

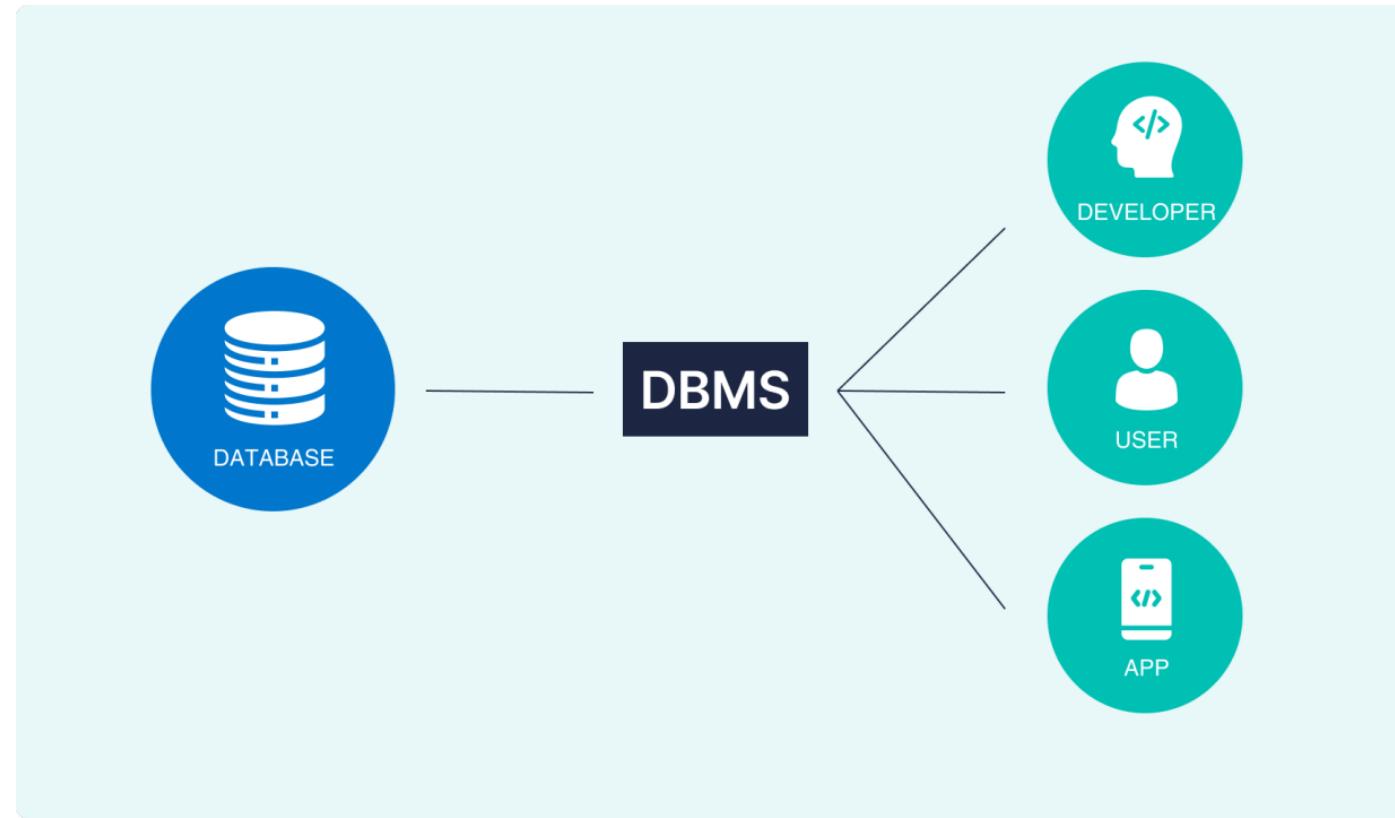
BASI DI DATI RELAZIONALI

Mauro Casadei

FONDAMENTI DEI DATABASE

COS'È UN DATABASE?

- Un Database è definito 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.

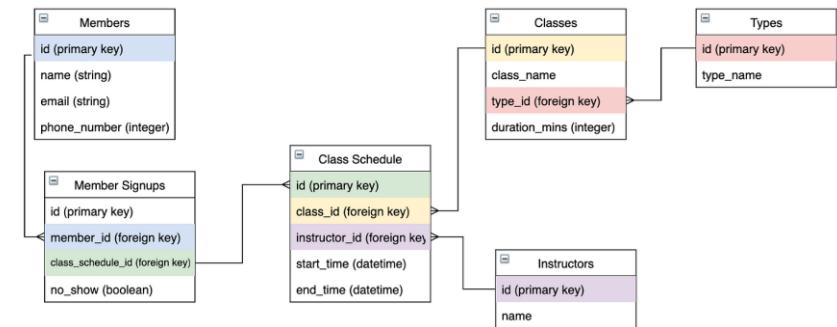
Alcuni Database
relazionali usati:

MySQL

PostgreSQL

Microsoft SQL Server

IBM DB2



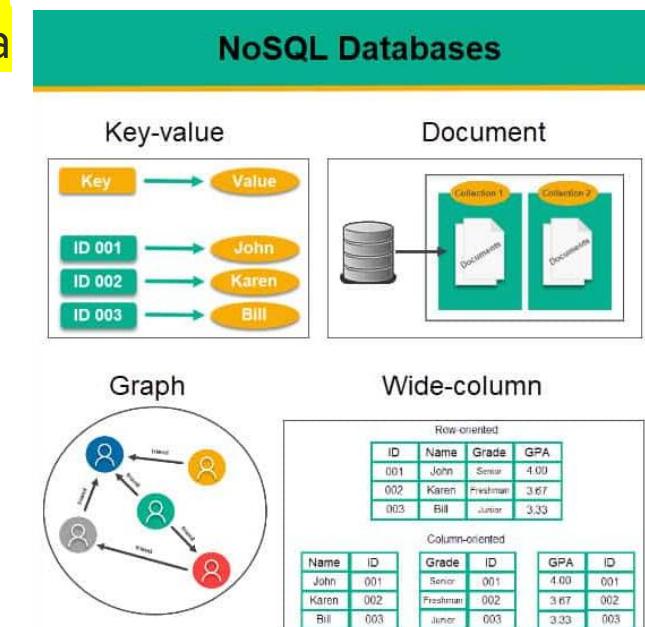
TIPI DI DATABASE

- 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 **mancanza 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 **mancanza** 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

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

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	

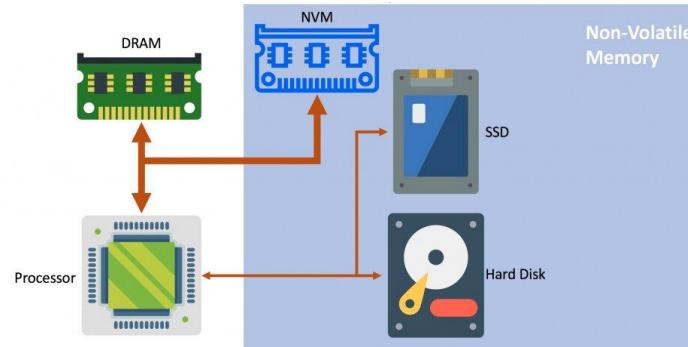
CONCETTI DI BASE

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

- **Persistenza**

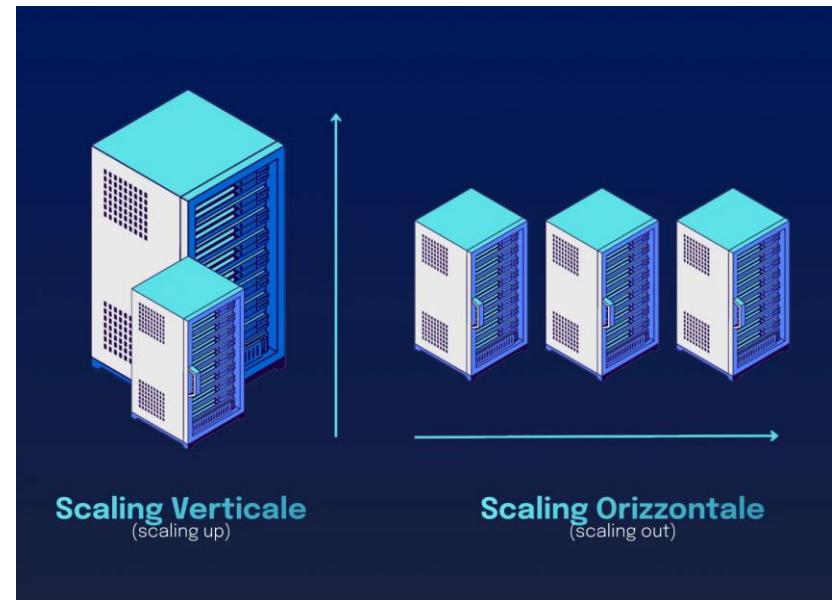
Nei Database il concetto di persistenza si riferisce alla **capacità di mantenere dati anche in caso di arresto anomalo o non.**

- 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	telefono
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 – 2NF – chiavi composte

Seconda Forma Normale

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

1. È già in **1NF**.

2. Ogni colonna deve dipendere **da tutta la chiave primaria**, non solo da una parte (**in caso di chiave composta**).

- Esempio di Tabella NON in 2NF
- Immagina una tabella per gestire gli ordini di un negozio:

order_id	product_id	product_name	quantity	unit_price
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**).

Tabella Orders

order_id	product_id	quantity
101	A1	2
101	A2	1
102	A1	1
102	A3	1

Tabella Products

product_id	product_name	unit_price
A1	Laptop	1200
A2	Mouse	20
A1	Keyboard	50



NORMALIZZAZIONE

QUIZ:

<https://gemini.google.com/share/4ocffffc4129>

Terza Forma Normale

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

È già in **2NF**.

