

# MAURO CASADEI

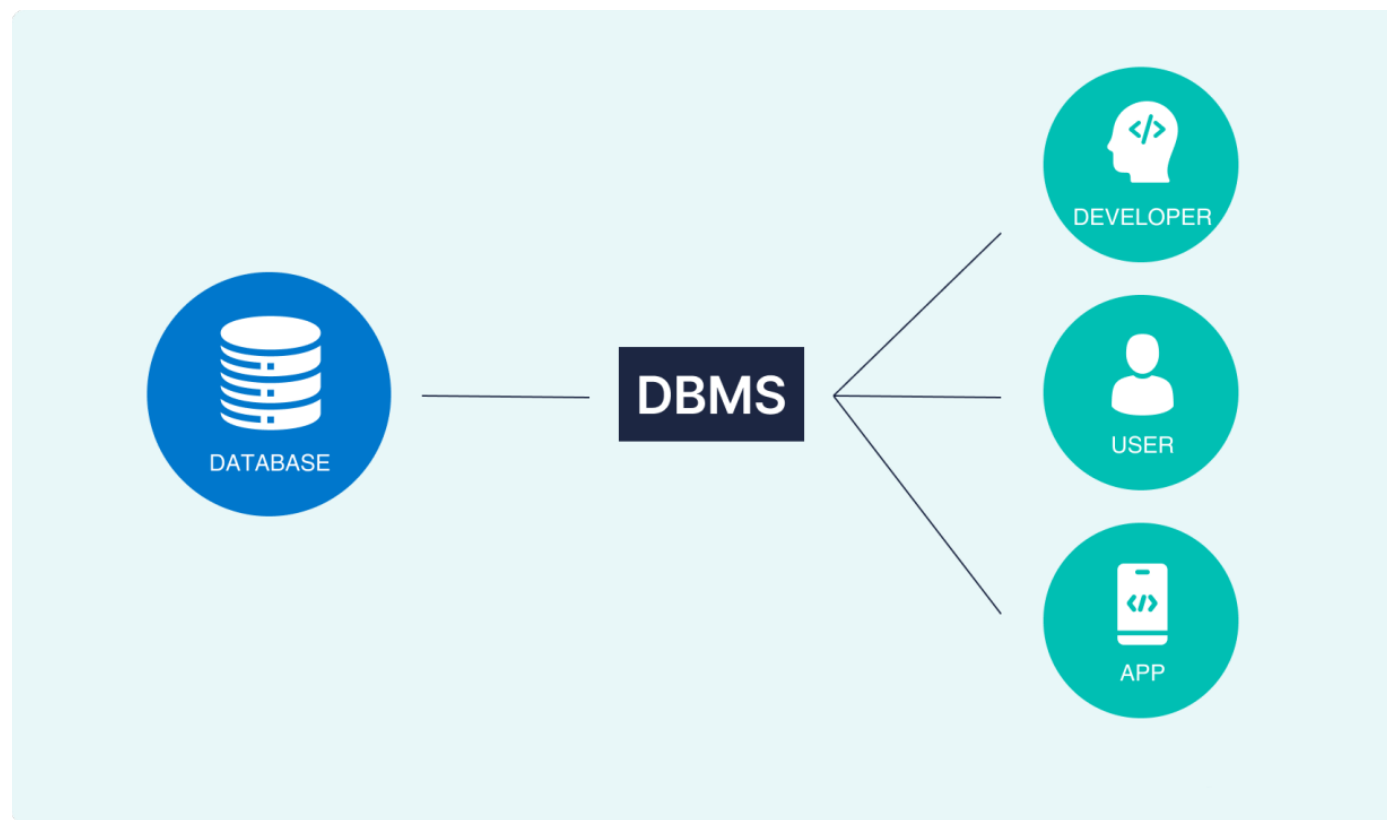
## BASI DI DATI RELAZIONALI

# BASI DI DATI RELAZIONALI

# CAPITOLO 1: FONDAMENTI DEI DATABASE

# COS'È UN DATABASE?

- Un Database è definita come una collezione di dati ordinato accessibile tramite un «DataBase Management System» (DBMS), il software che consente, tramite queries, l'interazione di utenti e applicazioni con il Database per la gestione e analisi di dati



# COS'È UN DATABASE?

- Alcuni esempi di Database:
- Un'azienda di vendita al dettaglio utilizza un database relazionale per memorizzare le informazioni sui prodotti, come codice a barre, descrizione, prezzo, quantità in magazzino, ecc.
- Un'azienda di assicurazioni come \*\*\*\* utilizza un database relazionale per memorizzare le informazioni sui clienti, come polizze, premi, sinistri, ecc.
- Un'azienda di e-commerce come \*\*\*\* utilizza un database relazionale per memorizzare le informazioni sui clienti, come nome, indirizzo, numero di telefono, ordini effettuati, ecc.
- Nell'ambito dei giochi online e dei social game, I Database tengono traccia dei tuoi punteggi, del tuo inventario e dello stato del gioco. Inoltre, tengono traccia di cose come la tua lista di amici, le chat in gioco e le transazioni, e le tue interazioni con altri giocatori.

# COS'È UN DATABASE SERVER

È un computer o un sistema che ospita un database e gestisce le richieste di accesso ai dati da parte di più client. È responsabile dell'archiviazione, della gestione e della sicurezza dei dati, oltre a garantire l'accesso simultaneo da parte di più utenti o applicazioni.

- **Esempi di database server:**MySQL Server
- Microsoft SQL Server
- PostgreSQL
- Oracle Database
- MongoDB Server

# COS'È UN DBMS / RDBMS?

Per DBMS ci si riferisce al software che consiste di **un'interfaccia** tra gli utenti di un database con le loro applicazioni e le risorse costruite dall' hardware e dagli archivi di dati presenti in un sistema di elaborazione

Alcune caratteristiche di questo software sono:

- **Indipendenza della struttura fisica dei dati**
- **I programmi applicativi sono indipendenti dai dati fisici, cioè è possibile modificare i supporti** con cui i dati sono registrati e le modalità di accesso alle memoria di massa senza modifiche alle applicazioni
- **Indipendenza della struttura logica dei dati**
- **I programmi applicativi sono indipendenti dalla struttura logica con cui i dati sono organizzati negli archivi: quindi è possibile apportare modifiche alla definizione delle strutture della base di dati senza modificarne il software applicativo**

# COS'È UN DBMS / RDBMS?

- Facilità di accesso
- il ritrovamento dei dati è facilitato e **svolto con grandi velocità**, anche nel caso di richieste provenienti contemporaneamente da più utenti.
- Integrità dei dati
- le **operazioni sui dati** richieste dagli utenti **vengono eseguite fino al loro completamento** per assicurare la consistenza dei dati
- Sicurezza dei dati
- sono previste **procedure di controllo per impedire accessi non autorizzati** ai dati contenuti nel database e di protezione da guasti accidentali
- Uso di linguaggi per la gestione del database
- il database viene gestito attraverso **comandi per la manipolazione dei dati contenuti in esso e comandi per effettuare interrogazioni alla base di dati al fine di ottenere le informazioni desiderate**



# TIPI DI DATABASE: RELAZIONALI E NOSQL

I Database che analizzeremo sono di 2 tipi, relazionali e noSQL

I Database relazionali sono quelli più comuni e sono caratterizzati da:

- organizzazione dei dati in **tabelle con una struttura fissa e definita**
- **divisione delle tabelle in righe e colonne**, ogni riga contenente un record e ogni colonna un attributo
- utilizzo di **relazioni logiche come «uno a molti» o «molti a molti» per collegare i dati di più tabelle**
- **utilizzo del linguaggio SQL** (Structured Query Language) per creare, modificare e interrogare i dati.

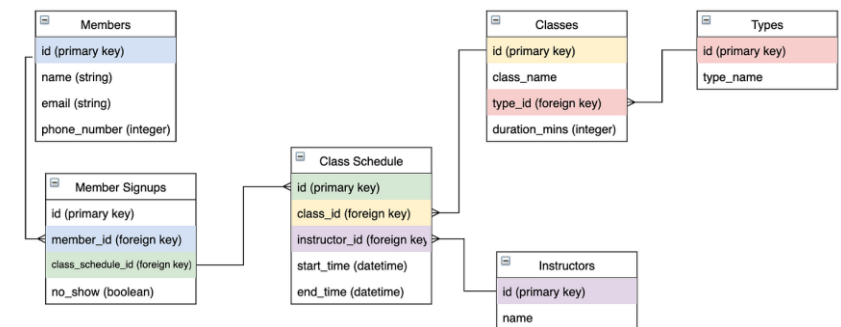
Alcuni Database relazionali usati:

MySQL

PostgreSQL

Microsoft SQL Server

IBM DB2

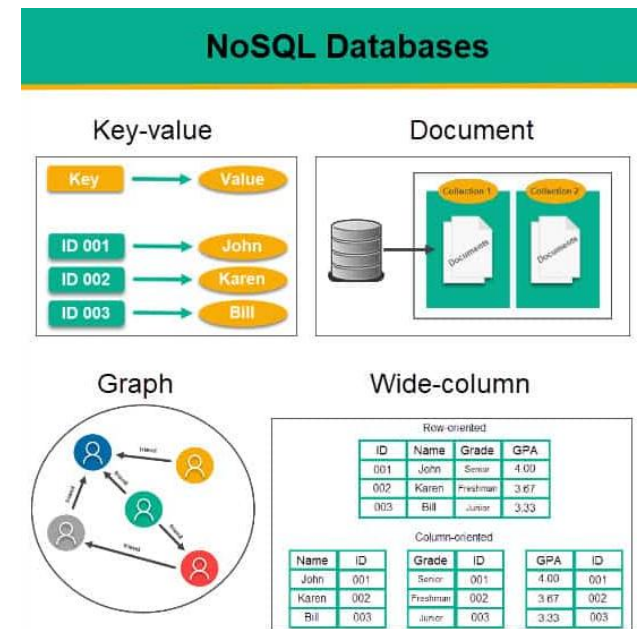


# TIPI DI DATABASE

- I Database **NoSQL** sono una tipologia di Database che, come indica il nome, **non usa il linguaggio SQL per consentire una gestione dei dati** all'interno di una struttura flessibile e variabile.
- Alcune caratteristiche di questi database sono:
  - La **mancaanza di vincoli** sulla struttura dei dati contenuti
  - La **scalabilità orizzontale**, che consente di incrementare in modo quasi illimitato la capacità del Database
  - La capacità di **contenere diversi formati come documenti e grafici**
  - La **mancaanza** di una definizione **di uno schema per i dati**, il che significa che i dati possono essere aggiunti o modificati senza dover modificare la struttura del database

## Alcuni Database NoSQL usati:

MongoDB  
Cassandra  
Redis  
Neo4j



# COMPONENTI DI UN DATABASE RELAZIONALE

La **tabella**, l'insieme organizzato di righe e colonne

Una **riga (record)**, rappresenta un record o una tupla di dati

Un **indice primario**, una colonna la quale utilità è di facilitare la ricerca di un record specifico e per questo il contenuto (chiave primaria) deve essere univoco da tutti gli altri record

Una **colonna**, rappresenta un campo del record

SALES				
purchase_number	date_of_purchase	customer_id	item_code	
1	03/09/2016	1	A_1	
2	02/12/2016	2	C_1	
3	15/04/2017	3	D_1	
4	24/05/2017	1	B_2	
5	25/05/2017	4	B_2	
6	06/06/2017	2	B_1	
7	10/06/2017	4	A_2	
8	13/06/2017	3	C_1	
9	20/07/2017	1	A_1	
10	11/08/2017	2	B_1	

Una **chiave esterna**, rappresenta un indice di un'altra tabella e serve a collegare più tabelle



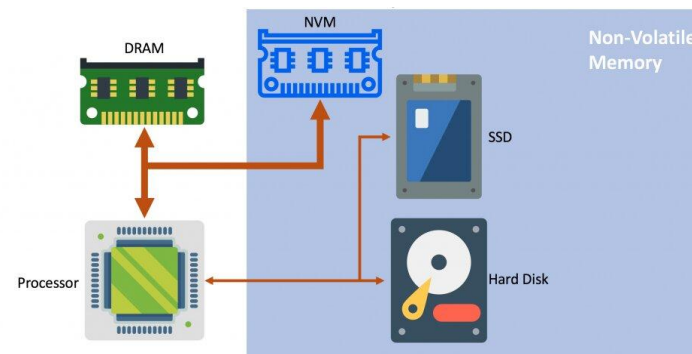
# CONCETTI DI BASE

- Per capire come funzionano i Database è necessario comprendere alcuni concetti base.

- **Persistenza**

Nei Database il concetto di persistenza si riferisce alla **capacità di mantenere dati anche in caso di arresto anomalo o non.**

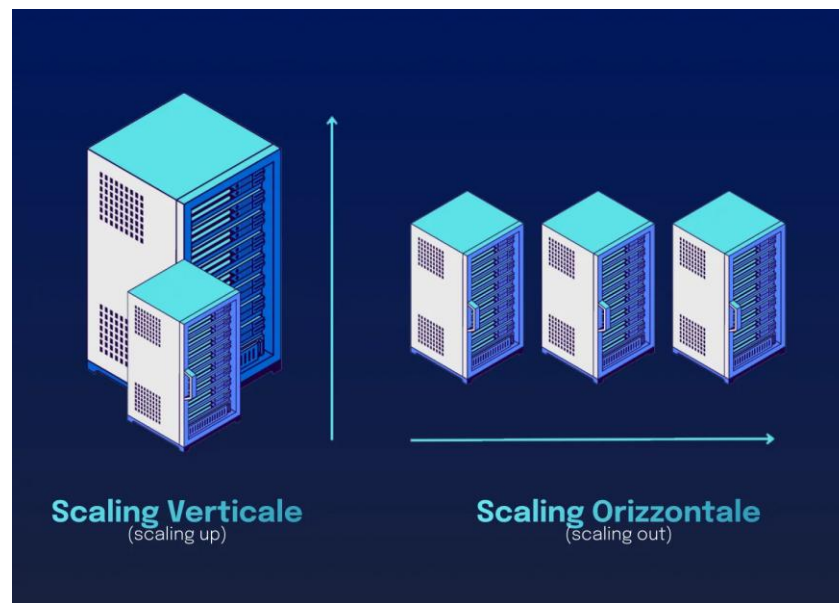
- I Database riescono a soddisfare questo concetto **memorizzando i dati all'interno di memoria non volatile come dischi rigidi o SSD** piuttosto che in memoria volatile come RAM o Cache



# CONCETTI DI BASE

- **Scalabilità**

- I Database **relazionali** sotto l'aspetto della scalabilità hanno la capacità di **scalare solo in maniera verticale**, cioè per **incrementare le prestazioni o la memoria del Database l'unica possibilità è quella di incrementare le risorse del server dove è installato**, con CPU o maggiore storage. Ciò può essere uno svantaggio nel caso si deve mantenere una quantità di dati molto elevata, sia a livello di costi che di prestazioni



# CONCETTI DI BASE

- Integrità
- I Database relazionali sotto l'aspetto dell'integrità hanno la capacità di garantire la coerenza e l'attendibilità dei dati attraverso la **definizione di vincoli e regole di integrità**. Tuttavia, ciò può essere limitato nel caso di dati molto complessi o distribuiti, dove la gestione dell'integrità può diventare difficile e costosa. Inoltre, **la perdita di dati o la corruzione dei dati può essere un problema serio nel caso di database relazionali, specialmente se non sono implementate adeguate misure di backup e ripristino.**

integrità sulle colonne	Restrizioni sui valori assunti da una colonna
integrità sulle tabelle	Restrizioni sui valori assunti da tutte le righe di una tabella
integrità referenziale	Restrizioni sui valori assunti dalle colonne in comune delle tabelle in relazione

# NORMALIZZAZIONE

La normalizzazione è un processo di progettazione dei database relazionali che mira a organizzare i dati per:

- Ridurre la **ridondanza**.
- Garantire la **coerenza**.
- Migliorare **l'efficienza** delle operazioni di aggiornamento, inserimento e cancellazione.

La normalizzazione si basa sulle **forme normali**, una serie di regole che determinano il livello di organizzazione di una tabella.

# NORMALIZZAZIONE

## Prima Forma Normale

Una tabella è in **Prima Forma Normale (1NF)** se:

1. Tutte le colonne contengono **valori atomici** (non divisibili).
2. Ogni riga è identificata da un **valore univoco** (es. una chiave primaria).

ID	Nome	Telefoni
1	Mario	123456, 789012
2	Anna	345678, 901234

VIOLAZIONE!!!

SOLUZIONE

ID	Nome	Telefono
1	Mario	123456
1	Mario	789012
2	Anna	345678
2	Anna	901234



# NORMALIZZAZIONE

## Seconda Forma Normale

- Una tabella è in **Seconda Forma Normale (2NF)** se:

1. È già in **1NF**.

2. Ogni attributo (non chiave) è **completamente dipendente** dalla chiave primaria.

**Avviene normalmente in caso di chiavi composte**

- Esempio di Tabella NON in 2NF

- Immagina una tabella per gestire gli ordini di un negozio:

OrderID	ProductID	ProductName	Quantity	UnitPrice
101	A1	Laptop	2	1200
101	A2	Mouse	1	20
102	A1	Laptop	1	1200
102	A3	Keyboard	1	50

- Analisi:

- La chiave primaria è composta da **OrderID e ProductID**.
- L'attributo **ProductName** e **UnitPrice** dipendono solo da **ProductID**, non dalla combinazione completa della chiave primaria (**OrderID, ProductID**).
- Questa dipendenza parziale viola la 2NF.

Tabella Orders

OrderID	ProductID	Quantity
101	A1	2
101	A2	1
102	A1	1
102	A3	1

Tabella Products

ProductID	ProductName	UnitPrice
A1	Laptop	1200
A2	Mouse	20
A3	Keyboard	50

# NORMALIZZAZIONE

## Terza Forma Normale

- Una tabella è in **Terza Forma Normale (3NF)** se:

1. È già in **2NF**.

2. Esempio corretto:

Tabella: **Ordini**

ID_Ordine	Cliente_ID	Prodotto	Prezzo
1	101	Laptop	1000
2	102	Smartphone	700
3	101	Tablet	500

Tabella: **Clienti**

Cliente_ID	Cliente	Indirizzo_Cliente	Città_Cliente	CAP_Cliente
101	Mario Rossi	Via Roma 10	Roma	00100
102	Anna Verdi	Corso Italia 50	Milano	20100

1. Non contiene **dipendenze transitive** (attributi non chiave che **dipendono da chiavi non chiave nella tabella**)

### Esempio di una Tabella che Viola la 3NF

Tabella: **Ordini**

ID_Ordine	Cliente	Indirizzo_Cliente	Città_Cliente	CAP_Cliente	Prodotto	Prezzo
1	Mario Rossi	Via Roma 10	Roma	00100	Laptop	1000
2	Anna Verdi	Corso Italia 50	Milano	20100	Smartphone	700
3	Mario Rossi	Via Roma 10	Roma	00100	Tablet	500

Nella tabella degli Ordini, c'è una dipendenza transitiva:

La chiave primaria è ID\_Ordine.

L'attributo Indirizzo\_Cliente dipende da Cliente, che a sua volta dipende da ID\_Ordine.

Questa è una dipendenza transitiva:

ID\_Ordine → Cliente → Indirizzo\_Cliente.

Secondo la 3NF, tutti gli attributi non chiave devono dipendere direttamente dalla chiave primaria (ID\_Ordine). Per risolvere, abbiamo separato i dati dei clienti in una nuova tabella.

# NORMALIZZAZIONE

## FORMA NORMALE DI BOYCE E CODD

- UNA TABELLA È IN **FORMA NORMALE DI BOYCE E CODD (BCNF)** SE:

1. È GIÀ IN **2NF**.
2. E IN ESSA TUTTI I DETERMINANTI POSSONO ESSERE CHIAVI CANDIDATE L CIOÈ OGNI ATTRIBUTO Y DAL QUALE DIPENDONO ALTRI ATTRIBUTI Z PUÒ SVOLGERE LA FUNZIONE DI CHIAVE.
3. ESEMPIO CORRETTO:

1. Tabella Corsi:

Corso	Professore
CS101	Dr. Rossi
CS102	Dr. Bianchi
MATH101	Dr. Verdi

2. Tabella Professori:

Professore	Dipartimento
Dr. Rossi	Informatica
Dr. Bianchi	Informatica
Dr. Verdi	Matematica

PER APPLICARE QUESTA FORMA NORMALE INIZIAMO DA UNA TABELLA

Corso	Professore	Dipartimento
CS101	Dr. Rossi	Informatica
CS102	Dr. Bianchi	Informatica
MATH101	Dr. Verdi	Matematica

QUESTA TABELLA CONTIENE 2 DIPENDENZE:

- PROFESSORE DIPENDE DAL CORSO
- DIPARTIMENTO DIPENDE DAL PROFESSORE

IN QUESTO CASO LA COLONNA PROFESSORE NON PUÒ ESSERE UNA CHIAVE DATO CHE PIÙ DI UN CORSO PUÒ AVERE LO STESSO PROFESSORE, QUESTO VIOLA LA BCNF.

DIVIDENDO LA TABELLA IN DUE DOVE UNA CONTIENE LA DETERMINANTE DEL PROFESSORE E L'ALTRA LA DETERMINANTE DEL DIPARTIMENTO

# CAPITOLO 2: COME FUNZIONANO I DATABASE

# COME FUNZIONANO I DATABASE

I database sono il cuore delle moderne applicazioni informatiche, gestendo enormi quantità di dati in modo efficiente, sicuro e organizzato.

- Architettura e gestione dello **storage**.
- Gestione delle **transazioni** e paradigmi ACID vs BASE.
- **Ottimizzazione** tramite **indici** e tecniche di **caching**.
- **Concorrenza** e isolamento delle transazioni.
- **Sicurezza** e controllo degli accessi.
- Operazioni **CRUD** e paradigmi di programmazione.

