Linguaggio SQL

Database, DBMS e RDBMS

Il termine database, tradotto in italiano con banca dati o base di dati indica un insieme di dati riguardanti uno stesso argomento, o più argomenti correlati tra loro, strutturata in modo tale da consentire l'uso dei dati stessi (e il loro aggiornamento) da parte di applicazioni software. La base di dati, oltre ai dati veri e propri, deve contenere anche le informazioni sulle loro rappresentazioni e sulle relazioni che li legano.

Un DBMS (Database Management System) è un sistema software in grado di gestire una collezione di dati che sono condivisi da più applicazioni e utenti. Un DBMS deve:

- gestire una grande quantità di dati.
- gestire la condivisione dei dati tra diversi utenti ed applicazioni.
- gestire le transazioni.
- garantire l'affidabilità dei dati, cioè la capacità di ripristino a fronte di malfunzionamenti (resilienza), meccanismi di salvataggio (backup) e ripristino (recovery).
- offrire una "visione strutturata" dei dati in base al modello logico usato.
- garantire la privatezza dei dati in base a dei meccanismi di autorizzazione.

Un RDBMS (Relational Database Management System) è un sistema relazionale per la gestione di basi di dati ed indica un DBMS basato sul modello relazionale ed è stato introdotto da Edgar F.Codd. I requisiti minimi per cui un DBMS può essere chiamato RDBMS sono:

- deve presentare i dati all'utente sotto forma di relazioni (una presentazione a tabelle può soddisfare questa proprietà).
- deve fornire operatori relazionali per manipolare i dati in forma tabellare.

Linguaggio SQL

SQL (Structured Query Language) è un linguaggio creato per l'accesso a informazioni memorizzate nei database. L'SQL nasce nel 1974 ad opera di Donald Chamberlin, nei laboratori dell'IBM e il suo DBMS relazionale più diffuso ancora oggi.

Più nel dettaglio, SQL è un linguaggio standardizzato per database basati sul modello relazionale (RDBMS), progettato per le seguenti operazioni:

1. creare e modificare schemi di database: DDL, Data Definition Language;
2. inserire, modificare e gestire dati memorizzati: DML, Data Manipulation Language;
3. interrogare i dati memorizzati: DQL, Data Query Language;
4. creare e gestire strumenti di controllo e accesso ai dati: DCL, Data Control Language.

Breve storia di SQL

Qui ci sono importanti punti di riferimento del storia dell'SQL:

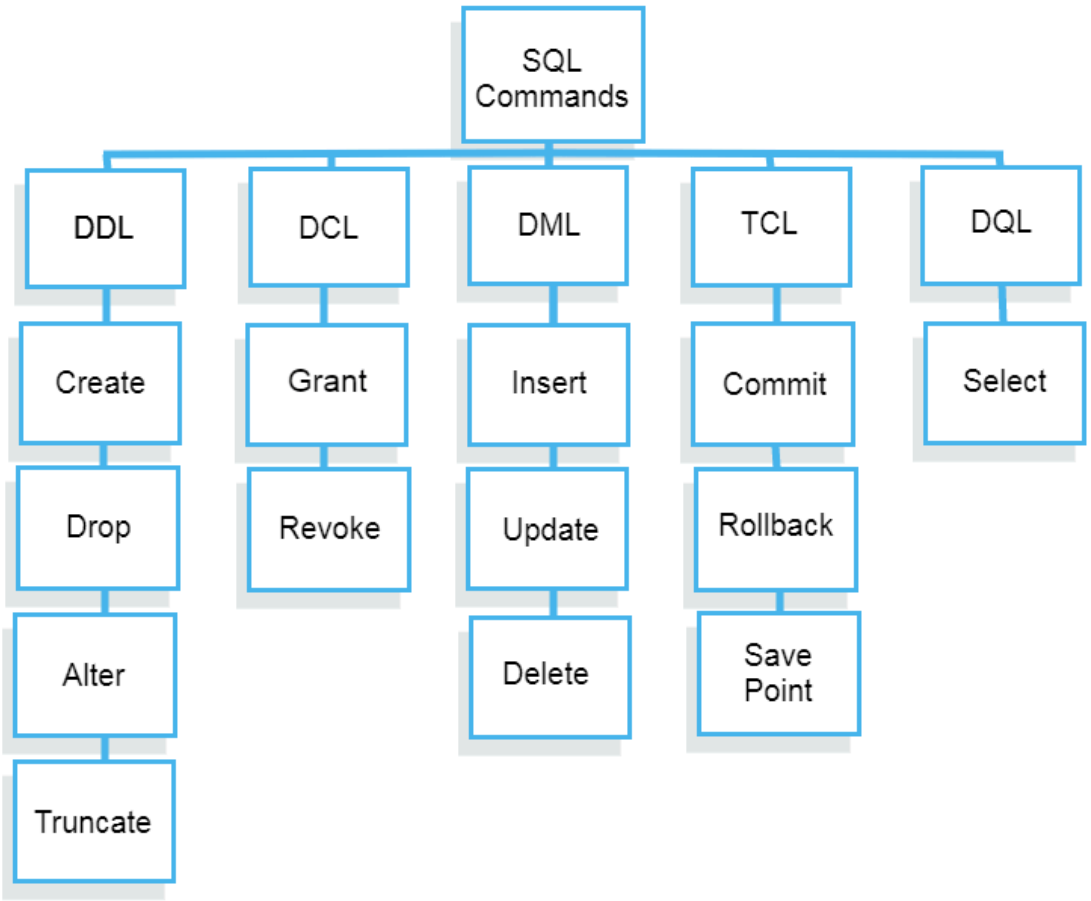
- 1970 – Il Dr. Edgar F. "Ted" Codd descrive un modello relazionale per i database.
- 1974 – Appare il linguaggio di query strutturato.
- 1978 - IBM ha rilasciato un prodotto chiamato System/R.
- 1986 - IBM ha sviluppato il prototipo di un database relazionale, standardizzato dall'ANSI.
- 1989- Lancio della prima versione in assoluto di SQL
- 1999 – Lancio di SQL 3 con funzionalità come trigger, orientamento agli oggetti, ecc.
- Funzioni della finestra SQL2003, funzionalità relative a XML, ecc.
- SQL2006: supporto per il linguaggio di query XML
- Supporto migliorato di SQL2011 per i database temporali

Struttura SQL

Essendo un linguaggio dichiarativo, SQL non richiede la stesura di sequenze di operazioni (come ad es. i linguaggi imperativi), piuttosto di specificare le proprietà logiche delle informazioni ricercate. Esso si divide in quattro sottoinsiemi:

- **DDL** (Data Definition Language) – DDL serve a creare, modificare o eliminare gli oggetti in un database. Sono i comandi DDL a definire la struttura del database e quindi dei dati ivi contenuti. Ma non fornisce gli strumenti per modificare i dati stessi: per tale scopo di usa il DML. L'utente deve avere i permessi necessari per agire sulla struttura del database e questi permessi vengono assegnati tramite il DCL (Data Control Language).