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

studente_id	nomestudente	citta	cap
1	Marco	Pesaro	61121

- Il problema:
- StudentID → Citta
- Citta → CAP
- Quindi:
- CAP dipende da Citta
- ma Citta non è chiave primaria ✗
- ➡ Questa è dipendenza transitiva .

studenti		
id	nomestudente	citta_id
1	Marco	1

citta		
id	citta	cap
1	Pesaro	61121

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: INSTALLAZIONE MYSQL

- <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 |
+-----+
| test |
+-----+
```

```
mysql> USE biblioteca;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_biblioteca |
+-----+
| books
| books_loans
| users
+-----+
3 rows in set (0.00 sec)
```

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.

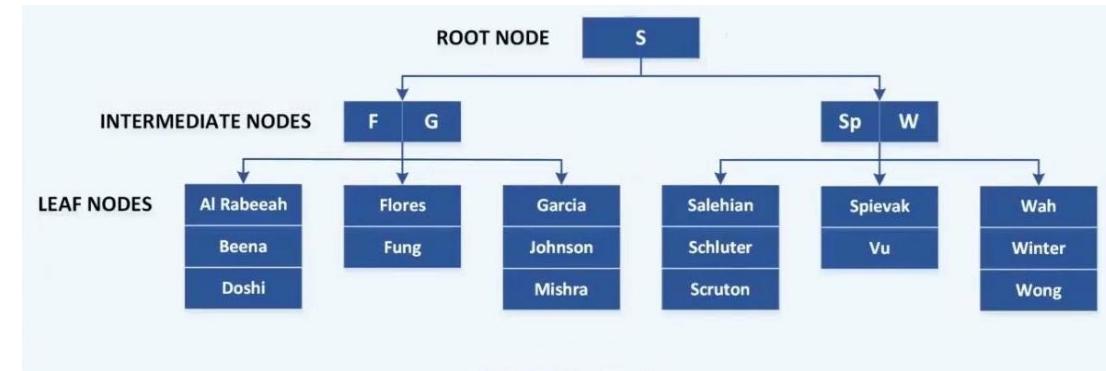
- Tipi di Indici:

- **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:

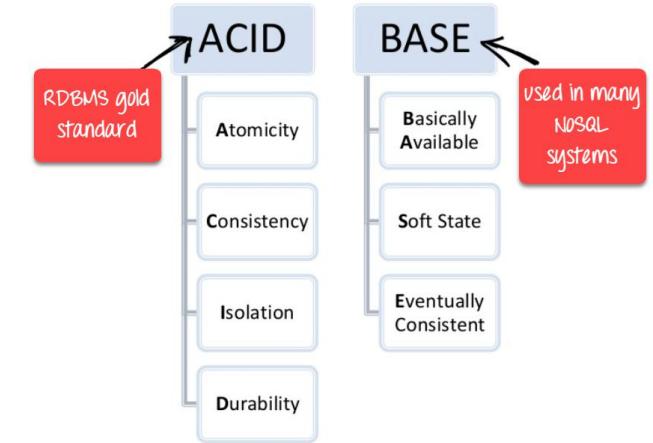
- Modello ACID (Relazionale):

- 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** (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** (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**.

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)
- Il **dettaglio ordine (righe del prodotto)** di un **ordine** è un'entità debole perché non può esistere senza un Ordine.

ENTITA ASSOCIATIVE (TABELLA PIVOT)

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 compositi** vengono compilati da altri attributi, che possono essere semplici o meno.
 - Es: Un indirizzo è un attributo composito 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}

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:

- {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 molti-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.

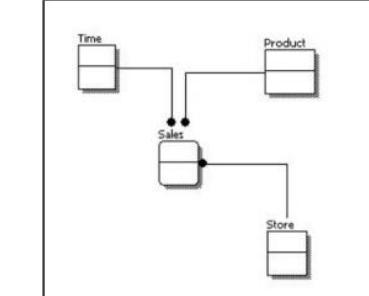
▪ Modello Fisico

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

non è più un
modello er
ma un
database
REALE

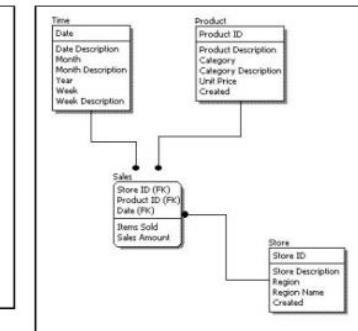
Modello ER
Concettuale

Conceptual Model Design



Modello ER
Logico

Logical Model Design



Modello ER
Logico

Physical Model Design

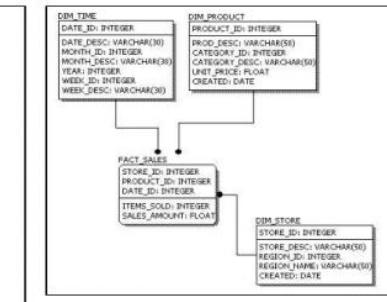
