusi i NULL	<code>SELECT COUNT(*) FROM prodotti</code>	Include NULL
<code>COUNT([ALL] col)</code>	Conta i valori non-NULL della colonna specificata	<code>SELECT COUNT(email) FROM utenti</code>	Esclude NULL
<code>COUNT(DISTINCT col)</code>	Conta i valori distinti e non-NULL della colonna	<code>SELECT COUNT(DISTINCT voto) FROM esami</code>	Esclude NULL , considera i distinti
<code>MIN(col)</code>	Restituisce il valore minimo della colonna	<code>SELECT MIN(voto) FROM esami</code>	Ignora NULL
<code>MAX(col)</code>	Restituisce il valore massimo della colonna	<code>SELECT MAX(voto) FROM esami</code>	Ignora NULL
<code>AVG(col)</code>	Calcola la media dei valori numerici della colonna	<code>SELECT AVG(reddito) FROM persone</code>	Ignora NULL
<code>SUM(col)</code>	Calcola la somma totale dei valori numerici della colonna	<code>SELECT SUM(voto) FROM esami</code>	Ignora NULL

Molto frequentemente gli operatori aggregati si applicano a sottoinsiemi di righe prelevati dalla clausola GROUP BY .

Example

```
-- CONTEGGIO ORDINI CLIENTE
SELECT cliente, COUNT(*) AS numero_ordini
FROM ordini
GROUP BY cliente;

-- MEDIA ESAMI
SELECT studente, AVG(voto) AS media_voti
FROM esami
GROUP BY studente
HAVING AVG(voto) >= 26;

-- SOMMA TOTALE ACQUISTI CLIENTE
SELECT cliente, SUM(importo) AS totale_speso
FROM ordini
GROUP BY cliente;

-- MEDIA REDDITO FIGLI PER PADRE
SELECT p.padre, AVG(f.reddito) AS media_reddito_figli
FROM persone f
JOIN paternita p ON p.figlio = f.nome
WHERE f.eta < 30
GROUP BY p.padre
HAVING AVG(f.reddito) > 20000;
```

2.4.6 Interrogazioni insiemistiche

Le **interrogazioni insiemistiche** in SQL consentono di combinare i risultati di due o più istruzioni SELECT in un unico set di risultati. Si basano sui principi dell'algebra relazionale, dove gli operatori agiscono su relazioni e producono nuove relazioni.

```
SelectSQL {<UNION | INTERSECT | EXCEPT> [ALL] SelectSQL}
```

- 1. SelectSQL indica una normale query di selezione
- 2. [ALL] permette di specificare se si vogliono includere tutti i duplicati (ALL) oppure no (default senza ALL elimina duplicati).

Operatore	Scopo Principale	Note Chiave / Dettagli	Esempio Sintetico
UNION ALL	Unisce i risultati di due o più SELECT eliminando duplicati	Deve avere stesso numero e tipi di colonne; elimina duplicati di default	sql SELECT col1 FROM tab1 UNION SELECT col1 FROM tab2;
INTERSECT	Restituisce solo le righe presenti in entrambi i SELECT	Supportato da alcuni DBMS; equivalente a JOIN o sottoquery con IN	sql SELECT col FROM tab1 INTERSECT SELECT col FROM tab2;
EXCEPT	Restituisce righe presenti nel primo SELECT ma non nel secondo	Conosciuto anche come MINUS in Oracle; equivalente a sottoquery con NOT IN	sql SELECT col FROM tab1 EXCEPT SELECT col FROM tab2;

Example

```
-- UNION ALL
SELECT evento FROM log_giorno1
UNION ALL
SELECT evento FROM log_giorno2;

-- INTERSECT
SELECT prodotto_id FROM negozio_A
INTERSECT
SELECT prodotto_id FROM negozio_B;

-- COMBINAZIONE DI OPERATORI
SELECT prodotto_id FROM negozio_A
UNION ALL
SELECT prodotto_id FROM negozio_B
EXCEPT
SELECT prodotto_id FROM negozio_C;
```

2.4.7 Interrogazioni nidificate

Le interrogazioni nidificate (**subquery**) sono istruzioni SELECT complete inserite all'interno di un'altra interrogazione SQL. Fungono da componenti di una query più grande, fornendo un set di dati che la query esterna utilizza per la sua elaborazione. La nidificazione può avvenire sia nella clausola `SELECT` che nella clausola `FROM`.

2.4.7.1 Nidificate semplici (non correlate)

L'interrogazione nidificata (o **interna**) viene **eseguita una sola volta** prima dell'interrogazione esterna.

Il **risultato della subquery** viene memorizzato in una tabella temporanea, a cui la **query esterna accede** per eseguire i suoi controlli. Questa interpretazione è valida quando la subquery non fa riferimento a variabili (o attributi) definite nell'interrogazione esterna.

Sostanzialmente la nidificata è semplice quando la subquery non usa dati della query esterna.