- **DML** (Data Manipulation Language) – DML fornisce i comandi per inserire, modificare, eliminare o leggere i dati all'interno delle tabelle di un database. La struttura di questi dati deve già essere stata definita tramite il DDL. Esempi di comandi sono SELECT,INSERT, UPDATE, DELETE ecc.
- **DCL** (Data Control Language) – DCL serve a fornire o revocare agli utenti i permessi necessari per poter utilizzare i comandi DML e DDL, oltre agli stessi comandi DCL (che gli servono per poter a sua volta modificare i permessi su alcuni oggetti).
- **DQL** (Data Query Language) – DQL serve per creare query sui database e sui sistemi informativi da parte degli utenti oltre a rendere possibile l'estrazione di informazioni dal database interrogando la base dei dati interfacciandosi dunque con l'utente e le sue richieste di servizio.



Operatori SQL

Infine, gli operatori, messi a disposizione da SQL standard si dividono in quattro categorie:

- Operatori di confronto
- Operatori aritmetici
- Operatori condizionali
- Operatori logici

Differenza tra DDL, DML, DCL e DQL

Ricapitolando, in informatica SQL (Structured Query Language) `e un linguaggio standardizzato per database basati sul modello relazionale (RDBMS) che racchiude un insieme di linguaggi:

1. DDL (Data Definition Language)
   Permette di creare, modificare o eliminare gli oggetti in un database ovvero agire sullo schema di database. Ad esempio i comandi CREATE TABLE.., DROP TABLE.., ecc.
2. DML (Data Manipulation Language)
   Consente di leggere, inserire, modificare o eliminare i dati in un database. Ad esempio i comandi INSERT INTO.., DELETE FROM.., ecc.
3. DCL (Data Control Language)
   Utilizzato per fornire o revocare agli utenti i permessi necessari per poter utilizzare i comandi Data Manipulation Language (DML) e Data Definition Language (DDL), oltre agli stessi comandi DCL (che servono a loro volta a modificare i permessi su alcuni oggetti). Ad esempio i comandi GRANT.. ON.. TO.. ecc.
4. TCL (Transaction Control Language) è un sottoinsieme di SQL che consente di gestire le transazioni in un database. TCL è un insieme di comandi che garantiscono la coerenza e l'integrità dei dati durante le operazioni.
5. DQL (Data Query Language)
   Usato per creare query sui database e sui sistemi informativi da parte degli utenti. Serve per rendere possibile l'estrazione di informazioni dal database interrogando la base dei dati interfacciandosi dunque con l'utente e le sue richieste di servizio. Ad esempio i comandi SELECT.. FROM.. WHERE.. ecc.

DDL

Si usa il DDL per costruire e modificare l'architettura del tuo database: le tabelle, le colonne, gli indici, le viste e altri elementi che contengono e organizzano i dati.

I comandi DDL principali sono:

1. `CREATE`: Utilizzato per creare nuovi oggetti nel database.
2. `ALTER`: Utilizzato per modificare la struttura di oggetti esistenti.
3. `DROP`: Utilizzato per eliminare oggetti esistenti dal database.
4. `TRUNCATE`: Utilizzato per rimuovere tutti i record da una tabella rapidamente (spesso considerato DDL per le sue implicazioni).

Vediamoli nel dettaglio:

CREATE

Il comando `CREATE` è usato per creare nuovi oggetti nel database. Gli oggetti più comuni che creerai sono tabelle, indici e viste.

a) `CREATE DATABASE` (La sintassi può variare leggermente tra i sistemi DBMS) Crea un nuovo database logico.

SQL

CREATE DATABASE NomeDatabase;

b) `CREATE TABLE` Questo è forse il comando DDL più utilizzato. Definisce una nuova tabella, specificando le sue colonne, i tipi di dati per ciascuna colonna e i vincoli (constraints) che garantiscono l'integrità dei dati.

Sintassi di Base:

```
CREATE TABLE NomeTabella (
    NomeColonna1 TipoDati [Vincoli],
    NomeColonna2 TipoDati [Vincoli],
    ...
    NomeColonnaN TipoDati [Vincoli],
    [VincoliTabella]
);
```

Esempio Approfondito

```
CREATE TABLE Utenti (
```

```
    UtenteID INT PRIMARY KEY AUTO_INCREMENT, -- Chiave primaria, auto-incrementante (sintassi specifica può variare, es. IDENTITY in SQL Server, SERIAL in PostgreSQL)

    Nome VARCHAR(50) NOT NULL,            -- Stringa di max 50 caratteri, non può essere NULL

    Cognome VARCHAR(50) NOT NULL,

    Email VARCHAR(100) UNIQUE NOT NULL,      -- Deve essere unica in tutta la tabella e non NULL

    DataRegistrazione DATE DEFAULT CURRENT_DATE, -- Data, con valore predefinito alla data corrente

    Eta INT CHECK (Eta >= 18),            -- Numero intero, con vincolo che sia >= 18

    RepartoID INT,                        -- Colonna per chiave esterna



    -- Vincolo a livello di tabella per la chiave esterna

    FOREIGN KEY (RepartoID) REFERENCES Reparti(RepartoID)

        ON DELETE SET NULL -- Se un reparto viene eliminato, imposta RepartoID a NULL negli utenti correlati

        ON UPDATE CASCADE  -- Se l'ID di un reparto cambia, aggiornalo anche qui);
```

- **Tipi di Dati Comuni:** `INT` (interi), `VARCHAR(n)` (stringhe variabili), `CHAR(n)` (stringhe fisse), `TEXT` (testo lungo), `DATE`, `DATETIME` / `TIMESTAMP`, `BOOLEAN`, `DECIMAL(p,s)` (numeri decimali), `FLOAT`/`REAL` (numeri in virgola mobile).
- **Vincoli (Constraints) Comuni:**
  - `NOT NULL`: La colonna non può contenere valori NULL.
  - `UNIQUE`: Tutti i valori nella colonna devono essere unici.
  - `PRIMARY KEY`: Identifica univocamente ogni riga nella tabella (combina `NOT NULL` e `UNIQUE`). Una tabella può avere una sola chiave primaria.
  - `FOREIGN KEY`: Garantisce l'integrità referenziale collegando una colonna (o più) a una `PRIMARY KEY` di un'altra tabella. Definisce azioni come `ON DELETE` (es. `CASCADE`, `SET NULL`, `RESTRICT`) e `ON UPDATE`.
  - `CHECK`: Assicura che i valori in una colonna soddisfino una condizione specifica.
  - `DEFAULT`: Fornisce un valore predefinito per la colonna se non ne viene specificato uno durante l'inserimento (`INSERT`).

c) `CREATE INDEX` Crea un indice su una o più colonne di una tabella per velocizzare le query (`SELECT`). Gli indici sono strutture dati separate che permettono al database di trovare le righe più rapidamente.

-- Indice sulla colonna Email per ricerche veloci

```
CREATE INDEX idx_utenti_email ON Utenti (Email);
```

-- Indice composito su Nome e Cognome

```
CREATE INDEX idx_utenti_nome_cognome ON Utenti (Nome, Cognome);
```

Nota: Le chiavi primarie (`PRIMARY KEY`) e i vincoli `UNIQUE` solitamente creano automaticamente un indice.

d) **`CREATE VIEW`** Crea una vista, che è una tabella virtuale basata sul risultato di una query SQL salvata. Utile per semplificare query complesse, restringere l'accesso ai dati o presentare i dati in un formato specifico.

```
CREATE VIEW VistaUtentiAttivi AS

SELECT UtenteID, Nome, Cognome, Email

FROM Utenti

WHERE DataRegistrazione >= '2024-01-01'; -- Esempio di filtro
```

ALTER

Il comando `ALTER` viene utilizzato per modificare la struttura di un oggetto esistente. È molto potente ma va usato con cautela, specialmente su tabelle con molti dati, perché può richiedere tempo e bloccare la tabella.

a) **`ALTER TABLE`** Permette di aggiungere, modificare o eliminare colonne e vincoli in una tabella esistente.