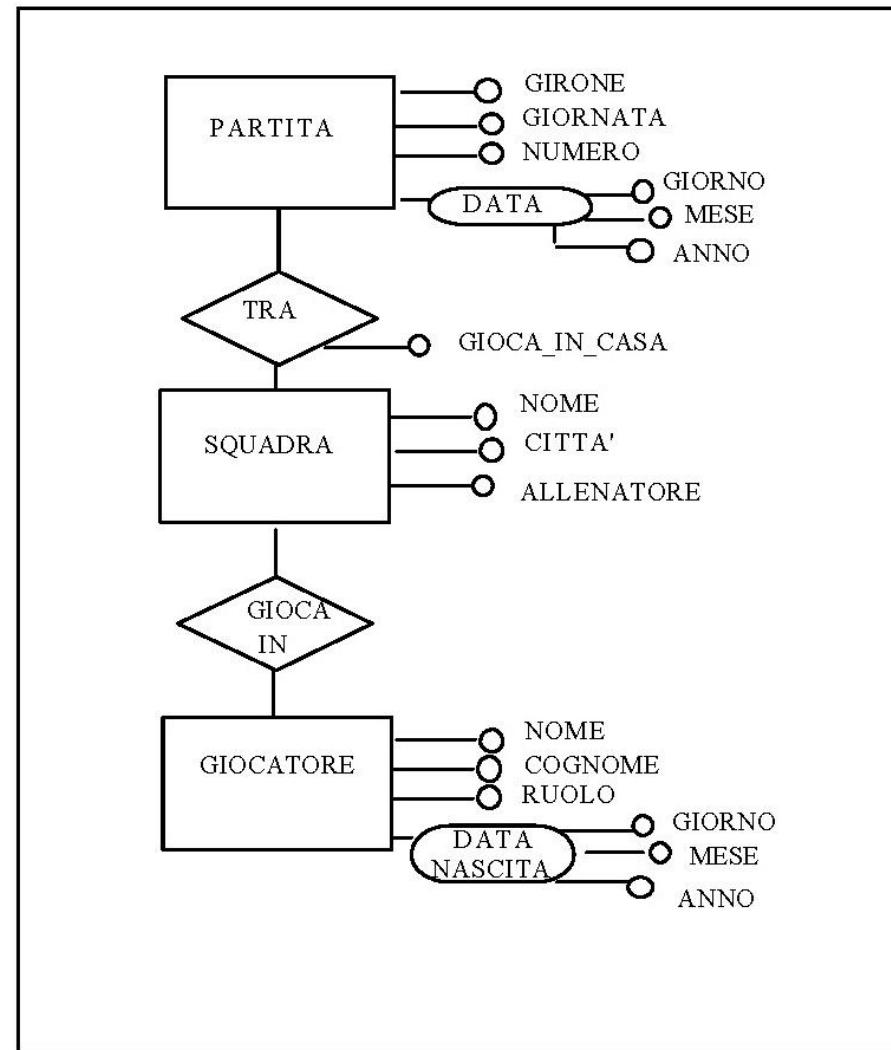


DIAGRAMMA ER – MODELLO LOGICO



RELAZIONI IDENTIFYING E NON IDENTIFYING

▪ Relazione Non Identifying (tratteggiata)

1:n

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:n

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) :

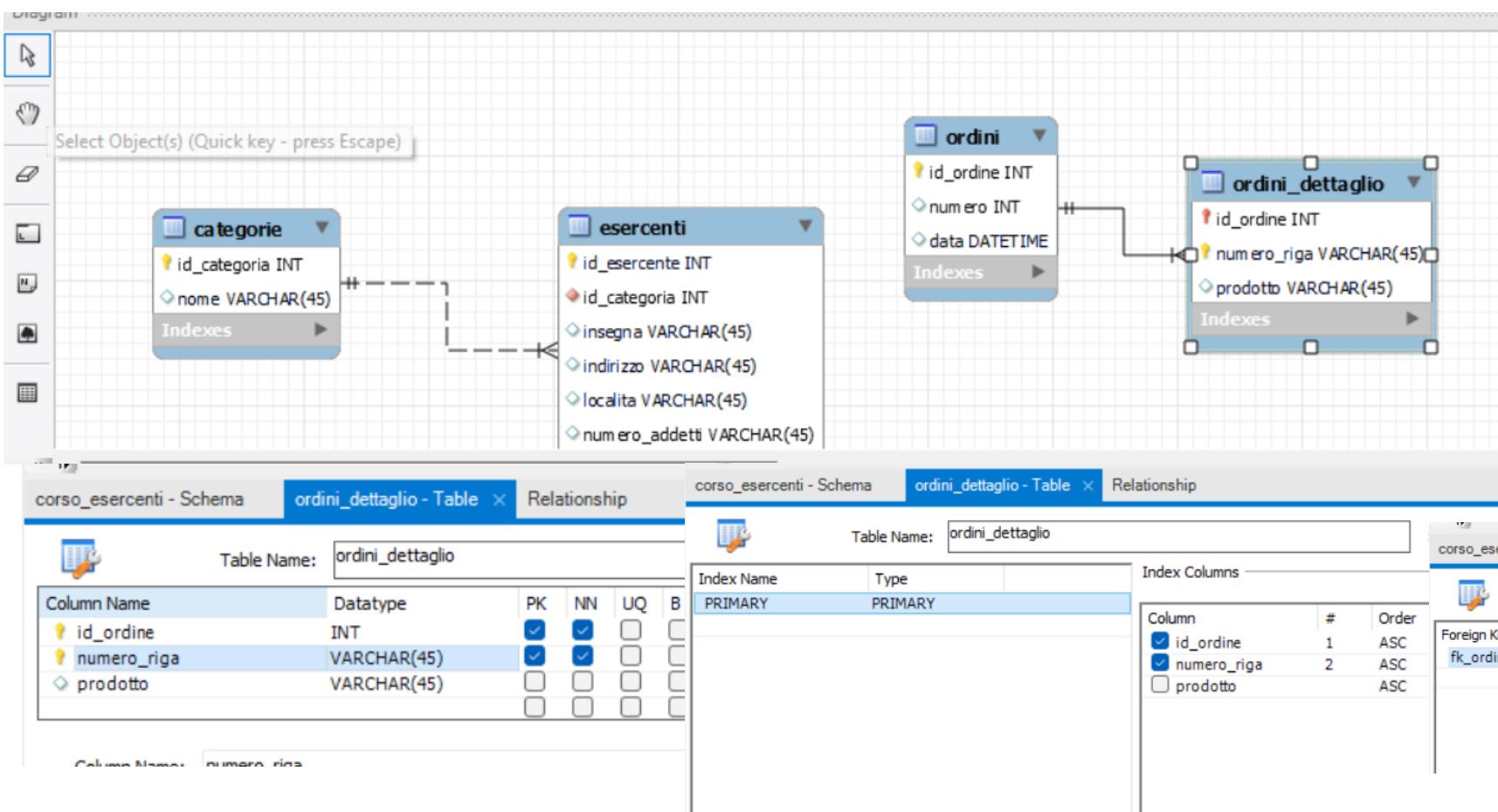
id_ordine (PK, FK)

id_prodotto (PK)

Quantita

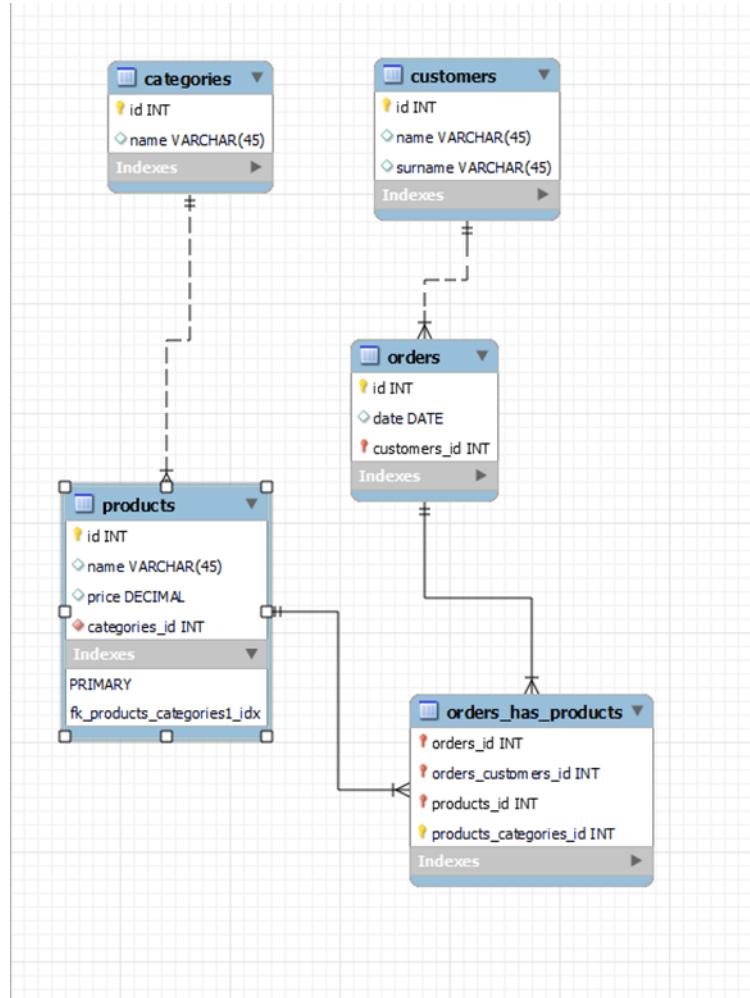
la chiave primaria di dettagli_ordine include una parte della chiave primaria di ordini, **un dettaglio ordine non può esistere senza ordine**

ESEMPIO IDENTIFYING

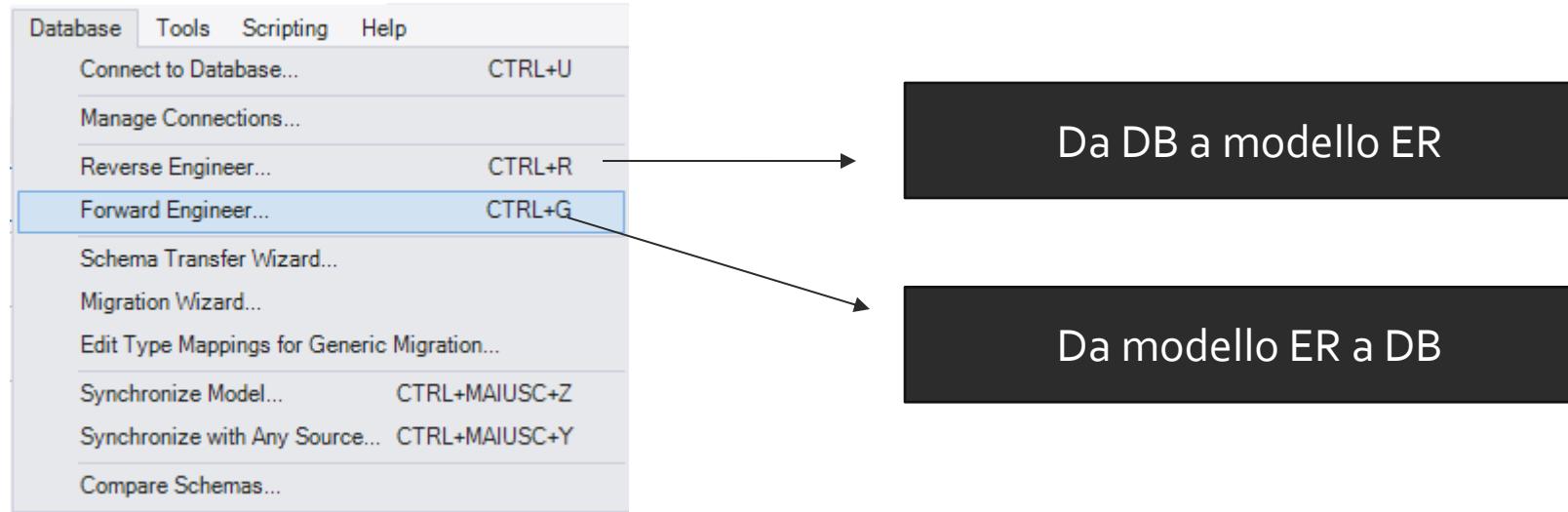


- ✓ Relazione identifying tra ordini → ordini_dettaglio
- ✓ Primary key composta (id_ordine, numero_riga)
- ✓ FK definita sulla sola colonna che punta alla tabella padre
- ✓ Colonna numero_riga NOT NULL, come richiesto da una PK

ESEMPIO DA EER DIAGRAM (FISICO) MYSQL WORKBENCH



MYSQL WORKBENCH E IL MODELLO ER



CLI DI MYSQL

- Il **CLI di MySQL** è il **Command Line Interface**, cioè il programma a riga di comando che permette di usare MySQL senza grafica, direttamente dal terminale.
- In altre parole:
-  **MySQL CLI = il client da terminale per eseguire comandi MySQL**
- È l'alternativa “testuale” agli strumenti grafici come MySQL Workbench o phpMyAdmin.

CLI DI MYSQL

- Se MySQL non è nel PATH, devi usare il percorso completo (es. Windows):
▪ cd "C:\Program Files\MySQL\MySQL Server 8.0\bin"
- mysql -u root -p
▪ Enter password: ****
- Welcome to the MySQL monitor...
- mysql> USE corso_esercenti;
- mysql> SELECT * FROM categorie;
- mysql> EXIT;

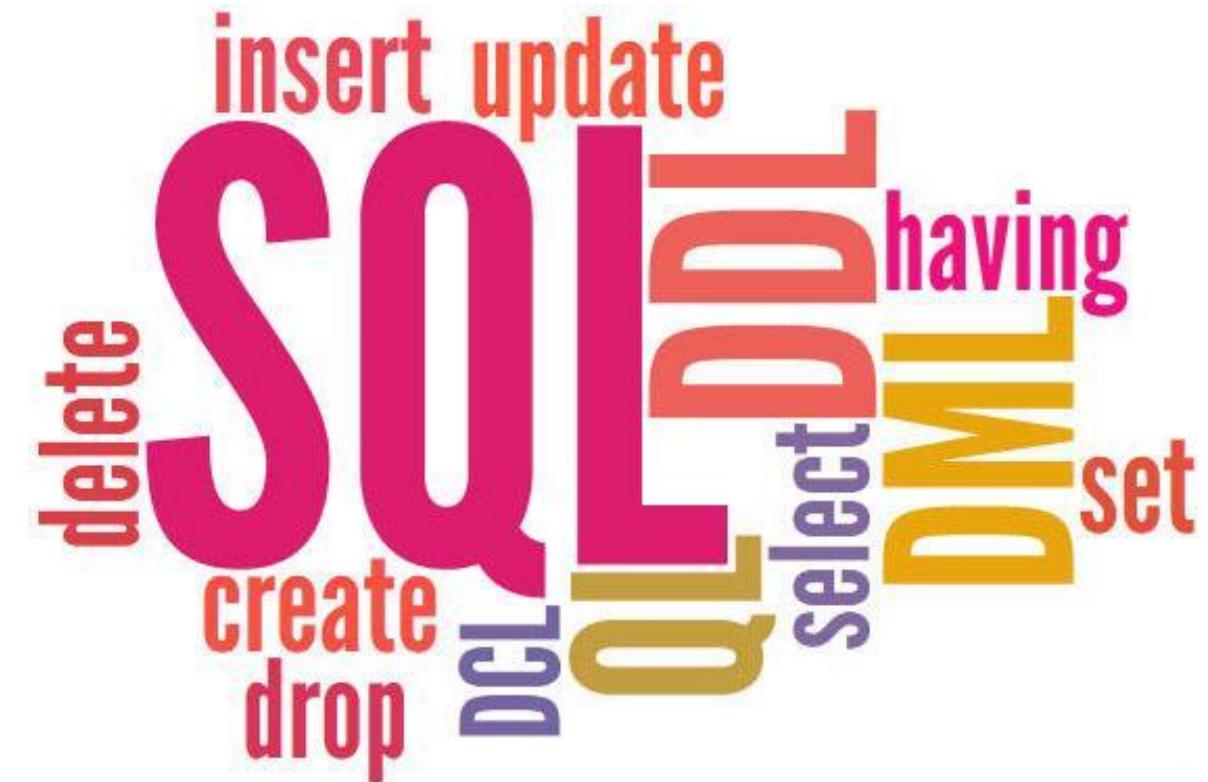
STRUTTURA DEI LINGUAGGI DI DATABASE

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 **CREATE** ha svariati usi, quelli più comuni sono:

- **CREATE DATABASE** `db_name`

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

- **CREATE TABLE** «db_name».<«table_name»>

(«field_name» «FieldType»,)

Questo comando crea una tabella all'interno del Database «db_name» una tabella di nome «table_name» inserendoci le colonne «field_name» di tipo «FieldType»