Example

```
SELECT nome
FROM studenti
WHERE corso IN (SELECT corso FROM corsi_attivi);
```

La subquery è **indipendente**, cioè non dipende da ogni singolo studente, quindi viene eseguita **una volta sola**.

2.4.7.2 Nidificate complesse (correlate)

La subquery fa riferimento a dati (generalmente una colonna) della query esterna. 💡 **Viene eseguita una volta per ogni riga** della query esterna, perché il suo risultato **dipende da quella riga**.

Example

🎯 Obiettivo della query:

Selezionare tutte le persone che ****condividono lo stesso nome e cognome con qualcun altro****, ma hanno un ****codice fiscale diverso****.

```
SELECT *
FROM Persona P
WHERE EXISTS (
    SELECT *
    FROM Persona P1
    WHERE P1.NOME = P.NOME AND
          P1.Cognome = P.Cognome AND
          P1.CodFiscale <> P.CodFiscale
)
```

1. La query principale scorre tutte le righe della tabella `Persona` con alias `P` ;
2. Per ogni riga `P` , esegue la subquery correlata che cerca una persona `P1` con lo stesso nome e cognome, ma codice fiscale diverso
3. Se esiste almeno una riga del genere, la subquery restituisce qualcosa (in quanto `EXISTS` è `TRUE`)
4. Se `EXISTS` è `TRUE` , allora la riga contenente `P` viene inclusa nel risultato finale

Da questo esempio è evidente che non è possibile eseguire l'interrogazione nidificata prima di valutare l'interrogazione più esterna, in quanto senza avere associato un valore alla variabile `P` , l'interrogazione nidificata non risulta completamente definita.

Keyword	Descrizione	Confronto su più valori	Esempio
IN	Presenza nella lista	Sì	val IN (subquery)
NOT IN	Assenza nella lista	Sì	val NOT IN (subquery)
ANY / SOME	Valore confrontato con uno qualsiasi della lista	Sì	val > ANY (subquery)
ALL	Valore confrontato con tutti i valori della lista	Sì	val > ALL (subquery)
EXISTS	Verifica se la subquery restituisce almeno un valore	No	EXISTS (subquery)
NOT EXISTS	Verifica se la subquery non restituisce nessun valore	No	NOT EXISTS (subquery)

3 Caratteristiche avanzate

3.1 Viste

Alcune query possono essere semplificate mediante l'utilizzo delle **viste**, delle tabelle virtuali il cui contenuto dipende dal contenuto delle altre tabelle della base di dati. Le viste vengono definite associando un nome ed una lista di attributi al risultato dell'esecuzione di un'interrogazione.

```
CREATE VIEW NomeVista [(ListaAttributi)] AS SelectSQL [WITH [LOCAL | CASCADED] CHECK OPTION]
```

L'interrogazione deve restituire un insieme di attributi compatibile con gli attributi definiti nello schema della vista: l'ordine nella clausola `SELECT` deve corrispondere all'ordine degli attributi nello schema.

Example

```
--VISTA 'ImpiegatiAmmin'
CREATE VIEW ImpiegatiAmmin(matricola, nome, cognome, stipendio) AS
SELECT matricola, nome, cognome, stipendio
FROM impiegato
WHERE dipart = 'Amministrazione' AND stipendio > 10

-- VISTA 'ImpiegatoAmminPoveri' ottenuta a partire dalla vista precedente
CREATE VIEW ImpiegatiAmminPoveri AS
SELECT *
FROM ImpiegatiAmmin
WHERE stipendio < 50
WITH CHECK OPTION
```

Mentre le viste possono essere trattate come relazioni base per le operazioni di interrogazione (DQL), lo stesso non si può dire per le **operazioni di aggiornamento** (DML: INSERT, UPDATE, DELETE). In molti casi, **non è possibile stabilire facilmente** una semantica univoca per gli **aggiornamenti** su una vista, perché una singola operazione su una vista **potrebbe non corrispondere univocamente** a un insieme di modifiche sulle relazioni base sottostanti.

Ad esempio, si consideri una vista creata unendo due tabelle **Afferenza** (Impiegato, Dipartimento) e **Direzione** (Dipartimento, Direttore): Impiegato, Direttore. L'inserimento di una tupla in questa vista **non corrisponderebbe** univocamente a un insieme di aggiornamenti sulle relazioni base, poiché **non sarebbe disponibile un valore per l'attributo Dipartimento**, necessario per stabilire la corrispondenza tra le due relazioni sottostanti. Per questo motivo, molti sistemi pongono **forti limitazioni** sulla possibilità di specificare aggiornamenti sulle viste.

Generalmente i problemi sorgono quando la vista è definita tramite un **JOIN** tra più tabelle. LO standard prevede che una vista sia aggiornabile solamente quando una riga di ciascun tabella di base corrisponde ad una riga della vista.