Sintassi Comuni:

Aggiungere una Colonna:

```
ALTER TABLE NomeTabella

ADD COLUMN NomeNuovaColonna TipoDati [Vincoli];
```

Esempio:

```
ALTER TABLE Utenti

ADD COLUMN Telefono VARCHAR(20) NULL; -- Aggiunge colonna Telefono, ammette NULL
```

Eliminare una Colonna:

```

ALTER TABLE NomeTabella

DROP COLUMN NomeColonnaDaEliminare;
```

Esempio:

```
ALTER TABLE Utenti
```

```
DROP COLUMN Eta; -- Attenzione: perdita irreversibile dei dati in quella colonna!
```

Modificare il Tipo di Dati o i Vincoli di una Colonna: (La sintassi varia molto tra i DBMS: `MODIFY COLUMN`, `ALTER COLUMN TYPE`, etc.)

SQL
-- Esempio generico (potrebbe essere diverso in MySQL, PostgreSQL, SQL Server, Oracle)

```
ALTER TABLE NomeTabella

ALTER COLUMN NomeColonna TipoNuovoDati [NuoviVincoli];
```

Esempio (sintassi simile a PostgreSQL/SQL Server):

```
ALTER TABLE Utenti

ALTER COLUMN Nome TYPE VARCHAR(70); -- Modifica la lunghezza massima del Nome
```

Esempio (sintassi simile a MySQL):

```
ALTER TABLE Utenti

MODIFY COLUMN Nome VARCHAR(70) NOT NULL; -- Modifica tipo e vincolo NOT NULL
```

Aggiungere un Vincolo:

```
ALTER TABLE NomeTabella

ADD CONSTRAINT NomeVincolo TIPO_VINCOLO (Colonne);
```

Esempio:

```
ALTER TABLE Utenti

ADD CONSTRAINT chk_telefono CHECK (Telefono LIKE '+%'); -- Aggiunge un vincolo CHECK sul formato del telefono
```

Eliminare un Vincolo:

```
ALTER TABLE NomeTabella

DROP CONSTRAINT NomeVincolo;
```

Esempio:

```
ALTER TABLE Utenti

DROP CONSTRAINT chk_telefono;
```

b) `ALTER DATABASE`, `ALTER INDEX`, `ALTER VIEW` Esistono anche comandi `ALTER` per altri oggetti, ma sono meno comuni o hanno sintassi più specifiche del DBMS (es. rinominare un database, ricostruire un indice).

DROP

Il comando `DROP` è usato per eliminare definitivamente oggetti dal database. **Usare con estrema cautela**, poiché l'operazione è generalmente irreversibile.

a) **`DROP TABLE`** Elimina una tabella e tutti i suoi dati, indici, trigger e vincoli associati.

```
DROP TABLE NomeTabella;
```

Esempio:

```
DROP TABLE Utenti; -- Elimina la tabella Utenti e tutti i suoi dati!
```

Opzione utile: `IF EXISTS` previene un errore se la tabella non esiste.

```
DROP TABLE IF EXISTS Utenti;
```

Attenzione alle Chiavi Esterne: Se altre tabelle fanno riferimento alla tabella che stai eliminando tramite `FOREIGN KEY`, il `DROP` potrebbe fallire a meno che tu non elimini prima le chiavi esterne o usi opzioni specifiche come `CASCADE` (pericoloso, elimina anche gli oggetti dipendenti).

b) **`DROP INDEX`** Elimina un indice.

```
DROP INDEX NomeIndice ON NomeTabella; -- Sintassi comune (può variare)
```

Esempio:

```
DROP INDEX idx_utenti_email ON Utenti;
```

c) **`DROP VIEW`** Elimina una vista.

```
DROP VIEW NomeVista;
```

Esempio:

```
DROP VIEW VistaUtentiAttivi;
```

d) **`DROP DATABASE`** Elimina un intero database e tutti gli oggetti al suo interno. **Operazione estremamente distruttiva!**

```
DROP DATABASE NomeDatabase;
```

TRUNCATE TABLE

Anche se tecnicamente rimuove dati (come `DELETE`, che è DML), `TRUNCATE` è spesso considerato parte del DDL o trattato in modo simile perché:

- È molto più veloce di `DELETE FROM NomeTabella` (senza `WHERE`) perché di solito non registra le singole eliminazioni di riga.
- È un'operazione minimamente loggata (in molti DBMS).
- Resetta i contatori auto-incrementanti (in molti DBMS).
- Non attiva i trigger `ON DELETE` (in molti DBMS).
- È irreversibile nella maggior parte dei casi (non si può fare `ROLLBACK` come con `DELETE` all'interno di una transazione standard in molti sistemi).

Sintassi:

```
TRUNCATE TABLE NomeTabella;
```

Esempio:

```
TRUNCATE TABLE LogEventi; -- Svuota rapidamente la tabella LogEventi
```

Quando usarlo: Utile per svuotare rapidamente tabelle di log, tabelle temporanee o per resettare una tabella durante lo sviluppo/test. Da non usare se serve poter annullare l'operazione o se devono attivarsi trigger specifici per ogni riga eliminata.

Considerazioni Importanti e Best Practices con DDL:

- **Irreversibilità:** Molte operazioni DDL (`DROP`, `TRUNCATE`, `ALTER TABLE DROP COLUMN`) sono difficili o impossibili da annullare. Esegui sempre backup prima di modifiche strutturali importanti.
- **Blocchi (Locking):** Comandi come `ALTER TABLE` possono bloccare la tabella per la durata dell'operazione, impattando le performance e la disponibilità dell'applicazione. Pianifica le modifiche durante finestre di manutenzione.
- **Script e Controllo Versione:** Gestisci le modifiche allo schema del database tramite script SQL salvati in un sistema di controllo versione (come Git). Questo permette di tracciare le modifiche, collaborare e distribuire gli aggiornamenti in modo controllato. Utilizza strumenti di migrazione (es. Flyway, Liquibase) per automatizzare l'applicazione degli script DDL.
- **Idempotenza:** Scrivi script DDL in modo che possano essere eseguiti più volte senza causare errori o effetti collaterali indesiderati. Usa clausole come `IF NOT EXISTS` per `CREATE` e `IF EXISTS` per `DROP`.

- **Dipendenze:** Sii consapevole delle dipendenze tra oggetti (es. viste che dipendono da tabelle, chiavi esterne). Un comando `DROP` o `ALTER` può fallire o avere effetti a cascata se ci sono oggetti dipendenti.
- **Specificità del DBMS:** Anche se SQL è uno standard, ci sono variazioni significative nella sintassi DDL tra diversi sistemi di gestione di database (MySQL, PostgreSQL, SQL Server, Oracle, SQLite, ecc.). Consulta sempre la documentazione specifica del tuo DBMS.

DQL

È interessante notare che, mentre DDL (Data Definition Language) e DML (Data Manipulation Language) sono categorie ampiamente riconosciute e distinte, il termine DQL è spesso usato per riferirsi specificamente all'unico comando SQL il cui scopo primario è l'interrogazione o il recupero dei dati: il comando SELECT.