```
CREATE DATABASE database_prova;
```

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

LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

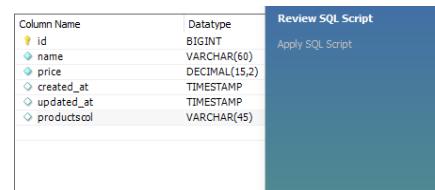
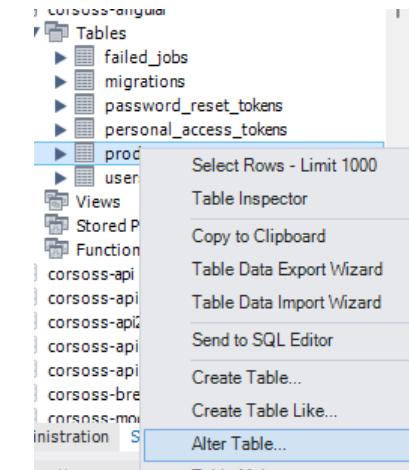
- Il comando **ALTER** viene per modificare tabelle o colonne già presenti:

- 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



```
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 **ALTER** viene per modificare tabelle o colonne già presenti:

- ALTER TABLE** «table_name»

Il comando viene usato per aggiungere, eliminare o modificare le colonne di una tabella con un nome definito dall'argomento TableName

- ALTER COLUMN** «column_name» «FieldType»

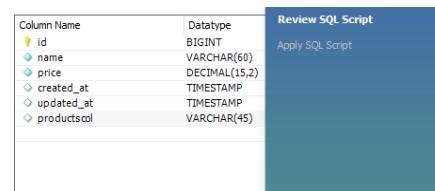
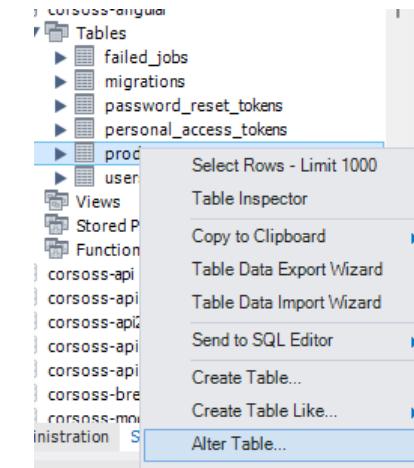
Questo comando è usato dopo ALTER TABLE per modificare il tipo di data contenuto in ColumnName al tipo FieldType

```
ALTER TABLE clienti  
ADD data_nascita DATE;
```

```
ALTER TABLE database_prova.clienti  
ALTER COLUMN telefono VARCHAR(20);
```

```
ALTER TABLE database_prova.clienti  
DROP COLUMN fax;
```

```
6 ALTER table 'ricambi'. 'tabella_di_prova' CHANGE column 'email' 'email' varchar(215);
```



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
```

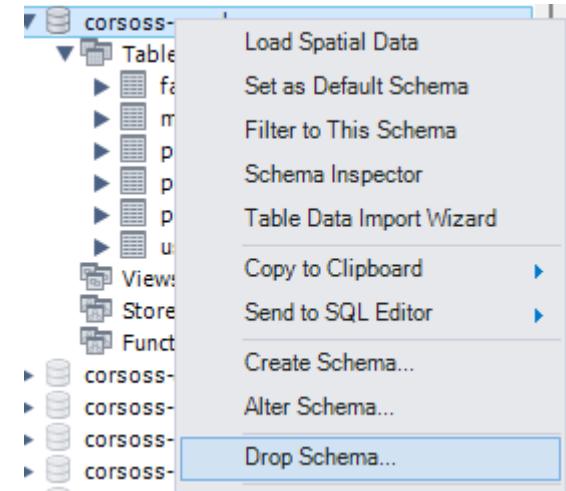
```
ALTER TABLE clienti  
RENAME COLUMN data_nascita TO data_di_nascita;
```

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** «table_name»
Il comando viene usato per eliminare una tabella e i dati in essa
- **DROP COLUMN** «column_name»
Questo comando è usato dopo ALTER TABLE per eliminare la colonna ColumnName e i suoi contenuti

```
DROP DATABASE database_prova;
```

```
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

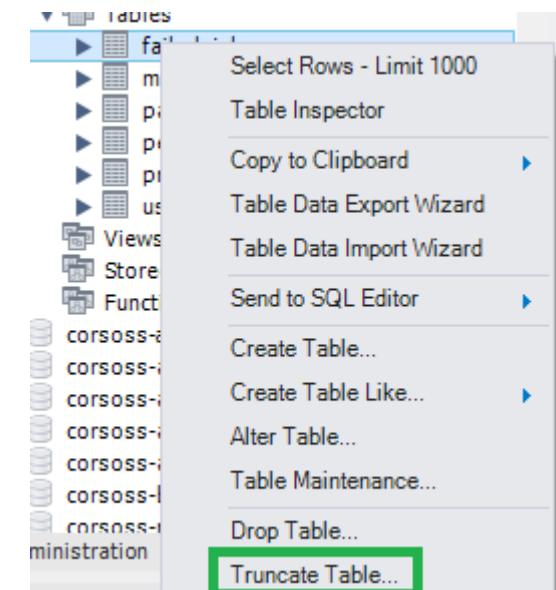
- TRUNCATE TABLE «table_name»

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;

```
SET FOREIGN_KEY_CHECKS=0;
SET
UNIQUE_CHECKS=0;TRUNCATE
`ricambi`.`workshops`;
SET FOREIGN_KEY_CHECKS=1;
SET UNIQUE_CHECKS=1;
```



LINGUAGGI DDL (DATA DEFINITION LANGUAGE)

- Il comando **RENAME** viene usato dopo per rinominare colonne o tavelle
- 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 ricambi.workshops rename column 'pec1' to 'pec';
```

```
ALTER TABLE clienti  
RENAME COLUMN telefono TO numero_telefono;
```

```
ALTER TABLE clienti  
RENAME TABLE clienti TO nuovi_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 «table_name»**

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 «table_name»**

Con questo comando si possono recuperare solo le colonne desiderate, riducendo il numero di dati visualizzati.

- **SELECT * FROM «table_name» WHERE «condition»**

Utilizzando la clausola WHERE, è possibile filtrare le righe in base a condizioni specifiche

- **SELECT * FROM «table_name» ORDER BY «column_name» [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(«column_name») AS «Alias» FROM «table_name»**

Questo consente di effettuare calcoli durante la selezione sulle colonne selezionate

ORDINE DI ESECUZIONE DEGLI ELEMENTI IN UNA QUERY

ordine di esecuzione DEGLI ELEMENTI in una QUERY:

FROM / JOIN: Vengono identificate le tabelle e unite.

WHERE: Filtra le singole righe prima del raggruppamento.

GROUP BY: Raggruppa le righe filtrate.

HAVING: Filtra i gruppi creati (agisce sui gruppi, non sulle righe).

SELECT: Proietta le colonne finali e calcola espressioni/funzioni di aggregazione.

ORDER BY: Ordina il risultato finale.