- La **WITH CHECK OPTION** è una clausola opzionale che può essere aggiunta alla definizione di una vista. Il suo scopo è **imporre** che tutte le righe inserite o aggiornate tramite la vista rispettino le condizioni definite nella clausola **WHERE** della vista stessa.

I qualificatori **LOCAL** e **CASCADED** specificano il livello di applicazione del **CHECK OPTION** quando si ha a che fare con una gerarchia di viste:

- **LOCAL** : Applica il controllo solo alle condizioni specificate nella definizione della vista corrente (quella che si sta creando o modificando, ovvero quella più esterna) e **non si propaga** alle viste sottostanti nella catena di dipendenza.
- **CASCADED** : Applica il controllo non solo alle condizioni della vista corrente, ma anche a tutte le condizioni di selezione di tutte le **viste sottostanti** nella catena su cui la vista è basata. È un comportamento "**a cascata**" che garantisce la coerenza attraverso l'intera gerarchia di viste.

3.2 Stored Procedures (procedure)

Le stored procedure sono una caratteristica avanzata dei sistemi di gestione di basi di dati (DBMS) che permettono di raggruppare istruzioni SQL per essere eseguite come una singola unità logica. Si va effettivamente a creare una funzione, assegnando un nome ad un'istruzione SQL.

Una stored procedure è un **blocco di codice** SQL (lo standard SQL-2 definisce procedure semplici formate da un solo comando SQL), o un'istruzione SQL a cui è associato un **nome**, con la possibilità di specificare parametri per lo scambio di informazioni. Il loro scopo principale è aumentare la comprensibilità del programma e riutilizzare la logica di business. In molti sistemi commerciali, una stored procedure può contenere tutti i costrutti di un linguaggio procedurale, rendendola computazionalmente completa.

Sono uno strumento potente per incapsulare logica di business direttamente nel database, migliorando la modularità e le prestazioni di certe operazioni.

☰ Example

```
PROCEDURE AssegnaCitta(  
:Dip VARCHAR(20),  
:Citta VARCHAR(20))  
UPDATE Dipartimento  
SET Citta = :Citta  
WHERE Nome = :Dip;
```

La procedura può essere invocata avendo cura di associare un valore ai parametri:

```
$ AssegnaCitta('Informatica', 'Napoli')
```

Tali procedure possono essere utilizzate per eseguire operazioni in un trigger, non restituiscono alcun valore.

3.3 Funzioni scalari

Le **funzioni scalari** in SQL sono strumenti di utilità che operano su **singoli valori** (o "tuple" a livello concettuale) e restituiscono un singolo risultato. Sono distinte dagli operatori aggregati, che invece operano su insiemi di tuple per produrre un singolo valore (ad esempio, COUNT, MIN, MAX, AVG, SUM).

3.3.1 Funzioni temporali

Permettono la gestione di informazioni temporali.

- CURRENT_DATE : restituisce la **data corrente**;
- CURRENT_TIME : restituisce l'ora corrente;
- CURRENT_TIMESTAMP :
- EXTRACT (YEAR, DATA) : estrae una specifica parte (eg. anno) da un valore di data

3.3.2 Funzioni di manipolazione di stringhe

- CHAR_LENGTH : restituisce la lunghezza della stringa
- LOWER : converte stringa in **minuscolo**
- UPPER : converte stringa in **maiuscolo**
- SUBSTRING : restituisce parte della stringa

3.3.3 Funzioni di conversione del dominio

Consentono di convertire un valore da un tipo di dato a un altro.

- CAST : consente di convertire un valore in un altro dominio (CAST (data AS CHAR(10)))

3.3.4 Funzioni matematiche

Eseguono operazioni matematiche su valori numerici.

- `ABS()` : restituisce il **valore assoluto**
- `SQRT()` : calcola la **radice quadrata**

3.3.5 Funzioni condizionali

3.3.5.1 Funzione `COALESCE`

Restituisce il **primo valore** `NOT NULL` da un elenco di espressioni. Se tutti gli argomenti sono `NULL` allora restituisce `NULL`. La funzione viene solitamente utilizzata per convertire i valori nulli in valori predefiniti dichiarati dal programmatore.

Example

Estrarre nomi, cognomi e dipartimenti cui afferiscono gli impiegati, usando la stringa 'Ignoto' nel caso in cui non si conosca il dipartimento.

```
SELECT nome, cognome, COALESCE(Dipart, 'Ignoto')
FROM impiegato
```

3.3.5.2 Funzione `NULLIF`

La funzione richiede come argomenti un'espressione e un valore costante; se l'espressione è uguale al valore costante, la funzione restituisce `NULL`, altrimenti restituisce il valore dell'espressione.

Può essere utilizzata per filtrare valori indesiderati o per esempio per gestire una divisione per 0.