In molte classificazioni, SELECT viene considerato parte integrante del DML, poiché tecnicamente "manipola" i dati nel senso che li legge, li filtra, li combina e li presenta. Tuttavia, dato il suo ruolo centrale e specifico nel recuperare informazioni piuttosto che modificarle, a volte viene isolato sotto l'etichetta DQL.

Quindi, una guida approfondita su DQL è, in sostanza, una guida approfondita sul comando SELECT e tutte le sue potenti clausole. Riprenderemo ed espanderemo quanto visto nella guida DML, concentrandoci esclusivamente sull'arte di interrogare il database.

Guida Approfondita a SQL DQL (Il Comando SELECT)

Il DQL (Data Query Language), rappresentato dal comando SELECT, è lo strumento fondamentale per estrarre informazioni significative dai dati memorizzati nel tuo database. Permette di porre domande al database e ottenere risposte sotto forma di set di risultati (tabelle di dati).

La Struttura Fondamentale del Comando SELECT:

```
SELECT [DISTINCT] espressione_colonna1 [AS alias1], espressione_colonna2 [AS alias2], ... | *

FROM NomeTabella [AS alias_tabella]

[JOIN TipoJoin AltraTabella [AS alias_altra_tabella] ON CondizioneJoin]...

[WHERE CondizioneFiltroRighe]

[GROUP BY ColonnaRaggruppamento1, ColonnaRaggruppamento2, ...]

[HAVING CondizioneFiltroGruppi]

[ORDER BY ColonnaOrdinamento1 [ASC | DESC], ColonnaOrdinamento2 [ASC | DESC], ...]

[LIMIT NumeroRighe OFFSET SaltoRighe]; -- Sintassi comune, può variare (es. TOP, FETCH)
```

Analizziamo in dettaglio ogni componente:

SELECT - La Lista di Selezione:

Specifica cosa vuoi vedere nel risultato.

- **NomeColonna**: Seleziona colonne specifiche.

```
SELECT Nome, Email FROM Utenti;
```

- *** (Asterisco)**: Seleziona tutte le colonne dalla/e tabella/e specificate in FROM. Utile per esplorazione rapida, ma sconsigliato in codice di produzione (meno performante, meno robusto a cambiamenti di schema, trasferisce dati non necessari).

```
SELECT * FROM Ordini;
```

- **DISTINCT**: Elimina le righe duplicate dal set di risultati, basandosi sui valori delle colonne selezionate.

```
SELECT DISTINCT Citta FROM Clienti; -- Mostra ogni città una sola volta
```

- **Espressioni**: Puoi usare calcoli, concatenazioni, chiamate a funzioni.
    - **Funzioni Scalari**: Operano su valori singoli (es. **UPPER(), LOWER(), CONCAT(), LENGTH(), DATE_FORMAT(), COALESCE()**).

```
SELECT UPPER(Nome) AS NomeMaiuscolo, COALESCE(Telefono, 'Non Fornito') FROM Utenti;
```

    - **Funzioni di Aggregazione**: Operano su un gruppo di righe per restituire un singolo valore (usate spesso con **GROUP BY**). Le principali sono **COUNT(), SUM(), AVG(), MIN(), MAX()**.

```
SELECT COUNT(*) FROM Utenti; -- Numero totale di utenti

SELECT AVG(Importo) FROM Fatture WHERE Pagato = TRUE; -- Importo medio delle fatture pagate
```

- **AS Alias**: Assegna un nome temporaneo e più leggibile a una colonna o a un'espressione nel risultato. Fondamentale per chiarezza e per riferirsi a colonne calcolate in **ORDER BY**.

```
SELECT Nome + ' ' + Cognome AS NomeCompleto FROM Utenti; -- (Sintassi concatenazione varia, es. || o CONCAT())
```

FROM - La Sorgente dei Dati:

- Specifica la tabella (o le tabelle) da cui recuperare i dati.

```
SELECT UtenteID, Nome FROM Utenti;
```

- Puoi assegnare un alias anche alla tabella (**FROM Utenti AS u**) per rendere più concisi i riferimenti alle colonne, specialmente nei **JOIN**.

JOIN - Combinare Dati da Più Tabelle:

- Permette di collegare righe da due o più tabelle basandosi su una condizione di join (tipicamente l'uguaglianza tra chiave primaria di una tabella e chiave esterna dell'altra).

- INNER JOIN: Restituisce solo le righe per cui esiste una corrispondenza in entrambe le tabelle unite. È il tipo di join più comune.

```
SELECT o.OrdineID, o.DataOrdine, u.Nome, u.Email

FROM Ordini AS o

INNER JOIN Utenti AS u ON o.UtenteID = u.UtenteID; -- Ordini con i dati dell'utente che li ha effettuati
```

- LEFT JOIN (o LEFT OUTER JOIN): Restituisce tutte le righe della tabella a sinistra (FROM) e le righe corrispondenti della tabella a destra (JOIN). Se non c'è corrispondenza a destra, le colonne della tabella destra avranno valore NULL. Utile per trovare dati che non hanno una corrispondenza.

```
SELECT u.UtenteID, u.Nome, o.OrdineID

FROM Utenti AS u

LEFT JOIN Ordini AS o ON u.UtenteID = o.UtenteID; -- Mostra tutti gli utenti, e i loro ordini se presenti (altrimenti OrdineID sarà NULL)
```

- RIGHT JOIN (o RIGHT OUTER JOIN): Speculare al LEFT JOIN. Restituisce tutte le righe della tabella a destra e le corrispondenti a sinistra (o NULL). Meno comune, spesso si preferisce riformulare con LEFT JOIN.

- FULL OUTER JOIN: Restituisce tutte le righe da entrambe le tabelle. Se manca la corrispondenza da un lato, le colonne di quel lato saranno NULL. Utile per vedere tutti i dati da entrambe le parti, con o senza corrispondenze.

- CROSS JOIN: Produce il prodotto cartesiano delle tabelle (ogni riga della prima tabella combinata con ogni riga della seconda). Raramente usato intenzionalmente, può generare risultati enormi.

- Condizione ON: Specifica come le tabelle sono collegate.

WHERE - Filtrare le Righe:

- Restringe le righe restituite basandosi su condizioni logiche applicate prima di qualsiasi raggruppamento.

- Usa operatori di confronto (=, >, <, >=, <=, != / <>), logici (AND, OR, NOT), e specifici (LIKE, IN, BETWEEN, IS NULL).

```
SELECT * FROM Prodotti WHERE Prezzo > 100 AND Categoria = 'Elettronica';
```

```
SELECT * FROM Eventi WHERE NomeEvento LIKE '%Concerto%';

SELECT * FROM Task WHERE DataScadenza IS NULL OR Stato != 'Completato';
```

5. **GROUP BY** - Raggruppare Righe:

- Aggrega righe multiple in una singola riga di riepilogo, basandosi sui valori comuni in una o più colonne.
- È quasi sempre usato insieme a funzioni di aggregazione (**COUNT**, **SUM**, **AVG**, etc.) applicate alle colonne non presenti nella clausola **GROUP BY**.

```
SELECT Categoria, COUNT(*) AS NumeroProdotti, AVG(Prezzo) AS PrezzoMedio

FROM Prodotti

GROUP BY Categoria; -- Statistiche per categoria
```

- Importante: Tutte le colonne nella **SELECT** list devono essere o funzioni di aggregazione o colonne presenti nella clausola **GROUP BY**.