# ARCHITETTURA DI UN DATABASE

Un database è progettato su un'architettura stratificata che include:

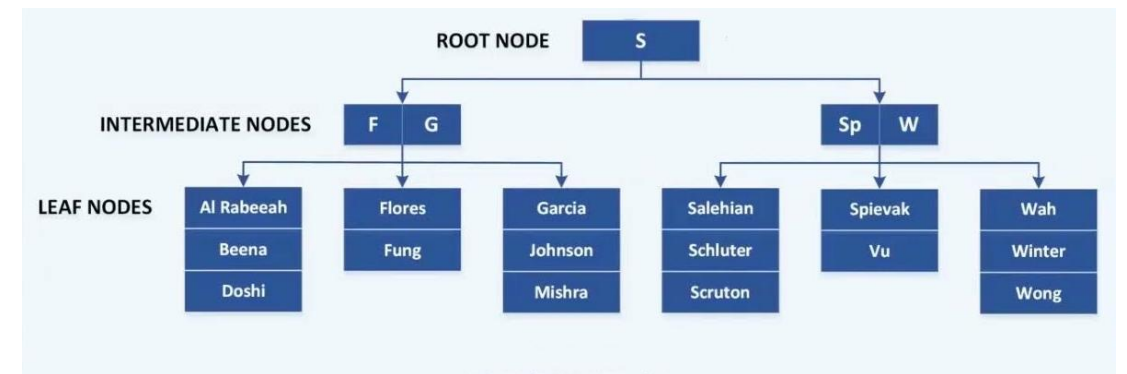
- Livelli di **Storage**: **Dati memorizzati fisicamente su dischi o SSD** e gestiti logicamente in tabelle e colonne.
- Indici: Strutture specializzate che **migliorano la velocità di accesso ai dati**.
- **Cache**: **Memorizzazione temporanea dei dati più frequentemente usati** per accelerare l'accesso.

Questa architettura consente di bilanciare efficienza e scalabilità, garantendo **prestazioni elevate anche con grandi volumi di dati**.

# OTTIMIZZAZIONE E INDICI NEI DATABASE

Gli indici sono strumenti essenziali per **ottimizzare le prestazioni del database**.

- Cosa sono gli indici?
  - Strutture di dati che migliorano la velocità delle query di ricerca, simili all'indice di un libro.
- Tipi di Indici:
  - **B-Tree**: Per ricerche e ordinamenti efficienti
    - (BETWEEN, <, >, ORDER BY, LIKE 'prefix%')
  - **Hash**: Per accesso diretto a valori specifici.
    - Ottimizzato per confronti di uguaglianza (es. =).
- Gestione degli Indici:
  - Gli indici devono **essere aggiornati in tempo reale**.
  - **Possono aumentare il costo di operazioni come INSERT e UPDATE.**



Una progettazione ottimale degli indici riduce il tempo di risposta e migliora l'esperienza dell'utente.

# GESTIONE DELLA CONCORRENZA

La gestione della concorrenza garantisce che più transazioni possano accedere ai dati in parallelo senza interferenze.

- Locking:
  - Shared Lock: Permette letture condivise, ma blocca le scritture.
  - Exclusive Lock: Impedisce sia letture che scritture da altre transazioni.
- Isolamento delle Transazioni



# SICUREZZA E ACCESSO AI DATI

La sicurezza dei dati è una priorità nei database, con strumenti per controllare chi può accedere e cosa può fare:

- **Autenticazione**: Verifica **dell'identità degli utenti** tramite credenziali.
- **Autorizzazione**: Controllo dei **privilegi di accesso** e modifica ai dati.
- **Crittografia**: Protezione dei **dati in transito** e a riposo.
- **Audit**: Registrazione delle attività per **identificare eventuali violazioni**.

Una buona sicurezza combina **politiche di accesso rigorose e tecnologie avanzate** per proteggere i dati sensibili.

# GESTIONE DELLE TRANSAZIONI – ACID VS BASE

La gestione delle transazioni è essenziale per mantenere l'integrità dei dati:

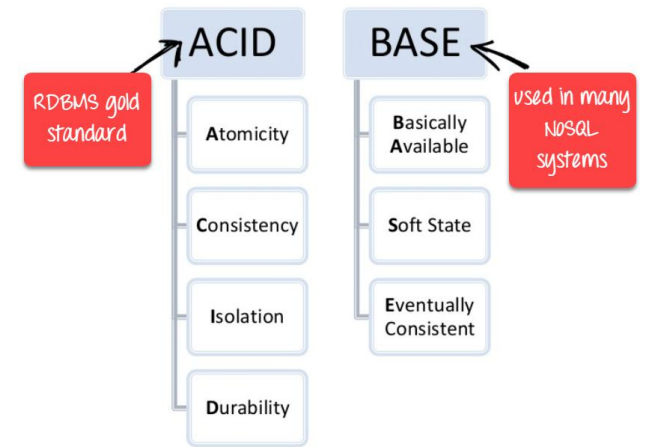
- Modello ACID (Relazionale):

- Atomicità: Le transazioni sono indivisibili.
- Consistenza: I dati restano validi.
- Isolamento: Nessuna interferenza tra transazioni.
- Durabilità: Le modifiche sono permanenti.

- Modello BASE (NoSQL):

- Basically Available: sempre disponibili per essere letti o scritti anche in caso di guasti parziali del sistema
- Soft State: Stato temporaneamente incoerente perché i dati sono distribuiti su più nodi e potrebbero non essere sincronizzati in tempo reale.
- Eventual Consistency: nel tempo, i dati diventeranno coerenti su tutti i nodi, ma non immediatamente.

Questi modelli si adattano a diverse necessità: ACID per applicazioni critiche, BASE per sistemi distribuiti e scalabili.



# NOMENCLATURE E STANDARD IN PROGRAMMAZIONE

- **Pascal Case** (es.: UserName)

- Definizione: Ogni parola inizia con una lettera maiuscola.
- Pro:
  - Facile da leggere e intuitivo.
  - Convenzione comune in linguaggi orientati agli oggetti (es.: C#, .NET).
- Contro:
  - **Meno comune nei database SQL.**
  - Può risultare incoerente se si lavora con ambienti che usano diversi stili.

- **Camel Case** (es.: userName)

- Definizione: La prima parola inizia con una lettera minuscola, e ogni parola successiva inizia con una maiuscola.
- Pro:
  - Comune in linguaggi come JavaScript e Java.
  - Facile da associare a variabili in codice applicativo.
- Contro:
  - **Meno usato nei database**, dove spesso si preferisce uno stile più chiaro come lo **snake case**.

# NOMENCLATURE E STANDARD IN PROGRAMMAZIONE

- **Snake Case** (es.: user\_name)
  - Definizione: Le parole sono tutte minuscole e separate da un underscore \_.
  - Pro:
    - Convenzione più diffusa nei database relazionali (SQL).
    - Evita problemi di case sensitivity nei database che non fanno distinzione tra maiuscole e minuscole.
    - Facile da leggere, soprattutto in query complesse.
  - Contro:
    - Meno elegante se il progetto utilizza linguaggi orientati agli oggetti che preferiscono Pascal o Camel Case.
- customer\_id
- order\_id
- order\_number

# DIAGRAMMI ERD

- Un diagramma entità-relazione (diagramma ER o ER DIAGRAM) è una rappresentazione visiva di come gli elementi di un database si relazionano tra loro. Gli ERD sono un tipo specializzato di diagramma di flusso che spiega i tipi di relazione tra diverse entità all'interno di un sistema.
- stabiliscono il modo in cui le entità del mondo reale verranno modellate in un database relazionale

- Un'entità ERD è qualcosa di definibile, come una persona, un ruolo, un evento, un concetto o un oggetto, che può avere informazioni
- ESEMPIO: utenti, persone, prodotti, fatture, ordini ETC...
- Le entità sono classificate come **forti o deboli**.
- Un'entità **forte** è un'entità che **può esistere autonomamente**, senza la necessità di altre entità per definirla.
- Un'entità **debole**, invece, **dipende da un'altra entità** per la sua esistenza e non ha una propria chiave primaria.
- Entità forte: **clienti**: cliente è un'entità forte perché può esistere autonomamente. Ogni cliente ha un identificatore unico, ad esempio un codice cliente (cliente\_id)
- L'ORDINE di un cliente (**ordini**) è un'entità debole perché non può esistere senza un Cliente. Un ordine è identificato attraverso la combinazione del suo ID (ordine\_id) e il cliente\_id (chiave esterna che collega l'ordine al cliente).

# ENTITA ASSOCIATIVE

Un'entità associativa, utilizzata per rappresentare una relazione multi-a-molti tra due (o più) entità principali, è una sorta di "ponte" tra entità correlate.

**Ha una chiave primaria che è solitamente una combinazione delle chiavi primarie delle entità coinvolte nella relazione.**

Può avere anche propri attributi che non appartengono alle entità

- Entità forti:

- **studenti** (con studente\_id come chiave primaria)
- **corsi** (con corso\_id come chiave primaria)

- Entità Associativa:

- Tabella **iscrizioni** (entità associativa che rappresenta l'iscrizione dello studente al corso).
- Entità associativa: **iscrizioni**
  - Attributi: data\_iscrizione, voto\_finale
  - Chiave primaria: combinazione di **studente\_id e corso\_id** (questi sono le chiavi esterne dalle entità studenti e corsi).

- Gli attributi sono qualità, proprietà e caratteristiche che definiscono un'entità o un tipo di entità. In un progetto ERD classico, gli attributi vengono visualizzati come ovali e vengono visualizzati accanto all'entità corrispondente in un ER
  - **Gli attributi semplici** non possono essere semplificati o suddivisi in ulteriori attributi
    - ES: CAP
  - **Gli attributi composti** vengono compilati da altri attributi, che possono essere semplici o meno.
    - Es: Un indirizzo è un attributo composto contenente un numero civico e una via/piazza
  - Gli **attributi derivati** sono calcolati in base ad altri attributi. Il valore della busta paga di un dipendente deriva dalle ore lavorate, dalla durata del periodo di retribuzione e dal salario
    - Ellisse tratteggiate



# ATTRIBUTI CHIAVE

- Le chiavi di entità sono gli attributi che definiscono in modo univoco ciascuna entità in un set di dati
- **Superchiave:** uno o più attributi che possono essere utilizzati per identificare univocamente una riga nella tabella
- **Chiave candidata:** la superchiave la più piccola combinazione di colonne che identifica univocamente ogni riga
- **Chiave primaria:** la chiave candidata scelta per definire in modo univoco un set di entità. Poiché la chiave primaria è ciò che distingue ogni entità, non è possibile che due voci in un database condividano lo stesso valore di chiave primaria
  - In un diagramma ER, la chiave primaria di ogni entità sarà sottolineata
- **Chiave esterna:** un attributo che identifica la relazione di un'entità con un'altra. Le entità deboli si basano su chiavi esterne per definirsi come entità forti

## Esempi di superchiavi:

- {ID\_Studente}
- {Codice\_Fiscale}
- {Email}
- {ID\_Studente, Nome}
- {ID\_Studente, Cognome, Telefono}

## Esempi di chiavi candidate:

- {ID\_Studente}
- {Codice\_Fiscale}
- {Email}

## Esempio di chiave primaria:

- Si sceglie {ID\_Studente}

- Le relazioni sono le linee collegate che collegano tra loro le entità in un ERD. Indicano il modo in cui le entità all'interno di un ERD sono associate tra loro
- **Cardinalità delle relazioni**
  - Le **relazioni uno a uno (1:1)** indicano che un record all'interno di un'entità può essere referenziato solo da un record dell'altra entità.
    - Il rapporto tra ITS e presidente è un rapporto uno a uno
  - Le **relazioni uno-a-molti (1:M)** descrivono situazioni in cui ogni record all'interno di un'entità si riferisce a più record di un'altra entità
    - Categorie e prodotti sono 1:M
  - Le **relazioni multi-a-molti (M:M)** mostrano che uno o più record all'interno di entrambe le entità possono essere connessi.
    - Prodotti e Ordini (un ordine può avere più Prodotto e un prodotto essere in più ordini)

# MODELLI ENTITÀ-RELAZIONE

- **I modelli ER concettuali** offrono una visione di alto livello dei dati.  
I modelli di **dati concettuali** solitamente **contengono entità e relazioni, senza addentrarsi ulteriormente nelle tabelle** e nella cardinalità del database
- **I modelli ER logici** sono simili ai modelli concettuali, ma con un più dettagli.
  - **vengono definite le colonne o gli attributi di ciascuna entità,**
- **I modelli ER fisici** sono i **progetti concreti** per i progetti di progettazione di database. Includono la quantità massima di dettagli, ad esempio la cardinalità e le chiavi primarie ed esterne.

# MODELLO CONCETTUALE, LOGICO E FISICO

## • Modello Concettuale

- Rappresenta l'idea generale del sistema informativo e i concetti principali da gestire.

## • Modello Logico

- Specifica la struttura dei dati in termini logici, indipendente da un particolare sistema di database, ma con dettagli più precisi rispetto al modello concettuale.

## • Modello Fisico

- Rappresenta la struttura effettiva del database così come verrà implementata in un DBMS specifico.

Modello ER  
Concettuale

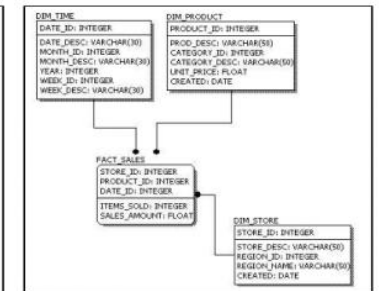
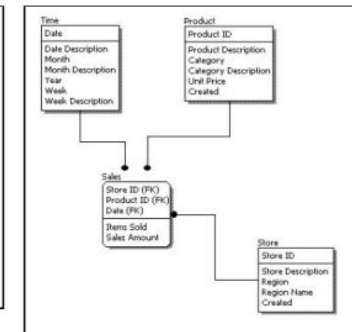
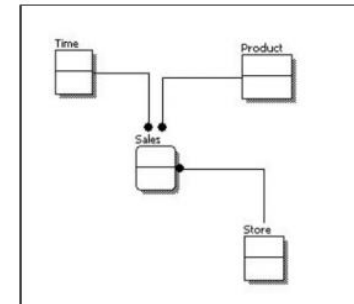
Modello ER  
Logico

non è più un  
modello er  
ma un  
database  
REALE

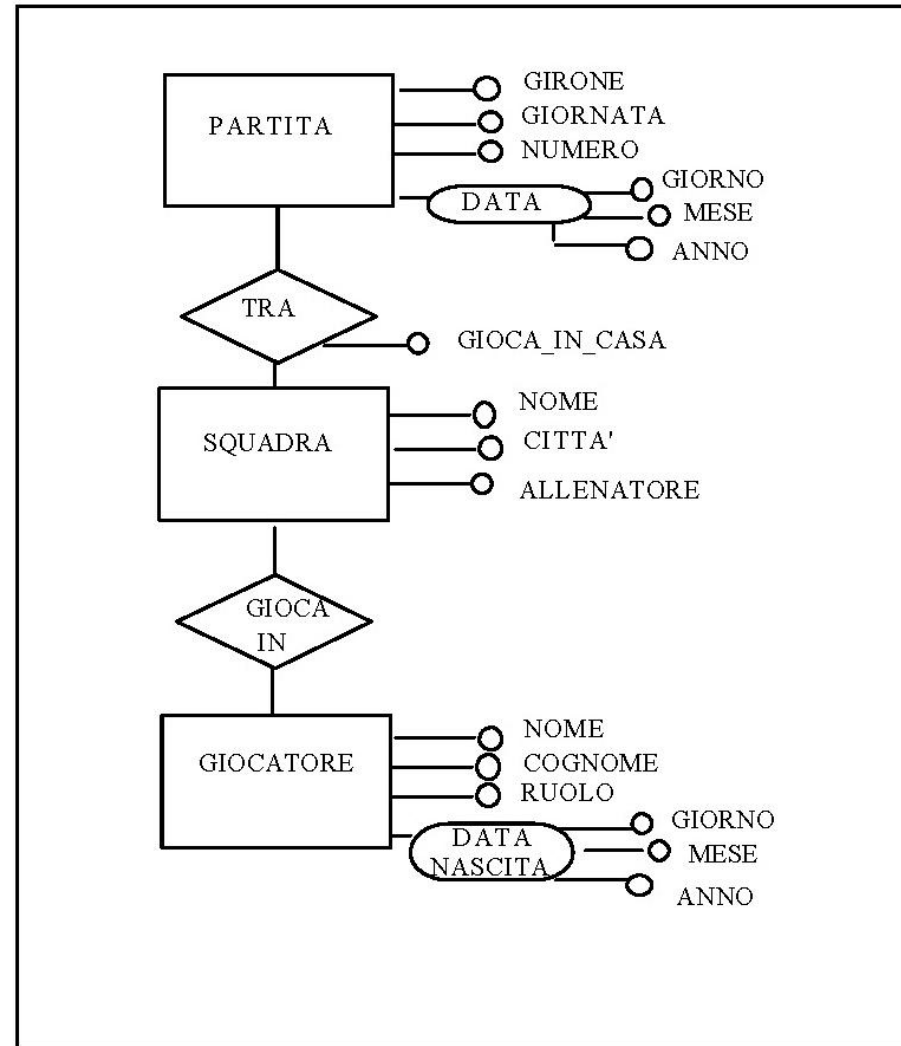
Conceptual Model Design

Logical Model Design

Physical Model Design



# DIAGRAMMA ER – MODELLO LOGICO



# RELAZIONI IDENTIFYING E NON IDENTIFYING

## • Relazione Non Identifying (tratteggiata)



1. È una relazione in cui l'entità figlia **non dipende dal genitore** per la propria identificazione.

- Tabella clienti (Padre):

- id\_cliente (PK)

- nome

- cognome

- Tabella ordini (Figlia):

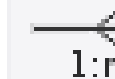
- id\_ordine (PK)

- id\_cliente (FK)

- data\_acquisto

La chiave primaria della tabella ordini (id\_ordine) è indipendente dalla chiave primaria della tabella clienti (id\_cliente)

## Relazione Identifying



1. È una relazione in cui l'entità figlia **dipende strettamente dall'entità genitore** per la propria identificazione.

Tabella ordini (Padre):

id\_ordine (PK)

data\_acquisto

Tabella dettagli\_ordine (Figlia):

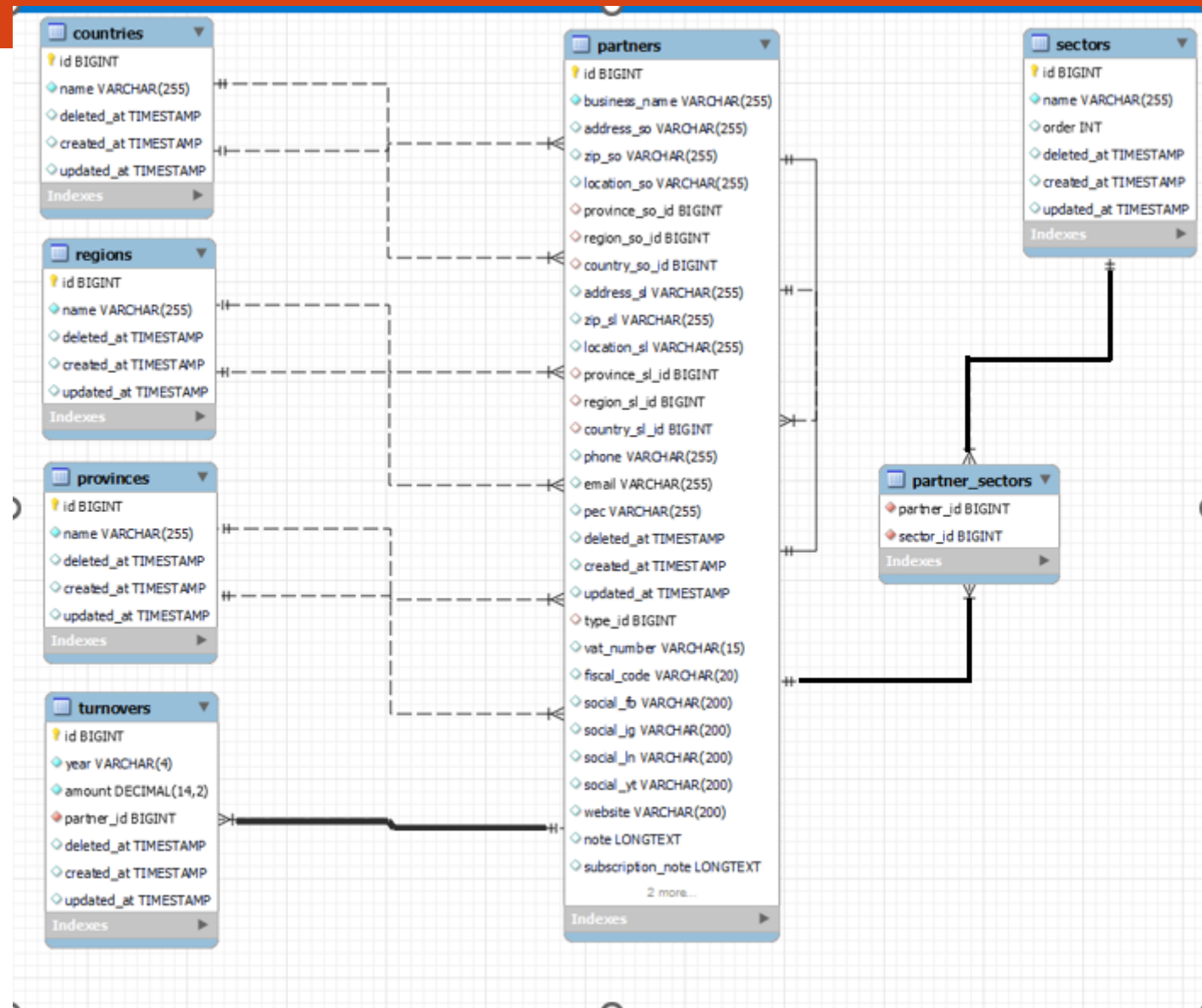
id\_ordine (PK, FK)

id\_prodotto (PK)