Example

Estrarre nomi, cognomi e dipartimenti cui afferiscono gli impiegati, restituendo il valore `NULL` quando `Dipart` possiede il valore 'Ignoto'.

```
SELECT nome, cognome, NULLIF(Dipart, 'Ignoto')
FROM impiegato
```

3.3.5.3 Funzione `CASE`

Funzione condizionale potente in SQL, che consente di implementare logiche *"se-allora-altrimenti"* direttamente nelle query. Restituisce un singolo valore basato sulla valutazione di condizioni definite.

```
CASE espressione
{WHEN valore THEN esprRisultato}
[ELSE esprRisultato]
END
```

```
CASE {WHEN condizione THEN espressione}
[ELSE espressione]
```

Example

Estrarre l'ammontare delle tasse annuali per un veicolo

```
SELECT targa,
CASE tipo
  WHEN 'auto' THEN 2.58 * Kwatt
  ELSE NULL
END AS tassa
FROM veicolo
WHERE anno > 1975
```


3.4 Funzioni in PostgreSQL

A differenza delle procedure, le funzioni **restituiscono** un **valore** e sono definite in postgresQL tramite il linguaggio PL/pgSQL :


```
CREATE FUNCTION NomeFunzione(TipoArgomenti) RETURNS TipoRitorno
AS 'function body text'
LANGUAGE plpgsql;
```

Si tratta di un linguaggio strutturato a blocchi nella forma:

```
[ <<label>> ]
[ DECLARE dichiarazioni ]
BEGIN
  statements
END [label];
```



Il Linguaggio PL/pgSQL



- Il linguaggio PL/pgSQL e' un linguaggio strutturato a blocchi
- Ciascun blocco ha la forma:

```
DECLARE
  < dichiarazioni di variabili >
BEGIN
  < istruzioni >
END
```

- I blocchi possono essere annidati
- Dichiarazioni e istruzioni devono terminare con un punto e virgola

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Tortone, *Basi di dati*, 5e
©2018 McGraw-Hill Education (Italy) S.r.l.

Example

```
CREATE FUNCTION circ(r INTEGER) RETURNS NUMERIC AS $BODY$
DECLARE
  costante_pi CONSTANT NUMERIC := pi();
  risultato NUMERIC;
```

```
BEGIN
    risultato := 2 * costante_pi * r;
    RETURN risultato;
END;
$BODY$
LANGUAGE plpgsql;
```

È possibile definire delle variabili per salvare il risultato di una query:


```
SELECT <espressione> INTO <variabile> FROM ...
```

Example


```
-- Salvataggio della risposta in una variabile
DECLARE
    risultato corso.crediti INTEGER;
BEGIN
    SELECT SUM(crediti) INTO risultato
    FROM corso JOIN frequenza USING(id_corso)
    WHERE id_studente = studente
```

Tali funzioni supportano anche il lancio delle eccezioni, tramite il quale è possibile arrestare l'esecuzione:

```
RAISE EXCEPTION 'MessaggioEccezione'
```



Il Linguaggio PL/pgSQL: Esempio



```

DECLARE
    minimo INTEGER;
BEGIN
    minimo := (SELECT MIN(voto) FROM frequenza
               Where Corso='Basi di Dati');
    IF (minimo>27) THEN
        RAISE EXCEPTION 'Corso troppo facile';
    ENDIF;
END

```

RAISE EXCEPTION: Solleva un'eccezione con il messaggio d'errore <messaggio> ed arresta l'esecuzione della funzione.

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e
©2018 McGraw-Hill Education (Italy) S.r.l.

3.5 Trigger

I **trigger** SQL (chiamati anche **regole attive**) sono procedure eseguite autonomamente dal sistema di gestione di basi di dati (DBMS) in **risposta** a **specifici eventi** di modifica dei dati su una data tabella o vista. I DBMS

tradizionali sono passivi ed eseguono operazioni solo su richiesta, mentre un DBMS attivo, come PostgreSQL, ha capacità reattive (**basi dati attive**) e può definire e utilizzare i trigger.

Genericamente servono a garantire il soddisfacimento dei vincoli di integrità referenziale e/o specificare meccanismi di reazione ad hoc in caso di violazione dei vincoli, specificare regole aziendali.



Il concetto di trigger

- Paradigma: Evento-Condizione-Azione
 - Quando un **evento** si verifica
 - Se la **condizione** è vera
 - Allora l'**azione** è eseguita
- Questo modello consente computazioni reattive
- Non è il solo tipo di regole:
 - Vincoli di integrità
 - Regole datalog
 - Regole di business
- Problema: è difficile realizzare applicazioni complesse

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e ©2018 McGraw-Hill Education (Italy) S.r.l. 18

Sono basati sul paradigma Evento-Condizione-Azione (ECA):


- **Evento**: operazione che attiva il trigger. Solitamente si tratta di una **modifica** dello stato del database, come un `INSERT` (inserimento), `UPDATE` (modifica) o `DELETE` (cancellazione) su una tabella specifica.
- **Condizione**: **predicato** che viene valutato per determinare se l'azione del trigger debba essere eseguita;
- **Azione**: funzione o una sequenza di istruzioni SQL da eseguire se la condizione è vera

```
CREATE TRIGGER NomeTrigger
Modo Evento ON TabellaTarget
[REFERENCING Referenza]
[FOR EACH Livello]
[WHEN (PredicatoSQL)]
StatementProceduraleSQL
```