6. **HAVING** - Filtrare i Gruppi:

- Simile a **WHERE**, ma filtra i risultati dopo che le righe sono state raggruppate da **GROUP BY** e le funzioni di aggregazione sono state calcolate.
- Permette di applicare condizioni sui risultati delle funzioni di aggregazione.

```
SELECT Categoria, COUNT(*) AS NumeroProdotti

FROM Prodotti

GROUP BY Categoria

HAVING COUNT(*) >= 10; -- Mostra solo le categorie con almeno 10 prodotti
```

7. **ORDER BY** - Ordinare i Risultati:

- Dispone le righe del set di risultati finale in un ordine specifico.
- **ASC**: Ordine ascendente (A-Z, 0-9, date meno recenti prima). È il default.
- **DESC**: Ordine discendente (Z-A, 9-0, date più recenti prima).
- Puoi ordinare per più colonne (l'ordinamento secondario si applica quando i valori della colonna primaria sono uguali). Puoi usare alias definiti nella **SELECT** list.

```
SELECT Nome, Cognome, DataNascita FROM Pazienti ORDER BY DataNascita DESC, Cognome ASC;
```

8. **LIMIT / TOP / Workspace** - Limitare il Numero di Righe:

- Restringe il numero di righe restituite dal risultato finale. La sintassi varia:
    - **LIMIT N**: (MySQL, PostgreSQL, SQLite) Restituisce le prime N righe.
    - **LIMIT N OFFSET M**: (MySQL, PostgreSQL, SQLite) Salta le prime M righe e restituisce le successive N (utile per la paginazione).
    - **TOP N**: (SQL Server) Messo dopo SELECT [DISTINCT].
    - **Workspace FIRST N ROWS ONLY**: (Standard SQL, Oracle, DB2) Messo alla fine della query.

```
-- Primi 5 prodotti più costosi (MySQL/PostgreSQL)

SELECT NomeProdotto, Prezzo FROM Prodotti ORDER BY Prezzo DESC LIMIT 5;



-- Prodotti dalla posizione 11 alla 20 (paginazione) (MySQL/PostgreSQL)

SELECT NomeProdotto, Prezzo FROM Prodotti ORDER BY NomeProdotto LIMIT 10 OFFSET 10;
```

Best Practices per DQL (**SELECT**):

- Sii Specifico: Seleziona solo le colonne che ti servono (SELECT col1, col2) invece di SELECT *. Migliora performance e leggibilità.
- Filtra Presto: Usa WHERE per ridurre il numero di righe il prima possibile.
- Indici: Assicurati che ci siano indici appropriati (creati con DDL) sulle colonne usate frequentemente in WHERE, JOIN, e ORDER BY per velocizzare le query.
- JOIN con Cautela: Comprendi i tipi di join e usali correttamente. Evita CROSS JOIN non necessari. Assicurati che le condizioni ON usino colonne indicizzate.
- Leggibilità: Formatta le query in modo chiaro (indentazione, a capo), usa alias significativi e commenta le parti complesse.
- Test: Prova le tue query su dati di test per assicurarti che restituiscano i risultati attesi e valutane le performance.

In sintesi, il DQL, incarnato dal comando **SELECT**, è lo strumento essenziale per estrarre valore e conoscenza dai tuoi dati. La sua flessibilità e potenza derivano dalla combinazione delle sue clausole, permettendoti di formulare interrogazioni da semplici a estremamente complesse. Padroneggiare SELECT è una competenza fondamentale nell'ambito dei database.

Ricorda che la sintassi esatta di alcune clausole (specialmente LIMIT/TOP/Workspace) e funzioni può variare leggermente tra i diversi

sistemi di gestione di database (DBMS). Consulta sempre la documentazione specifica del tuo sistema.

DML

Dopo aver esplorato il DDL per definire la struttura, passiamo ora al DML (Data Manipulation Language), l'insieme di comandi SQL utilizzati per interagire con i dati all'interno delle strutture definite dal DDL.

Il DML ti permette di:

- Recuperare dati (interrogare).
- Inserire nuovi dati.
- Modificare dati esistenti.
- Eliminare dati esistenti.

I comandi DML fondamentali sono:

1. **SELECT**: Recupera dati dalle tabelle.
2. **INSERT**: Aggiunge nuove righe (record) a una tabella.
3. **UPDATE**: Modifica i dati nelle righe esistenti di una tabella.
4. **DELETE**: Rimuove righe esistenti da una tabella.

Vediamoli in dettaglio:

SELECT

È il comando DML più utilizzato e versatile. Serve per estrarre dati da una o più tabelle. La sua potenza risiede nelle numerose clausole che permettono di filtrare, ordinare, raggruppare e combinare i dati.

Sintassi di Base:

```
SELECT [DISTINCT] NomeColonna1, NomeColonna2, ... | *

FROM NomeTabella

[WHERE CondizioneFiltro]

[GROUP BY ColonnaRaggruppamento]

[HAVING CondizioneFiltroGruppi]

[ORDER BY ColonnaOrdinamento [ASC | DESC]]

[LIMIT NumeroRighe]; -- O TOP, FETCH FIRST ecc. a seconda del DBMS
```

Spiegazione delle Clausole Principali:

- **SELECT [DISTINCT]** colonne | *: Specifica le colonne da restituire.
    - *****: Seleziona tutte le colonne della tabella.

- DISTINCT: Restituisce solo righe uniche (elimina i duplicati basati sulle colonne selezionate).
- Puoi usare funzioni (es. COUNT(), SUM(), AVG(), UPPER(), CONCAT()) e alias (AS NuovoNome).

```
SELECT UtenteID, Nome AS NomeUtente, Email FROM Utenti;

SELECT COUNT(*) AS NumeroTotaleUtenti FROM Utenti;

SELECT DISTINCT RepartoID FROM Utenti WHERE RepartoID IS NOT NULL;
```

- FROM NomeTabella: Indica la tabella (o le tabelle unite tramite JOIN) da cui estrarre i dati.
- WHERE CondizioneFiltro: Filtra le righe basandosi su una condizione logica. Usa operatori come =, >, <, >=, <=, != o <>, LIKE (pattern matching), IN (lista di valori), BETWEEN (intervallo), IS NULL, IS NOT NULL, combinati con AND, OR, NOT.

```
SELECT Nome, Cognome, Email FROM Utenti WHERE Eta >= 30;

SELECT * FROM Utenti WHERE Nome LIKE 'Mar%'; -- Nomi che iniziano con Mar

SELECT * FROM Utenti WHERE RepartoID IN (1, 3);

SELECT * FROM Utenti WHERE DataRegistrazione BETWEEN '2024-01-01' AND '2024-12-31';
```

- JOIN: Combina righe da due o più tabelle basandosi su una colonna correlata (solitamente chiavi primarie ed esterne).
  - INNER JOIN: Restituisce solo le righe che hanno corrispondenza in entrambe le tabelle.
  - LEFT JOIN (o LEFT OUTER JOIN): Restituisce tutte le righe della tabella di sinistra e le righe corrispondenti della tabella di destra (o NULL se non c'è corrispondenza).
  - RIGHT JOIN (o RIGHT OUTER JOIN): Restituisce tutte le righe della tabella di destra e le righe corrispondenti della tabella di sinistra.
  - FULL OUTER JOIN: Restituisce tutte le righe quando c'è una corrispondenza in una delle tabelle (sinistra o destra).

```
SELECT u.Nome, u.Cognome, r.NomeReparto

FROM Utenti u

INNER JOIN Reparti r ON u.RepartoID = r.RepartoID; -- Unisce Utenti e Reparti



SELECT u.Nome, u.Cognome, r.NomeReparto

FROM Utenti u
```

```
LEFT JOIN Reparti r ON u.RepartoID = r.RepartoID; -- Mostra tutti gli utenti, anche quelli senza reparto assegnato
```

- **GROUP BY** ColonnaRaggruppamento: Raggruppa righe che hanno lo stesso valore in una o più colonne. Solitamente usato con funzioni di aggregazione (**COUNT**, **SUM**, **AVG**, **MIN**, **MAX**) per eseguire calcoli su ciascun gruppo.

```
SELECT RepartoID, COUNT(*) AS NumeroUtenti

FROM Utenti

WHERE RepartoID IS NOT NULL

GROUP BY RepartoID; -- Conta utenti per reparto
```

- **HAVING** CondizioneFiltroGruppi: Filtra i gruppi creati da **GROUP BY** basandosi su una condizione applicata ai risultati delle funzioni di aggregazione. **WHERE** filtra le righe prima del raggruppamento, **HAVING** filtra i gruppi dopo.

```
SELECT RepartoID, COUNT(*) AS NumeroUtenti

FROM Utenti

WHERE RepartoID IS NOT NULL

GROUP BY RepartoID

HAVING COUNT(*) > 10; -- Mostra solo reparti con più di 10 utenti
```

- **ORDER BY** ColonnaOrdinamento [**ASC** | **DESC**]: Ordina le righe del risultato basandosi su una o più colonne. **ASC** (ascendente, predefinito) o **DESC** (discendente).

```
SELECT Nome, Cognome, DataRegistrazione

FROM Utenti

ORDER BY DataRegistrazione DESC, Cognome ASC; -- Ordina per data (più recente prima), poi per cognome (A-Z)
```

- **LIMIT** NumeroRighe (o **TOP** N in SQL Server, **Workspace FIRST N ROWS ONLY** in standard SQL/Oracle/DB2): Limita il numero di righe restituite. Utile per la paginazione o per vedere solo i primi/ultimi risultati.

```
SELECT * FROM Utenti ORDER BY UtenteID DESC LIMIT 10; -- Gli ultimi 10 utenti registrati (MySQL/PostgreSQL)

-- SELECT TOP 10 * FROM Utenti ORDER BY UtenteID DESC; -- (SQL Server)
```

Subquery (Query Annidate):

Una **SELECT** può essere usata all'interno di un'altra query **DML** (in **SELECT**, FROM, WHERE, HAVING).

```
-- Seleziona utenti il cui RepartoID è presente nella tabella Reparti con Budget > 50000
```

```
SELECT Nome, Cognome FROM Utenti

WHERE RepartoID IN (SELECT RepartoID FROM Reparti WHERE Budget > 50000);
```

INSERT

Il comando INSERT aggiunge una o più nuove righe a una tabella.

Sintassi Principali:

- Inserire una riga specificando colonne e valori: (Metodo consigliato per chiarezza e robustezza)

```
INSERT INTO NomeTabella (Colonna1, Colonna2, Colonna3)

VALUES (Valore1, Valore2, Valore3);
```

- Esempio:

```
INSERT INTO Utenti (Nome, Cognome, Email, Eta, RepartoID)

VALUES ('Mario', 'Rossi', 'm.rossi@example.com', 35, 1);
```

- Nota: Le colonne non specificate riceveranno il loro valore DEFAULT o NULL (se permesso). Le colonne auto-incrementanti di solito non vengono specificate.

- Inserire una riga senza specificare le colonne: (Richiede che i valori siano forniti nello stesso ordine delle colonne nella definizione della tabella e per tutte le colonne non auto-incrementanti/senza default). Meno robusto se la struttura della tabella cambia.

```
INSERT INTO NomeTabella VALUES (Valore1, Valore2, Valore3, ...);
```

- Inserire più righe con un singolo comando: (Sintassi può variare leggermente)

```
INSERT INTO NomeTabella (Colonna1, Colonna2) VALUES

(ValoreA1, ValoreA2),

(ValoreB1, ValoreB2),

(ValoreC1, ValoreC2);
```

- Inserire i risultati di una query SELECT:

```
INSERT INTO NomeTabellaDestinazione (Colonna1, Colonna2)

SELECT ColonnaA, ColonnaB

FROM AltraTabella

WHERE Condizione;
```

- Esempio: Copiare utenti da una tabella temporanea 'NuoviUtenti' a 'Utenti'

```
INSERT INTO Utenti (Nome, Cognome, Email)

SELECT Nome, Cognome, EmailDaVerificare

FROM NuoviUtenti

WHERE EmailVerificata = TRUE;
```

UPDATE

Il comando UPDATE modifica i valori nelle colonne di righe esistenti che soddisfano una certa condizione.

Sintassi:

```
UPDATE NomeTabella

SET Colonna1 = NuovoValore1,

   Colonna2 = NuovoValore2,

   ...

WHERE CondizioneFiltro; -- ATTENZIONE ALLA CLAUSOLA WHERE!
```

- **SET**: Specifica le colonne da aggiornare e i nuovi valori. Puoi usare valori letterali, espressioni o valori da altre colonne.
- **WHERE**: Specifica quali righe aggiornare. È *FONDAMENTALE*! Se omessa, **UPDATE** modificherà *TUTTE* le righe della tabella.

Esempio:

```
-- Aggiorna l'email e l'età dell'utente con UtenteID = 5
UPDATE Utenti
SET Email = 'nuova.email@example.com',
    Eta = 36
WHERE UtenteID = 5;

-- Aumenta lo stipendio del 10% per tutti gli utenti del RepartoID = 2
UPDATE Utenti
SET Stipendio = Stipendio * 1.10
WHERE RepartoID = 2;

-- ATTENZIONE: Aggiorna TUTTE le email nella tabella (probabilmente un errore!)
-- UPDATE Utenti SET Email = 'default@example.com';
```

DELETE

Il comando DELETE rimuove una o più righe esistenti da una tabella basandosi su una condizione.

Sintassi:

```
DELETE FROM NomeTabella
WHERE CondizioneFiltro; -- ATTENZIONE ALLA CLAUSOLA WHERE!
```

- **WHERE**: Specifica quali righe eliminare. È *FONDAMENTALE*! Se omessa, **DELETE** rimuoverà *TUTTE* le righe della tabella.

Esempio:

```
-- Elimina l'utente con UtenteID = 10
DELETE FROM Utenti
WHERE UtenteID = 10;

-- Elimina tutti gli utenti registrati prima del 2020
DELETE FROM Utenti
WHERE DataRegistrazione < '2020-01-01';

-- ATTENZIONE: Elimina TUTTE le righe dalla tabella Utenti!
```

```
-- DELETE FROM Utenti;
```

Differenza tra **DELETE** e **TRUNCATE TABLE** (DDL):

- **DELETE** è **DML**, **TRUNCATE** è **DDL** (o simile).
- **DELETE** rimuove le righe una per una (può essere lento su tabelle grandi) e registra l'operazione (permette **ROLLBACK** all'interno di una transazione). Può attivare trigger **ON DELETE**.
- **TRUNCATE** è molto più veloce (rimuove tutte le righe deallocando le pagine dati), è minimamente loggato (spesso non permette **ROLLBACK** facile), di solito non attiva trigger e può resettare contatori **AUTO_INCREMENT**.