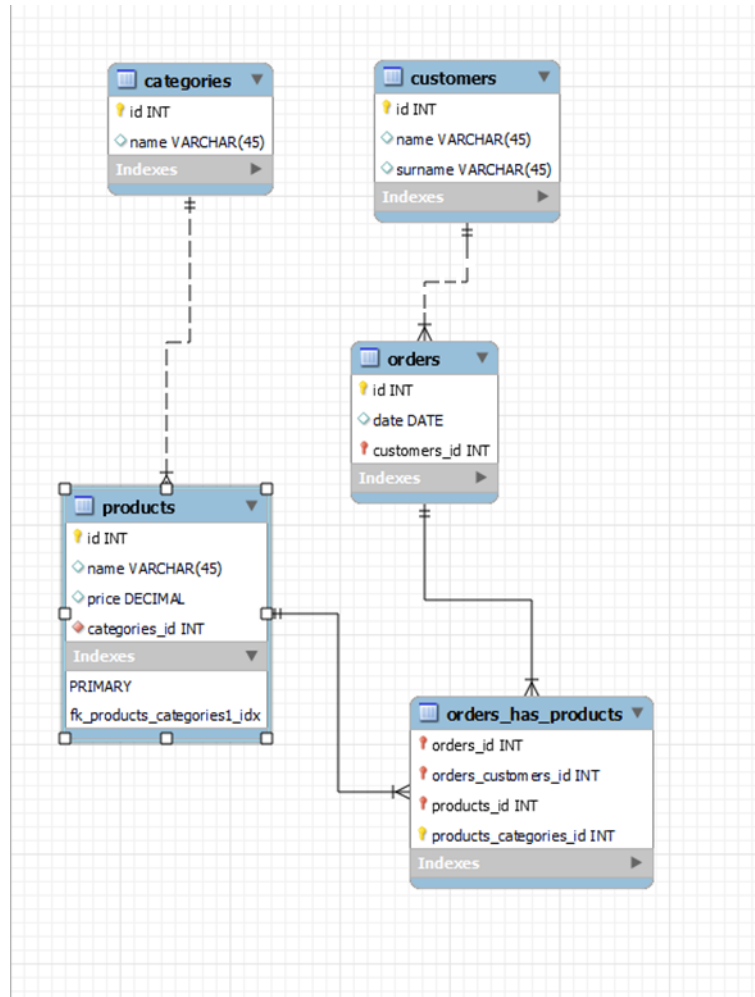
Quantita

la chiave primaria di dettagli\_ordine include una parte della chiave primaria di ordini, **un dettaglio ordine non può esistere senza ordine**

# ESEMPIO PARTNERS

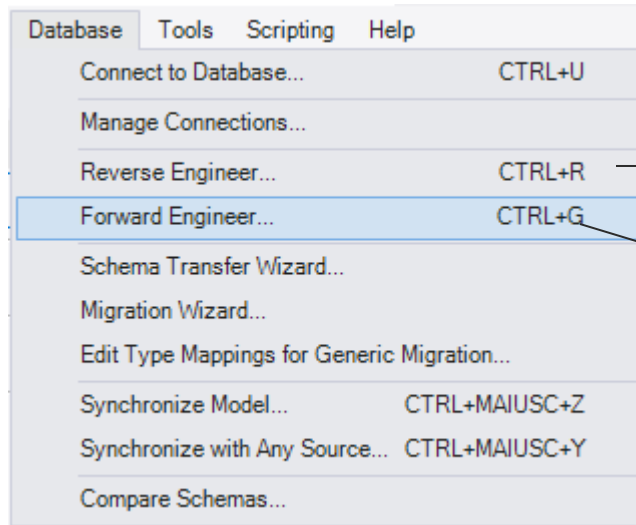


# ESEMPIO DA EER DIAGRAM (FISICO) MYSQL WORKBENCH





# MYSQL WORKBENCH E IL MODELLO ER



Da DB a modello ER

Da modello ER a DB

# CLI DI MYSQL

- Aprire cmd e posizionarsi nella cartella bin
- Cd C:\Program Files\MySQL\MySQL Server 8.2\bin
- > mysql mysql -u tuo\_utente -p
- > scrivere pwd
- > use nomedb;
- > select \* from tabella;

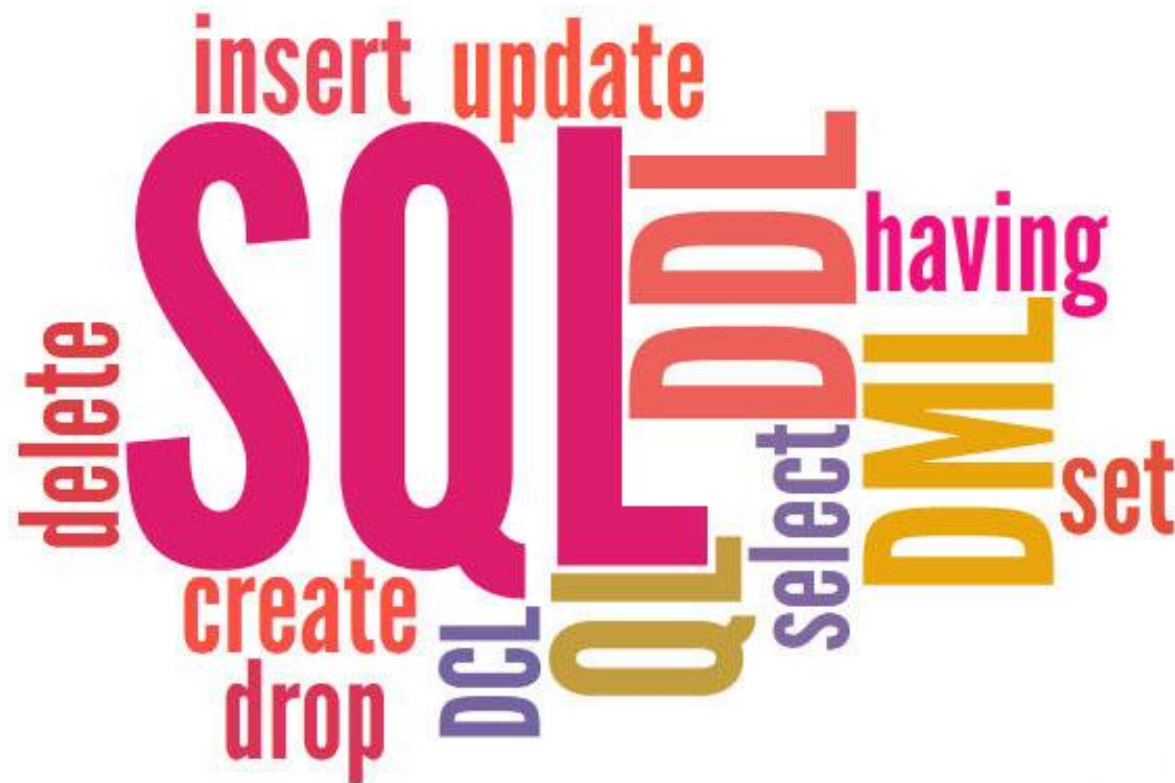
# CAPITOLO 3: STRUTTURA DEI LINGUAGGI DI DATABASE

# SQL E I LINGUAGGI RELAZIONALI

SQL, acronimo di **Structured Query Language**, è lo standard per l'interazione con i database relazionali.

Si distingue come un linguaggio dichiarativo, in cui si specifica cosa ottenere, delegando al sistema il come.

I database relazionali **si basano su un modello tabellare**, dove i dati sono organizzati in **righe** (tuple) e **colonne** (attributi). Questo approccio consente flessibilità e coerenza nella gestione di grandi quantità di informazioni, rendendo SQL uno strumento universale per molteplici sistemi, come MySQL, PostgreSQL e SQL Server.



# MYSQL - TIPI DI CAMPI: TESTI

Quando si crea o modifica un campo è necessario specificare il tipo di dato che sarà contenuto, alcuni dei più usati sono:

- **TINYTEXT**: fino a 255 byte
- **TEXT**: fino a 65,535 byte (circa 64 KB).
- **MEDIUMTEXT**: fino a 16,777,215 byte (circa 16 MB
- **LONGTEXT**: fino a 4,294,967,295 byte (circa 4 GB).

# MYSQL - TIPI DI CAMPI: NUMERI

- **TINYINT:** Intero molto piccolo.
  - Intervallo (SIGNED): -128 a 127
  - Intervallo (UNSIGNED): 0 a 255
  - Occupazione: 1 byte
- **SMALLINT:** Intero piccolo.
  - Intervallo (SIGNED): -32,768 a 32,767
  - Intervallo (UNSIGNED): 0 a 65,535
  - Occupazione: 2 byte
- **MEDIUMINT:** Intero medio.
  - Intervallo (SIGNED): -8,388,608 a 8,388,607
  - Intervallo (UNSIGNED): 0 a 16,777,215
  - Occupazione: 3 byte
- **INT (o INTEGER):** Intero standard.
  - Intervallo (SIGNED): -2,147,483,648 a 2,147,483,647
  - Intervallo (UNSIGNED): 0 a 4,294,967,295
  - Occupazione: 4 byte
- **BIGINT:** Intero molto grande.
  - Intervallo (SIGNED): -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
  - Intervallo (UNSIGNED): 0 a 18,446,744,073,709,551,615
  - Occupazione: 8 byte

# MYSQL - TIPI DI CAMPO: NUMERI DECIMALI

- **DECIMAL(p, s) o NUMERIC(p, s):** Tipo per numeri decimali con precisione esatta.
- **p** è la precisione, cioè il numero totale di cifre.
- **s** è la scala, cioè il numero di cifre dopo il punto decimale.
- Ad esempio, **DECIMAL(10, 2)** può memorizzare numeri con fino a 10 cifre totali, di cui 2 dopo il punto decimale (ad esempio, 12345678.90).
- **Occupazione:** La dimensione in byte dipende dalla precisione (p). Ad esempio:
  - **DECIMAL(10,2)** occupa 5 byte.
  - **DECIMAL(65,30)** occupa 20 byte.
- **FLOAT:** Tipo per numeri in virgola mobile con precisione approssimativa.
  - Occupazione: 4 byte
  - Intervallo: da circa  $-3.402823466E+38$  a  $3.402823466E+38$ .
- **DOUBLE:** Tipo per numeri in virgola mobile a doppia precisione (più preciso di FLOAT).
  - Occupazione: 8 byte
  - Intervallo: da circa  $-1.7976931348623157E+308$  a  $1.7976931348623157E+308$ .
  - Anche questo tipo è approssimato, ma con maggiore precisione rispetto a FLOAT

# RIEPILOGO TIPI NUMERICI IN MYSQL

Tipo	Intervallo (SIGNED)	Intervallo (UNSIGNED)	Occupazione
TINYINT	-128 a 127	0 a 255	1 byte
SMALLINT	-32,768 a 32,767	0 a 65,535	2 byte
MEDIUMINT	-8,388,608 a 8,388,607	0 a 16,777,215	3 byte
INT	-2,147,483,648 a 2,147,483,647	0 a 4,294,967,295	4 byte
BIGINT	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	0 a 18,446,744,073,709,551,615	8 byte
DECIMAL	Variabile (p, s)	Variabile (p, s)	Variabile, dipende dalla precisione
FLOAT	-3.402823466E+38 a 3.402823466E+38	-	4 byte
DOUBLE	-1.7976931348623157E+308 a 1.7976931348623157E+308	-	8 byte

## Quando usare quale tipo:

- Usa `INT` per numeri interi.
- Usa `DECIMAL` per valori monetari o quando la precisione esatta è importante.
- Usa `FLOAT` o `DOUBLE` per valori scientifici o per quando una piccola imprecisione è accettabile.





# MYSQL - TIPI PER DATE E ORARI

- **DATE**: Memorizza una data nel formato YYYY-MM-DD.
  - Intervallo: 1000-01-01 a 9999-12-31
  - Occupazione: 3 byte
  - Esempio: 2024-12-07
- **DATETIME**: Memorizza una data e un'ora nel formato YYYY-MM-DD HH:MM:SS.
  - Intervallo: 1000-01-01 00:00:00 a 9999-12-31 23:59:59
  - Occupazione: 8 byte
  - Esempio: 2024-12-07 15:30:00
- **YEAR**: Memorizza un anno nel formato YYYY.
  - Intervallo: 1901 a 2155
  - Occupazione: 1 byte
- **TIMESTAMP**: Memorizza una data e un'ora con l'ora UTC, che può essere automaticamente aggiornato dal database. Simile a DATETIME, ma con una gestione speciale dei fusi orari.
  - Intervallo: 1970-01-01 00:00:01 UTC a 2038-01-19 03:14:07 UTC
  - Occupazione: 4 byte
  - Esempio: 2024-12-07 15:30:00
- **TIME**: Memorizza solo l'ora nel formato HH:MM:SS.
  - Intervallo: -838:59:59 a 838:59:59
  - Occupazione: 3 byte
  - Esempio: 15:30:00

# MYSQL - TIPI PER VALORI BOLEANI

- **BOOLEAN** (alias TINYINT(1)): Memorizza valori booleani, dove 0 rappresenta FALSE e 1 rappresenta TRUE. In realtà, MySQL lo memorizza come un intero di 1 byte, ma è comunemente usato per valori logici.
- Intervallo: 0 (FALSE) o 1 (TRUE)
- Occupazione: 1 byte
- Esempio: TRUE o FALSE

# MYSQL - TIPI PER BINARI (BLOB)

- **BLOB** (Binary Large Object): Memorizza dati binari di dimensione variabile, come file immagine o video.
  - Occupazione: Varia in base alla lunghezza del contenuto binario (fino a 65,535 byte).
  - Esempio: Un'immagine, un file PDF.
- **TINYBLOB**: Memorizza dati binari fino a 255 byte.
  - Occupazione: 1 byte + lunghezza del contenuto
  - Esempio: Piccole immagini o file binari.
- **MEDIUMBLOB**: Memorizza dati binari fino a 16,777,215 byte.
  - Occupazione: 3 byte + lunghezza del contenuto
  - Esempio: Video o file audio.
- **LONGBLOB**: Memorizza dati binari molto grandi, fino a 4 GB.
  - Occupazione: 4 byte + lunghezza del contenuto
  - Esempio: File video, file di grandi dimensioni.

# MYSQL - TIPI PER ENUM E SET

- **ENUM**: Tipo di dato per valori predefiniti, utilizzato per memorizzare una lista di valori possibili (come un campo che può essere solo "Sì" o "No").
  - Esempio: ENUM('Sì', 'No')
  - **UN SOLO VALORE**
  - Occupazione: 1 byte per un massimo di 255 valori.
- **SET**: Tipo di dato per memorizzare **UNO O PIÙ VALORI** da un insieme di valori predefiniti.
  - Esempio: SET('Red', 'Green', 'Blue')
  - Occupazione: Varia a seconda del numero di valori selezionati.

# MYSQL - TIPI DI DATI JSON

- **JSON**: Memorizza dati JSON in formato nativo. Permette di archiviare oggetti o array JSON e di eseguire operazioni come l'estrazione di dati da un campo JSON.
  - Esempio: {"name": "John", "age": 30}
  - Occupazione: Dipende dal contenuto JSON.

# RIEPILOGO: QUANDO USARE QUALE TIPO

- DATE, DATETIME, TIMESTAMP: Per memorizzare dati relativi a date e orari.
- BOOLEAN: Per memorizzare valori logici (vero o falso).
- CHAR, VARCHAR: Per memorizzare stringhe, scegli CHAR per lunghezze fisse e VARCHAR per lunghezze variabili.
- TEXT e BLOB: Per dati di testo o binari di grandi dimensioni.
- ENUM e SET: Per valori predefiniti e set di valori.
- JSON: Per memorizzare oggetti o array JSON.

# LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

I linguaggi DDL sono utilizzati per **definire la struttura del Database**. Con questi comandi possiamo creare nuove tabelle, modificare quelle esistenti e cancellare oggetti non più necessari.

Ad esempio, **con il comando CREATE** si possono definire le tabelle, specificando colonne, tipi di dati e vincoli come chiavi primarie ed esterne. Il **comando ALTER** consente di **modificare una struttura già esistente**, ad esempio aggiungendo una nuova colonna. Infine, **il comando DROP** elimina completamente un oggetto, come una tabella o un indice, dal Database.

Questi strumenti sono essenziali nella fase di progettazione e sviluppo del Database, garantendo che le strutture soddisfino i requisiti dell'applicazione.

# LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

Il comando CREATE ha svariati usi, quelli più comuni sono:

- CREATE DATABASE «DBName»

Il comando viene usato per creare un Database con un nome definito dall'argomento DBName

- CREATE TABLE «DBName».«TableName»

(«FieldName» «FieldType», ....)

Questo comando crea una tabella all'interno del Database «DBName» una tabella di nome «TableName» inserendoci le colonne «FieldName» di tipo «FieldType»

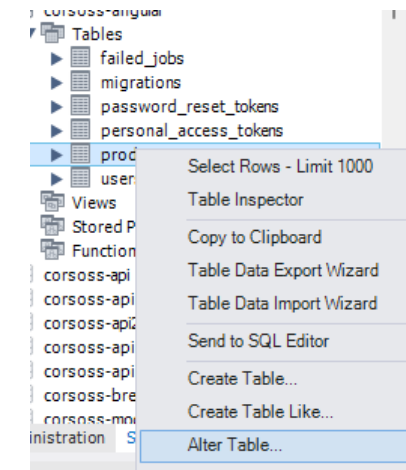
```
CREATE DATABASE nome_database;
```

```
CREATE TABLE Ordini (  
    OrdineID INT PRIMARY KEY,  
    ClienteID INT NOT NULL,  
    DataOrdine DATE,  
    FOREIGN KEY (ClienteID) REFERENCES Clienti(ID)  
);
```



# LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

- Il comando ALTER viene per modificare tabelle o colonne già presenti:
- ALTER TABLE «TableName»  
Il comando viene usato per aggiungere, eliminare o modificare le colonne di una tabella con un nome definito dall'argomento TableName
- ALTER COLUMN «ColumnName» «FieldType»  
Questo comando è usato dopo ALTER TABLE per modificare il tipo di data contenuto in ColumnName al tipo FieldType



Column Name	Datatype
Id	BIGINT
name	VARCHAR(60)
price	DECIMAL(15,2)
created_at	TIMESTAMP
updated_at	TIMESTAMP
productscol	VARCHAR(45)

Review the SQL Script to be Applied on the Database

```
1 ALTER TABLE 'corsoss-angular'. 'products'  
2 ADD COLUMN 'productscol' VARCHAR(45) NULL AFTER 'updated_at';  
3
```

```
ALTER TABLE Clienti  
ADD DataNascita DATE;
```

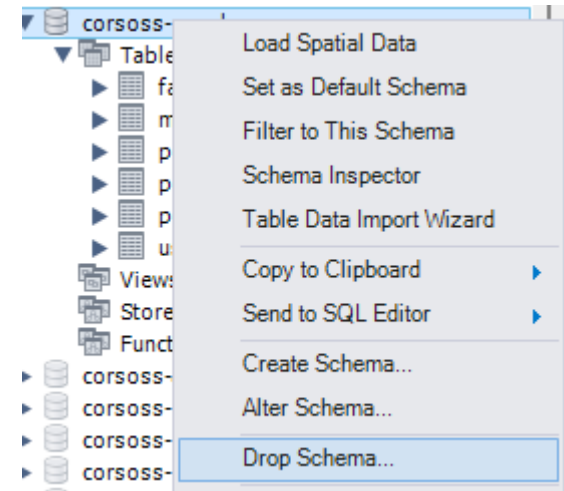
```
ALTER TABLE Clienti  
ALTER COLUMN Telefono VARCHAR(20);
```

```
ALTER TABLE Clienti  
DROP COLUMN Fax;
```

```
ALTER TABLE Clienti  
RENAME COLUMN DataNascita TO DataDiNascita;
```

# LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

- Il comando DROP viene usato principalmente eliminare tabelle o colonne, prestare attenzione quando si utilizza dato che l'operazione è irreversibile.
- DROP TABLE «TableName»  
Il comando viene usato per eliminare una tabella e i dati in essa
- DROP COLUMN «ColumnName»  
Questo comando è usato dopo ALTER TABLE per eliminare la colonna ColumnName e i suoi contenuti

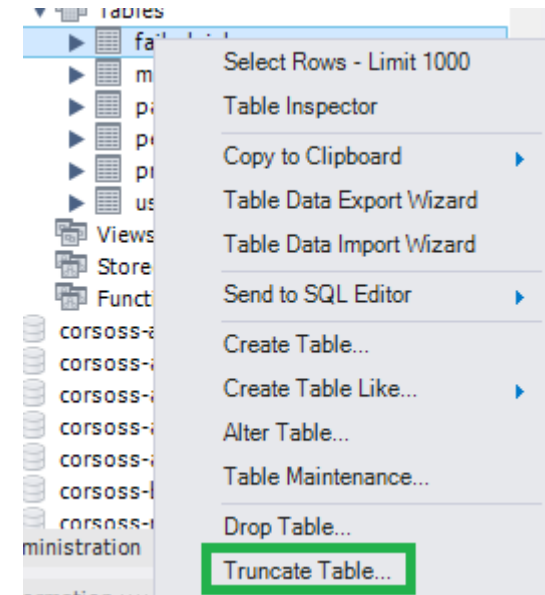


```
DROP DATABASE GestioneClienti;
```

```
ALTER TABLE Clienti  
DROP COLUMN Fax;
```

# LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

- Il comando TRUNCATE viene usato per eliminare i dati della colonna senza influire sulla struttura
- TRUNCATE TABLE «TableName»  
Il comando viene usato per eliminare i dati della tabella TableName, mantenendo le colonne, questo comando a differenza di delete reimposta gli id auto-incrementali a 1.  
Inoltre ci sono delle limitazioni quando si usa questo comando, **se sono presenti chiavi esterne non questo comando ritorna un errore**, inoltre **non ha un filtro per i dati da eliminare**, in questo caso è necessario usare il comando DELETE con WHERE