1. **Modo**:

- **BEFORE** : Il trigger viene **attivato prima** che l'operazione di modifica (`INSERT`, `UPDATE`, `DELETE`) sia eseguita. **Non possono modificare direttamente** lo stato del database, ma possono **condizionare** i valori della riga (`NEW` in modalità riga) o annullare l'operazione. Sono utili per verificare le modifiche prima che avvengano.
- **AFTER** : Il trigger viene **attivato dopo** che l'operazione di modifica è stata eseguita e gli eventuali **vincoli di integrità** sono stati **controllati**. Questa è la modalità più comune e adatta alla maggior parte delle applicazioni.
- **INSTEAD OF** : Questa modalità è specifica per le **viste**. Invece di eseguire l'operazione di `INSERT`, `UPDATE` o `DELETE` direttamente sulla vista (che potrebbe non essere aggiornabile), il trigger **INSTEAD OF** viene **eseguito al posto di tale operazione**. Questo permette di definire una logica complessa nella funzione del trigger per **instradare** le modifiche alle tabelle sottostanti, rendendo così aggiornabili viste che altrimenti non lo sarebbero (ad esempio, viste basate su più tabelle o che usano funzioni di

aggregazione)



Esempio “before” e “after”


- 1. “Conditioner” (agisce prima dell'update e della verifica di integrità)

```
create trigger LimitaAumenti
before update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```
- 2. “Re-installer” (agisce dopo l'update)


```
create trigger LimitaAumenti
after update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e

©2018 McGraw-Hill Education (Italy) S.r.l.



2. **Evento**: Specifica l'operazione che fa scattare il trigger (INSERT , UPDATE o DELETE). È possibile **specificare più eventi** per un singolo trigger (es. INSERT OR UPDATE), una estensione PostgreSQL allo standard SQL. Per gli UPDATE , si può specificare OF Colonna per attivare il trigger solo quando colonne specifiche vengono modificate
3. **REFERENCING** Referenza : serve a definire alias per i dati delle righe o tabelle coinvolte nell'operazione che attiva il trigger




Clausola referencing

- Dipende dalla granularità
 - Se la modalità è row-level, ci sono due **variabili di transizione** (old and new) che rappresentano il valore precedente o successivo alla modifica di una tupla
 - Se la modalità è statement-level, ci sono due **tabelle di transizione** (old table and new table) che contengono i valori precedenti e successivi delle tuple modificate dallo statement
- **old e old table** non sono presenti con l'evento **insert**
- **new e new table** non sono presenti con l'evento **delete**

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e

©2018 McGraw-Hill Education (Italy) S.r.l.

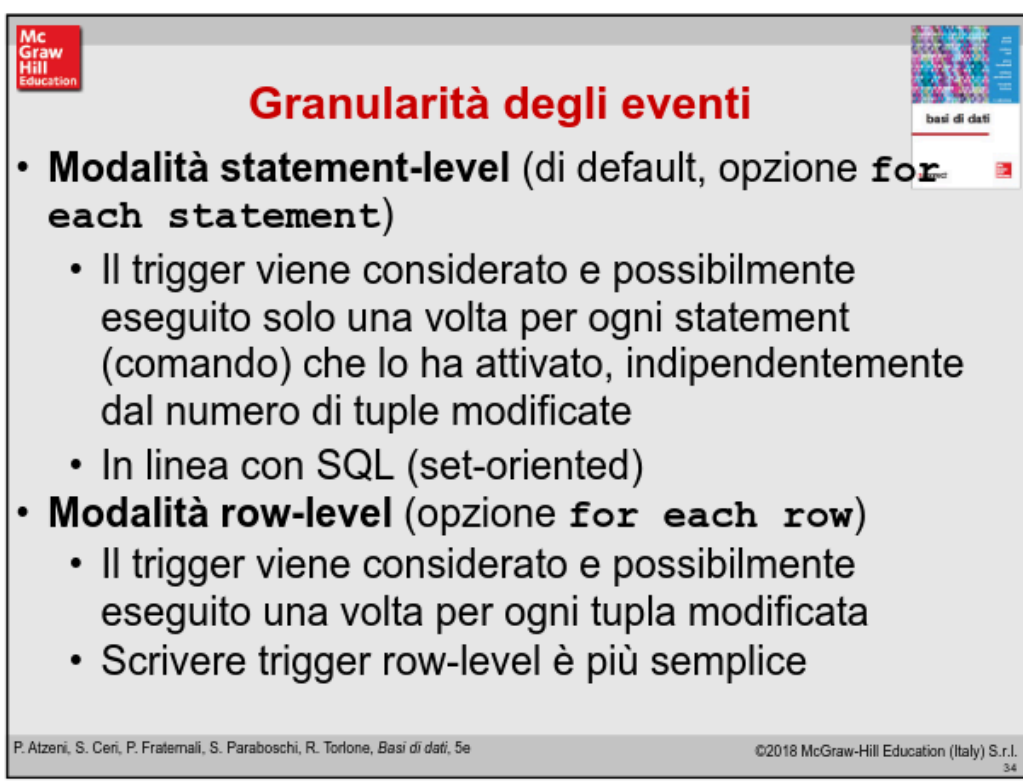


4. **Livello**: consente di impostare il livello di granularità:
 - ROW : La funzione del trigger viene eseguita **una volta per ogni singola riga** che viene influenzata dall'operazione di modifica. È spesso più semplice da scrivere per logiche che dipendono dai valori della riga specifica.

Vengono utilizzate le **variabili speciali** `OLD` e `NEW` (che sono di tipo `RECORD`) all'interno della funzione del trigger:

- `NEW` : Rappresenta la **riga** nella sua **nuova forma** (dopo un `INSERT` o `UPDATE`). Non è presente per le operazioni `DELETE` ;
- `OLD` : Rappresenta la **riga** nella sua **forma precedente** (prima di un `UPDATE` o `DELETE`). Non è presente per le operazioni `INSERT` ;
- Gli attributi individuali sono accessibili come `NEW.nome_colonna` o `OLD.nome_colonna`
- `STATEMENT` : La funzione del trigger viene eseguita **una sola volta per ogni istruzione SQL che attiva il trigger**, indipendentemente dal numero di righe modificate da tale istruzione. Questo è il comportamento **predefinito** se non specificato. È utile per logiche che riguardano l'operazione nel suo complesso, come l'aggiornamento di un contatore globale.

Vengono usate le tabelle di transizione `OLD TABLE` e `NEW TABLE` che contengono l'insieme di tutte le righe modificate dall'istruzione SQL. Per queste, la clausola `REFERENCING` diventa obbligatoria per assegnare un alias a queste tabelle temporanee. Ad esempio, `REFERENCING OLD TABLE AS vecchie_fatture` permette di riferirsi all'insieme delle righe cancellate come `vecchie_fatture` all'interno della funzione del trigger.



Granularità degli eventi

- **Modalità statement-level** (di default, opzione **for each statement**)
 - Il trigger viene considerato e possibilmente eseguito solo una volta per ogni statement (comando) che lo ha attivato, indipendentemente dal numero di tuple modificate
 - In linea con SQL (set-oriented)
- **Modalità row-level** (opzione **for each row**)
 - Il trigger viene considerato e possibilmente eseguito una volta per ogni tupla modificata
 - Scrivere trigger row-level è più semplice



P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e

©2018 McGraw-Hill Education (Italy) S.r.l.

34

5. `StatementProceduraleSQL` : indica l'azione che il trigger deve compiere. In PostgreSQL, questo significa chiamare una funzione (o "stored procedure") che è stata pre-definita. Generalmente si usa la

sintassi: EXECUTE PROCEDURE NomeProcedura(Argomenti) .



Trigger: Funzioni che specificano l'azione



La funzione che specifica l'azione di un trigger può essere scritta:

- In un linguaggio procedurale nativo del DBMS, ovvero "compatibile" con il modello logico del DB e con SQL (e.g. PL/SQL per Oracle, PL/pgSQL per PostgreSQL, ecc.)
- In un linguaggio procedurale esterno (e.g. C)
- **Noi utilizzeremo trigger le cui funzioni sono scritte in PL/pgSQL: Estensione procedurale di SQL per PostgreSQL**

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e

©2018 McGraw-Hill Education (Italy) S.r.l.

25



Evento-Condizione-Azione

- **Evento**
 - Normalmente una modifica dello stato del database: insert, delete, update
 - Quando accade l'evento, il trigger è *attivato*
- **Condizione**
 - Un predicato che identifica se l'azione del trigger deve essere eseguita
 - Quando la condizione viene valutata, il trigger è *considerato*
- **Azione**
 - Una sequenza di update SQL o una procedura
 - Quando l'azione è eseguita anche il trigger è *eseguito*
- I DBMS forniscono tutti i componenti necessari. Basta integrarli.

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e

©2018 McGraw-Hill Education (Italy) S.r.l.

17

Se vi sono più trigger associati allo stesso evento vengono eseguiti nel seguente ordine:

1. Prima i BEFORE triggers
2. Poi viene eseguita la modifica e verificati i vincoli
3. Infine sono eseguiti gli AFTER triggers

Quando triggers appartengono alla stessa categoria, l'ordine di esecuzione è definito in base alla data di creazione (i trigger più vecchi hanno priorità più alta).