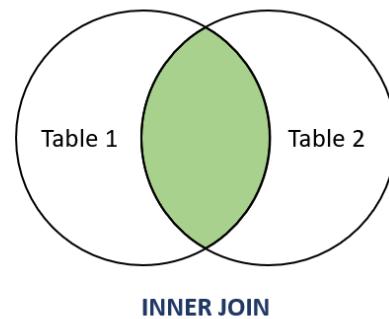
LINGUAGGI DML (DATA MANIPULATION LANGUAGE)

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

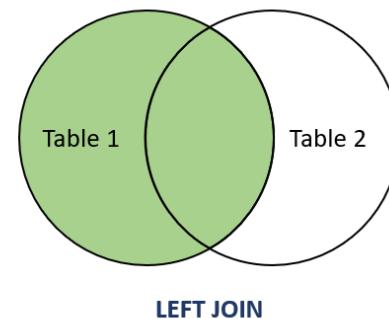
Tipologie principali di JOIN:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

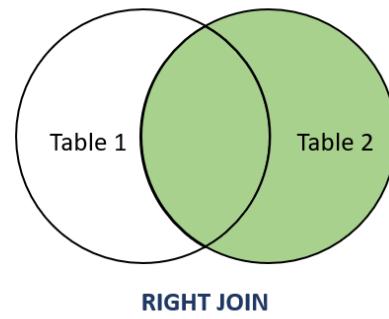
(in mysql non disponibile, utilizzare UNION)



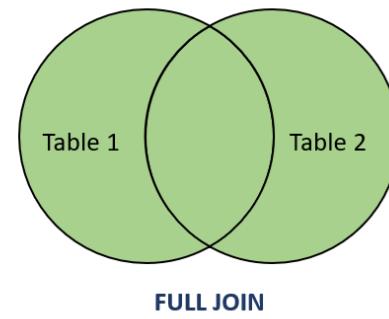
INNER JOIN



LEFT JOIN



RIGHT JOIN



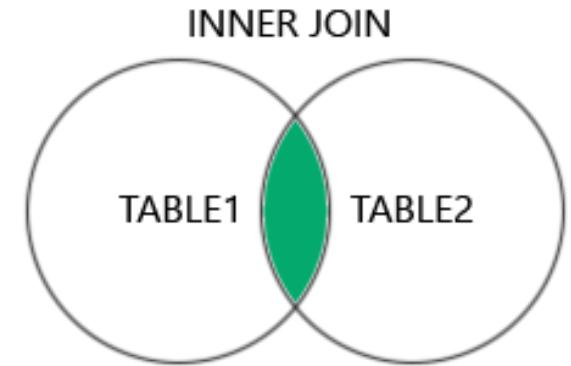
FULL JOIN

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:

```
ALTER TABLE clienti  
ADD COLUMN nome varchar;  
  
SELECT clienti.nome, ordini.id  
FROM clienti INNER JOIN ordini ON clienti.id = ordini.cliente_id;
```



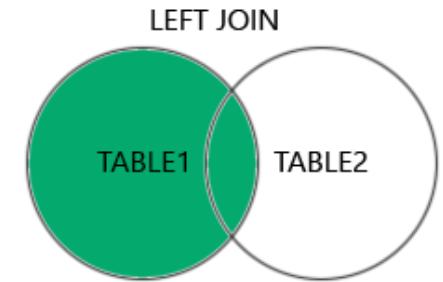
Così verrà mostrato il nome e l'id dell'ordine di tutti i clienti con almeno un ordine

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  
FROM clienti  
LEFT JOIN ordini  
ON clienti.id = ordini.cliente_id;
```



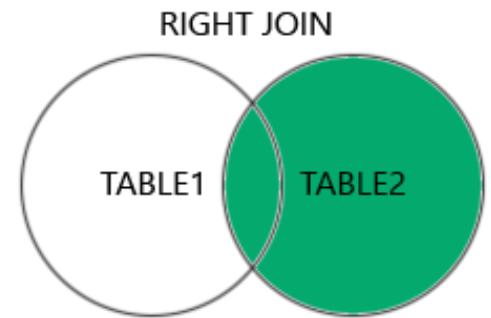
Così verrà mostrato il nome e l'id dell'ordine di tutti i clienti, se il cliente non ha un ordine associato verrà comunque mostrato, ma la colonna dell'id dell'ordine mostrerà NULL

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  
FROM clienti  
RIGHT JOIN ordini  
ON clienti.id = ordini.cliente_id;
```



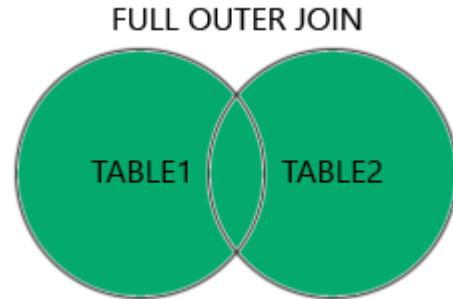
Così verrà mostrato il nome e l'id dell'ordine di tutti i clienti, se l'ordine non ha un cliente associato verrà comunque mostrato, ma la colonna del nome del cliente mostrerà NULL

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  
FROM clienti  
FULL JOIN ordini  
ON clienti.id = ordini.cliente_id;
```

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



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;
```

```
SELECT  
a.colonna_id AS id_a,  
b.colonna_id AS id_b,  
a.dato_a,  
b.dato_b  
FROM tabella_a AS a  
LEFT JOIN tabella_b AS b  
ON a.colonna_id = b.colonna_id  
  
UNION ALL  
  
SELECT  
a.colonna_id AS id_a,  
b.colonna_id AS id_b,  
a.dato_a,  
b.dato_b  
FROM tabella_a AS a  
RIGHT JOIN tabella_b AS b  
ON a.colonna_id = b.colonna_id  
WHERE a.colonna_id 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 eseguiremo

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

OPERATORI LOGICI

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

OVER

- La Clausola OVER (Window Functions)
- Definisce la "finestra" (il set di righe) su cui una funzione (come ROW_NUMBER(), AVG(), SUM()) deve operare.Componenti
- PARTITION BY (Opzionale)Raggruppa le righe.Il calcolo si resetta e riparte da 1 ogni volta che il valore di raggruppamento cambia
 - (es: PARTITION BY Dipartimento).
- ORDER BY (Spesso Obbligatorio)
- Definisce la sequenza.Determina l'ordine in cui le righe vengono contate o elaborate all'interno della finestra. Obbligatorio per le funzioni di ranking (ROW_NUMBER()).

```
select row_number() over (order by id), nam  
e from wines;
```

- SELECT
- c.nome,
- c.cognome,
- **ROW_NUMBER() OVER (**
- **ORDER BY id ASC -- Ordina le iscrizioni di quel corsista per modulo**
-) AS registrazione_progressiva
- FROM
- corsisti c
- ORDER BY
- registrazione_progressiva
ASC;

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

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

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.

```
select row_number() over (partition by producer_id, producer_id, name from wines;
```

MYSQL FUNCTIONS

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

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) → Estraе 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
- UPPER() converte in maiuscolo
- LOWER() converte in minuscolo
- **SELECT CONCAT('Hello', ' ', 'World'); -- Hello World**
- **SELECT CONCAT_WS('-', 'Hello', 'World'); -- Hello-World**

ESEMPI FUNZIONI SULLE STRINGHE

- SELECT CONCAT(cognome, nome) from customers;
- SELECT LENGTH(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_mesi; #2`
- `SELECT ADDDATE('2025-03-03', INTERVAL 10 DAY) AS nuova_data; #2025-03-13`

FUNZIONI NUMERICHE

- Per operazioni matematiche.
- ABS(x) → Valore assoluto
- ROUND(x, d) → Arrotonda al numero di decimali (d a quale decimale: 1= 1 decimale, 2=2 decimali)
- `SELECT id, ROUND(prezzo, 1) AS costo_arrotondato FROM corsi; // 1.23 = 1.2`
- CEIL(x) → Arrotonda per eccesso (unità)
- FLOOR(x) → Arrotonda per difetto (unità)
- MOD(x, y) → Resto della divisione
- RAND() → Numero casuale
- POWER(x, y) → Elevamento a potenza
- SQRT(x) → 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 sum(quantity*unit_price) from  
ricambi_mc.order_item where order_id=1
```

NON necessita di GROUP BY perché SUM
aggrega già

```
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;`

<code>id</code>	<code>montante</code>	<code>IF(montante>5000,'OK','KO')</code>
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)` → Estraе 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`

- Se ERROR 1064 (42000) ELIMINARE CARATTERI SPORCHI O BOM CON MODALITA
ESADECIMALE

ESEMPI FUNZIONI JSON

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

```
| JSON_UNQUOTE(  
| Mauro  
|/
```

RIMOSSED 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

- `SELECT` `CONCAT('-',TRIM('Maria'),'-');`

-Maria-

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 != '';
;`

✓ Mostro le righe 0 - 0 (1 del totale, La query ha impiegato 0,0004 secondi.)

`SELECT REGEXP_LIKE('abcdefgaadds', '[A-Z]{10}');`

Profiling [Modifica inline] [Modifica] [Spiega SQL] [Crea il codice PHP] [Aggiorna]

Mostra tutti

Numero di righe: 25

Filtra righe: Cerca nella tabella

Opzioni extra

`REGEXP_LIKE('abcdefgaadds', '[A-Z]{10}')`

1

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

REGEX

- Elementi Fondamentali:
- Caratteri Letterali:
- Ogni lettera, numero, simbolo o spazio ha un significato letterale.