**TRUNCATE TABLE Prodotti;**

# LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

- Il comando RENAME viene usato dopo per rinominare colonne o tabelle
- ALTER table RENAME COLUMN "old\_name" to "new\_name"  
Rinomina la colonna old\_name a new\_name
- RENAME TABLE "old\_name" to "new\_name"  
Rinomina la tabella old\_name a new\_name

```
ALTER TABLE Clienti  
RENAME COLUMN Telefono TO NumeroTelefono;
```

```
RENAME TABLE ClientiVecchi TO Clienti;
```

# LINGUAGGI DDL – PROPRIETÀ DEI CAMPI

Oltre al nome e al tipo di colonna è possibile inserire anche argomenti come:

- PRIMARY KEY  
Definisce la **colonna come identificatore UNIVOCO della tabella**
- NOT NULL  
**Previene valori NULL**
- UNIQUE  
Garantisce **l'unicità dei valori della colonna**
- B: definisce se una colonna deve essere definita in binario (utilizzato per BLOB etc..)
- ZF= Zerofill - ovvero ad esempio INT(5) di valore 42 è memorizzato come 00042 \_ **DEPRECATO DA MYSQL 8.0**
- UNSIGNED: definisce se il numero deve avere il segno
- DEFAULT valore  
Definisce un **valore predefinito** nel caso non venga inserito nella generazione di valori
- FOREIGN KEY  
**Collega una colonna a una colonna primaria** di un'altra tabella per mantenere l'integrità referenziale.
- AUTO\_INCREMENT  
Quando viene inserito un nuovo valore nella tabella questa colonna **verrà automaticamente riempita con un numero autoincrementante**
- ON DELETE/UPDATE CASCADE  
Usato insieme alle foreign key, **consente la propagazione delle modifiche o rimozione dei dati** quando viene modificata la chiave primaria legata alla colonna
- **G = generated column:** totale DECIMAL(10,2) GENERATED ALWAYS AS (prezzo + iva) VIRTUAL

# DDL ESEMPI

- **CREATE TABLE IF NOT EXISTS** `docenti` (
- `id` **INT NOT NULL AUTO\_INCREMENT**,
- `cognome` **VARCHAR(45) NULL**,
- `nome` **VARCHAR(45) NULL**,
- `cellulare` **VARCHAR(20) NULL**,
- **PRIMARY KEY** (`id`))
- **ENGINE = InnoDB;**

- **CREATE TABLE IF NOT EXISTS** `docenti\_has\_corsi` (
- `docenti\_id` **INT NOT NULL**,
- `corsi\_id` **INT NOT NULL**,
- **PRIMARY KEY** (`docenti\_id`, `corsi\_id`),
- **INDEX** `fk\_docenti\_has\_corsi\_corsi1\_idx` (`corsi\_id` ASC) **VISIBLE**,
- **INDEX** `fk\_docenti\_has\_corsi\_docenti\_idx` (`docenti\_id` ASC) **VISIBLE**,
- **CONSTRAINT** `fk\_docenti\_has\_corsi\_docenti`
- **FOREIGN KEY** (`docenti\_id`)
- **REFERENCES** `corso\_ss5`.`docenti` (`id`)
- **ON DELETE** NO ACTION
- **ON UPDATE** NO ACTION,
- **CONSTRAINT** `fk\_docenti\_has\_corsi\_corsi1`
- **FOREIGN KEY** (`corsi\_id`)
- **REFERENCES** `corso\_ss5`.`corsi` (`id`)
- **ON DELETE** NO ACTION
- **ON UPDATE** NO ACTION)
- **ENGINE = InnoDB;**

# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

- I linguaggi DML sono utilizzati per gestire i dati all'interno di un Database.
- Con questi comandi possiamo leggere, inserire, aggiornare ed eliminare informazioni nelle tabelle.
- Ad esempio, con il comando **SELECT** è possibile recuperare i dati desiderati, applicando filtri e ordinamenti per ottenere informazioni specifiche. Il comando **INSERT** consente di aggiungere nuove righe a una tabella, specificando i valori per ogni colonna. Il comando **UPDATE** permette di modificare i dati esistenti, ad esempio aggiornando l'indirizzo di un cliente. Infine, il comando **DELETE** elimina le righe che soddisfano determinati criteri, liberando spazio senza alterare la struttura della tabella.
- Questi strumenti sono essenziali per la gestione quotidiana dei dati, garantendo che le informazioni possano essere manipolate e consultate in modo efficace per soddisfare le esigenze operative e analitiche dell'applicazione.

# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

Il comando SELECT è uno dei più utilizzati in SQL e **permette di leggere i dati da una o più tabelle all'interno di un database**. Ecco alcuni dei suoi usi più comuni:

- **SELECT \* FROM «TableName»**  
Questo comando seleziona tutte le colonne e tutte le righe della tabella specificata, permettendo una visualizzazione completa dei dati contenuti in essa.
- **SELECT «Column1», «Column2» FROM «TableName»**  
Con questo comando si possono recuperare solo le colonne desiderate, riducendo il numero di dati visualizzati.
- **SELECT \* FROM «TableName» WHERE «Condition»**  
Utilizzando la clausola WHERE, è possibile filtrare le righe in base a condizioni specifiche
- **SELECT \* FROM «TableName» ORDER BY «ColumnName» [ASC|DESC]**  
Questo comando ordina i dati in base a una o più colonne, in ordine crescente (ASC) o decrescente (DESC).
- **SELECT AVG/SUM/MIN/MAX(«ColumnName») AS «Alias» FROM «TableName»**  
Questo consente di effettuare calcoli durante la selezione sulle colonne selezionate

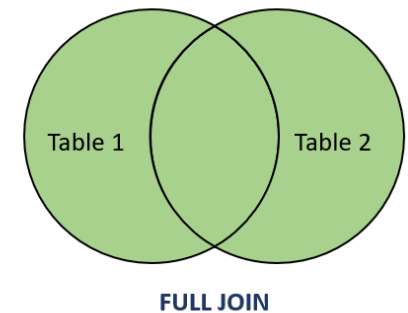
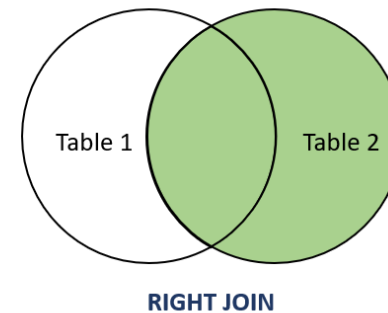
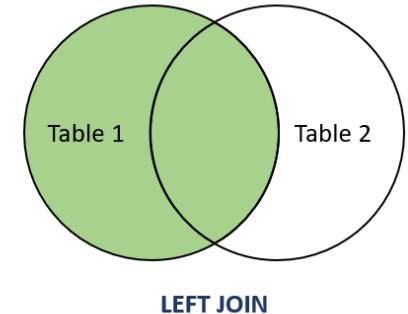
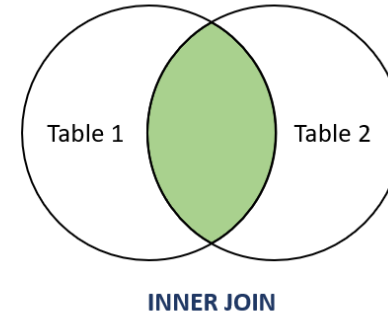


# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

I JOIN sono operazioni SQL utilizzabili insieme al SELECT che **consentono di combinare i dati di due o più tabelle** basandosi su una relazione definita tra di esse.

Tipologie principali di JOIN:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN  
(in mysql non disponibile, utilizzare UNION)

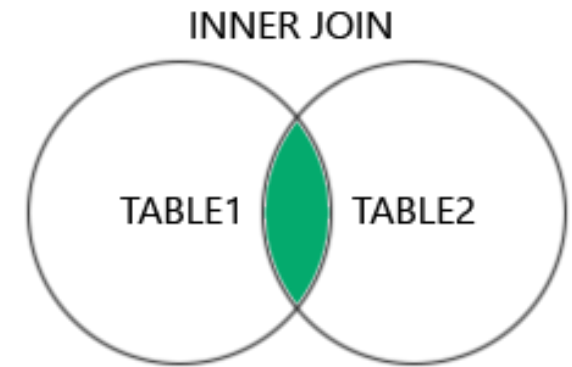


# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

INNER JOIN o solo JOIN restituisce solo le righe con corrispondenze tra le due tabelle.

- Se volessi trovare i clienti che hanno effettuato degli ordini:

```
SELECT Clienti.Nome, Ordini.ID_Ordine  
FROM Clienti  
INNER JOIN Ordini  
ON Clienti.ID_Cliente = Ordini.ID_Cliente;
```



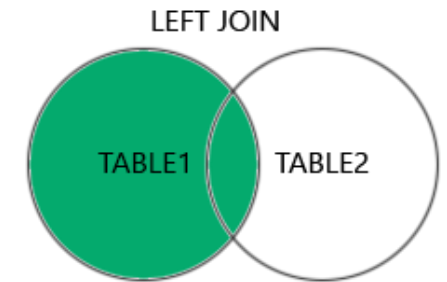
Così verrà mostrato il nome e l'id dell'ordine di tutti i clienti con almeno un ordine

# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

LEFT JOIN restituisce tutte le righe della prima tabella e le corrispondenze dalla seconda tabella.

- Se non ci sono corrispondenze, i valori della seconda tabella saranno NULL.
- Se volessi mostrare tutti clienti insieme all'id degli ordini se presente:

```
SELECT Clienti.Nome, Ordini.ID_Ordine  
FROM Clienti  
LEFT JOIN Ordini  
ON Clienti.ID_Cliente = Ordini.ID_Cliente;
```



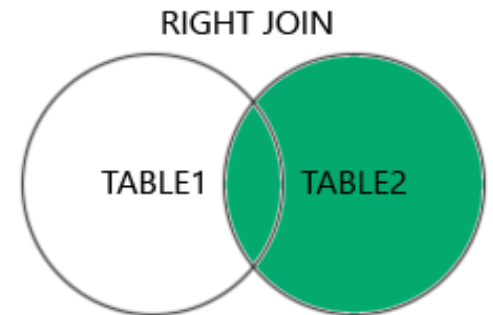
Così verrà mostrato il nome e l'id dell'ordine di tutti i clienti, se il cliente non ha un ordine associato verrà comunque mostrato, ma la colonna dell'id dell'ordine mostrerà NULL

# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

RIGHT JOIN restituisce tutte le righe della seconda tabella e le corrispondenze dalla prima tabella.

- Se non ci sono corrispondenze, i valori della prima tabella saranno NULL.
- Se volessi mostrare tutti gli ordini insieme al cliente se presente:

```
SELECT Clienti.Nome, Ordini.ID_Ordine  
FROM Clienti  
RIGHT JOIN Ordini  
ON Clienti.ID_Cliente = Ordini.ID_Cliente;
```



Così verrà mostrato il nome e l'id dell'ordine di tutti i clienti, se l'ordine non ha un cliente associato verrà comunque mostrato, ma la colonna del nome del cliente mostrerà NULL

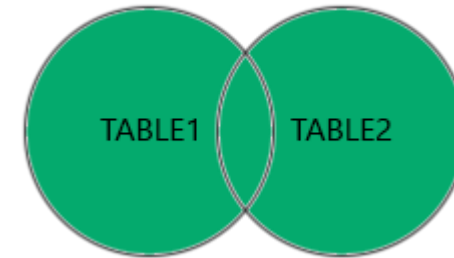
# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

- FULL JOIN è l'unione di RIGHT e LEFT JOIN in quanto restituisce tutte le righe di entrambe le tabelle, incluse quelle senza corrispondenze.
- Se non ci sono corrispondenze, i valori saranno NULL.
- Se volessi mostrare tutti gli ordini e clienti con le corrispondenze:

```
SELECT Clienti.Nome, Ordini.ID_Ordine
FROM Clienti
FULL JOIN Ordini
ON Clienti.ID_Cliente = Ordini.ID_Cliente;
```

- Così verranno mostrate tutte le righe di entrambe le righe di entrambe le tabelle, le colonne che non hanno corrispondenze mostreranno NULL

FULL OUTER JOIN



## Come implementare la Full Join in MySQL

Come detto all'inizio, l'operatore *Full Join* non è disponibile su MySQL, tuttavia possiamo raggiungere lo stesso output in questo modo:

```
SELECT  A.CodiceA,
        B.CodiceB,
        A.ValoreA,
        B.ValoreB
FROM    TabellaA AS A
LEFT JOIN TabellaB AS B
      ON A.CodiceA = B.CodiceB
UNION ALL
SELECT  A.CodiceA,
        B.CodiceB,
        A.ValoreA,
        B.ValoreB
FROM    TabellaA AS A
RIGHT JOIN TabellaB AS B
      ON A.CodiceA = B.CodiceB
WHERE   A.CodiceA IS NULL;
```

# WHERE

- La clausola WHERE viene utilizzata per filtrare le righe prima che vengano applicate funzioni di aggregazione o altre operazioni
- Condizioni con operatori logici (AND, OR, NOT).
- Filtri basati su valori (=, >, <, LIKE, ecc.).
- Filtri su sottogruppi usando subquery.

```
SELECT *  
FROM prodotti  
WHERE prezzo BETWEEN 10 AND 50  
AND disponibilita = 'in stock';
```

# ALIAS (AS)

Con il SELECT possiamo assegnare dei nomi temporanei alle colonne usando degli alias per rendere più leggibile il nome delle colonne che vengono selezionate oppure per dare un nome al risultato delle funzioni che eseguiamo

Per dare un alias alla colonna possiamo aggiungere AS dopo la colonna che abbiamo selezionato seguito dal nome della colonna, tra virgolette singole nel caso sia un nome contenente spazi

```
SELECT first_name as Nome  
FROM customer  
WHERE customer_id < 30;
```

# OPERATORI DI CONFRONTO

Operatore	Descrizione	Esempio	Risultato
=	Uguaglianza	<code>categoria = 'Sport'</code>	Restituisce righe con categoria "Sport".
<> o !=	Diverso da	<code>categoria &lt;&gt; 'Casa'</code>	Restituisce righe con categoria diversa da "Casa".
>	Maggiore di	<code>prezzo &gt; 100</code>	Restituisce righe con prezzo maggiore di 100.
<	Minore di	<code>prezzo &lt; 50</code>	Restituisce righe con prezzo minore di 50.
>=	Maggiore o uguale a	<code>prezzo &gt;= 75</code>	Restituisce righe con prezzo $\geq 75$ .
<=	Minore o uguale a	<code>prezzo &lt;= 30</code>	Restituisce righe con prezzo $\leq 30$ .
BETWEEN	Compreso tra due valori (inclusi)	<code>prezzo BETWEEN 10 AND 50</code>	Restituisce righe con prezzo tra 10 e 50.
IN	Valore contenuto in una lista	<code>categoria IN ('A', 'B')</code>	Restituisce righe con categoria "A" o "B".
NOT IN	Valore non contenuto in una lista	<code>categoria NOT IN ('X')</code>	Restituisce righe senza categoria "X".
LIKE	Cerca valori corrispondenti a un pattern	<code>nome LIKE 'Mario%'</code>	Restituisce righe con nomi che iniziano con "Mario".
IS NULL	Verifica valori nulli	<code>descrizione IS NULL</code>	Restituisce righe con descrizione nulla.
IS NOT NULL	Verifica valori non nulli	<code>descrizione IS NOT NULL</code>	Restituisce righe con descrizione non nulla.





# OPERATORI LOGICI

Operatore	Descrizione	Esempio	Risultato
AND	Tutte le condizioni devono essere vere	<code>prezzo &gt; 50 AND categoria = 'Elettronica'</code>	Restituisce righe con prezzo > 50 e categoria "Elettronica".
OR	Almeno una condizione deve essere vera	<code>prezzo &lt; 30 OR disponibilita = 'in stock'</code>	Restituisce righe con prezzo < 30 o disponibili in stock.
NOT	Inverte il risultato di una condizione	<code>NOT (prezzo &gt; 100)</code>	Restituisce righe con prezzo ≤ 100.
AND NOT	Combinazione: tutte vere tranne una	<code>prezzo &gt; 50 AND NOT categoria = 'Sport'</code>	Restituisce righe con prezzo > 50 che non sono categoria "Sport".

- gli operatori logici hanno una priorità (es. NOT è valutato prima di AND, che a sua volta è valutato prima di OR)

# OPERATORI ARITMETICI

▪ +, -, \*, /, % per eseguire calcoli o confronti derivati

```
SELECT prodotto, prezzo * 0.9 AS prezzo_scontato  
FROM prodotti  
WHERE prezzo > 50;
```

# GROUP BY

- La clausola GROUP BY viene utilizzata per raggruppare righe che hanno valori identici in una o più colonne.
- Spesso associata a funzioni di aggregazione (SUM, COUNT, AVG, ecc.) per calcolare statistiche su ciascun gruppo

```
SELECT categoria, COUNT(*) AS numero_prodotti  
FROM prodotti  
GROUP BY categoria;
```

## Esempio con GROUP BY

sql

[Copia](#) [Modifica](#)

```
SELECT customer_id, COUNT(*) AS totale_ordini  
FROM orders  
GROUP BY customer_id;
```

✓ Questa query restituisce una riga per ogni `customer_id`, mostrando il numero totale di ordini per ciascun cliente.

## Output

customer_id	totale_ordini
1	5
2	3
3	7

⚠ Con `GROUP BY`, le righe vengono aggregate in un unico risultato per ogni gruppo.

# PARTITION BY

- Divide i dati in gruppi o esegue calcoli come SUM ma NON aggrega
- Usato con funzioni di finestra (ROW\_NUMBER(), RANK(), SUM() OVER(), ecc.)
- Mantiene il numero originale delle righe, aggiungendo valori calcolati per ogni gruppo

customer_id	count(*)
30	2
38	3
44	1
49	1
63	2
78	2
80	1
98	1
337	4
488	2
777	2
831	1

customer_id	row_number() over (PARTITION by customer_id)
30	1
30	2
38	1
38	2
38	3
44	1
49	1
63	1
63	2
78	1
78	2
80	1
98	1
337	1
337	2
337	3
337	4
488	1
488	2
777	1
777	2
831	1

differenza tra partition by e group by

## Esempio con PARTITION BY

```
sql
SELECT
  customer_id,
  order_id,
  order_date,
  ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC) AS numero_ordine
FROM orders;
```

✓ Questa query assegna un numero progressivo a ogni ordine per ciascun cliente, senza aggregare i dati.

## Output

customer_id	order_id	order_date	numero_ordine
1	105	2024-01-10	1
1	102	2023-12-15	2
1	100	2023-11-20	3
2	205	2024-01-08	1
2	202	2023-12-10	2

⚠ Con PARTITION BY, le righe rimangono inalterate, ma viene calcolato un valore per ciascun gruppo.



# MYSQL FUNCTIONS

- MySQL offre molte funzioni integrate per lavorare con dati numerici, stringhe, date, JSON e altro; possiamo dividerle per categorie:
- Stringhe
- Date

# FUNZIONI SULLE STRINGHE

- Manipolano e analizzano stringhe di testo.
- CONCAT(str1, str2, ...) → Concatena stringhe
- LENGTH(str) → Lunghezza della stringa in byte
- CHAR\_LENGTH(str) → Numero di caratteri LUNGHEZZA in CARATTERI
- SUBSTRING(str, start, length) → Estrae una sottostringa – parte da 1
- LOCATE(substr, str) → Trova la posizione di una sottostringa
- REPLACE(str, str\_OLD, str\_NEW) → Sostituisce una parte di stringa
- CONCAT\_WS('-', str1, str2, ...) → Concatena stringhe con un separatore
- SELECT CONCAT('Hello', ' ', 'World'); -- Hello World
- SELECT CONCAT\_WS('-', 'Hello', 'World'); -- Hello-World

# ESEMPI FUNZIONI SULLE STRINGHE

- SELECT CONCAT(cognome,nome) from customers;
- SELECT LENGHT(cognome) from customers;
- SELECT LENGTH('abc'), CHAR\_LENGTH('abc');
- -- Output: 3, 3 (perché ogni carattere occupa 1 byte in UTF-8)
- SELECT LENGTH('è'), CHAR\_LENGTH('è');
- -- Output: 2, 1 (perché 'è' in UTF-8 occupa 2 byte ma è un solo carattere)
- SELECT LOCATE("a", cognome) from customers limit 10;
- SELECT cognome, REPLACE(cognome, 'ORIALI', 'CAMPI') AS cognome\_modificato FROM customers;

# FUNZIONI SULLE DATE E ORE

- Per manipolare e calcolare date e orari.
- NOW() → Data e ora attuale
- CURDATE() → Data attuale
- CURTIME() → Ora attuale
- DATE\_FORMAT(date, format) → Formatta una data
- DATEDIFF(date1, date2) → Differenza in giorni
- TIMESTAMPDIFF(UNIT, date1, date2) → Differenza tra date in unità
- ADDDATE(date, INTERVAL value unit) → Aggiunge un intervallo a una data
- `SELECT DATE_FORMAT(NOW(), '%d/%m/%Y'); -- 30/01/2025`



# ESEMPI FUNZIONI SULLE DATE

- SELECT NOW(); # 2025-03-03 08:12:17
- SELECT CURDATE(); #2025-03-03
- SELECT CURTIME(); #08:11:42
- SELECT DATE\_FORMAT('2025-03-03', '%d/%m/%Y') AS data\_formattata;
- SELECT DATEDIFF('2025-03-03', '2025-02-28') AS giorni; #3
- SELECT TIMESTAMPDIFF(DAY, '2025-01-01', '2025-03-03') AS differenza\_giorni; #61
- SELECT TIMESTAMPDIFF(MONTH, '2025-01-01', '2025-03-03') AS differenza\_giorni; #2
- SELECT ADDDATE('2025-03-03', INTERVAL 10 DAY) AS nuova\_data; # 2025-03-13

# FUNZIONI NUMERICHE

- Per operazioni matematiche.
- $ABS(x) \rightarrow$  Valore assoluto
- $ROUND(x, d) \rightarrow$  Arrotonda al numero di decimali (d a quale decimale: 1= 1 decimale, 2=2 decimale)
- `SELECT id, ROUND(prezzo, 1) AS costo_arrotondato FROM corsi; // 1,23 = 1,2`
- $CEIL(x) \rightarrow$  Arrotonda per eccesso (unità)
- $FLOOR(x) \rightarrow$  Arrotonda per difetto (unità)
- $MOD(x, y) \rightarrow$  Resto della divisione
- $RAND() \rightarrow$  Numero casuale
- $POWER(x, y) \rightarrow$  Elevamento a potenza
- $SQRT(x) \rightarrow$  Radice quadrata
- `SELECT ROUND(3.14159, 2); -- 3.14`

# ESEMPI FUNZIONI SUI NUMERI

- SELECT ABS (-2) ; #2
- SELECT ROUND (2.1234, 2) ; #2.12
- SELECT CEIL (2.1234) ; #3
- SELECT FLOOR (2.1234) ; #2
- SELECT MOD (10, 3) ; #1
- SELECT RAND () ; #NUMERO CASUALE DA 0 A 1
- SELECT FLOOR (RAND () \* 100) + 1 AS numero\_casuale;
  - NUMERO CASUALE DA 1 A 100
- SELECT POWER (5, 2) ; #25
- SELECT SQRT (25) ; #5

# FUNZIONI DI AGGREGAZIONE

- SUM: Somma.
- COUNT: Conteggio righe.
- AVG: Media.
- MIN/MAX: Valori minimo e massimo.

```
SELECT AVG(prezzo) AS prezzo_medio  
FROM prodotti;
```

# ESEMPI FUNZIONI AGGREGAZIONE

- SELECT SUM(montante) from pratiche; #1250125.25
- SELECT COUNT(\*) from pratiche; #1.256
- SELECT AVG(montante) from pratiche; 995.32
- SELECT MIN(montante) from pratiche; # 250.00
- SELECT MAX(montante) from pratiche; #40000.00

# COUNT

LA FUNZIONE COUNT() SVOLGE UNA SEMPLICE CONTA DELLE RIGHE DI UNA COLONNA O DELL'INTERA TABELLA IN BASE ALL'ARGOMENTO INSERITO.