3.5.1 Trigger in PL/pgSQL (postgresql)

3.5.1.1 Creazione della trigger function

```


CREATE [ OR REPLACE ] FUNCTION <nome_funzione> ()
RETURNS TRIGGER AS $BODY$ -- Devo ritornare il tipo Trigger!
DECLARE
    -- dichiarazione variabili
BEGIN
    -- istruzioni
END;
$BODY$
LANGUAGE plpgsql;

```

- \$body\$ può essere sostituito con qualunque altro delimitatore: \$EZ\$
- La sezione DECLARE può NON essere presente

⚠ Warning

Una trigger function non prende in input alcun argomento!



PL/pgSQL da usare in un trigger Un esempio per il DB Segreteria

L'attributo crediti della tabella corso può essere NULL: uno studente non dovrebbe potersi iscrivere ad un corso con crediti non noti. Definiamo un trigger che generi tale vincolo

```

CREATE FUNCTION iscrizione_valida()
RETURNS TRIGGER AS
$BODY$
DECLARE
    c INTEGER;
BEGIN
    SELECT crediti INTO c FROM corso WHERE id_corso=NEW.id_corso
    IF c IS NULL THEN RAISE EXCEPTION $$'Non è possibile iscriversi
    al corso % perchè tale corso e' in via di denizione'$$, NEW.id_corso;
    ELSE RETURN NEW;
    END IF;
END;
$BODY$
LANGUAGE PLPGSQL;

```

Il trigger, passa la nuova riga su cui effettuare il controllo nella variabile NEW.

Se la funzione non rileva eccezioni, restituisce la riga da inserire nella variabile NEW.

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, Basi di dati, 5e

©2018 McGraw-Hill Education (Italy) S.r.l.
30

Il blocco BEGIN-END può restituire un valore tramite RETURN :

- Trigger BEFORE e FOR EACH ROW : il valore di ritorno può essere NULL (annullamento dell'operazione) o una variabile di tipo RECORD
- Trigger AFTER o FOR EACH STATEMENT : valore di ritorno ignorato

3.5.1.2 Creazione del trigger

```

CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
ON table_name
[FOR [EACH] {ROW | STATEMENT}]
EXECUTE PROCEDURE trigger_function

```

Example


```
--Trigger ROW-LEVEL
CREATE TRIGGER AccountMonitor
AFTER UPDATE ON Account
FOR EACH ROW
WHEN new.Totale > old.Totale
INSERT VALUES (
    new.NumeroConto,
    new.Totale-old.Totale
) INTO Accredito
```

Il Trigger controlla il totale nuovo su account e se > del vecchio inserisce una nuova riga in una tabella diversa Accredito. New e Old si riferiscono alla riga della tabella referenziata dal Trigger, nell'esempio Account

Example

-- Trigger STATEMENT-LEVEL

```
CREATE TRIGGER ArchiviaFattureCancellate
AFTER DELETE ON Fattura
REFERENCING OLD TABLE AS VECCHIE_FATTURE
INSERT INTO FattureCancellate
(SELECT * FROM VECCHIE_FATTURE)
```



Esempio di trigger statement-level

```
create trigger ArchiviaFattureCancellate
after delete on Fattura
/* Stiamo omettendo [for each statement] */
REFERENCING OLD TABLE AS
VECCHIE_FATTURE
insert into FattureCancellate
(select *
from VECCHIE_FATTURE)
```

Osservare con for each statement il referencing è obbligatorio

Il Trigger dopo un delete di una o più fatture, ovvero quando termina lo statement, sulla tabella **Fattura**, allora inserisce all'interno della tabella fatturecancellate le righe cancellate che risultano dalla variabile old denominata attraverso il referencing VECCHIE_FATTURE.

P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, *Basi di dati*, 5e

©2018 McGraw-Hill Education (Italy) S.r.l.

37

Example

```
CREATE TRIGGER auto_order
AFTER UPDATE ON materiale
```

```

FOR EACH ROW
WHEN (OLD.qntDisponibile >= OLD.qntLimite AND NEW.qntDisponibile < NEW.qntLimite)
EXECUTE FUNCTION auto_order_function();
CREATE OR REPLACE FUNCTION auto_order_function()
RETURNS TRIGGER AS $BODY$
BEGIN
/*Verifico se é già presente un ordine con questo ID*/
IF NOT EXISTS (
SELECT *
FROM ordine WHERE idPezzo = NEW.idPezzo
) THEN
INSERT INTO ordine (idPezzo, qntRiordino, dataOrdine)
VALUES(NEW.idPezzo, NEW.qntRiordino, CURRENT_DATE);
END IF;
RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql;

```

3.6 Controllo dell'accesso

Per default l'utente che crea una risorsa ha il completo accesso sulla stessa. È tuttavia possibile [modificare i permessi](#) di lettura e scrittura per ogni singolo utente o gruppo. SQL definisce vari tipi di privilegi che possono essere concessi o revocati:

- **INSERT** : permette di inserire nuovi oggetti;
- **UPDATE** : consente di modificare il contenuto di una risorsa;
- **DELETE** : permette di eliminare oggetti;
- **SELECT** : permette di leggere la risorsa;
- **REFERENCES** : permette la definizione di vincoli di integrità referenziale verso la risorsa
- **USAGE** : permette l'uso della risorsa in una definizione, come in un dominio

Gli elementi a cui un utente non può accedere devono essergli [nascosti](#). Per autorizzare un utente a vedere solo alcune ennuple di una relazione si definisce una [vista](#) con una condizione di selezione e si attribuiscono le autorizzazioni direttamente sulla vista, anziché sulla relazione di base.