```
1 • SELECT nome, cognome, cap_residenza, REGEXP_LIKE(cap_residenza, '[0-9]{5}')  
2 | as cap_corretto FROM corso_gestione_aula.corsiisti  
3 | where REGEXP_LIKE(cap_residenza, '[0-9]{5}') = 0;
```

nome	cognome	cap_residenza	cap_corretto
Marco	Rizzo	1123	0

```
SELECT REGEXP_LIKE(name, 'a*') FROM wines;  
SELECT REGEXP_LIKE(name, '.{2,}') FROM wines;
```

- Meta-caratteri:
- Simboli speciali con significati specifici nelle RegEx.
- Punto (.): Rappresenta qualsiasi carattere eccetto il ritorno a capo. Nelle [perde il suo significato]
- Asterisco (*): Indica zero o più occorrenze del carattere o gruppo precedente.
- Più (+): Indica una o più occorrenze del carattere o gruppo precedente.
- Punto interrogativo (?): Indica zero o una occorrenza del carattere o gruppo precedente.
- Accento circonflesso (^): Indica l'inizio della stringa.
- Dollaro (\$): Indica la fine della stringa.
- Barra verticale (|): Rappresenta un'opzione "o".
- Parentesi tonde (): Utilizzate per raggruppare parti dell'espressione.

REGEX

- Classi di Caratteri:
- Definiscono un insieme di caratteri.
- Parentesi quadre ([]): Contengono un set di caratteri da abbinare.
- Intervalli: Usati all'interno delle parentesi quadre per specificare un intervallo di caratteri.
- Sequenze di Escape:
- Utilizzate per rappresentare caratteri speciali o non stampabili.
- Backslash (\): Precede un meta-carattere per trattarlo come un carattere normale, ricordarsi nelle stringhe di mettere \\ altrimenti (singolo) \ sql lo rimuove

```
SELECT name FROM wines where REGEXP LIKE(name, '\d{1,}') = 1; # almeno 1 d
SELECT name FROM wines where REGEXP LIKE(name, '\Nd{1,}') = 1; # ignorato \, almeno 1 d
SELECT name FROM wines where REGEXP LIKE(name, '\\\d{1,}') = 1; # almeno 1 numero
```

REGEX

- Specificano il numero di occorrenze di un carattere o gruppo.
- Accolade ({}): Definiscono un numero esatto o un intervallo di ripetizioni.
- Questa sintassi di base fornisce le fondamenta per costruire espressioni regolari più complesse e potenti. Con la pratica, diventerà più facile creare pattern che rispondano alle tue esigenze specifiche, rendendo le RegEx uno strumento indispensabile per qualsiasi webmaster o professionista SEO.

- **Caratteri speciali**
- **\d** - Rappresenta una cifra (0-9).
- **\w** - Rappresenta una lettera, una cifra o un underscore (_). In pratica, \w corrisponde a qualsiasi carattere alfanumerico.
- **\s** - Rappresenta qualsiasi spazio bianco, inclusi spazi, tabulazioni e nuovi paragrafi.
- **^ Inizio della Stringa**
- **\$ Fine della stringa**
- **\D** - Rappresenta qualsiasi carattere che non sia una cifra (l'opposto di \d).
- **\W** - Rappresenta qualsiasi carattere che non sia una lettera, una cifra o un underscore (l'opposto di \w).
- **\S** - Rappresenta qualsiasi carattere che non sia uno spazio bianco (l'opposto di \s).

ESEMPI

- [a-z]{3}.* # almeno 3 lettere, seguite da qualsiasi altro carattere
- [a-z]{5} # almeno 5 lettere consecutive
- ([a-z]{2,})|([a-z]{5}) # Regex che cerca almeno 2 caratteri consecutivi o 5 lettere consecutive:
- [a-z]{6,}|\s{2,} // almeno 6 caratteri o 2 spazi consecutivi
- ^[A-Z0-9]{6}[A-Z]{8}[0-9]{2}[A-Z]{1}[0-9]{3}\\$ #codice fiscale
- ^[0-9]{5}\\$ #cap

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(*)  
FROM vini  
WHERE vn_IDGuida = 9  
GROUP BY vn_IDGuida;
```

```
SELECT vn_IDGuida, COUNT(*) AS totale  
FROM vini  
GROUP BY vn_IDGuida  
HAVING COUNT(*) > 10;
```

WHERE colonna = valore → giusto

HAVING COUNT / SUM / AVG ... → giusto

SUBQUERY EXISTS / IN ...

- 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);
```

```
SELECT name FROM wines w WHERE EXISTS (SELECT 1 FROM  
producers p WHERE p.id = w.producer_id AND p.company_name  
LIKE "%nov%");
```

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;
```

CAST

- Il CAST() in MySQL permette di convertire un valore da un tipo di dato a un altro.
- `SELECT CAST('2025-03-05 14:30:00' AS DATE);`
- Tipi di dati supportati
 - Numerici: SIGNED, UNSIGNED
 - Stringhe: CHAR, BINARY
 - Date/Time: DATE, DATETIME, TIME
- Esempi pratici:
 - `SELECT CAST('2025-03-05 14:30:00' AS DATE);`
 - `SELECT CAST(12345 AS CHAR);`
 - `SELECT CAST('100.55960' AS DECIMAL(5,2));`
 - `SELECT CAST('100.50' AS UNSIGNED);`

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** «table_name» **VALUES** (valore1, valore2, ...)

Questo comando aggiunge una nuova riga specificando i valori per tutte le colonne nella stessa sequenza definita dalla tabella.

- **INSERT INTO** «table_name» («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** «table_name» («column1», «column2») **SELECT** «column1», «column2»
FROM «other_table» **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 «table_name» SET «column_name» = valore**
Questo comando modifica il valore di una colonna per tutte le righe della tabella.
- **UPDATE «table_name» SET «column_name» = valore WHERE «Condition»**
Utilizzando la clausola WHERE, è possibile applicare modifiche solo alle righe che soddisfano una determinata condizione.
- **UPDATE «table_name» SET «column_name» = «column_name» + 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 «table_name»**

Questo comando rimuove tutte le righe della tabella, a differenza di TRUNCATE questo non reimposta l'auto-increment a 0 (zero)

- **DELETE FROM «table_name» WHERE «Condition»**

Utilizzando la clausola WHERE, è possibile eliminare solo le righe che soddisfano una determinata condizione.

- **Attenzione: una DELETE senza WHERE cancellerà 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 «table_name» TO «user_name»

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 «table_name» FROM «user_name»

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

LINGUAGGI TCL (TRANSACTION CONTROL LANGUAGE)

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

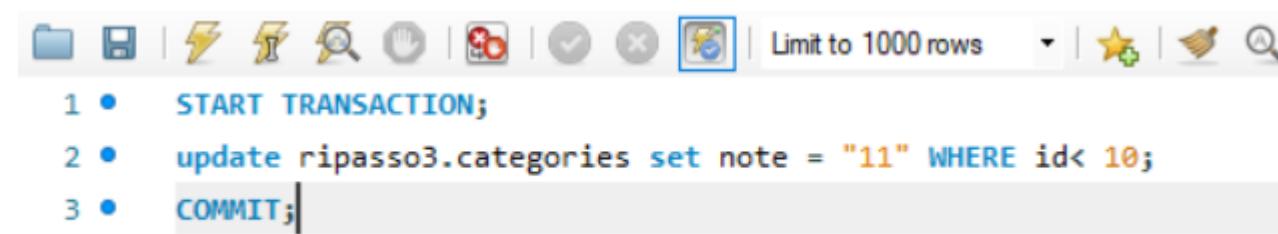
Permettono di controllare quando i cambiamenti ai dati vengono salvati o annullati.

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**



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

LINGUAGGI TCL (TRANSACTION CONTROL LANGUAGE)

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

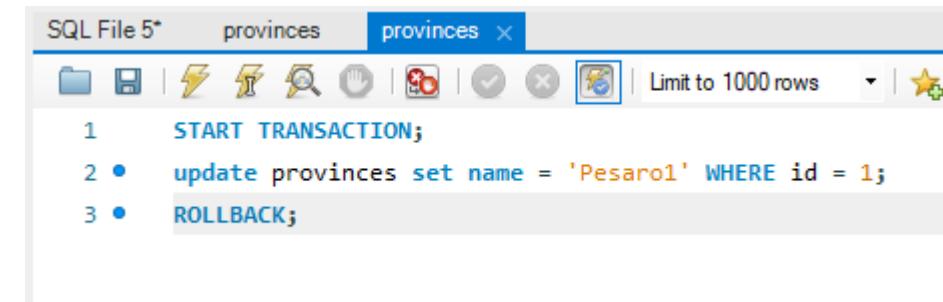
```
START TRANSACTION;  
UPDATE Prodotti SET Prezzo = 20 WHERE ID_Prodotto = 1;  
COMMIT;
```

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

LINGUAGGI TCL (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;
```



The screenshot shows a MySQL Workbench interface with a transaction log. The log contains three entries:

- 1 START TRANSACTION;
- 2 • update provinces set name = 'Pesaro1' WHERE id = 1;
- 3 • ROLLBACK;

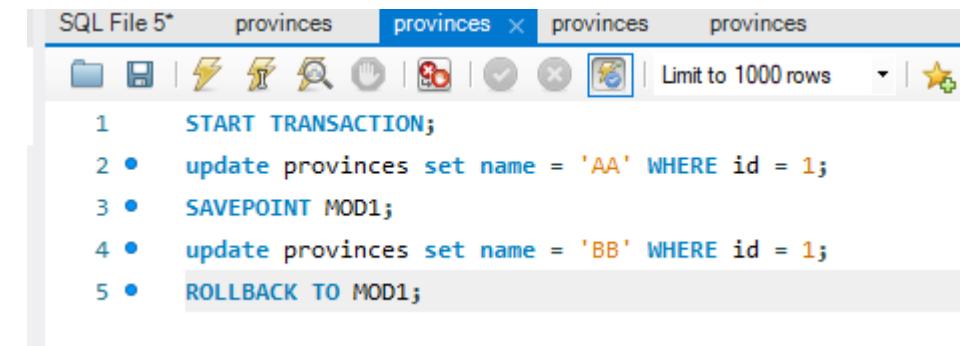
The 'Limit to 1000 rows' button is visible at the top right of the log area.

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

LINGUAGGI TCL (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;
```



```
SQL File 5* provinces provinces provinces provinces  
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;
```

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')

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

```
GRANT SELECT ON *.* TO 'guest'@'%'
```

```
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

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 tavole 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)
```

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

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

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 MySQL 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;

ESEMPIO

- DELIMITER @
- CREATE PROCEDURE calcola_iva(IN imponibile DECIMAL(16,2), IN aliquota DECIMAL(5,2), OUT totale DECIMAL(16,2))
- BEGIN
 - -- Calcola il totale
 - SELECT imponibile + (imponibile * aliquota / 100) INTO totale;
- END @