MERGE (Comando Avanzato/Specifico del DBMS)

Il comando **MERGE** (o sintassi equivalenti come INSERT ... ON CONFLICT in PostgreSQL o INSERT ... ON DUPLICATE KEY UPDATE in MySQL) esegue operazioni INSERT, UPDATE o DELETE su una tabella di destinazione basate sul confronto con una tabella o query sorgente. È utile per sincronizzare dati ("UPSERT").

Concetto (Sintassi varia molto):

```
MERGE INTO TabellaDestinazione AS Target

USING FonteDati AS Source -- (Può essere una tabella, vista o query)

ON (Target.Chiave = Source.Chiave) -- Condizione di join

WHEN MATCHED THEN -- Se la riga esiste già nella destinazione

    UPDATE SET Target.Colonna1 = Source.Valore1, Target.Colonna2 = Source.Valore2 -- Aggiorna

    -- [DELETE WHERE CondizioneDelete] -- Opzionalmente, elimina

WHEN NOT MATCHED BY TARGET THEN -- Se la riga esiste nella sorgente ma non nella destinazione

    INSERT (Colonna1, Colonna2) VALUES (Source.Valore1, Source.Valore2); -- Inserisci

-- [WHEN NOT MATCHED BY SOURCE THEN ...] -- Meno comune, agisce su righe solo nella destinazione
```

Considerazioni Importanti e Best Practices con DML:

- WHERE è Cruciale: Non sottolineerò mai abbastanza l'importanza della clausola WHERE in UPDATE e DELETE per evitare modifiche o

cancellazioni accidentali di massa. Testa sempre le condizioni WHERE con un SELECT prima di eseguire UPDATE o DELETE.

- Transazioni: Le operazioni DML (INSERT, UPDATE, DELETE) sono tipicamente eseguite all'interno di transazioni. Una transazione è un'unità di lavoro atomica: o tutte le operazioni al suo interno hanno successo (COMMIT) o vengono annullate (ROLLBACK) in caso di errore, mantenendo la consistenza del database. Impara a usare BEGIN TRANSACTION, COMMIT, ROLLBACK.

- Performance:

    - Le query SELECT complesse o su tabelle grandi beneficiano enormemente degli indici (creati con DDL CREATE INDEX) sulle colonne usate in WHERE, JOIN e ORDER BY.

    - UPDATE e DELETE di molte righe possono essere operazioni intensive. Eseguirle in batch più piccoli o durante periodi di basso carico può essere una buona strategia.

- Sicurezza (SQL Injection): Quando costruisci query DML usando input fornito dall'utente (es. da un'applicazione web), non concatenare mai direttamente l'input nella stringa SQL. Questo apre la porta agli attacchi di SQL Injection. Usa sempre query parametrizzate o prepared statements forniti dal tuo linguaggio di programmazione/driver del database.

- Leggibilità: Formatta le tue query SQL in modo leggibile, usa alias significativi per tabelle e colonne (AS) e aggiungi commenti (-- per riga singola, /* ... */ per blocco) per spiegare parti complesse.

Il DML è il cuore dell'interazione quotidiana con un database relazionale. Padroneggiare questi comandi, specialmente SELECT, è essenziale per qualsiasi sviluppatore, analista o amministratore di database. Spero questa guida approfondita ti sia stata utile!

DCL

Dopo aver definito la struttura (DDL), manipolato i dati (DML) e interrogato il database (DQL/SELECT), il DCL si occupa di un aspetto fondamentale: la **sicurezza** e il **controllo degli accessi**. In pratica, il DCL definisce chi può fare cosa all'interno del database.

I comandi principali del DCL sono:

1. GRANT: Concede permessi o privilegi agli utenti o ai ruoli.

2. REVOKE: Rimuove permessi o privilegi precedentemente concessi.

3. DENY (Presente in alcuni DBMS come SQL Server): Nega esplicitamente un permesso, sovrascrivendo eventuali GRANT.

Vediamo questi comandi e i concetti associati in modo approfondito.

Concetti Fondamentali di Sicurezza SQL

Prima di addentrarci nei comandi, è essenziale comprendere alcuni concetti chiave:

- Utenti (Users/Logins): Rappresentano le identità che possono connettersi al database. Possono essere persone fisiche, applicazioni o servizi. La creazione e gestione degli utenti stessi avviene spesso tramite comandi specifici del DBMS (es. **CREATE USER, CREATE LOGIN**) che a volte sono considerati parte del DCL o comandi amministrativi separati.

- Ruoli (Roles): Sono "contenitori" di privilegi. Invece di assegnare permessi individualmente a ciascun utente, è buona pratica assegnare i permessi a un ruolo e poi assegnare quel ruolo agli utenti. Questo semplifica enormemente l'amministrazione: per modificare i permessi di molti utenti, basta modificare il ruolo. Molti DBMS hanno ruoli predefiniti (es. **public, db_owner, read_only**). È possibile creare ruoli personalizzati (es. CREATE ROLE).
- **Privilegi (Privileges/Permissions):** Sono le azioni specifiche che un utente o un ruolo è autorizzato a compiere su determinati oggetti del database. Possono essere:
    - **A livello di Oggetto:** Permessi su tabelle, viste, procedure, funzioni, ecc. Esempi comuni:
        - SELECT: Permesso di leggere dati da una tabella/vista.
        - INSERT: Permesso di inserire nuove righe.
        - UPDATE: Permesso di modificare righe esistenti.
        - DELETE: Permesso di eliminare righe.
        - REFERENCES: Permesso di creare chiavi esterne che fanno riferimento a una tabella.
        - EXECUTE: Permesso di eseguire una procedura memorizzata o una funzione.
        - ALTER: Permesso di modificare la struttura dell'oggetto (DDL).

        - DROP: Permesso di eliminare l'oggetto (DDL).
    - A livello di Sistema/Database: Permessi più ampi che riguardano l'intero database o il server. Esempi: **CREATE TABLE, CREATE VIEW, CREATE USER, BACKUP DATABASE**, ecc.
- **Proprietà dell'Oggetto (Object Ownership):** Di solito, l'utente che crea un oggetto (es. una tabella) ne diventa il proprietario e ha automaticamente tutti i permessi su di esso.

GRANT

Il comando **GRANT** viene utilizzato per concedere uno o più privilegi a un utente o a un ruolo.

Sintassi Generale:

```
GRANT privilegio1 [, privilegio2, ...] | ALL PRIVILEGES

ON TipoOggetto NomeOggetto -- (es. TABLE MiaTabella, DATABASE MioDB, PROCEDURE MiaProc)

TO utente1 | ruolo1 [, utente2 | ruolo2, ...];
```

Sintassi Specifica per Colonne (alcuni DBMS):

```
GRANT privilegio(colonna1 [, colonna2, ...]) -- Es. GRANT SELECT(Nome, Email), UPDATE(Telefono)

ON TABLE NomeTabella

TO utente1 | ruolo1;
```

Sintassi per Concedere un Ruolo a un Utente:

```
GRANT NomeRuolo TO NomeUtente;
```

Clausola Opzionale **WITH GRANT OPTION**:

```
GRANT SELECT ON TABLE Clienti TO utente_responsabile

WITH GRANT OPTION;
```

Questa clausola permette **all'utente_responsabile** di concedere a sua volta il permesso SELECT sulla tabella Clienti ad altri utenti o ruoli. Usare con estrema cautela, poiché può rendere difficile tracciare chi ha effettivamente i permessi.

Esempi:

```
-- Concedere il permesso di lettura (SELECT) sulla tabella Prodotti al ruolo 'analisti'

GRANT SELECT ON TABLE Prodotti TO analisti;
```

```
-- Concedere i permessi di inserimento e aggiornamento sulla tabella Ordini all'utente 'app_service'

GRANT INSERT, UPDATE ON TABLE Ordini TO app_service;

-- Concedere tutti i permessi sulla tabella LogErrori all'utente 'admin_user' (usare con cautela!)

GRANT ALL PRIVILEGES ON TABLE LogErrori TO admin_user;

-- Concedere il permesso di eseguire la procedura 'CalcolaBonus' al ruolo 'manager'

GRANT EXECUTE ON PROCEDURE CalcolaBonus TO manager;

-- Concedere il ruolo 'sviluppatore' all'utente 'mario_rossi'

GRANT sviluppatore TO mario_rossi;

-- Concedere il permesso di creare nuove tabelle nel database corrente all'utente 'db_architect' (privilegio di sistema)

GRANT CREATE TABLE TO db_architect;
```

REVOKE

Il comando **REVOKE** viene utilizzato per rimuovere privilegi precedentemente concessi tramite **GRANT**.

Sintassi Generale:

```
REVOKE privilegio1 [, privilegio2, ...] | ALL PRIVILEGES

ON TipoOggetto NomeOggetto

FROM utente1 | ruolo1 [, utente2 | ruolo2, ...];
```

Sintassi Specifica per Colonne (alcuni DBMS):

```
REVOKE privilegio(colonna1 [, colonna2, ...])

ON TABLE NomeTabella

FROM utente1 | ruolo1
```

Sintassi per Rimuovere un Ruolo da un Utente:

```
REVOKE NomeRuolo FROM NomeUtente;
```

Revocare l'Opzione **GRANT OPTION** (sintassi può variare):

```
-- Rimuove solo la capacità di concedere il permesso, ma lascia il permesso stesso all'utente

REVOKE GRANT OPTION FOR SELECT ON TABLE Clienti FROM utente_responsabile;
```

Clausole **`CASCADE`** / **`RESTRICT`** (Comportamento varia tra DBMS):

- `REVOKE ... CASCADE`: Se l'utente/ruolo da cui si sta revocando un permesso (che aveva `WITH GRANT OPTION`) lo aveva a sua volta concesso ad altri, `CASCADE` revoca il permesso anche a questi ultimi.

- `REVOKE ... RESTRICT` (o comportamento predefinito in alcuni sistemi): Impedisce la revoca se ci sono dipendenze (es. se l'utente ha concesso il permesso ad altri).

Esempi:

```
-- Rimuovere il permesso SELECT sulla tabella Prodotti dal ruolo 'analisti'

REVOKE SELECT ON TABLE Prodotti FROM analisti;

-- Rimuovere i permessi INSERT e UPDATE sulla tabella Ordini dall'utente 'app_service'

REVOKE INSERT, UPDATE ON TABLE Ordini FROM app_service;

-- Rimuovere il permesso di eseguire la procedura 'CalcolaBonus' dal ruolo 'manager'

REVOKE EXECUTE ON PROCEDURE CalcolaBonus FROM manager;

-- Rimuovere il ruolo 'sviluppatore' dall'utente 'mario_rossi'

REVOKE sviluppatore FROM mario_rossi;
```

DENY (Principalmente in SQL Server)

In alcuni sistemi come SQL Server, esiste anche `DENY`. Mentre `REVOKE` semplicemente rimuove un permesso precedentemente concesso (o negato), `DENY` crea una **negazione esplicita** che ha la precedenza su qualsiasi `GRANT`.

Sintassi (SQL Server):

```
DENY privilegio1 [, privilegio2, ...] | ALL PRIVILEGES

ON TipoOggetto NomeOggetto [(colonna1, ...)]

TO utente1 | ruolo1 [, utente2 | ruolo2, ...]

[CASCADE]; -- Per propagare la negazione a chi aveva ricevuto il permesso dall'utente negato
```

Differenza Chiave:

- Se un utente appartiene a due ruoli, RuoloA (con `GRANT SELECT` su TabellaX) e RuoloB (con `REVOKE SELECT` su TabellaX), l'utente *potrebbe* comunque avere accesso tramite RuoloA.
- Se invece RuoloB avesse `DENY SELECT` su TabellaX, l'utente *non avrebbe* accesso, perché `DENY` sovrascrive `GRANT`.

Esempio (SQL Server):

```
-- Impedire esplicitamente all'utente 'utente_limitato' di cancellare dati dalla tabella LogImportanti,

-- anche se facesse parte di un ruolo che ha il permesso DELETE.

DENY DELETE ON TABLE LogImportanti TO utente_limitato;
```

Best Practices per DCL:

- **Principio del Minimo Privilegio (Principle of Least Privilege):** Concedi agli utenti e alle applicazioni solo i permessi strettamente necessari per svolgere le loro funzioni. Evita di assegnare privilegi ampi (`ALL PRIVILEGES`) o ruoli amministrativi potenti se non indispensabili.
- **Usa i Ruoli:** Gestisci i permessi tramite ruoli personalizzati. È molto più scalabile e manutenibile che gestire i permessi per ogni singolo utente.
- **Cautela con `WITH GRANT OPTION`:** Limita l'uso di questa opzione per prevenire la diffusione incontrollata dei permessi.
- **Non Usare Account Condivisi:** Ogni utente/applicazione dovrebbe avere il proprio account per tracciabilità e responsabilità.
- **Audit Regolari:** Controlla periodicamente i permessi assegnati per assicurarti che siano ancora appropriati e per rimuovere quelli non più necessari.
- **Separazione dei Compiti (Separation of Duties):** Usa account/ruoli diversi per compiti diversi (es. un utente per l'amministrazione, uno per le operazioni dell'applicazione, uno per il reporting).
- **Sicurezza a Livello Applicativo:** La sicurezza del database (DCL) dovrebbe essere integrata con misure di sicurezza a livello di applicazione (autenticazione, autorizzazione a livello di funzionalità).

In conclusione, il DCL è uno strumento vitale per proteggere l'integrità e la confidenzialità dei dati nel tuo database. Comprendere e applicare correttamente `GRANT` e `REVOKE` (e `DENY` dove applicabile), insieme a una buona strategia basata sui ruoli e sul principio del minimo privilegio, è fondamentale per qualsiasi amministratore di database o sviluppatore responsabile della sicurezza.

Ricorda che l'implementazione specifica dei comandi DCL, dei privilegi disponibili e della gestione di utenti/ruoli può variare significativamente tra i diversi sistemi DBMS (es. Oracle, SQL Server, MySQL, PostgreSQL). Consulta sempre la documentazione ufficiale del tuo sistema.