NEL CASO STIAMO CONTANDO LE RIGHE DI UNA COLONNA I VALORI NULL VERRANNO IGNORATI NEL CONTEGGIO FINALE MENTRE SE STIAMO ESEGUENDO COUNT SULLA TABELLA I VALORI NULL VERRANNO CONSIDERATI

```
SELECT COUNT(*) as "Pagamenti Totali"  
FROM payments;
```

```
SELECT COUNT(payment_done) as "Pagati"  
FROM payments;
```

# SUM

LA FUNZIONE SUM() RITORNA LA SOMMA DEI VALORI CONTENUTI NELLA COLONNA SPECIFICATA, OPPURE DI UN'OPERAZIONE TRA COLONNE, DI UN INSIEME DI RIGHE.

QUESTA FUNZIONE PUÒ ESSERE USATA SOLO IN COLONNE CHE CONTENGONO ESCLUSIVAMENTE VALORI NUMERICI ED OGNI NULL VERRÀ IGNORATO

```
SELECT SUM(payment_amount) as "Totale pagamenti"  
FROM payments;
```

```
SELECT SUM(payment_amount) as "Totale pagamenti Cliente 1"  
FROM payment  
WHERE client_id = 1;
```

# AVG

LA FUNZIONE AVG() RITORNA LA MEDIA DEI VALORI CONTENUTI NELLA COLONNA SPECIFICATA, OPPURE DI UN'OPERAZIONE TRA COLONNE, DI UN INSIEME DI RIGHE.

QUESTA FUNZIONE PUÒ ESSERE USATA SOLO IN COLONNE CHE CONTENGONO ESCLUSIVAMENTE VALORI NUMERICI ED OGNI NULL VERRÀ IGNORATO, IL RISULTATO DELLA FUNZIONE USERÀ LO STESSO TIPO DI VALORE DELLA COLONNA SELEZIONATA PER CUI SE STIAMO FACENDO LA MEDIA DI UNA COLONNA DI INT IL RISULTATO SARÀ UN INT SENZA VIRGOLE.

NEL CASO VOGLIAMO AVERE UN DATO PIÙ PRECISO POSSIAMO USARE UN CAST PER MOSTRARE UN RISULTATO PIÙ PRECISO

```
SELECT AVG(age) as "Età media"  
FROM payment;
```

```
SELECT AVG(CAST(age AS FLOAT)) as "Età media precisa"  
FROM payment;
```



# MIN MAX

LE FUNZIONI MIN() E MAX() RITORNA IL VALORE PIÙ ALTO O PIÙ BASSO NELLA COLONNA SPECIFICATA, OPPURE DI UN'OPERAZIONE TRA COLONNE, DI UN INSIEME DI RIGHE.

QUESTA FUNZIONE PUÒ ESSERE USATA SIA IN COLONNE NUMERICHE CHE IN COLONNE CON STRINGHE, IN QUEST'ULTIMO MIN RITORNA LA PRIMA STRINGA IN ORDINE ALFABETICO MENTRE MAX RITORNA L'ULTIMA.

I VALORI NULL NON VENGONO CONSIDERATI

```
SELECT MIN(payment_amount) as "Pagamento più basso"  
FROM payment;
```

```
SELECT MAX(payment_amount) as "Pagamento più alto"  
FROM payment;
```

# FUNZIONI DI CONTROLLO DEL FLUSSO

- IF(condition, true\_value, false\_value)
- IFNULL(value, default\_value) → Sostituisce NULL
  - SELECT id, nome, IFNULL(email, 'NULL') FROM corsisti;
- SELECT IF(10 > 5, 'Yes', 'No'); -- Yes

▪ SELECT id, montante, IF(montante > 5000, 'OK', 'KO') from pratiche;

id	montante	IF(montante>5000,'OK','KO')
6	14280.00	OK
7	36000.00	OK
8	10800.00	OK
10	2400.00	KO
11	16320.00	OK
13	9600.00	OK

# FUNZIONI JSON

- Per lavorare con JSON in MySQL.
- `JSON_OBJECT(key, value, ...)` → Crea un oggetto JSON
- `JSON_ARRAY(value, ...)` → Crea un array JSON
- `JSON_EXTRACT(json, path)` → Estrae un valore
- `JSON_UNQUOTE(json_extract(...))` → Rimuove virgolette
- `SELECT JSON_EXTRACT('{\"name\": \"Alice\", \"age\": 25}', '$.name'); -- \"Alice\"`

# ESEMPI FUNZIONI JSON

- `SELECT JSON_OBJECT('montante',montante) from pratiche;`

`JSON_OBJECT('montante',montante)`

`{"montante": 14280.00}`

`{"montante": 36000.00}`

`{"montante": 10800.00}`

`{"montante": 2400.00}`

- `SELECT JSON_ARRAY(montante, importo_rata) from pratiche;`

`JSON_ARRAY(montante, importo_rata)`

`[14280.00, 119.00]`

`[36000.00, 300.00]`

`[10800.00, 150.00]`

- `SELECT JSON_EXTRACT('{ "nome": "Mauro", "cognome": "Casadei" }', '$.nome');`

`JSON_EXTRACT('{ "nome": "Mauro"`

`"Mauro"`

- `SELECT JSON_EXTRACT('{ "persona": { "nome": "Mauro", "cognome": "Casadei" } }', '$.persona.nome');`

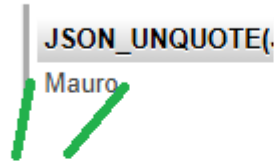
`JSON_EXTRACT('{ "perso`

`"Mauro"`

- `$` è la radice da cui partire

# ESEMPI FUNZIONI JSON

- SELECT JSON\_UNQUOTE (JSON\_EXTRACT ( ' {"persona": {"nome": "Mauro", "cognome": "Casadei"}} ', '\$.persona.nome' ) ) ;



The diagram illustrates the operation of the `JSON_UNQUOTE` function. It shows the function name `JSON_UNQUOTE(` in a grey box, followed by the string `"Mauro"` in green. A green checkmark is placed next to the result `Mauro`, indicating that the function successfully removes the double quotes from the input string.

RIMOSSO I doppi apici  
da "Mauro" a Mauro

# FUNZIONI DI GESTIONE TESTO E REGEX

- Per lavorare con pattern e sostituzioni.
- UPPER(str) → Maiuscolo
- LOWER(str) → Minuscolo
- TRIM(str) → Rimuove spazi
- REGEXP\_LIKE(str, pattern) → Controlla una regex
- `SELECT UPPER('hello'); -- HELLO`

# ESEMPI FUNZIONI TESTO

- SELECT UPPER('ciao mondo'); #CIAO MONDO
- SELECT LOWER('CIAO MONDO'); #ciao mondo
- SELECT TRIM(' ciao mondo '); #ciao mondo
- SELECT REGEXP\_LIKE('abc123', '[0-9]'); #1
- SELECT email, REGEXP\_LIKE(email, '^[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\$')  
from customers where email IS NOT NULL and email != ''  
;

[REDACTED]@libero.it	1
no mail	0
[REDACTED]@libero.it	1

# FUNZIONI DI GESTIONE DEL SISTEMA

- Per ottenere informazioni su MySQL.
- USER() → Utente corrente
- DATABASE() → Nome del database attuale
- VERSION() → Versione di MySQL
- **SELECT DATABASE();**



# ESEMPI FUNZIONI DEL SISTEMA

- SELECT USER() ; # root@localhost
- SELECT VERSION() ; # 8.2
- SELECT DATABASE() ; #test

# ORDER BY

- La clausola ORDER BY in SQL viene utilizzata per ordinare i risultati di una query in base a uno o più campi. Può essere usata in combinazione con i modificatori ASC (ordine crescente) e DESC (ordine decrescente).
- ASC (Ordinamento Crescente)
- DESC (Ordinamento Decrescente)
- Ordinamento su Più Colonne

```
SELECT *  
FROM prodotti  
ORDER BY categoria ASC, prezzo DESC;
```

```
SELECT colonna1, colonna2, ...  
FROM tabella  
ORDER BY colonna [ASC|DESC]:
```

```
SELECT nome, prezzo  
FROM prodotti  
ORDER BY prezzo DESC;
```

# DISTINCT

Questo argomento che possiamo aggiungere al SELECT prima delle colonne filtrerà i risultati in modo che vengano ritornate solo entry uniche ignorando tutti i duplicati.

Un utilizzo comune di DISTINCT è insieme a COUNT per contare il numero di entry uniche presenti in una determinata colonna

```
SELECT DISTINCT categoria  
FROM prodotti;
```

# UNION

Usando UNION possiamo combinare i risultati di due SELECT applicando DISTINCT automaticamente

Per usare UNION è necessario che entrambi i risultati dei SELECT contengano lo stesso numero di colonne.

Se abbiamo due colonne sulle quali vogliamo forzare l'unione possiamo usare ALIAS per dare lo stesso nome alle due colonne

Nel caso volessimo prevenire il DISTINCT possiamo usare UNION ALL

```
SELECT nome FROM clienti
UNION
SELECT nome FROM fornitori;
```

# ESEMPIO UNION

- SELECT numero\_pratica, 'tus' FROM pratiche\_tus where id < 10
- UNION
- SELECT numero\_pratica, 'cqs' FROM pratiche\_cqs where id < 10;

numero_pratica	tus
3	tus
4	tus
1	tus
2	tus
5	tus
6	tus
7	tus
1	cqs
2	cqs
3	cqs

# HAVING

- La clausola HAVING viene usata per filtrare gruppi di dati generati da GROUP BY, mentre WHERE si applica ai dati prima della raggruppamento

```
SELECT categoria, COUNT(*) AS numero_prodotti  
FROM prodotti  
GROUP BY categoria  
HAVING COUNT(*) > 5;
```

```
select vn_IDGuida,count(vn_IDGuida) from vini group by vn_IDGuida having vn_IDGuida =9
```

# SUBQUERY

- Una sottoquery è una query nidificata all'interno di un'altra query. Può essere usata per calcolare valori intermedi o filtrare dati.
- Per eseguire una subquery basta aggiungere un SELECT all'interno di un altro

```
SELECT nome
FROM clienti
WHERE id IN (
    SELECT cliente_id
    FROM ordini
    WHERE totale > 100
);
```

Qui possiamo usare una subquery unita all'utilizzo di una funzione di aggregazione per filtrare i prodotti con un prezzo superiore alla media

- In questo caso il WHERE si baserà sul risultato dalla subquery che richiede gli id dei clienti con un totale di ordini maggiore di 100

```
SELECT nome, prezzo
FROM prodotti
WHERE prezzo > (SELECT AVG(prezzo) FROM prodotti);
```

# LIMIT

- La clausola LIMIT viene utilizzata per limitare il numero di righe restituite da una query. È utile quando si desidera visualizzare solo una parte dei risultati, ad esempio per implementare la paginazione o testare query con dataset grandi.

```
SELECT *  
FROM prodotti  
LIMIT 5;
```



# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

Il comando INSERT viene utilizzato per **aggiungere nuove righe a una tabella nel database**. Consente di specificare i valori per una o più colonne, creando nuovi record. Ecco alcuni dei suoi usi più comuni:

- `INSERT INTO «TableName» VALUES (valore1, valore2, ...)`  
Questo comando aggiunge una nuova riga specificando i valori per tutte le colonne nella stessa sequenza definita dalla tabella.
- `INSERT INTO «TableName» («Column1», «Column2») VALUES (valore1, valore2)`  
Con questo comando, si possono specificare solo alcune colonne della tabella, lasciando le altre con valori predefiniti o NULL. È possibile anche inserire più valori
- `INSERT INTO «TableName» («Column1», «Column2») SELECT «Column1», «Column2» FROM «OtherTable» WHERE «Condition»`  
Questo comando consente di copiare dati da un'altra tabella

# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

Il comando **UPDATE** viene utilizzato insieme a **SET** per modificare i dati esistenti all'interno di una tabella. Consente di aggiornare una o più colonne per una o più righe. Ecco i suoi usi più comuni:

- **UPDATE «TableName» SET «ColumnName» = valore**  
Questo comando modifica il valore di una colonna per tutte le righe della tabella.
- **UPDATE «TableName» SET «ColumnName» = valore WHERE «Condition»**  
Utilizzando la clausola **WHERE**, è possibile applicare modifiche solo alle righe che soddisfano una determinata condizione.
- **UPDATE «TableName» SET «ColumnName» = «ColumnName» + valore WHERE «Condition»**  
È possibile aggiornare una colonna basandosi su calcoli o valori esistenti.
- **Attenzione: una UPDATE senza WHERE aggiornerà TUTTE le righe della tabella**

# LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

Il comando DELETE viene utilizzato per **eliminare righe da una tabella**, permettendo di specificare quali record rimuovere tramite condizioni. A differenza di TRUNCATE, che elimina tutte le righe, DELETE offre maggiore controllo grazie all'uso della clausola WHERE:

- **DELETE FROM «TableName»**  
Questo comando rimuove tutte le righe della tabella, a differenza di TRUNCATE questo non reimposta l'auto-increment a 0
- **DELETE FROM «TableName» WHERE «Condition»**  
Utilizzando la clausola WHERE, è possibile eliminare solo le righe che soddisfano una determinata condizione.
- **Attenzione: una DELETE senza WHERE aggiornerà TUTTE le righe della tabella**

# LINGUAGGI DCL (DATA CONTROL LANGUAGE)

Il linguaggio DCL è utilizzato per **gestire i permessi e il controllo degli accessi all'interno di un database**. Con i comandi DCL, gli amministratori di database possono definire chi ha il diritto di eseguire determinate operazioni, come la lettura, la modifica, o l'eliminazione dei dati. I comandi DCL più comuni sono GRANT e REVOKE

- GRANT «privilege1», «privilege2» ON «TableName» TO «UserName»

Il comando GRANT **permette di assegnare permessi a uno o più utenti per eseguire operazioni su oggetti del database**. È possibile specificare quali tipi di operazioni possono essere effettuate, come SELECT, INSERT, UPDATE e DELETE

- REVOKE «privilege1», «privilege2» ON «TableName» FROM «UserName»

Il comando REVOKE viene utilizzato per rimuovere i permessi precedentemente concessi. Quando un permesso viene revocato, l'utente non potrà più eseguire l'operazione associata su quella tabella o oggetto

# LINGUAGGITCL (TRANSACTION CONTROL LANGUAGE)

I comandi TCL (Transaction Control Language) sono utilizzati per gestire le transazioni nei database relazionali.

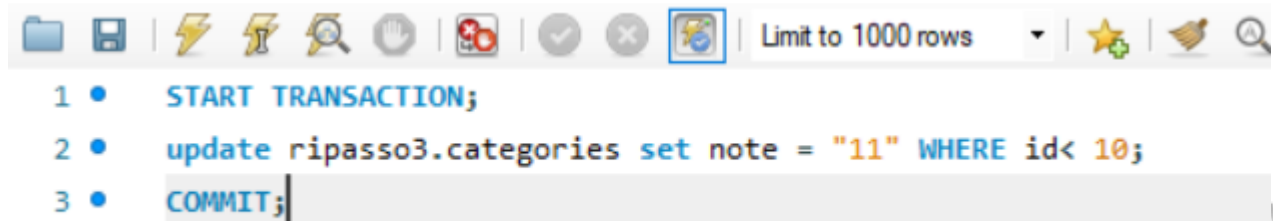
Permettono di controllare quando i cambiamenti ai dati vengono salvati o annullati.

Fondamentali per garantire la consistenza e la coerenza dei dati in scenari multi-utente.

Le operazioni principali sono:

- **COMMIT**: Salva permanentemente i cambiamenti fatti durante una transazione.
- **ROLLBACK**: Annulla i cambiamenti fatti durante una transazione.
- **SAVEPOINT**: Imposta un punto di ripristino all'interno di una transazione.

Tutti questi comandi sono disponibili solo dopo aver iniziato una transazione con il comando **START TRANSACTION**



The screenshot shows a SQL query editor interface with a toolbar at the top containing icons for file operations, execution, and search. Below the toolbar, a list of SQL commands is displayed, numbered 1 through 3. The commands are: 1. `START TRANSACTION;`, 2. `update ripasso3.categories set note = "11" WHERE id< 10;`, and 3. `COMMIT;`. The text "Limit to 1000 rows" is visible in the toolbar. In the bottom right corner, there is a logo for "ware" with the tagline "your business" and a stylized arrow icon.

```
1 • START TRANSACTION;  
2 • update ripasso3.categories set note = "11" WHERE id< 10;  
3 • COMMIT;
```

# LINGUAGGITCL (TRANSACTION CONTROL LANGUAGE)

Il comando **COMMIT** finalizza le modifiche fatte ai dati durante una transazione, rendendole permanenti nel database e chiudendo la transazione.

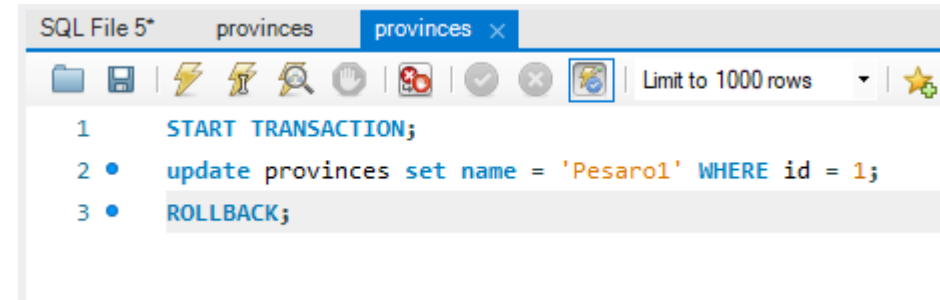
```
START TRANSACTION;  
UPDATE Prodotti SET Prezzo = 20 WHERE ID_Prodotto = 1;  
COMMIT;
```

In questo esempio l'UPDATE non viene salvato definitivamente fino a quando non viene eseguito COMMIT una volta salvato non sarà più possibile ritornare i dati allo stato precedente

# LINGUAGGITCL (TRANSACTION CONTROL LANGUAGE)

Il comando **ROLLBACK** annulla le modifiche fatte durante una transazione, riportando i dati allo stato precedente chiudendo la transazione.

```
START TRANSACTION;  
UPDATE Prodotti SET Prezzo = 15 WHERE ID_Prodotto = 2;  
ROLLBACK;
```

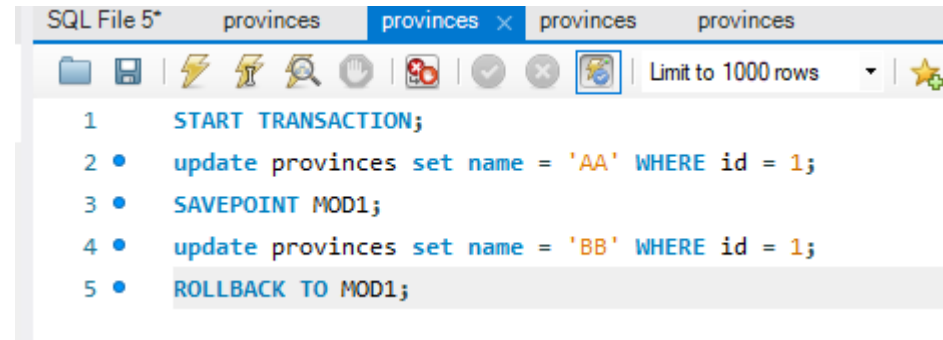


In questo esempio l'UPDATE è temporaneamente salvato nel database, ma quando rollback viene eseguito dato che è all'interno della transazione l'UPDATE verrà annullato e i dati modificati ritorneranno al loro stato originale

# LINGUAGGITCL (TRANSACTION CONTROL LANGUAGE)

Il comando **SAVEPOINT** crea un punto di ripristino all'interno di una transazione, permettendo di annullare solo una parte delle modifiche.

```
START TRANSACTION;
UPDATE Prodotti SET Prezzo = 10 WHERE ID_Prodotto = 1;
SAVEPOINT PrimaModifica;
UPDATE Prodotti SET Prezzo = 5 WHERE ID_Prodotto = 2;
ROLLBACK TO PrimaModifica;
```



The screenshot shows a SQL editor window titled 'SQL File 5\*' with a tab labeled 'provinces'. The editor contains the following SQL code:

```
1 START TRANSACTION;
2 • update provinces set name = 'AA' WHERE id = 1;
3 • SAVEPOINT MOD1;
4 • update provinces set name = 'BB' WHERE id = 1;
5 • ROLLBACK TO MOD1;
```

The editor interface includes a toolbar with icons for file operations, a search bar, and a 'Limit to 1000 rows' dropdown menu.

In questo esempio i due UPDATE sono nella stessa transazione ma in due punti diversi, usando il ROLLBACK TO possiamo specificare il punto sul quale ripristinare, questo però lascerà la transazione aperta e rimane poi da fare COMMIT per il resto dei dati



# LINGUAGGI DCL (DATA CONTROLLING LANGUAGE)

I comandi DCL (Data Controlling Language) sono utilizzati 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).

Questi comandi sono utili per avere un controllo migliore sulle attività del database nel caso abbiamo molti utenti che lo utilizzano

Le operazioni principali sono:

- **GRANT**: fornisce uno o più permessi a un determinato utente su un determinato oggetto del database (es: il permesso di inserimento in una tabella).
- **REVOKE**: revoca uno o più permessi a un determinato utente su un determinato tipo di oggetti (es: il permesso di cancellazione da una tabella).

# LINGUAGGI DCL (DATA CONTROLLING LANGUAGE)

Il comando **GRANT** è utilizzato per assegnare permessi agli utenti o ai ruoli su oggetti di database, come tabelle, viste o interi schemi.

```
GRANT SELECT, INSERT, UPDATE  
ON employees  
TO 'user123'@'localhost';
```

In questo esempio, stiamo concedendo i privilegi per utilizzare i comandi SELECT, INSERT e UPDATE sulla tabella 'employees' all'utente 'user123' ma solo sulla macchina dove è contenuto il database ('localhost')

```
GRANT ALL PRIVILEGES  
ON company.*  
TO 'user123'@'%';
```

In questo esempio invece stiamo concedendo controllo completo su tutte le tabelle del database 'company' all'utente 'user123' da qualsiasi macchina che ha accesso al database

In entrambi gli esempi dobbiamo finalizzare i cambiamenti ai permessi utilizzando il comando FLUSH PRIVILEGES

# LINGUAGGI DCL (DATA CONTROLLING LANGUAGE)

Il comando **GRANT** può essere usato anche per consentire ad altri utenti di eseguire GRANT nel caso vogliamo consentire ad altri utenti la possibilità di gestire i privilegi

Questo può essere realizzato aggiungendo WITH GRANT OPTION alla fine di un comando GRANT

Questo permesso è l'unico permesso che non viene dato con il comando GRANT ALL PRIVILEGES

```
GRANT ALL PRIVILEGES  
ON company.*  
TO 'user123'@'%'  
WITH GRANT OPTION
```

# LINGUAGGI DCL (DATA CONTROLLING LANGUAGE)

Il comando **REVOKE** è utilizzato al contrario di GRANT per rimuovere i permessi di un utente di utilizzare certi comandi.

```
REVOKE INSERT, UPDATE  
ON employees  
FROM 'user123'@'localhost';
```

In questo esempio, stiamo rimuovendo i privilegi per utilizzare i comandi INSERT e UPDATE sulla tabella 'employees' all'utente 'user123' ma solo sulla macchina dove è contenuto il database ('localhost')

```
REVOKE ALL PRIVILEGES  
ON company.*  
FROM 'user123'@'%';
```

In questo esempio invece stiamo revocando il controllo completo su tutte le tabelle del database 'company' all'utente 'user123' da qualsiasi macchina che ha accesso al database

```
REVOKE GRANT OPTION  
ON employees  
FROM 'user123'@'localhost';
```

Nel caso vogliamo revocare i permessi di utilizzo del comando GRANT dobbiamo specificare GRANT OPTION come revoca

Come per GRANT dobbiamo finalizzare i cambiamenti ai permessi utilizzando il comando FLUSH PRIVILEGES

# VARIABILI LOCALI

MySQL consente di salvare variabili o risultati di query all'interno di variabili locali temporanee che possiamo richiamare in altre query.

Per far questo possiamo far uso del comando INTO seguito dal nome della variabile che in MySQL si presentano con una @ all'inizio per differenziarle dai nomi delle tabelle o colonne.

Una volta salvata la variabile, per il resto della sessione corrente possiamo chiamarla con SELECT o usarla per altre query.

```
select count(*) into @numero from customers;  
select @numero;
```

Questo esegue il conteggio delle righe nella tabella aziende e salva il risultato all'interno di una variabile @n\_aziende

```
mysql> SELECT @n_aziende;  
+-----+  
| @n_aziende |  
+-----+  
|          5 |  
+-----+  
1 row in set (0.00 sec)
```

Se in seguito all'assegnazione selezioniamo la variabile possiamo notare che ritornerà il risultato della query

È importante ricordare che la variabile non è dinamica, quindi se aggiungiamo una nuova azienda la variabile rimarrà uguale al valore assegnato al momento del SELECT

# ESEMPI DI VARIABILI

- `SELECT count(*) as numero_clienti into @numero_clienti from customers;`
- `SELECT @numero_clienti;` @numero\_clienti  
96109
- `SELECT sum(montante) into @montante_cliente_id24 from practice_cqs where customer_id=24;`  
`select round(@montante_cliente_id24 * 10 / 100,2);` 10 / 100,2)  
1428.00

# STORED PROCEDURES

Oltre alle variabili è possibile salvare query o operazioni intere all'interno di procedure che vengono salvate nel database in modo da poterle usare in seguito tramite comandi.

In MSOL le Stored Procedures sono disponibili soltanto a partire dalla versione 5

- Si dividono in 2 gruppi:
- **Procedure**: non devono restituire valori ma accettare parametri di input e di output.
- **Funzioni** (User Defined Functions o più semplicemente UDF): restituiscono un valore e accettano parametri di input ed output

# STORED PROCEDURES

Nella definizione delle Stored Procedures è prevista l'introduzione di tre diversi parametri.

**IN:** rappresenta gli argomenti in ingresso della routine; a questo parametro viene assegnato un valore quando viene invocato il sotto-programma; il parametro utilizzato non subirà in seguito modifiche.

**OUT:** è il parametro relativo ai valori che vengono assegnati con l'uscita dalla procedura; questi parametri diventano disponibili per gli utenti

**INOUT:** rappresenta una combinazione tra i due parametri precedenti.



# DELIMITER

In MySQL si usa «;» per indicare il termine di un'istruzione ma quando scriviamo blocchi di istruzioni o istruzioni multipli all'interno di un BEGIN ... END, usare il delimitatore base potrebbe essere interpretato in maniera sbagliata dal parser di MySQL causando errori. Per evitare questo per la scrittura di procedure più complesse possiamo fare uso di DELIMITER, questa istruzione consente di cambiare temporaneamente il delimitatore in uno personalizzato, dopo il quale creiamo la procedura nuova utilizzando all'interno il delimitatore base, terminata la definizione la chiudiamo con il delimitatore temporaneo

```
DELIMITER //
```

```
CREATE PROCEDURE esempio()  
BEGIN  
    SELECT 'Ciao, mondo';  
END;  
//  
  
DELIMITER ;
```

```
DELIMITER $$  
  
CREATE PROCEDURE calcola_totale(IN prezzo DECIMAL(10,2), IN aliquota DECIMAL(5,2), OUT totale DECIMAL(10,2))  
BEGIN  
    SET totale = prezzo + (prezzo * aliquota / 100);  
END$$  
  
DELIMITER ;
```

# STORED PROCEDURES

- DELIMITER //
- CREATE PROCEDURE CONTA\_CLIENTI(OUT count\_result INT)
- BEGIN
- SELECT COUNT(\*) INTO count\_result FROM customers;
- END // -- Qui deve essere usato // invece di ;
- DELIMITER ;
- CALL CONTA\_CLIENTI(@count);
- SELECT @count; -- Per vedere il valore restituito
- CREATE PROCEDURE `conta\_corsi\_area\_didattica`(IN \_id INT, OUT num INT)
- BEGIN
- SELECT COUNT(id) into num from corsi where aree\_didattiche\_id = \_id;
- 
- END
- SET @num = 0;
- CALL conta\_corsi\_area\_didattica(1, @num);
- SELECT @num;

- **#procedura per contare i clienti nati tra 2 date passate in input**

- DELIMITER ~

- CREATE PROCEDURE totale\_clienti\_nascita(IN \_from DATE, IN \_to DATE, OUT \_num int)

- BEGIN

- SELECT COUNT(\*) INTO num FROM customers where data\_nascita BETWEEN \_from AND \_to;

- END ~

- DELIMITER ;

- SET @\_num=0;

- CALL totale\_clienti\_nascita("1990-01-01", "1990-12-31", @\_num);

- SELECT @\_num;

- **#procedure per calcolare il montante di un cliente**

- DELIMITER ~

- CREATE PROCEDURE montante\_cliente(IN \_id INT, OUT \_montante INT)

- BEGIN

- SELECT round(sum(montante),2) into \_montante from practice\_cqs where id = \_id;

- END;

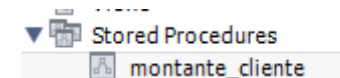
- DELIMITER ;

- #richiamo la procedura

- SET @\_montante = 0;

- CALL montante\_cliente(24, @\_montante);

- SELECT @\_montante;



# STORED FUNCTION

Oltre alle procedure è possibile salvare delle funzioni nel database, queste calcolano e restituiscono un valore, risultando simile ad una funzione in un linguaggio di programmazione.

La struttura di una stored function è la seguente:

Indichiamo che tipo di dato  
ritornare alla fine della funzione

Indichiamo se la funzione è  
deterministica  
(DETERMINISTIC) o no  
(NONDETERMINISTIC), cioè se  
dati gli stessi due valori di input  
il risultato della funzione  
rimane lo stesso o no

```
CREATE FUNCTION calcola_iva(prezzo DECIMAL(10,2), aliquota DECIMAL(5,2))  
RETURNS DECIMAL(10,2)  
DETERMINISTIC  
BEGIN  
    RETURN prezzo * aliquota / 100;  
END;
```

Creiamo la funzione con  
CREATE FUNCTION seguito dal  
nome della funzione e gli  
argomenti necessari, insieme  
alla tipologia di dato

Quando vogliamo ritornare il  
valore al termine della funzione  
dobbiamo precedere il valore  
con il comando RETURN  
seguito dal valore o variabile

La funzione, una volta creata è chiamabile in qualsiasi query  
usando il nome di essa seguita da i suoi argomenti, se presenti.

```
SELECT calcola_iva(100, 22);
```

# ESEMPIO

- DELIMITER \$\$
- **CREATE FUNCTION** get\_fullname(\_id INT)
- returns TEXT DETERMINISTIC
- BEGIN
- DECLARE fullname TEXT;
- SELECT concat(cognome, ' ', nome)  
into fullname FROM corsisti where id =  
\_id;
- return fullname;
- END \$\$
- DELIMITER ;

- DELIMITER ~
- #DROP function calcola\_sconto ~
- **CREATE FUNCTION** calcola\_sconto(\_importo  
DECIMAL(18,2), \_sconto\_percentuale  
DECIMAL(18,2))
- RETURNS DECIMAL(18,2)
- DETERMINISTIC
- BEGIN
- RETURN ROUND(\_importo - \_importo \*  
\_sconto\_percentuale / 100, 2);
- END ~
- DELIMITER ;
- **SELECT calcola\_sconto(25.50, 10.50);**

# STORED FUNCTION

le differenze tra procedura e funzione sono:

- Output

Le funzioni **ritornano un singolo valore alla fine delle istruzioni** mentre le procedure non possono ritornare direttamente valori (OUT non è un ritorno diretto)

- Chiamata

Per eseguire una procedura dobbiamo usare un comando specifico (CALL) mentre **per le funzioni possiamo anche usarle all'interno delle query**


- Usi Principali

Le funzioni possono essere usate per **calcolare o trasformare dati** mentre le procedure vengono usate per eseguire operazioni complesse

## **Conclusione: quando usare le stored procedure?**

Caso d'uso	Stored Procedure	Query Normale
Query complessa con più operazioni	✓ Sì, è più efficiente	✗ No
Query semplice su una tabella indicizzata	✗ No	✓ Sì
Evitiamo il traffico tra Codice e MySQL	✓ Sì	✗ No
Necessità di sicurezza avanzata	✓ Sì	✗ No
Utilizzo della cache MySQL	✗ No	✓ Sì

### Regola generale:

- Per query complesse e operazioni ripetute, le stored procedure sono migliori.
- Per query semplici su dati ben indicizzati, una query normale può essere più veloce. 

# SIGNAL

Un comando utile che possiamo usare durante la creazione di funzioni o procedure è il comando SIGNAL, questo consente di gestire errori personalizzati e consente di implementare delle logiche di controllo.

Quando usiamo SIGNAL dobbiamo seguirlo con SQLSTATE che indica che verrà ritornato un codice stato di SQL, il codice stato a 5 cifre tra virgolette e opzionalmente il messaggio d'errore.

Il codice stato che usiamo indica il tipo di errore che vogliamo ritornare, nella maggior parte dei casi verrà usato il codice '45000' che indica un errore generico ma è possibile usare altri codici già definiti disponibile in

<https://www.ibm.com/docs/it/i/7.5?topic=code-s-listing-sqlstate-values>

```
SIGNAL SQLSTATE '45000'  
SET MESSAGE_TEXT = 'Errore personalizzato: operazione non consentita.';
```

```
ERROR 1644 (45000): Errore personalizzato: operazione non consentita.
```

```
CREATE PROCEDURE verifica_utente(id_utente INT)  
BEGIN  
    DECLARE utente_trovato INT;  
    SELECT COUNT(*) INTO utente_trovato FROM utenti WHERE id = id_utente;  
  
    IF utente_trovato = 0 THEN  
        SIGNAL SQLSTATE '45000'  
        SET MESSAGE_TEXT = 'Utente non trovato.';  
    END IF;  
END;
```



# SIGNALS - ESEMPIO

```
• DELIMITER //
```

```
• CREATE PROCEDURE CONTA_CORSI_2(IN min INT, OUT  
  count_result INT)
```

```
• BEGIN
```

```
•     SELECT COUNT(*) INTO count_result FROM  
  corsi;
```

```
•     IF count_result < min THEN
```

```
•         SIGNAL SQLSTATE '45000'
```

```
•         SET MYSQL_ERRNO = 1001,  
  MESSAGE_TEXT = 'Non ci sono corsi';
```

```
•     END IF;
```

```
• END;
```

```
• DELIMITER ;
```

```
• CALL CONTA_CORSI_2(5,@count);
```

```
• SELECT @count; -- Per vedere il valore  
  restituito
```

```
• DELIMITER ~
```

```
• #DROP PROCEDURE calcola sconto ~
```

```
• CREATE PROCEDURE calcolā sconto(IN _importo  
  DECIMAL(18,2),IN _sconto_percentuale  
  DECIMAL(18,2), OUT _risultato DECIMAL(18,2))
```

```
• BEGIN
```

```
•     IF _sconto_percentuale > 100 THEN
```

```
•         SIGNAL SQLSTATE '45000'
```

```
•         SET MESSAGE_TEXT = 'Errore: lo sconto  
  non può essere maggiore del 100%';
```

```
•     ELSE
```

```
•         -- Calcolo lo sconto e salvo il  
  risultato nella variabile di output
```

```
•         SELECT ROUND(_importo - _importo *  
  _sconto_percentuale / 100, 2) INTO _risultato;
```

```
•     END IF;
```

```
• END ~
```

```
• DELIMITER ;
```

```
• SET @risultato = 0;
```

```
• CALL calcola sconto(25.50, 10.50, @risultato);
```

```
• SELECT @risultato;
```

```
• CALL calcola sconto(25.50, 100.50,  
  @risultato);
```

```
• SELECT @risultato;
```

# TRIGGER

MySQL consente anche l'automatizzazione di query e operazioni tramite trigger, queste sono query che possiamo legare alle operazioni delle tabelle e che vengono eseguite prima o dopo l'operazione impostata.

La struttura dei trigger è la seguente:

Usiamo il comando CREATE TRIGGER seguito dal nome del trigger

Inseriamo il tipo di evento e su quale tabella vogliamo effettuare la query

Quando andiamo a scrivere il trigger abbiamo disponibili OLD e NEW per gestire i dati delle tabelle: OLD si riferisce al dato già presente in tabella che stiamo andando a modificare o eliminare, NEW si riferisce ai dati in entrata della query

Terminiamo la creazione del trigger con END

```
CREATE TRIGGER before_insert_cliente
BEFORE INSERT ON clienti
FOR EACH ROW
BEGIN
    IF NEW.data_creazione IS NULL THEN
        SET NEW.data_creazione = NOW();
    END IF;
END;
```

Prima di iniziare la query dobbiamo inserire FOR EACH ROW, questo indica di eseguire la query del trigger ad ogni riga influenzata dalla query iniziale e come per le Stored Procedures iniziamo la query con BEGIN

- INSERT INTO `crm\_its`.`corsi` (`nome`, `data\_inizio`, `data\_fine`)
- VALUES ('ciao', NULL, '2025-09-30');

- DELIMITER \$\$
- CREATE TRIGGER before\_insert\_corsi
- BEFORE INSERT ON corsi
- FOR EACH ROW
- BEGIN
- -- Se il campo data\_inizio non è stato fornito, generiamo un errore
- IF NEW.data\_inizio IS NULL THEN
- SIGNAL SQLSTATE '45000'
- SET MYSQL\_ERRNO = 1002, MESSAGE\_TEXT = 'Il campo data\_inizio è obbligatorio.';
- END IF;
- 
- -- Se inserito\_il non è stato fornito, imposta il timestamp corrente
- IF NEW.inserito\_il IS NULL THEN
- SET NEW.inserito\_il = NOW();
- END IF;
- END\$\$
- DELIMITER ;

27 21:20:27 INSERT INTO `cm\_its`.`corsi` (`nome`, `data\_inizio`, `data\_fine`) VALUES ('ciao', NULL, '2025-09-30') Error Code: 1002. Il campo data\_inizio è obbligatorio.

# TRIGGER

I trigger vanno legati ad una operazione su una specifica tabella, possiamo inoltre specificare se vogliamo eseguire il trigger prima (BEFORE) o dopo (AFTER) la query originale, le operazioni sul quale possiamo legare i trigger sono INSERT, UPDATE e DELETE.

```
CREATE TRIGGER before_insert_clienti
BEFORE INSERT ON clienti
FOR EACH ROW
BEGIN
    IF NEW.email NOT LIKE '%@%' THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Email non valida.';
    END IF;
END;
```

```
CREATE TRIGGER before_update_clienti
BEFORE UPDATE ON clienti
FOR EACH ROW
BEGIN
    IF OLD.email = NEW.email THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Email non modificata.';
    END IF;
END;
```

```
CREATE TRIGGER before_delete_clienti
BEFORE DELETE ON clienti
FOR EACH ROW
BEGIN
    IF OLD.id = 1 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'ID 1 non eliminabile.';
    END IF;
END;
```

```
CREATE TRIGGER after_insert_clienti
AFTER INSERT ON clienti
FOR EACH ROW
BEGIN
    INSERT INTO log_operazioni (operazione, id_cliente, dettagli)
    VALUES ('Inserimento', NEW.id, CONCAT('Cliente: ', NEW.nome));
END;
```

```
CREATE TRIGGER after_update_clienti
AFTER UPDATE ON clienti
FOR EACH ROW
BEGIN
    INSERT INTO log_operazioni (operazione, id_cliente, dettagli)
    VALUES ('Aggiornamento', NEW.id, CONCAT('Da: ', OLD.nome, ' A: ', NEW.nome));
END;
```

```
CREATE TRIGGER after_delete_clienti
AFTER DELETE ON clienti
FOR EACH ROW
BEGIN
    INSERT INTO log_operazioni (operazione, id_cliente, dettagli)
    VALUES ('Eliminazione', OLD.id, CONCAT('Cliente: ', OLD.nome));
END;
```

# TRIGGER

- È possibile avere più di un trigger legato allo stesso evento, in questo caso i trigger verranno eseguiti nell'ordine di creazione ma questo può essere modificato usando una clausola **FOLLOWS** o **PRECEDES** durante la creazione del trigger, queste, seguite dal nome del trigger già esistente consentono di gestire se il trigger che stiamo creando deve essere eseguito dopo (**FOLLOWS**) o prima (**PRECEDES**).

In questo caso `inizializza_data` verrà eseguito prima di `controllo_email`, impostando la data di creazione prima del controllo della email

```
CREATE TRIGGER controllo_email
BEFORE INSERT ON clienti
FOR EACH ROW
BEGIN
    IF NEW.email NOT LIKE '%@%' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Email non valida.';
    END IF;
END;
```

```
CREATE TRIGGER inizializza_data
BEFORE INSERT ON clienti
FOR EACH ROW
PRECEDES controllo_email
BEGIN
    IF NEW.data_creazione IS NULL THEN
        SET NEW.data_creazione = NOW();
    END IF;
END;
```

# VISTE

- Una vista (VIEW) in MySQL è una tabella virtuale basata su una query. Ti permette di semplificare query complesse, migliorare la sicurezza (nascondendo alcune colonne) e riutilizzare codice SQL in modo più efficiente.
- CREATE TABLE corsi (
  - id INT AUTO\_INCREMENT PRIMARY KEY,
  - nome VARCHAR(255) NOT NULL,
  - data\_inizio DATE,
  - data\_fine DATE,
  - inserito\_il TIMESTAMP DEFAULT CURRENT\_TIMESTAMP
- );
- Se vogliamo creare una vista che mostri solo i corsi attivi (cioè quelli con data\_fine non passata), possiamo fare così:
  - CREATE VIEW corsi\_attivi AS
  - SELECT id, nome, data\_inizio, data\_fine
  - FROM corsi
  - WHERE data\_fine >= CURDATE();

# VISTE

- Si possono assegnare permessi (GRANT) alle viste proprio come sulle tabelle. Questo è utile per limitare l'accesso ai dati sensibili.
- `GRANT SELECT ON crm_its.corsi_attivi TO 'user_read'@'localhost';`
- `REVOKE SELECT ON crm_its.corsi FROM 'user_read'@'localhost';`

# CAPITOLO 4: MYSQL INSTALLAZIONE E CONFIGURAZIONE



# SCELTA DI UN DATABASE RELAZIONALE

- Per i prossimi capitoli sarà necessario avere un database disponibile in locale. Per questo abbiamo diverse scelte sul database possibili:
- **MySQL – DBMS di proprietà di ORACLE**  
Il database più popolare, compatibile con molti linguaggi di programmazione, supporta le transazioni ed è usato in molte applicazioni
- **PostgreSQL**  
Supporta più tipi di dati come JSON, XML e Array, offre una scalabilità migliore e indici più avanzati ma ha meno compatibilità con applicazioni rispetto a MySQL
- **SQLite**  
Un database leggero e contenuto in un singolo file, usato principalmente in applicazioni mobile o dove non è disponibile molta memoria
- Per questo corso useremo **MySQL** vista la maggiore compatibilità con applicazioni nel mondo del lavoro e nelle prossime slide andremo a installare un'istanza in locale insieme ad un DBMS

# SCELTA DI UN DATABASE RELAZIONALE

- <https://dev.mysql.com/downloads/>

[Download MySQL Community Edition »](#)

- Per interagire con questi database o crearne di nuovi avremmo bisogno di un DBMS, in questo caso useremo MysqlWorkbench per la leggerezza e completezza delle feature
- Da cmd (terminale):

```
C:\Program Files\MySQL\MySQL Server 8.2\bin>mysql --version
mysql Ver 8.2.0 for Win64 on x86_64 (MySQL Community Server - GPL)
```

```
C:\Program Files\MySQL\MySQL Server 8.2\bin>mysql -u root -p
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 951
Server version: 8.2.0 MySQL Community Server - GPL
```

```
mysql> SHOW DATABASES;
```

Database
information_schema
mysql
performance_schema
sys

```
mysql> USE biblioteca;
```

```
Database changed
```

```
mysql> SHOW TABLES;
```

Tables_in_biblioteca
books
books_loans
users

3 rows in set (0.00 sec)

# IMPORTARE DATI IN MYSQL WORKBENCH

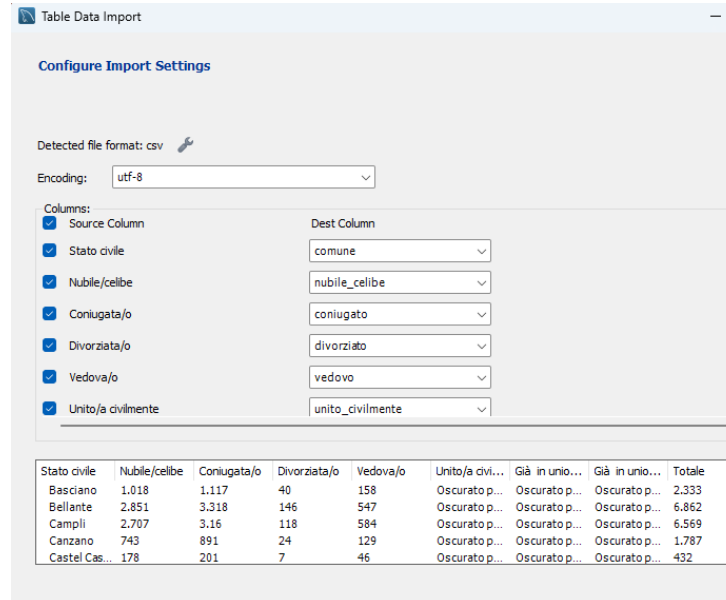
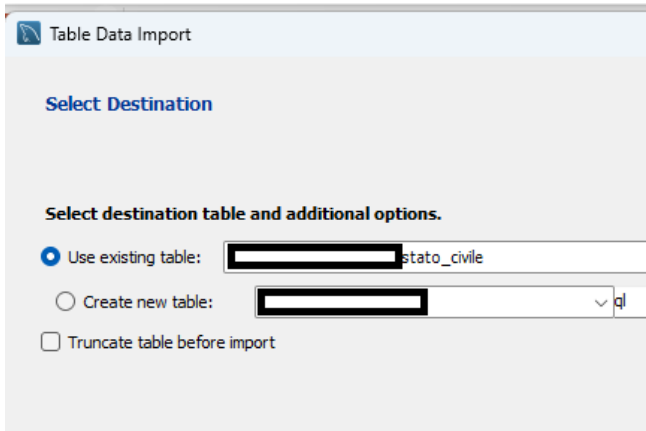
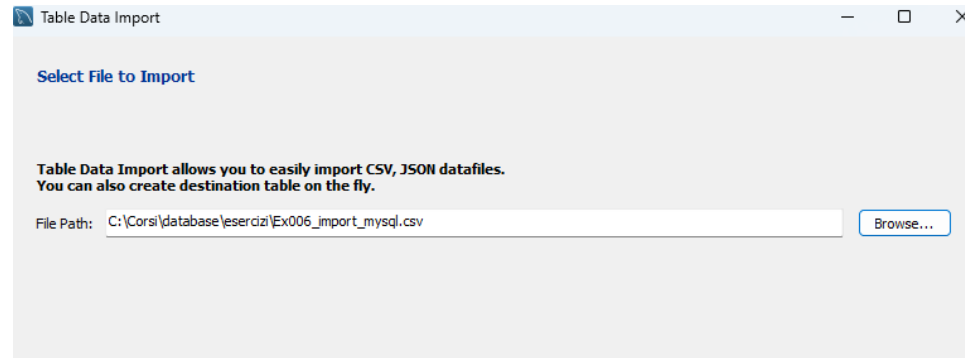
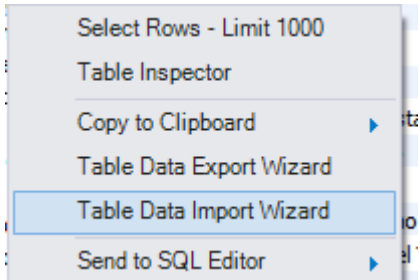
MYSQL WORKBENCH CONSENTE L'IMPORTAZIONE DI DATI DA TABELLE EXCEL CSV, QUESTO FORMATO È ABBASTANZA COMUNE NEI DATI SCARICATI DA SITI DI DATI OPEN.

PER INIZIARE L'IMPORTAZIONE:

- SELEZIONARE LE TABELLE DEL DATABASE CON IL TASTO DESTRO E SELEZIONARE "TABLE DATA IMPORT WIZARD"
- SELEZIONARE IL FILE CSV SCARICATO DA UN SITO DI OPEN DATA
- SELEZIONARE SE VOGLIAMO CREARE UNA NUOVA TABELLA PER I DATI IMPORTATI O USARNE UNA GIÀ ESISTENTE
- DOPO UNA PRIMA ANALISI DEL FILE VERRANNO MOSTRATE LE COLONNE CHE VERRANNO IMPORTATE, E SARÀ POSSIBILE SCEGLIERE QUALI COLONNE ESCLUDERE DALL'IMPORTAZIONE O CAMBIARE IL TIPO DI DATO

ESEGUITA LA QUERY E AGGIORNANDO LA CONNESSIONE I DATI IMPORTATI VERRANNO MOSTRATI ALL'INTERNO DELLA TABELLA SCELTA

# IMPORTARE CON IMPORT WIZARD DI MYSQL WORKBENCH



```
CREATE TABLE stato_civile  
(  
    comune VARCHAR(100),  
    nubile_celibe INT,  
    coniugato INT,  
    divorziato INT,  
    vedovo INT,  
    unito_civilmente INT,  
    unione_civile_decesso  
    INT,  
    unione_civile_scioglimento  
    INT,  
    totale INT  
);
```

# CAPITOLO 5: UTILIZZO DI PYTHON CON I DATABASE RELAZIONALI

# LIBRERIA PER COLLEGARE PYTHON CON MYSQL

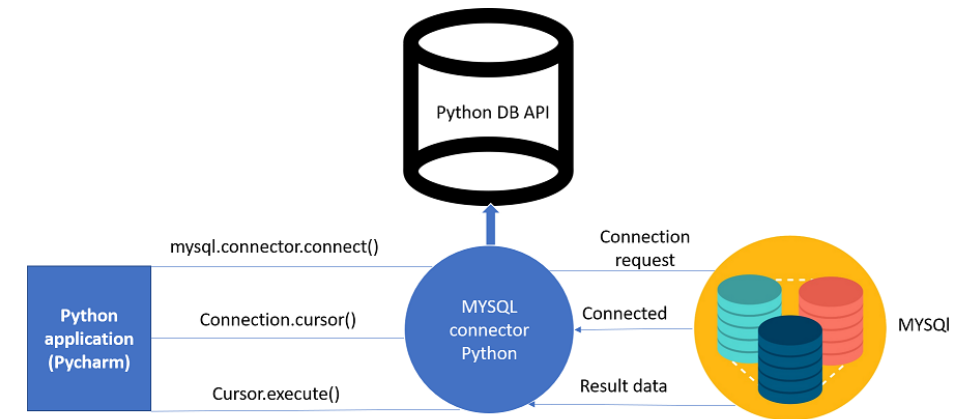
- Python è uno dei linguaggi di programmazione più utilizzati per gestire database grazie alla sua semplicità e vasta gamma di librerie.
- È ideale per creare applicazioni che interagiscono con database relazionali, come MySQL, PostgreSQL o SQLite.
- In questo capitolo, ci concentreremo sull'utilizzo della libreria **mysql-connector-python** per connettere Python a un database MySQL, eseguire query e manipolare i dati.
- Esistono altre librerie che svolgono la stessa funzione ma mysql-connector-python è l'unica libreria ufficiale di MySQL non che la più aggiornata per le ultime versioni del database, inoltre fornisce metodi più semplici per la gestione del collegamento e query

```
• #1. installare ambiente virtuale:
• #
• # pip install virtualenv
• # virtualenv env
• #env\Scripts\activate
• #oppure se già installata virtualenv
• # python -m venv env
• # source env/bin/activate      # Su
  Linux/macOS
• # env\Scripts\activate        # Su
  Windows
•
  # 2. installare connector
• # pip install mysql-connector-python
• #python --version
• #pip show mysql-connector-python
```

```
pip install mysql-connector-python
```

# CONNESSIONE & CURSORE

- La libreria mysql-connector-python così come altre librerie Python che interagiscono con i database fanno uso di 2 oggetti per svolgere l'elaborazione dei dati:
- **Connessione**  
Questo oggetto **rappresenta la connessione con il database**, richiede l'indirizzo del database, il nome e password dell'utente e il database al quale connettersi e consente la creazione di uno o più cursori i quali comunicheranno con la connessione che poi interagirà con il database per eseguire le query
- **Cursore**  
Questo oggetto, che viene generato dalla connessione, è ciò che **contiene le query e i risultati di essa dopo l'esecuzione**, il cursore interagirà solamente con la connessione dalla quale è stato generato, se si hanno più connessioni è necessario generare un cursore per ogni connessione che vogliamo usare



# CONNESSIONE AL DATABASE

- Per connettersi al Database e generare un oggetto connessione possiamo usare la funzione «connect\_sql» passando gli argomenti necessari.

```
def connect_sql(host, user, password, database):  
    conn = mysql.connector.connect(  
        host=host,  
        user=user,  
        password=password,  
        database=database  
    )  
    if conn.is_connected():  
        print ("Connessione Riuscita")  
    return conn
```

- Qui c'è un esempio di come applicare la funzione di connessione nel caso vogliamo chiedere all'utente a quale database si vuole collegare

```
sql_cursor.close()  
sql_conn.close()
```

In questo esempio possiamo definire una funzione che richiede gli argomenti necessari e ritorna la connessione. Possiamo anche interagire con la connessione per confermare il successo del comando. Questa funzione è utile nel caso vogliamo generare più connessioni o anche come funzione generica per più

script

```
sql_host = input("Inserisci l'host del database SQL: ")  
sql_user = input("Inserisci il nome utente per il database SQL: ")  
sql_password = input("Inserisci la password per il database SQL: ")  
sql_database = input("Inserisci il nome del database SQL: ")  
sql_conn = connect_sql(sql_host, sql_user, sql_password, sql_database)  
sql_cursor = sql_conn.cursor()
```

Alla fine dello script è necessario chiudere il cursore e la connessione per evitare che la connessione rimanga aperta nel database causando rallentamenti a lungo termine



# CONNESSIONE AL DATABASE

- Con la connessione possiamo usare una serie di funzioni o proprietà, alcune delle più importanti sono:
- `start_transaction()`, `commit()`, `rollback()`
  - Queste funzioni servono per utilizzare le transazioni senza chiamare direttamente la funzione tramite il cursore
- `Is_connected()`
  - Restituisce true se la connessione è attiva altrimenti viene restituito false
- `Reconnect(attempts = *, delay = *)`
  - Usabile quando manca la connessione al database, impostando il numero di tentativi (attempts) e il ritardo tra ogni tentativo (delay) verrà tentata la connessione con il database
- Database
  - Questa proprietà stampa il database selezionato attualmente

# UTILIZZO CURSORE

- Una volta generata la connessione possiamo creare un cursore utilizzando «oggettoConnessione».cursor()
- Questo genera un cursore legato alla connessione e con questo possiamo eseguire query e ricevere risultati:

- ```
sql_cursor = sql_conn.cursor()
```

Qui generiamo un cursore collegato alla connessione

- Con questo cursore possiamo eseguire qualsiasi comando SQL passandolo come argomento nella funzione execute

```
sql_cursor.execute("SQL COMMAND")
```

- ```
results = sql_cursor.fetchall()  
results = sql_cursor.fetchone()  
results = sql_cursor.fetchmany()
```

Dopo l'esecuzione possiamo usare uno dei vari fetch per ottenere i risultati

# UTILIZZO CURSORE

- Il cursore ha disponibili diverse metodi e proprietà utili per lo sviluppo, alcune delle quali sono:
- `execute(«SQL», «Values»), executemany(«SQL», «Values»)`
  - Il metodo esegue la query o comando inserito come argomento usando i valori se inseriti e salva i risultati nel cursore, una volta eseguito un SELECT non è possibile eseguire altri comandi su quel cursore fino a che non sono stati recuperati tutti i risultati. La variante `executemany` esegue la query per ogni valore nell'array dell'argomento `values`
- `fetchall()`
  - Il metodo prende tutti i risultati o quelli rimasti della query eseguita con `execute` e li ritorna come lista di tuple a meno che non specificato diversamente dagli argomenti del cursore
- `fetchone()`
  - Il metodo prende la prossima riga dei risultati e la ritorna come tupla, poi passa alla prossima riga
- `fetchmany(size=*)`
  - Questo metodo svolge la stessa funzione di `fetchone` ma consente un argomento `size` per configurare il numero di righe ritornate
- `column_names`
  - Questa proprietà ritorna i nomi delle colonne dei risultati
- `rowcount`
  - Questa proprietà ritorna il numero di colonne contenute nel risultato oppure il numero di righe modificate con INSERT o UPDATE
- `lastrowid`
  - Questa proprietà ritorna il contenuto della colonna AUTO\_INCREMENT dell'ultimo UPDATE o INSERT

# POSTGRESQL

UNO DEI DATABASE ALTERNATIVI A MYSQL PER APPLICAZIONI DI GRANDI DIMENSIONI PIÙ USATO È POSTGRESQL, UN DATABASE COMPLETAMENTE OPEN-SOURCE CONOSCIUTO PER LA SUA ROBUSTEZZA, MAGGIORE COMPATIBILITÀ NATIVA CON ALTRI LINGUAGGI, MAGGIORE CONFORMITÀ CON GLI STANDARD SQL PIÙ RECENTI E MAGGIORE SUPPORTO PER ESTENSIONI ESTERNE.

A DIFFERENZA DI MYSQL CHE UN DATABASE RELAZIONALE PURO, POSTGRESQL È UN DATABASE RELAZIONALE AD OGGETTI CHE UTILIZZA CONCETTI DELLA PROGRAMMAZIONE AD OGGETTI NELLA GESTIONE DEI DATI COME LA DEFINIZIONE DEI TIPI DI DATI E EREDITARIETÀ



# POSTGRESQL E MYSQL

Per Standard SQL si intende una serie di linee guida non obbligatorie per rendere più uniforme il linguaggio tra i vari DBMS

Caratteristica	PostgreSQL	MySQL
<b>Orientamento</b>	Database orientato agli standard e altamente estensibile.	Database semplice e veloce, orientato alle applicazioni.
<b>Standard SQL</b>	Conforme agli standard SQL più recenti (SQL:2011 e oltre).	Parzialmente conforme agli standard SQL.
<b>Modularità</b>	Supporta estensioni personalizzate (es. PostGIS).	Meno flessibile in termini di estensibilità.

# POSTGRESQL E MYSQL

Il modello MVCC è una tecnica di gestione dei dati di database che consente di ridurre al minimo i blocchi durante le operazioni di lettura e scrittura se effettuate da più persone allo stesso tempo

Caratteristica	PostgreSQL	MySQL
Transazioni complesse	Ottimizzato per query complesse e grandi dataset.	Ottimizzato per letture rapide e operazioni semplici.
Concorrenza	Usa il modello MVCC avanzato per ridurre i blocchi.	Concorrenza buona ma può richiedere più blocchi.
Velocità	Può essere più lento per operazioni semplici.	Generalmente più veloce per operazioni leggere.

Il modello MVCC funziona utilizzando un sistema che **salva molteplici copie del dato in elaborazione come versioni modificate**, questo consente, oltre ad essere una modalità alternativa di applicare transazioni, anche di poter consentire la lettura e scrittura dei dati, in modo anche contemporaneo, ad esempio mentre sta vendendo modificata una entry del database dei clienti, possiamo leggere da un altro punto di accesso il dato vecchio evitando blocchi nel caso vengano effettuate entrambe le operazioni allo stesso tempo.

# POSTGRESQL E MYSQL

PostgreSQL consente l'utilizzo di estensioni per aggiungere funzionalità extra come supporto per dati geografici, o una crittografia dei dati senza necessità di criptarli durante l'inserimento

Un'altra funzionalità è la possibilità di creare tipi di dati personalizzati per semplificare gli inserimenti.

Caratteristica	PostgreSQL	MySQL
Supporto JSON	JSON e JSONB con capacità avanzate (es. indici).	Supporto JSON, ma meno funzionale rispetto a PostgreSQL.
Estensioni	Supporta estensioni come PostGIS, CUBE, ecc.	Estensioni più limitate.
Tipi di dati personalizzati	Permette di definire tipi di dati customizzati.	Non supporta tipi di dati personalizzati.
Query	Supporta subquery, CTE, e query ricorsive avanzate.	Supporta subquery ma meno funzionali rispetto a PostgreSQL.

Tipi Compositi: rappresentano in un singolo elemento più oggetti

```
CREATE TYPE public.indirizzo AS
(
  via text,
  numero integer,
  citta text,
  cap text
);
```

Tipi enumerati: consente di impostare una serie di stringhe come valore

```
CREATE TYPE public.stato_cliente AS ENUM
('in_attesa', 'spedito', 'consegnato');
```

Domini: consente di impostare un tipo basato su uno già esistente ma con regole aggiuntive

```
CREATE DOMAIN public.cap_valido
AS text
NOT NULL;
ALTER DOMAIN public.cap_valido
ADD CONSTRAINT value CHECK (VALUE ~ '^\d{5}$');
```

# POSTGRESQL E MYSQL

UPSERT è una tipologia di INSERT che consente di inserire dati usando un indice specificato che, se già presente effettuerà un UPDATE al posto dell'operazione iniziale.

L'implementazione su PostgreSQL è effettuata utilizzando la clausola ON CONFLICT DO UPDATE

Caratteristica	PostgreSQL	MySQL
UPSERT	Supportato con <code>INSERT ... ON CONFLICT DO UPDATE</code> .	Supportato con <code>INSERT ... ON DUPLICATE KEY UPDATE</code> .
RETURNING	Permette di restituire righe modificate con <code>INSERT</code> , <code>UPDATE</code> , e <code>DELETE</code> tramite <code>RETURNING</code> .	Non supportato. Può essere simulato solo eseguendo una query aggiuntiva.
Limite nelle Query	Utilizza <code>LIMIT</code> e <code>OFFSET</code> per paginazione.	Supporta <code>LIMIT</code> e <code>OFFSET</code> , ma offre anche <code>SQL_CALC_FOUND_ROWS</code> (ora deprecato).

RETURNING è una clausola applicabile alle query di INSERT e UPDATE e consente la stampa delle righe inserite o modificate dalla query alla fine dell'esecuzione



# POSTGRESQL FULL JOIN

A differenza di MySQL che non implementa il FULL OUTER JOIN direttamente, PostgreSQL supporta la query direttamente senza dover eseguire molteplici query per lo stesso risultato.

Tabella `clienti`

id_cliente	nome
1	Mario
2	Lucia
3	Giovanni

Tabella `ordini`

id_ordine	id_cliente	prodotto
101	1	Smartphone
102	3	Laptop
103	4	Tablet

cliente_id	cliente_nome	ordine_id	prodotto_acquistato
1	Mario	101	Smartphone
2	Lucia	NULL	NULL
3	Giovanni	102	Laptop
NULL	NULL	103	Tablet

```
SELECT
    c.id_cliente AS cliente_id,
    c.nome AS cliente_nome,
    o.id_ordine AS ordine_id,
    o.prodotto AS prodotto_acquistato
FROM
    clienti c
FULL OUTER JOIN
    ordini o
ON
    c.id_cliente = o.id_cliente;
```

# POSTGRESQL E MYSQL

PostgreSQL supporta un numero di tipi di dati superiore a MySQL, grazie anche alle estensioni di terze parti.

Alcuni dei tipi più importanti sono:

- Booleani

A differenza di MySQL dove i booleani sono salvati come un TINYINT con un singolo carattere, PostgreSQL supporta i booleani con un tipo dedicato

- Intervalli

Un tipo di dato presente solo in PostgreSQL, rappresenta un intervallo di tempo rappresentato da una stringa contenente il numero e unità di tempo

- Array

Un altro tipo di dato che non esiste in MySQL, consente di memorizzare più valori in uno

- Geodati

Utilizzando l'estensione PostGIS possiamo consentire l'analisi e manipolazione di dati geografici come punti, linee, poligoni e coordinate geografiche, MySQL supporta i geodati ma non possiede la stessa gamma di funzioni di PostGIS

- JSON

Come per i geodati, questo tipo di dato è supportato anche in MySQL, ma in PostgreSQL è più ottimizzato e supporta un maggior numero di funzioni

# POSTGRESQL E MYSQL

La sicurezza dei database PostgreSQL è basata sull'applicazione di più metodi, oltre a quelli già presenti in MySQL come la protezione tramite password e il GRANT e REVOKE

- Ruoli  
PostgreSQL consente la creazione di ruoli assegnabili agli utenti, con i quali si possono gestire i diversi privilegi d'accesso ai dati
- Row-Level Security  
Consente di definire il controllo degli accessi ad ogni riga di una tabella
- Supporto Crittografia  
PostgreSQL supporta nativamente la cifratura SSL/TLS, espandibile alle colonne tramite l'estensione pgcrypto
- Log accessi e operazioni  
PostgreSQL consente di registrare ogni tentativo di connessione, accesso o modifica ai dati tramite i file di log. Per il log delle operazioni è possibile usare l'estensione pgaudit
- Responsabili  
Ogni volta che viene creato un oggetto viene automaticamente assegnato il possessore dell'oggetto, l'utente possessore ha accesso completo all'oggetto e può assegnare permessi ad altri utenti

Caratteristica	PostgreSQL	MySQL
Autenticazione	Supporta metodi avanzati come GSSAPI, SSPI, Kerberos.	Supporta autenticazione con plugin e password.
Controllo accessi	Controllo granulare a livello di colonna o riga.	Controllo accessi limitato a tabelle.



# TABLESPACE

In PostgreSQL è possibile salvare dati all'interno di cartelle del sistema, questo può essere utile per espandere la memoria del database all'interno di altri supporti di archiviazione come hhd o ssd.

Di base ogni database genera 2 tablespace predefiniti: pg\_default e pg\_global

Questi tablespace non modificabili vengono usati per salvare oggetti quando non è specificato un tablespace sul quale salvare i dati (pg\_default) e per salvare oggetti usati globalmente (pg\_global)

Nel caso vogliamo creare nuovi tablespaces possiamo usare il comando CREATE TABLESPACE

```
CREATE TABLESPACE fast_disk_space  
LOCATION '/data/fast_storage';
```

Una volta creato possiamo specificare, durante la creazione di database e tabelle il tablespace sul quale salvare i dati con l'argomento TABLESPACE

```
CREATE TABLE utenti (  
  id SERIAL,  
  nome TEXT  
) TABLESPACE fast_disk_space;
```

```
CREATE DATABASE analytics  
TABLESPACE fast_disk_space;
```

Limitazioni:

- Permessi di sistema: La directory specificata deve essere accessibile dall'utente di sistema che esegue PostgreSQL.
- Non supporta oggetti globali: Oggetti condivisi, come le tabelle di sistema globali, rimangono nel tablespace pg\_global.
- Spostare un oggetto tra tablespace: Non è possibile direttamente; è necessario ricreare l'oggetto in un nuovo tablespace.

# INDICI POSTGRE

Come in MySQL, PostgreSQL supporta gli indici B-Tree e Hash, ma sono disponibili anche altri tipi di indici per operazioni complesse:

- GIN (Generalized Inverted Index)  
Indice progettato per righe con molti valori, mappa ogni valore ad un insieme di righe.  
Ideale per: Array, Documenti JSON, Text Search, e Tipi Geometrici
- GiST (Generalized Search Tree)  
Indice simile al B-tree ma che consente l'utilizzo di dati e operazioni personalizzati  
Ideale per: Dati Spaziali, Intervalli, Similitudini Stringhe
- BRIN (Block Range INdices)  
Indice progettato per dati sequenziali, diverso da altri dato che salva i valori minimi e massimi per ogni blocco di memoria  
Ideale per: Dati Temporal, Dati Ordinati

# BENCHMARK

## ◆ 1. Performance e Scalabilità

Criterio	MySQL	PostgreSQL
Lettura	✓ Più veloce per letture ad alta frequenza	♦ Leggermente più lento su query semplici
Scrittura	✓ Ottimizzato per alte velocità	♦ Più sicuro ma leggermente più lento
Transazioni	♦ Supporta, ma con meno funzionalità	✓ Migliore gestione ACID
Scalabilità	✓ Più scalabile su cluster (con InnoDB)	♦ Più adatto a sistemi <b>single-node</b>

### 👉 Conclusione:

- Se hai molte letture e necessiti di alte prestazioni, MySQL è più leggero e veloce.
- Se usi molte transazioni complesse con garanzie ACID, PostgreSQL è più affidabile.

# BENCHMARK

## ◆ 2. Funzionalità Avanzate

Caratteristica	MySQL	PostgreSQL
Supporto ACID	♦ Sì, ma meno avanzato	✓ Completo e robusto
JSON & Document Store	✓ Buono (ma meno potente)	✓ Migliore supporto JSON
Full-Text Search	♦ Funziona, ma limitato	✓ Più potente (supporta ranking)
Stored Procedures	✓ Sì (PL/SQL, ma meno flessibile)	✓ PL/pgSQL e altri linguaggi
Supporto GIS	✓ Presente (ma limitato)	✓ Migliore per dati spaziali

# INSTALLAZIONE POSTGRESQL

Per installare PostgreSQL andiamo su  
<https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>

e selezioniamo il download appropriato per il sistema operativo.

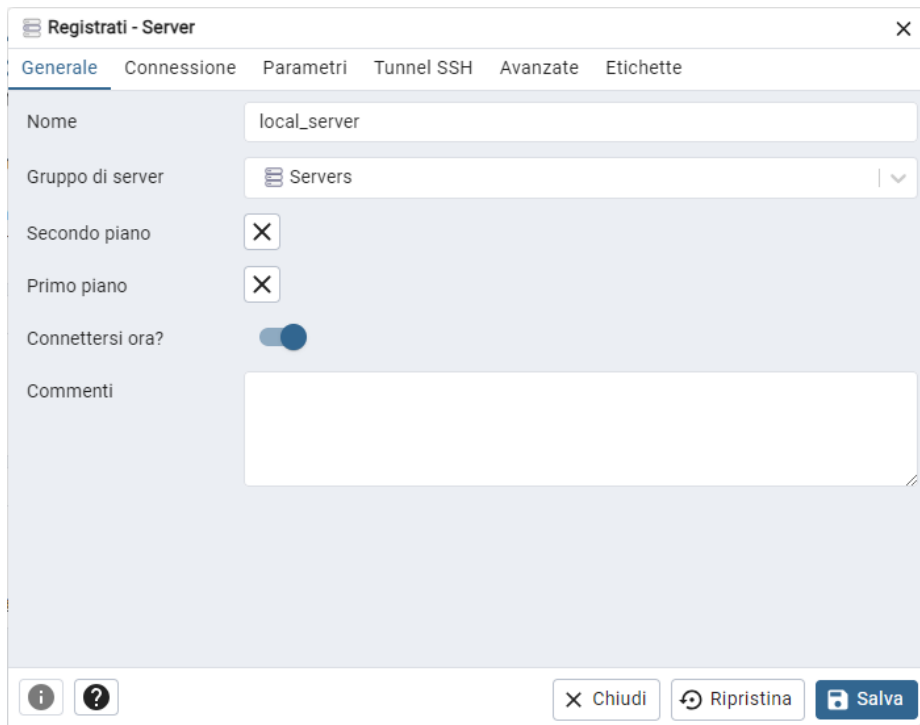
Questo installerà il server, il DBMS pgAdmin 4 e un gestore di estensioni «Stack Builder»

Lasciando tutte le impostazioni base, ci ritroveremo con un'istanza di PostgreSQL installata e funzionante nella porta 5432 con un utente amministratore «postgres»

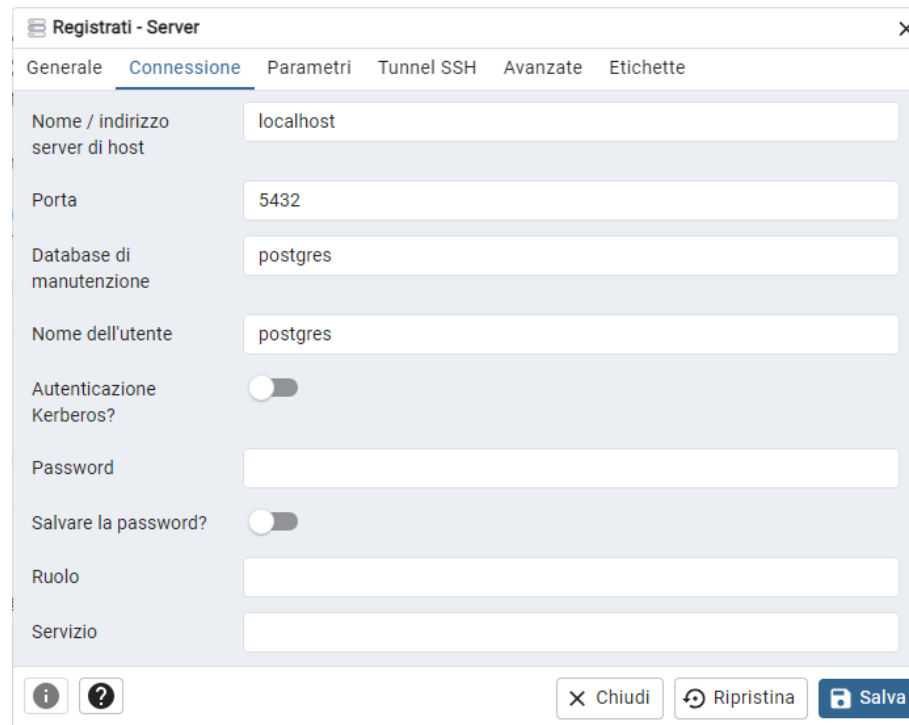


# PGADMIN 4

Una volta aperto pgAdmin, possiamo iniziare aggiungendo il server che abbiamo appena installato con «Aggiungi Nuovo Server», dove dopo aver dato un nome al collegamento, inseriamo come indirizzo nella tab connessione «localhost», inseriamo poi la password se è stata impostata



The screenshot shows the 'Registrati - Server' dialog box with the 'Generale' tab selected. The 'Nome' field contains 'local\_server'. The 'Gruppo di server' dropdown is set to 'Servers'. The 'Secondo piano' and 'Primo piano' checkboxes are both unchecked. The 'Connettersi ora?' toggle is turned on. The 'Commenti' field is empty. At the bottom, there are buttons for 'Chiudi', 'Ripristina', and 'Salva'.

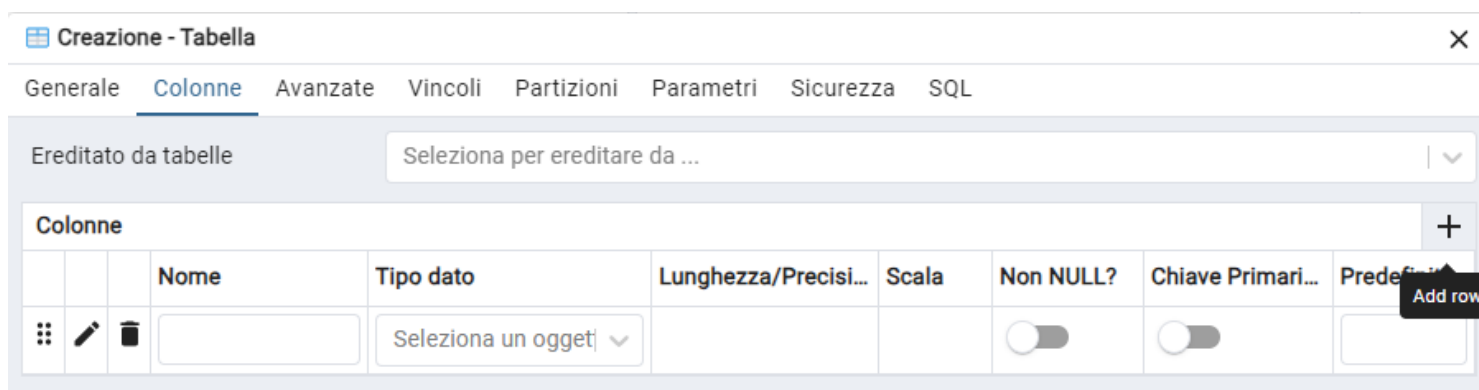
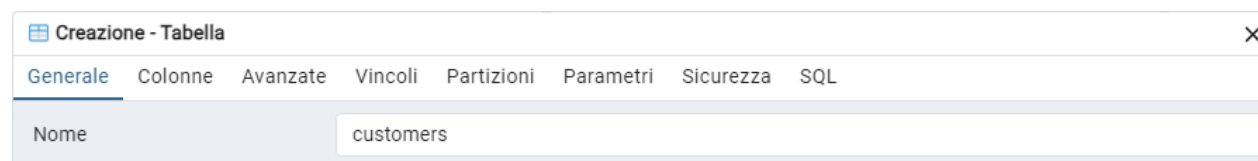
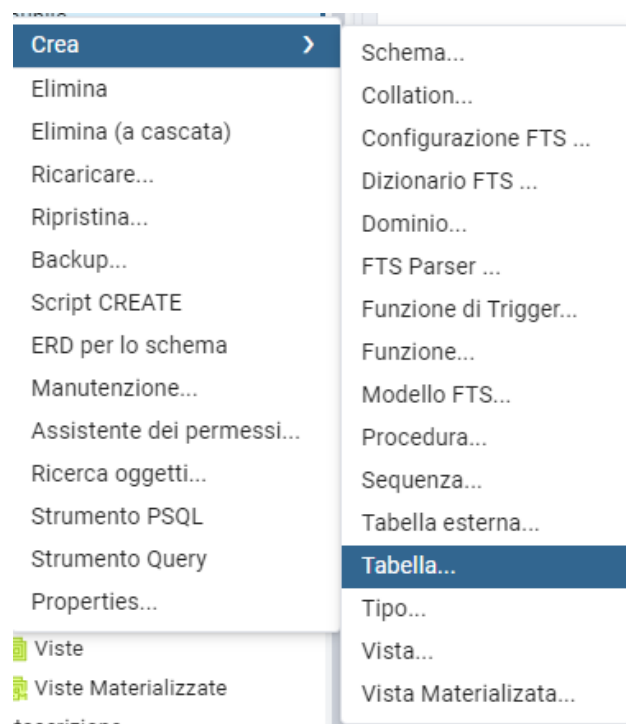


The screenshot shows the 'Registrati - Server' dialog box with the 'Connessione' tab selected. The 'Nome / indirizzo server di host' field contains 'localhost'. The 'Porta' field contains '5432'. The 'Database di manutenzione' field contains 'postgres'. The 'Nome dell'utente' field contains 'postgres'. The 'Autenticazione Kerberos?' toggle is turned off. The 'Password' field is empty. The 'Salvare la password?' toggle is turned off. The 'Ruolo' and 'Servizio' fields are empty. At the bottom, there are buttons for 'Chiudi', 'Ripristina', and 'Salva'.

# PGADMIN 4

Aggiunto il server verrà creato un database default, esso conterrà uno schema default «public», per creare tabelle su questo schema possiamo premere tasto destro sullo schema, «Crea -> Tabella», selezionare il nome della tabella e poi aprire la tab «Colonne» premere il tasto «+» per inserire una colonna.

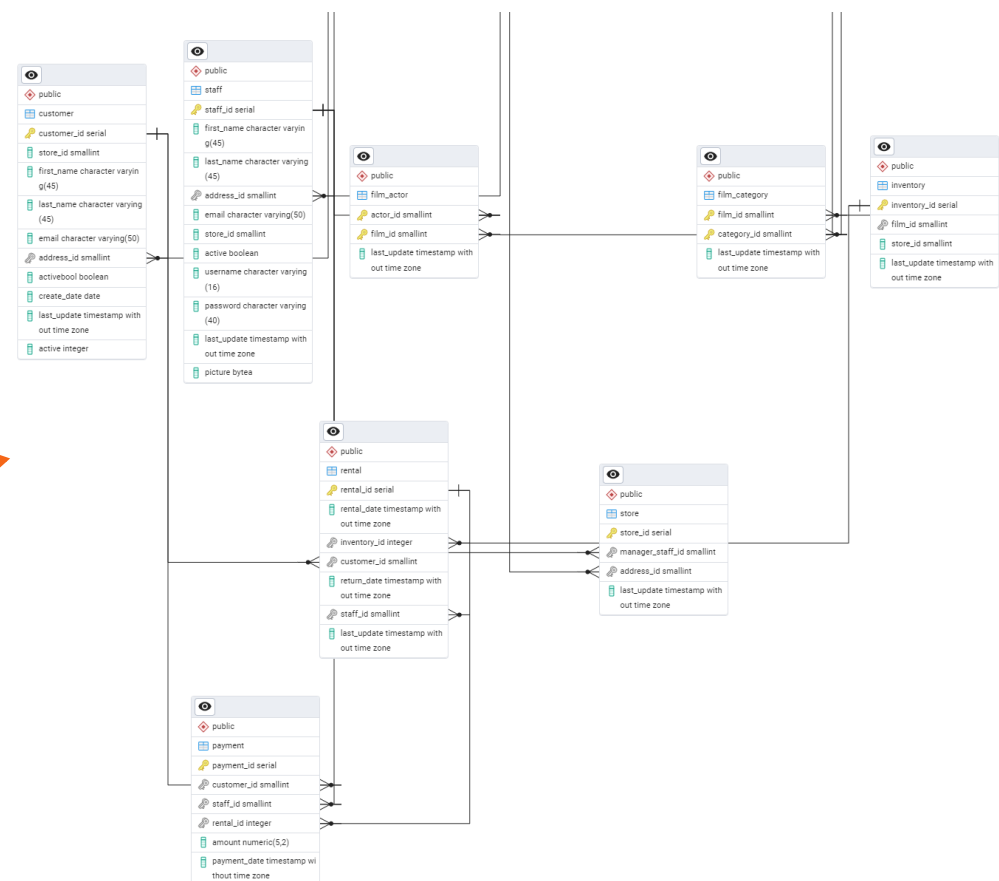
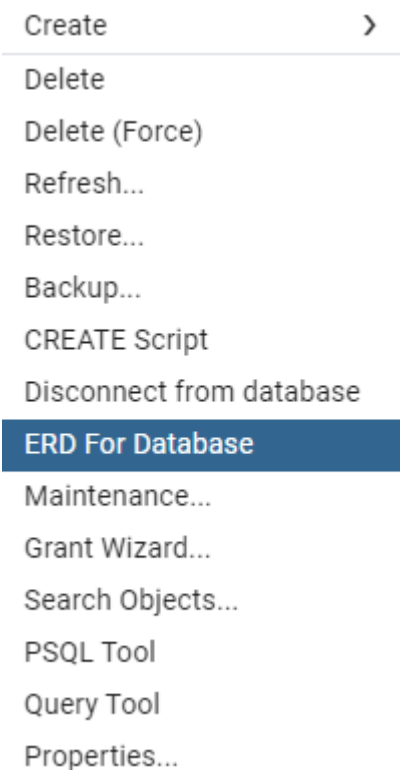
Se vogliamo impostare dei vincoli possiamo farlo dalla tab «Vincoli», qui possiamo impostare chiavi primarie, esterne o colonne uniche



# POSTGRESQL MODELLI ER

Come per altri database, PostgreSQL supporta la generazione di modelli ER fisici, tramite pgAdmin 4 o altri strumenti di terze parti (DBVisualizer, pgModeler, DBeaver).

Su pgAdmin 4:  
Cliccando mouse destro  
su un database:



# CORRISPONDENZA DI TIPI

Tipo in MySQL	Tipo in PostgreSQL	Descrizione
TINYINT	SMALLINT	Numeri interi piccoli (da -128 a 127 in MySQL; da -32,768 a 32,767 in PostgreSQL).
SMALLINT	SMALLINT	Numeri interi di media dimensione.
MEDIUMINT	INTEGER	Numeri interi di dimensione media (da -8,388,608 a 8,388,607 in MySQL; PostgreSQL usa <code>INTEGER</code> per intervalli più ampi).
INT , INTEGER	INTEGER	Numeri interi di base (da -2 miliardi a 2 miliardi in MySQL; da -2 miliardi a 2 miliardi in PostgreSQL).
BIGINT	BIGINT	Numeri interi molto grandi (da -9 quintilioni a 9 quintilioni).
FLOAT , DOUBLE	REAL , DOUBLE PRECISION	Numeri in virgola mobile (precisione variabile).
DECIMAL , NUMERIC	NUMERIC	Numeri decimali a precisione fissa (simile).
CHAR	CHAR , CHARACTER	Stringa di lunghezza fissa.
VARCHAR	VARCHAR , CHARACTER VARYING	Stringa di lunghezza variabile (simile, ma PostgreSQL usa <code>CHARACTER VARYING</code> come termine completo).
TEXT	TEXT	Testo di lunghezza variabile (molto grande).
BLOB , LONGBLOB	BYTEA	Dati binari (file, immagini, ecc.).
DATE	DATE	Data senza l'ora.
DATETIME	TIMESTAMP	Data e ora con precisione completa.
TIMESTAMP	TIMESTAMP	Data e ora (con aggiornamento automatico in PostgreSQL).
TIME	TIME	Ora del giorno (senza data).
YEAR	INTEGER	Anno (PostgreSQL non ha un tipo nativo per l'anno, ma <code>INTEGER</code> può essere usato).
ENUM	ENUM	Tipo di dato con un insieme di valori predefiniti.
SET	N/A	PostgreSQL non ha un tipo <code>SET</code> , ma può essere emulato con array o tabelle di supporto.