- DELIMITER ;
- SET @totale = 0; -- Inizializzazione della variabile @totale

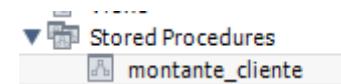
- CALL calcola_iva(100.52, 22, @totale); -- Chiamata alla stored procedure

- SELECT @totale; -- Visualizzare il risultato

ESEMPIO

- #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;



ESEMPIO _ calcolo totale ordini

- DELIMITER \$\$
- drop procedure order_total;
- create procedure order_total(IN p_order_id INT, OUT p_total DECIMAL(10,2))
- BEGIN
 - select sum(quantity*unit_price) into p_total from order_item where order_id=p_order_id group by order_id;
- END \$\$
- DELIMITER ;
- CALL order_total(1, @p_total);
- SELECT @p_total;

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

NON DETERMINISTIC: AD ESEMPIO UTILIZZANDO NOW() O CURDATE()

```
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

ESEMPIO CALCOLARE TOTALE ORDINE

- DELIMITER \$\$
- #DROP FUNCTION calculate_order;
- create function calculate_order(p_order_id INT)
- RETURNS DECIMAL(10,2)
- DETERMINISTIC
- BEGIN
- DECLARE total DECIMAL(10,2);
- SELECT SUM(unit_price * quantity) INTO total FROM order_item WHERE order_id=p_order_id GROUP BY order_id;
- RETURN total;
- END \$\$
- DELIMITER ;

- SELECT calculate_order(1);



Conclusion: quando usare le stored procedure?

Caso d'uso	Stored Procedure	Query Normale
Query complessa con più operazioni	<input checked="" type="checkbox"/> Sì, è più efficiente	<input type="checkbox"/> No
Query semplice su una tabella indicizzata	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Sì
Evitiamo il traffico tra Codice e MySQL	<input checked="" type="checkbox"/> Sì	<input type="checkbox"/> No
Necessità di sicurezza avanzata	<input checked="" type="checkbox"/> Sì	<input type="checkbox"/> No
Utilizzo della cache MySQL	<input type="checkbox"/> No	<input checked="" type="checkbox"/> 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-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 calcola_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

```
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;
```

Terminiamo la creazione del trigger con 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

TRIGGER ESEMPIO

- `INSERT INTO `crm_its`.`corsi` ('nome',
`data_inizio`, `data_fine`)`
- `VALUES ('ciao', NULL, '2025-09-30');`
- `DELIMITER $$`
- `CREATE TRIGGER trg_before_insert_corsi`
- `BEFORE INSERT ON corsi`
- `FOR EACH ROW`
- `BEGIN`
 - -- Se il campo data_inizio non è stato fornito, generiamo un errore
 - IF NEW.data_inizio IS NULL THEN
 - SIGNAL SQLSTATE '45000'
 - SET MYSQL_ERRNO = 1002, MESSAGE_TEXT = 'Il campo data_inizio è obbligatorio.';
 - END IF;
 -
 - -- Se inserito_il non è stato fornito, imposta il timestamp corrente
 - IF NEW.inserito_il IS NULL THEN
 - SET NEW.inserito_il = NOW();
 - END IF;
- `END$$`
- `DELIMITER ;`

27 21:20:27 INSERT INTO `crm_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;
```

ESEMPIO: trg_order_ai

- trg_ = prefisso del trigger
- CREATE TABLE `logs` (
 - `id` INT NOT NULL AUTO_INCREMENT,
 - `fk_id` INT NULL,
 - `table_name` VARCHAR(45) NULL,
 - `action` VARCHAR(45) NULL,
 - `created_at` TIMESTAMP NULL DEFAULT current_timestamp,
 - `message` text NULL,
- PRIMARY KEY (`id`);
- DELIMITER \$\$
- CREATE trigger trg_orders_ai
- AFTER INSERT ON orders
- FOR EACH ROW
- BEGIN
- INSERT INTO logs (fk_id, table_name, action, message)
- VALUES
- (NEW.id, 'orders', 'INSERT', CONCAT('inserimento di workshop_id: ', NEW.workshop_id));
- END \$\$
- DELIMITER ;

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 **FOLLOWERS** 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 (**FOLLOWERS**) o prima (**PRECEDES**).

```
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;
```

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

```
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;
```

ESEMPIO TRIGGER LOGS

- CREATE TABLE `LOGS` (
 - `ID` INT AUTO_INCREMENT PRIMARY KEY,
 - `timestamp` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 - `cliente_id` INT, -- ID del cliente inserito
 - `utente` VARCHAR(255) -- Utente MySQL che ha eseguito l'operazione
 -);
- CREATE DEFINER = CURRENT_USER TRIGGER `clienti_AFTER_INSERT` AFTER INSERT ON `clienti`
 - FOR EACH ROW
 - BEGIN
 - INSERT INTO `LOGS`(`cliente_id`, `utente`) VALUES (NEW.cliente_id, USER());
 - END

VIEW

- 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();

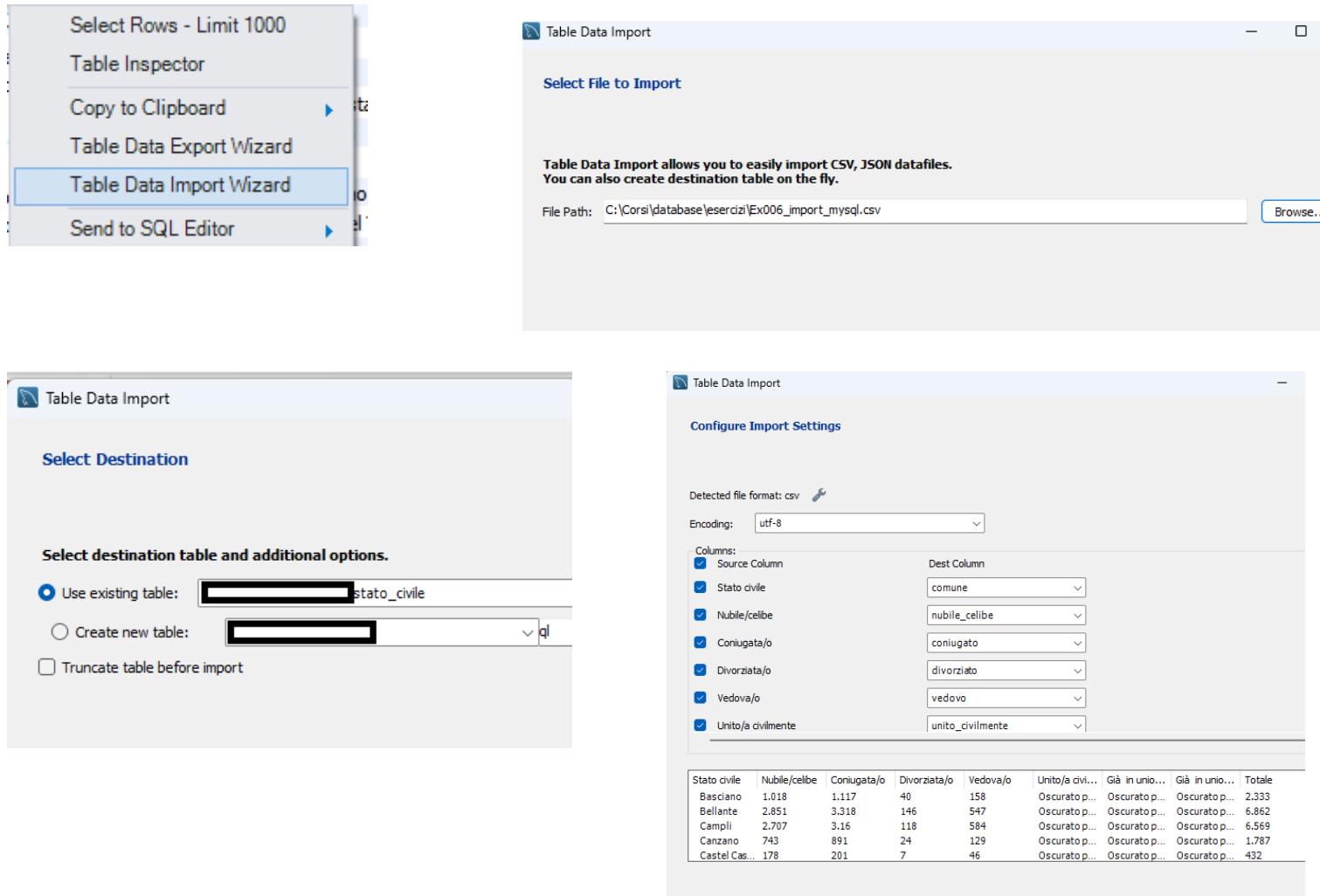
```
DROP VIEW workshop_total;
CREATE VIEW workshop_total as
SELECT workshop_id, SUM(quantity*unit_price)
as total from order_item JOIN orders on orders.id
= order_item.order_id group by workshop_id;

SELECT total FROM workshop_total WHERE
workshop_id=2;
```

VIEW E GRANT

- 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';**

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
);
```

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

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»

CORRISPONDENZA DI TIPI

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

POSTGRESQL E MYSQL

PostgreSQL consente l'utilizzo di estensioni per aggiungere funzionalità extra come supporto per dati geografici, o una criptografia 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.

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}$');
```

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 DI DATO (MySQL → PostgreSQL)

- **✗ Tipi NON esistenti in PostgreSQL**
- TINYINT
- MEDIUMINT
- YEAR
- SET
- **⚠ Tipi diversi o gestiti diversamente**
- ENUM
 - MySQL: inline nella colonna
 - PostgreSQL: **tipo da creare prima**
- BOOLEAN
 - MySQL: alias di TINYINT(1)
 - PostgreSQL: **boolean**
- JSON
 - MySQL: JSON
 - PostgreSQL: JSON e JSONB (più performante)
- **Equivalenze consigliate**
- TINYINT / MEDIUMINT → SMALLINT / INTEGER
- AUTO_INCREMENT → SERIAL / GENERATED AS IDENTITY