3.6.1 👍 Concessione di privilegi

```

GRANT <Privileges | ALL privileges>
ON Resource TO User [WITH GRANT OPTION]

```

- **GRANT OPTION** : specifica se il privilegio può essere trasmesso ad altri utenti. L'utente potrà a sua volta concedere lo stesso privilegio ad altri utenti

☰ Example

```

-- Concede il privilegio di SELECT sulla tabella employee all'utente user1
GRANT SELECT ON employee TO user1;

-- Concede i privilegi di SELECT e INSERT sulla tabella persona all'utente benfante
GRANT SELECT, INSERT ON persona TO benfante WITH GRANT OPTION;

-- Concede il privilegio di UPDATE solo sulla colonna nome della tabella persona

```

```
all'utente benefante
GRANT UPDATE(nome) ON persona TO benefante;
```

3.6.2 👉 Revoca di privilegi

```
REVOKE Privileges
ON Resource
FROM Users [RESTRICT | CASCADE]
```

☰ Example

```
-- Revoca il privilegio SELECT sulla tabella employee dall'utente user1
REVOKE SELECT ON employee FROM user1;

-- Revoca il privilegio INSERT sulla tabella persona dall'utente benefante
REVOKE INSERT ON persona FROM benefante;

-- Revoca tutti i privilegi sulla tabella employee dall'utente user1
REVOKE ALL ON employee FROM user1;
```

3.7 Transazioni

Una transazione è definita come una **unità elementare di lavoro** svolta da un programma applicativo su una base di dati. In un sistema di gestione di basi di dati (DBMS), una transazione è una **sequenza di operazioni** (lettura/scrittura) considerate **indivisibili**. Un sistema che offre un meccanismo per la definizione e l'esecuzione di transazioni è chiamato sistema transazionale (o OLTP).

Le transazioni sono caratterizzate da **quattro** proprietà **fondamentali**, spesso riassunte con l'acronimo **ACID**:

1. **Atomicità** (Atomicity): Una transazione è una sequenza di operazioni **indivisibile**, che deve essere eseguita per intero o per niente. Non sono ammessi stati intermedi parziali in quanto non è possibile lasciare la base di dati in uno stato intermedio o **inconsistente**. Ad esempio, in un trasferimento di fondi tra due conti (A e B), o avvengono sia il prelievo da A che il versamento su B, oppure nessuna delle due operazioni.
2. **Consistenza** (Consistency): Al termine dell'esecuzione di una transazione, i **vincoli di integrità** della base di dati devono essere **soddisfatti**. Se lo stato iniziale della base di dati è corretto, anche lo stato finale, dopo la transazione, deve essere corretto. **Violazioni temporanee** possono esistere "durante" l'esecuzione, ma se permangono alla fine, la transazione deve essere annullata (abortita) per intero.
3. **Isolamento** (Isolation): Ogni transazione deve essere eseguita in modo **isolato** e **indipendente** da tutte le altre transazioni concorrenti. L'esecuzione **concorrente** di un insieme di transazioni deve produrre un **risultato equivalente** a quello che si otterrebbe con una esecuzione sequenziale. Ciò implica che una transazione non espone i suoi stati intermedi, evitando il cosiddetto "**effetto domino**";
4. **Durabilità** (Durability): Una volta che una transazione è stata marcata come **completata** ("**commit**"), i **cambiamenti** che essa ha apportato alla base di dati **non devono** più essere **persi**. Questi effetti devono essere salvati su un supporto non volatile e garantiti anche in caso di guasti del sistema.

3.7.1 Gestione delle transazioni in SQL

In SQL, una transazione inizia con il primo comando SQL dopo la connessione alla base di dati o dopo la conclusione di una transazione precedente. Sebbene lo standard SQL includa un comando `START TRANSACTION` (non sempre obbligatorio o previsto in tutti i sistemi), la gestione si basa principalmente su due comandi per la conclusione:

- COMMIT [WORK] : **termina correttamente** la transazione, rendendo permanenti tutte le operazioni eseguite dall'inizio della transazione;
- ROLLBACK [WORK] : **annulla** la transazione, rinunciando all'esecuzione di tutte le operazioni specificate dopo l'inizio della transazione

Molti sistemi prevedono anche una modalità AUTOCOMMIT , dove **ogni singola operazione** SQL forma una **transazione** a sé stante. I record delle transazioni (BEGIN, COMMIT, ABORT, UPDATE, INSERT, DELETE) vengono scritti in un log per descrivere le attività di una transazione.