DDL (CREATE / ALTER / DROP)

- Sintassi MySQL non valida
- AUTO_INCREMENT
- ENGINE=InnoDB
- ALTER TABLE ... CHANGE column
- SET FOREIGN_KEY_CHECKS = 0
- Backtick `campo`
- PostgreSQL usa **doppi apici** "campo" (ed è case sensitive)
- TRUNCATE:
 - MySQL: ignora FK con settaggi
 - PostgreSQL: **richiede CASCADE**
- ENUM va creato **prima** del CREATE TABLE

DML (SELECT / INSERT / UPDATE / DELETE)

- **Funzioni MySQL inesistenti**
- IF(cond, a, b) -> CASE..WHEN
- DATE_FORMAT()
- TIMESTAMPDIFF()
- RAND()

- **Equivalenti PostgreSQL**
- DATE_FORMAT() → TO_CHAR()
- TIMESTAMPDIFF() → AGE() / EXTRACT
- RAND() → RANDOM()

- SELECT
- CASE
- WHEN id > 2 THEN 'magg'
- ELSE 'eq_o_min'
- END AS categoria_tipo, *
- FROM ricambi.categories
- ORDER BY id ASC;

- SELECT TO_CHAR(NOW(), 'DD/MM/YYYY');
- SELECT EXTRACT(YEAR FROM AGE(NOW(), data_nascita));

- SELECT RANDOM();

```
SET search_path TO ricambi, public;  
select * from workshops;
```

DML (SELECT / INSERT / UPDATE / DELETE)

- In PostgreSQL NON ESISTE **USE**, in mysql cambia il database utilizzato
- **SET search_path TO ricambi, public;**
- **search_path** è una lista ordinata di schemi.
- cerca l'oggetto nel **primo schema**
- se non lo trova, passa al **secondo**
- e così via
- **search_path = ricambi, public**
- Ordine di ricerca:
 - 1 ricambi
 - 2 public
-

```
SET search_path TO ricambi, public;  
select * from workshops;
```

SCHEMA di default: search_path

- Lo **search_path** è la **lista di schemi** in cui PostgreSQL cerca **tabelle, viste e funzioni** quando **NON specifichi lo schema** nel nome dell'oggetto.
- **SET search_path TO
nome_schema;**

FUNZIONI STRINGHE / DATE

- Nomi diversi
- LENGTH()
 - MySQL: byte
 - PostgreSQL: caratteri
- NOW() → ok in entrambi ma (PostgreSQL preferisce CURRENT_TIMESTAMP)
- In postgres:
 - SELECT LENGTH('è') AS lunghezza_caratteri;
 - In byte: SELECT OCTET_LENGTH('è') AS lunghezza_byte;
- MySQL: LENGTH() → byte
- PostgreSQL: LENGTH() → caratteri
- PostgreSQL: byte = OCTET_LENGTH()

FUNZIONI STRINGHE / DATE

- Nomi diversi
- LOCATE() → POSITION()
- NOW() → ok in entrambi (Postgres preferisce CURRENT_TIMESTAMP)
-  DATE
- MySQL è più permissivo su date “strane”
- PostgreSQL è molto più rigoroso (errori immediati)
- SELECT LOCATE('a', 'Mario');
- SELECT POSITION('a' IN 'Mario') AS posizione; #2
- SELECT NOW();
- SELECT DATE '2024-02-30';
- In mysql 2024-02-01
- In postgres: errore
- Uguale in mysql e postgres:
- INSERT INTO eventi (data_evento)
- VALUES ('2024-12-01');

POSTGRESQL OGGETTI DI BASE

- Oggetti:
- Tabelle = dati
- Constraint = regole
- Index = velocità
- Function = logica
- Trigger = automatismi
- View = lettura
- Schema = organizzazione

▼	❖ Schemas (2)
>	❖ public
▼	❖ ricambi
>	Aggregates
>	Collations
>	Domains
>	FTS Configurations
>	FTS Dictionaries
>	Aa FTS Parsers
>	FTS Templates
>	Foreign Tables
>	Functions
>	Materialized Views
>	Operators
>	Procedures
>	1.3 Sequences
>	Tables (9)
>	Trigger Functions (2)
>	Types
>	Views

- **Materialized Views**
- **👉 SELECT salvate + dati**
- Come una view
- Ma **memorizza i risultati**
- Va aggiornata manualmente

- **Funzioni SQL / PLpgSQL**
- Ritornano un valore
- Possono essere usate in SELECT, WHERE, ecc.
- **Trigger Functions**
- **👉 Funzioni SPECIALI per trigger**
- **NON si chiamano mai a mano**
- Sono eseguite **solo da un trigger**
- Devono fare RETURN NEW o RETURN OLD

- **Triggers**
- **👉 Intercettori di eventi**
- Scattano su:
 - INSERT
 - UPDATE
 - DELETE
- Collegano:
 - tabella → trigger function
- **Types**
- **👉 Tipi di dato personalizzati**
- ENUM
- Composite types

TRIGGER

- **1** CREATE TABLE ricambi.insert_log (
 - id SERIAL PRIMARY KEY,
 - table_name TEXT NOT NULL,
 - record_id INTEGER NOT NULL,
 - created_at TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
 -);
- **2** TRIGGER FUNCTION
 - CREATE OR REPLACE FUNCTION ricambi.log_insert()
 - RETURNS trigger
 - LANGUAGE plpgsql
 - AS \$\$
 - BEGIN
 - INSERT INTO ricambi.insert_log (table_name, record_id, created_at)
 - VALUES (TG_TABLE_NAME, NEW.id, CURRENT_TIMESTAMP);
 - RETURN NEW;
 - END;
 - \$\$;

- **3** CREATE TRIGGER **trg_orders_log_insert**
 - AFTER INSERT ON ricambi.orders
 - FOR EACH ROW
 - EXECUTE FUNCTION ricambi.log_insert();
- **4** INSERT INTO ricambi.orders (workshop_id, order_date, status)
 - VALUES (1, CURRENT_DATE, 'NEW');

FUNCTION e Procedure Language di PostgreSQL: PL/pgSQL

```
▪ set search_path to ricambi;  
▪ CREATE OR REPLACE FUNCTION ricambi.total_order(p_order_id  
INT)  
▪ RETURNS NUMERIC  
▪ LANGUAGE plpgsql  
▪ AS $$  
▪ BEGIN  
▪   RETURN (  
▪     SELECT SUM(quantity * unit_price)  
▪     FROM ricambi.order_item  
▪     WHERE order_id = p_order_id  
▪   );  
▪ END  
▪ $$;  
  
▪ select ricambi.total_order(1);
```

- PL/pgSQL (LANGUAGE plpgsql)
- È il **linguaggio procedurale** di PostgreSQL (simile a PL/SQL).
- **Caratteristiche**
 - ✓ variabili
 - ✓ IF / ELSE
 - ✓ LOOP
 - ✓ gestione errori
 - ✓ più istruzioni

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

POSTGRES 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

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.

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 INdexes)
Indice progettato per dati sequenziali, diverso da altri dato che salva i valori minimi e massimi per ogni blocco di memoria
Ideale per: Dati Temporali, Dati Ordinati

◆ 1. Performance e Scalabilità

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

👉 Conclusioni:

- 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.

◆ 2. Funzionalità Avanzate

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

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

Registrati - Server

Generale Connessione Parametri Tunnel SSH Avanzate Etichette

Nome	local_server
Gruppo di server	Servers
Secondo piano	X
Primo piano	X
Connettersi ora?	<input checked="" type="checkbox"/>
Commenti	

Chiudi **Ripristina** **Salva**

Registrati - Server

Generale **Connessione** Parametri Tunnel SSH Avanzate Etichette

Nome / indirizzo server di host	localhost
Porta	5432
Database di manutenzione	postgres
Nome dell'utente	postgres
Autenticazione Kerberos?	<input type="checkbox"/>
Password	
Salvare la password?	<input type="checkbox"/>
Ruolo	
Servizio	

Chiudi **Ripristina** **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

The screenshot shows the PGAdmin 4 interface. On the left, a context menu is open under the 'Crea' (Create) option, with 'Tabella...' selected. To the right, two windows are displayed: 'Creazione - Tabella' and 'Creazione - Tabella'. The top window shows the basic details for the table 'customers'. The bottom window shows the 'Colonne' (Columns) tab, where a new column can be added via a '+' button or an 'Add row' link.

Left Panel (Context Menu):

- Crea >
- Schema...
- Collation...
- Configurazione FTS ...
- Dizionario FTS ...
- Dominio...
- FTS Parser ...
- Funzione di Trigger...
- Funzione...
- Modello FTS...
- Procedura...
- Sequenza...
- Tabella esterna...
- Tabella...
- Tipo...
- Vista...
- Vista Materializzata...

Top Window (Creazione - Tabella):

Creazione - Tabella							
Generale	Colonne	Avanzate	Vincoli	Partizioni	Parametri	Sicurezza	SQL
Nome	customers						

Bottom Window (Creazione - Tabella):

Creazione - Tabella								
Generale	Colonne	Avanzate	Vincoli	Partizioni	Parametri	Sicurezza	SQL	
Ereditato da tabelle	Selezione per ereditare da ...							
Colonne								
	Nome	Tipo dato	Lunghezza/Precisi...	Scala	Non NULL?	Chiave Primari...	Predefinita...	
			Selezione un oggetto		<input type="checkbox"/>	<input type="checkbox"/>		

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), **NON ESISTE IL FORWARD ENGINEER**

Su pgAdmin 4:
Cliccando mouse destro
su un database:

