

INDICE argomenti

- [Cos'è un database](#)
- [Strutture Dati e Modelli di Database](#)
- [DBMS e RDBMS](#)
- [Il Modello RELAZIONALE](#)
- [Normalizzazione](#)
- [Tipi di dato](#)
- [Indici](#)
- [SQL - introduzione](#)
- [Installazione MySQL](#)
- [Accesso DBMS \(DBA\)](#)
- [DDL: 1 parte](#)
- [DCL](#)
- [Accesso DBMS \(user\)](#)
- [DDL: 2 parte](#)
- [DML](#)
- [Query](#)
- [Integrità referenziale](#)
- [UNION, INTERSECT e EXCEPT](#)
- [JOIN](#)
- [Funzioni](#)
- [Raggruppamenti](#)
- [Windows Function](#)
- [Viste](#)
- [Sub Query](#)
- [Backup/Restoring](#)
- [Indici - approfondimento](#)
- [Ottimizzazione Query](#)
- [Transaction](#)
- [Programmazione SQL - introduzione](#)
- [Triggers](#)
- [Stored Function](#)
- [Stored Procedures](#)
- [Events](#)

Strumenti

- MySQL Community Server 8.4
- Terminale/Shell: ambiente a riga di comando per interagire con il database tramite istruzioni SQL;
- MySQLWorkbench: ambiente grafico per interagire con il database;
- DBeaver: ambiente grafico per interagire con il database;
- Visual Studio Code con estensione MySQL Shell for VS Code;
- Documentazione ufficiale MySQL: <https://dev.mysql.com/doc/refman/8.4/en>

Cos'è un Database

Un database è una raccolta organizzata di dati archiviati in un formato facilmente accessibile.

- I database (o basi di dati) sono collezioni di dati tra loro correlati, utilizzate per rappresentare una porzione del mondo reale. Sono strutturati in modo tale da consentire la gestione efficiente dei dati, permettendo operazioni come inserimento, aggiornamento, ricerca e cancellazione delle informazioni.
- In informatica, il termine database indica una struttura organizzata di dati. I database (o più brevemente, DB) sono archivi dove le applicazioni memorizzano i dati in modo persistente¹ per poterli successivamente leggere, modificare o eliminare.

¹⁾ *In ambito informatico, la persistenza si riferisce alla capacità di un dato di essere conservato oltre la durata di esecuzione di un singolo programma, garantendo che i dati siano ancora disponibili in seguito.*

Dati e Informazione

- **Informazione:** dato elaborato con significato e contesto.

In senso generale, l'informazione è qualsiasi contenuto significativo che possiamo trasmettere, raccogliere o interpretare. Tuttavia, perché l'informazione diventi utile, deve essere organizzata e strutturata in una forma che ne consenta una facile elaborazione.

- **Dati:** rappresentazioni grezze di fatti, numeri o eventi.

Un dato è un'unità elementare di informazione, che può essere un numero, una parola, un'immagine, ecc.

Ad esempio, una stringa di caratteri come "Maria", oppure un numero come "30", sono dati grezzi. Da soli, però, non raccontano molto.

Per renderli significativi dobbiamo inserirli in un contesto strutturato, ad esempio: "Maria ha 30 anni."

Database file-server, client-server

database file-server

Sono semplici files, a cui possono facilmente accedere i programmi che li usano per inserire, visualizzare, modificare o cancellare i dati in essi contenuti.

- il sistema accede fisicamente al file;
- più il file è di grandi dimensioni maggiore il tempo di accesso;
- accesso contemporaneo da più utenti rallenta notevolmente il db;
 - *MS Access*,
 - *Filemaker*
 - ...;

database client-server

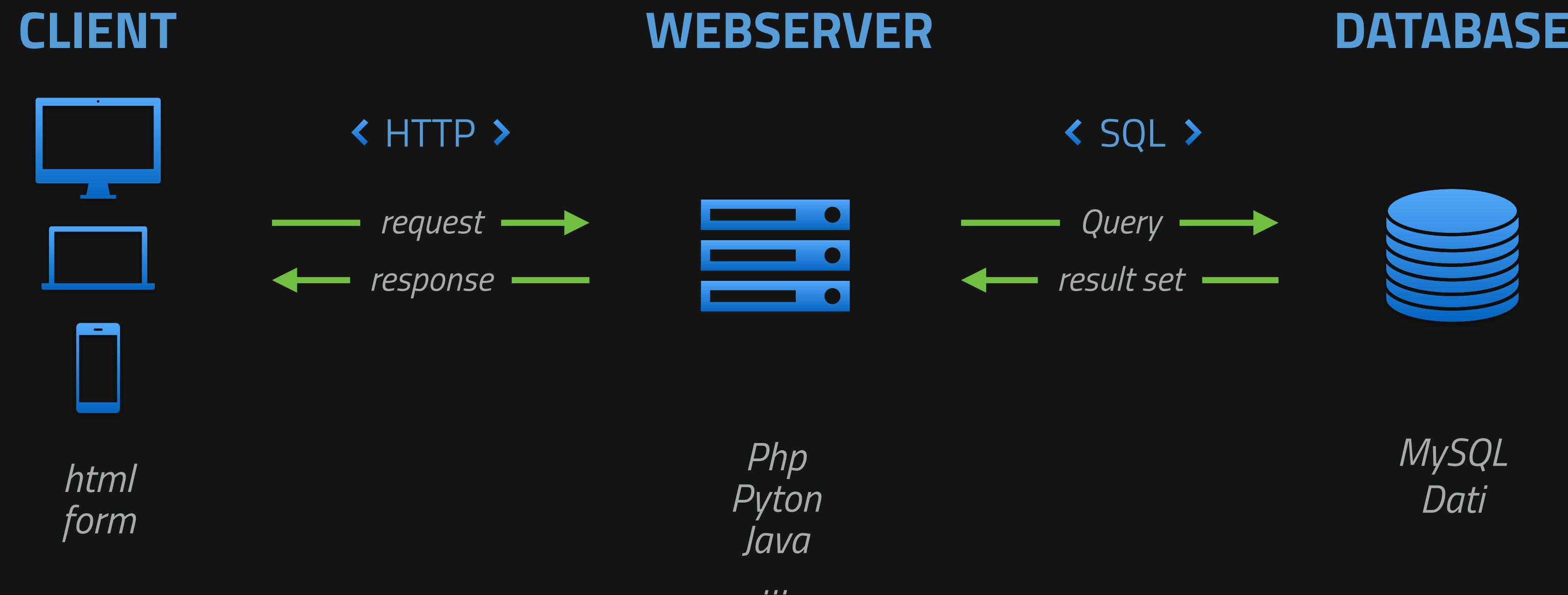
Rappresentano un servizio che mette a disposizione il software per interagire con i dati.

Viene gestito e manutenuto dai DBA (Database Administrator).

- *Microsoft SQL Server (RDBMS)*,
- *Oracle (RDBMS)*,
- *MySQL (RDBMS)*,
- *DB2 (RDBMS)*,
- *PostgreSQL (ORDBMS object-relational database system)*,
- *MongoDB (NoSQL Document Stores)*,
- *Neo4j (NoSQL Graph)*,
- ...;

client-server

Esempio di richiesta dati attraverso un form via https



VANTAGGI DATABASE CLIENT-SERVER

1. I clients **non accedono fisicamente al file** sul database, inviano solamente la loro query al motore del database ed il server restituisce solamente i dati richiesti.
2. **Velocità**: al crescere delle dimensioni del database il tempo di una query rimane identico, perché attraverso la LAN viaggiano e continueranno a viaggiare solamente la richiesta (query) ed i dati restituiti, la dimensione del database diventa alla fine irrilevante per il client.
3. Il motore del database è in grado di gestire tutte le **connessioni simultanee** da parte degli utenti, ed utilizzare al meglio le prestazioni dell'hardware.
4. **Sicurezza**. Se su un sistema file-server potrebbe succedere che in determinate situazioni il file arrivi ad essere corrotto (termine tecnico), questo non deve potere succedere, mai e per nessuna ragione, su un sistema client-server.
5. La sicurezza *viene garantita anche grazie alle funzioni che i db client-server normalmente offrono*. Tutte le tabelle di un sistema gestionale aziendale sono tra loro collegate, la mancanza della gestione delle relazioni può portare a grossi problemi circa l' **integrità dei dati**.

Strutture Dati e Modelli di Database

Le strutture dati rappresentano il modo in cui le informazioni vengono organizzate e memorizzate per facilitarne il recupero e la gestione.

Ogni struttura ha caratteristiche specifiche che la rendono più o meno adatta a determinati scenari applicativi.

Nei database, le strutture dati vengono implementate attraverso diversi modelli di organizzazione, ognuno pensato per esigenze specifiche.

Modelli di database più diffusi

Relazionale (RDBMS - Relational Database Management System)

- Organizza i dati in tabelle (righe e colonne).
- Relazioni tra le tabelle tramite chiavi primarie ed esterne.

In questo modello, i dati sono organizzati in tabelle (o relazioni), che possono essere pensate come fogli di calcolo.

Ogni tabella ha delle colonne (campi) e delle righe (record), dove le righe rappresentano entità specifiche e le colonne descrivono attributi delle entità.

Esempi: *MySQL, PostgreSQL, Oracle, SQL Server...*

Time Series Database

Le serie temporali sono una struttura in cui i dati vengono organizzati in base a un intervallo di tempo. Vengono utilizzate per tracciare cambiamenti e osservare tendenze nel tempo.

- Ottimizzato per la memorizzazione di dati temporali.
- Formato: Timestamp + Valore.

Esempi: *InfluxDB*, *TimescaleDB*, *OpenTSDB*.

Esempio: sensori di temperatura che registrano la temperatura ogni ora:

```
{  
  timestamp: "2025-02-21T12:00:00Z",  
  temperatura: 22.5  
}
```

Database Colonnare

- Memorizza i dati per colonne anziché per righe.
- Vantaggioso per analisi su grandi volumi di dati.

Esempi: *Apache Cassandra, Google Bigtable, Amazon Redshift*.

Esempio: Analisi di vendite con colonne separate per ogni mese:

Prodotto | Gennaio | Febbraio | Marzo

NoSQL (Not Only SQL)

- Pensato per scalabilità e flessibilità.
- Tipologie:
 - Document-oriented: Archivia dati in documenti JSON o BSON (es. *MongoDB, CouchDB*).
 - Key-Value Store: Struttura chiave-valore (es. *Redis, DynamoDB*).
 - Wide-Column Store: Variante del database colonna (es. *Apache HBase, ScyllaDB*).
 - Graph Database: Gestisce dati con nodi e connessioni (es. *Neo4j*).

Esempio: database documentale MongoDB che memorizza profili utente:

```
{  
  "nome": "Mario",  
  "cognome": "Rossi",  
  "email": "mario.rossi@example.com"  
}
```

DATABASE

uso dei database relazionali

DBMS

database management system

- È il software per la creazione e la manipolazione di un database.
È un **software di tipo server** (client-server) avente il compito di gestire uno o più database; questo vuol dire che il DBMS deve intervenire, **in qualità di intermediario**, in ogni operazione svolta sui database dai software che ne fanno utilizzo.
- Definisce gli utenti e gli amministratori di un database
- Fornisce **meccanismi di sicurezza, protezione** e controllo dell'**integrità dei dati**

Cosa fa un DBMS? (riepilogo)

Funzione	Descrizione
Gestione dati	Crea, legge, modifica, elimina
Sicurezza	Controllo accessi, permessi
Integrità	Regole sui dati, relazioni
Concorrenza	Accessi simultanei
Persistenza	Salvataggio stabile nel tempo
Architettura	Funziona come server in ambienti client-server

Il DBMS è l'intermediario tra i programmi e i dati, garantendo coerenza, sicurezza e prestazioni.

Componenti principali di un DBMS

Motore di archiviazione → gestisce come i dati sono salvati (es. InnoDB, MyISAM)

Motore delle query → interpreta ed esegue le istruzioni SQL

Gestore delle transazioni → garantisce che i dati siano consistenti

Controllo degli accessi → gestisce utenti, ruoli e permessi

Catalogo (metadati) → contiene le informazioni sulla struttura del database

RDBMS

relational database management system

- MySQL è un software appartenente alla famiglia dei DBMS. All'interno di questo gruppo di software è possibile identificare dei sotto-insiemi più specifici tra cui, ad esempio, quello dei **DBMS NoSQL** (MongoDB) e quello dei **RDBMS** a cui appartiene tra gli altri, appunto, MySQL.
- Gli RDBMS non sono altro che dei sistemi di gestione delle banche dati che operano in aderenza alla teoria relazionale secondo la quale il sistema deve operare sui dati mediante relazioni tra le diverse tabelle in cui questi vengono suddivisi e ordinati.
- Nel modello relazionale, infatti, i dati all'interno di un database *sono organizzati in differenti tavole le quali sono in relazione tra loro.*

Storage Engine

Gli **storage engine** rappresentano delle librerie che determinano il modo in cui i dati di una tabella saranno salvati su disco.

Ciò sarà determinante per valutare le prestazioni, l'affidabilità, le funzionalità offerte dalla tabella stessa, rendendola più o meno adatta a particolari utilizzi.

In pratica, scegliere un particolare storage engine significa scegliere il modo in cui i dati vengono gestiti.

MyISAM

Si tratta di un motore di memorizzazione veloce. Non supporta le transazioni. Non utilizza meccanismi di integrità referenziale.

- Adatto per le **ricerche full-text**;
- È **più veloce** poiché non è necessario tenere conto delle varie relazioni tra le tabelle;
- esegue il **lock sull'intera tabella**;
- ottimo se le tabelle vengono utilizzate principalmente in fase di lettura oppure se il **database è relativamente poco complesso**.

A partire dalla versione 5.5 di MySQL, *InnoDB è lo Storage Engine di default*. Prima era MyISAM.

InnoDB

Lo scopo di InnoDB è quello di associare maggiore sicurezza (intesa soprattutto come consistenza ed integrità dei dati) a performance elevate. Funzionalità peculiari:

- **Transizioni**: per transazione si intende la possibilità di un DBMS di svolgere più operazioni di modifica dei dati, facendo sì che i risultati diventino persistenti nel database solo in caso di successo di ogni singola operazione. In caso contrario, verranno annullate tutte le modifiche apportate;
- **Integrità referenziale**: conferiscono la possibilità di creare una relazione logica tra i dati di due tavelle, in modo da impedire modifiche all'una che renderebbero inconsistenti i dati dell'altra;
- esegue il **lock a livello di riga**;
- *Ricerche full-text* a partire da MySql 5.6.

Charset

I **character set** (insiemi di caratteri) sono i diversi sistemi attraverso i quali i caratteri alfanumerici, i segni di punteggiatura e tutti i simboli rappresentabili su un computer vengono memorizzati in un valore binario.

In ogni set di caratteri, ad un valore binario corrisponde un carattere ben preciso.

Con MySQL, a partire dalla versione 4.1, possiamo gestire i set di caratteri a livello di server, database, tabella e singola colonna, nonché di client e di connessione.

Ad ogni set di caratteri sono associate una o più **collation**, che rappresentano i modi possibili di confrontare le stringhe di caratteri facenti parte di quel character set.

In MySQL 8 il set di caratteri consigliato e predefinito è *utf8mb4* (UTF-8 vero (4 byte), oggi default).

La collation consigliata è *utf8mb4_0900_ai_ci* (accent-insensitive e case-insensitive)..

Supponiamo di avere un alfabeto di quattro lettere:

A, B, a, b.

Assegnamo ad ogni lettera un numero: A = 0, B = 1, a = 2, b = 3.

La lettera A è un *simbolo*, il numero 0 è la *codifica* per A, e la combinazione di tutte e quattro le lettere e la loro codifica è il **character set**.

Supponiamo che vogliamo confrontare due valori di stringa, A e B. Il modo più semplice per farlo è quello di guardare le codifiche: 0 per A e 1 per B.

Poiché 0 è minore di 1, diciamo A è inferiore a B.

Quello che abbiamo appena fatto è applicare un metodo di confronto per il nostro set di caratteri.

La **collation** è un insieme di regole (una sola regola in questo caso): "Confronta le codifiche".

IL MODELLO RELAZIONALE

- La **tabella** è la struttura dati fondamentale di un database relazionale;
- Con le tabelle si rappresentano le **entità** e le **relazioni** del modello concettuale*;
- La tabella è composta da campi (colonne o **attributi**) e da record (righe o **tuple**);
- Ogni *campo* rappresenta un **attributo** dell'entità/ relazione;
- Per ogni campo viene individuato un suo **dominio** (*tipo di dati*): alfanumerico, numerico, data...
- Ogni *record* rappresenta una *istanza* (o occorrenza o *tupla*) dell'entità/relazione;
- Garantisce l'**integrità referenziale**;

***Modello concettuale**: trasformazione di **specifiche** in linguaggio naturale (che definiscono la realtà descritta dal DB) in uno schema grafico chiamato **Diagramma E-R** che utilizza due concetti fondamentali: *Entità* e *Associazione/Relazione*.

Progettazione

Modello Concettuale

Definizione: rappresenta i dati e le loro relazioni a un livello astratto, senza preoccuparsi di dettagli tecnici o implementativi.

Obiettivo: descrivere la struttura dei dati in modo comprensibile per utenti e analisti, senza vincoli tecnologici.

Strumento tipico: diagrammi Entity-Relationship (ER) o UML class diagrams (Unified modeling language).

Indipendenza dal DBMS: Questo modello non dipende dal database relazionale, NoSQL o altro.

Modello Logico

Definizione: traduce il modello concettuale in una struttura più dettagliata e aderente alle regole di un particolare tipo di database (es. relazionale).

Obiettivo: definire tavelli, attributi, chiavi primarie e chiavi esterne, mantenendo ancora un livello di astrazione dalla tecnologia specifica.

Strumento tipico: Schema relazionale (tabelle, colonne, vincoli).

Dipendenza dal tipo di database: nel caso del modello relazionale, si definiscono relazioni tra tavelli con chiavi primarie e chiavi esterne.

Modello Fisico

Definizione: rappresenta la struttura effettiva del database come verrà implementata su un determinato DBMS.

Obiettivo: ottimizzare il database per le prestazioni, definendo dettagli come tipi di dati specifici, indici, partizionamento, ecc.

Strumento tipico: DDL (Data Definition Language) SQL per creare tabelle, vincoli, indici.

Dipendenza dal DBMS: strettamente legato a un DBMS specifico (es. MySQL, PostgreSQL, Oracle).

Fasi della progettazione

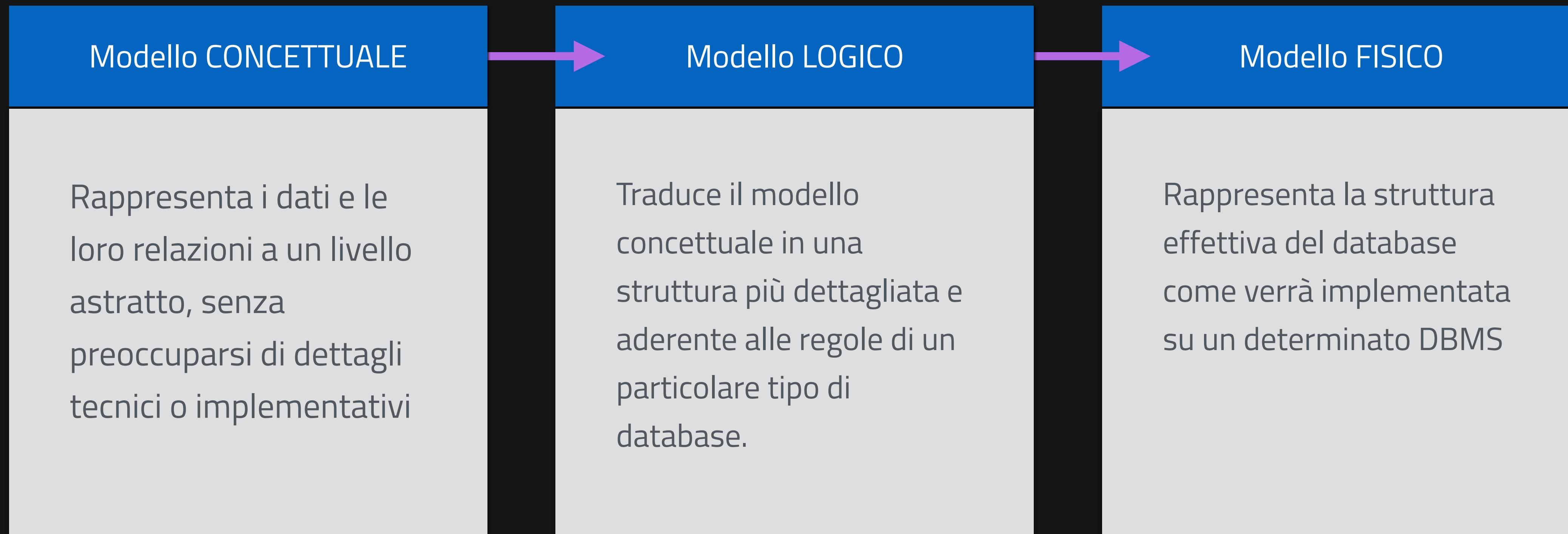


Diagramma E-R, simboli

Entità

Concetto fondamentale, generale, per la realtà che si sta modellando.

Rappresenta *classi di oggetti* (fatti, cose, persone, ...) che hanno *proprietà comuni* ed *esistenza autonoma* ai fini dell'applicazione di interesse.

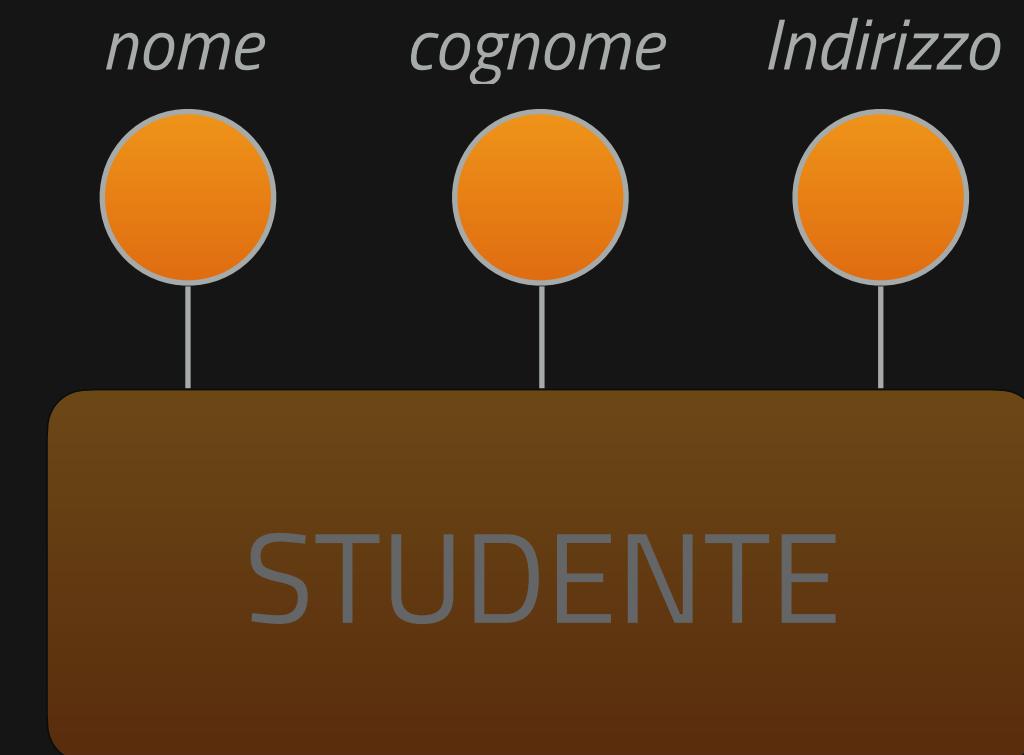
- Identificata da un rettangolo

STUDENTE

Attributi:

Caratteristiche specifiche di un'entità, utili (o necessarie) nella realtà da modellare

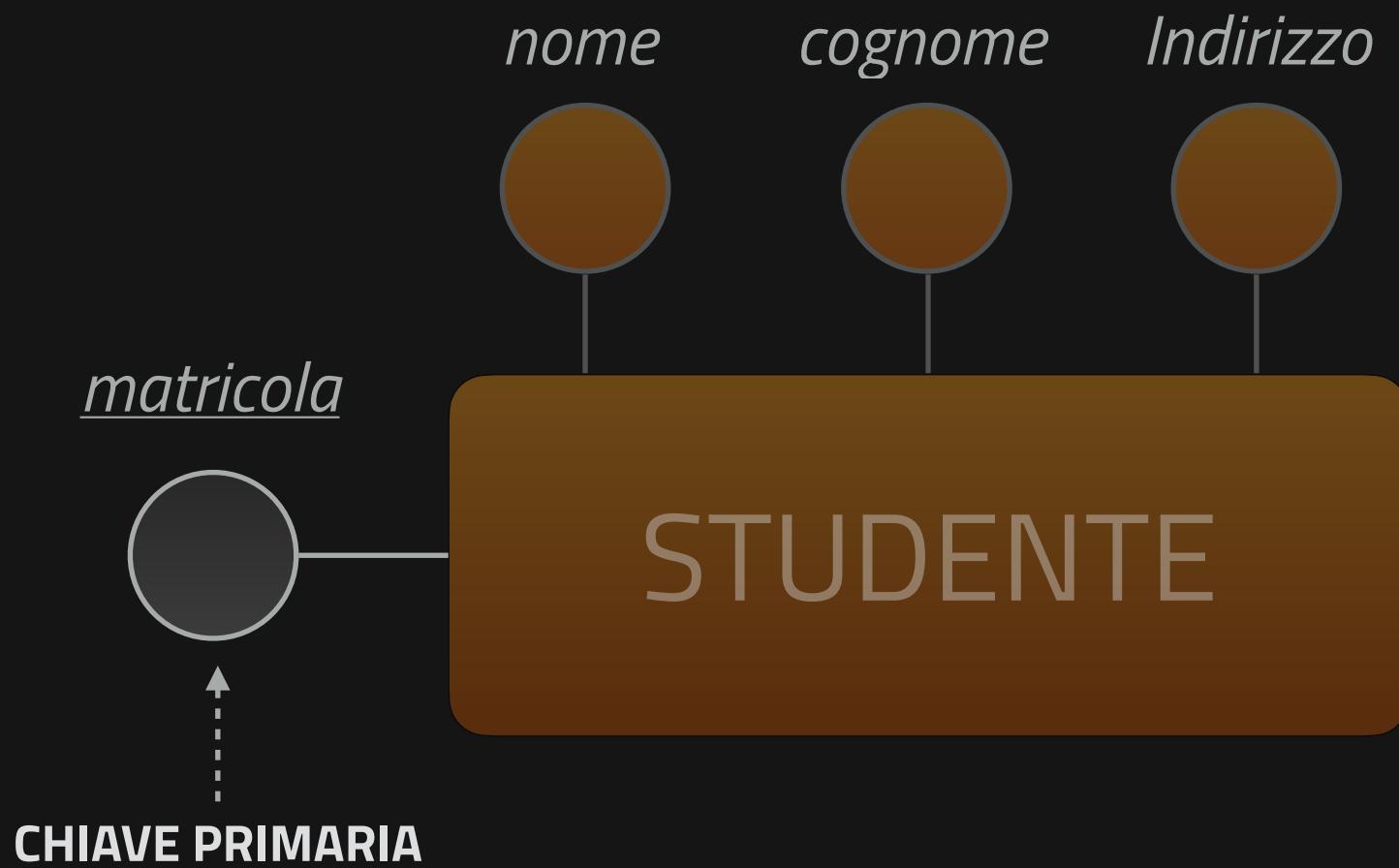
- identificata da un cerchio collegato all'entità



L'insieme di attributi che garantisce **l'univocità** delle istanze di un'entità è detta: **Chiave Primaria**.

È indicata come: **PRIMARY KEY** o **PK**

Identificata graficamente con un *cerchio pieno* collegato all'entità e relativo *nome attributo sottolineato*



Caratteristiche

- L'insieme dei campi i cui valori identificano univocamente un record all'interno di una tabella è detto Chiave Primaria. Quando la *chiave primaria* è *composta da un solo campo*, si parla di *campo chiave*.
- Quando non è possibile trovare un campo chiave tra gli attributi di una entità, si definisce un campo univoco di tipo numerico che si auto-incrementa (contatore): ID (identifier).

Esempi di campo chiave: *matricola*, *codice fiscale*, etc.

Istanze di un'Entità

Specifici dati, oggetti appartenenti ad un'entità

- non sono rappresentate nel Diagramma E-R
ma si intendono contenute in ogni entità:
- Carlo Rossi, via Verdi* è un'*istanza*
dell'**entità ALUNNO** (**attributi**: *Nome, Cognome, Indirizzo*)



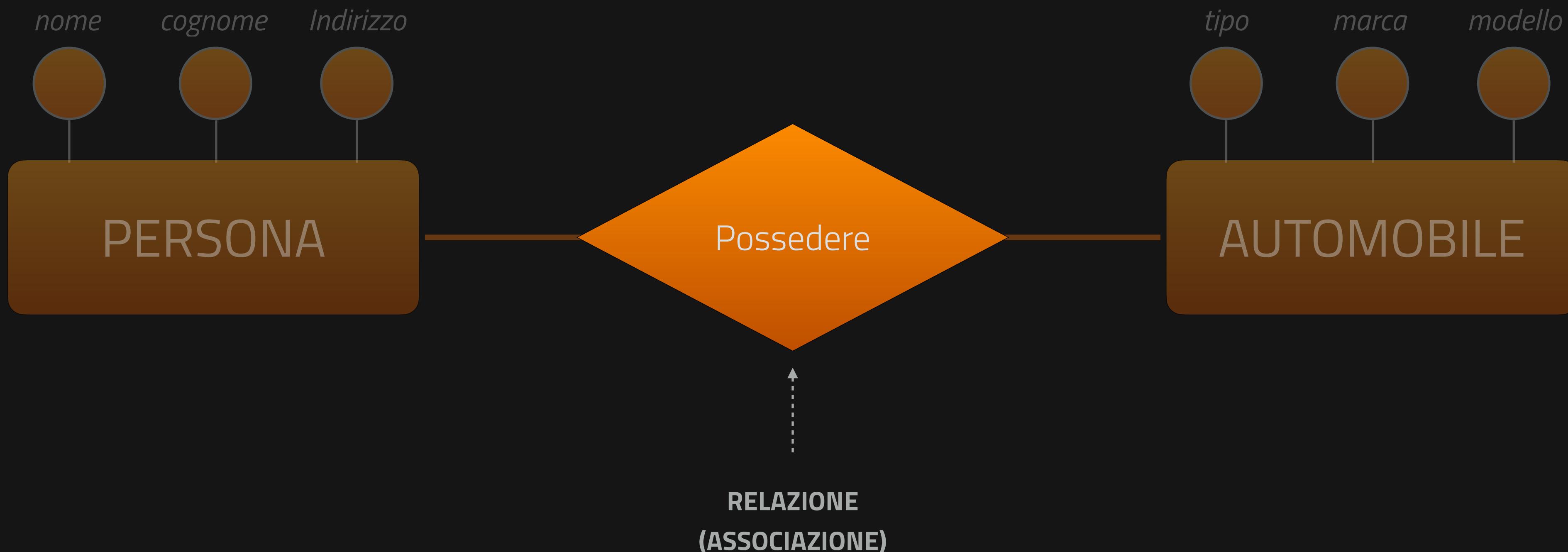
Possiamo considerare le entità come insiemi all'interno dei quali sono contenuti *oggetti* (le istanze) ciascuno con specifiche *caratteristiche* (valore degli attributi).



Relazioni (Associazioni)

Collegamenti logici che uniscono due o più entità nella realtà descritta dal database

- identificata da un rombo collegato alle due entità:



Cinema/teatro

Relazione 1,1 (uno a uno)



- Uno spettatore *occupa* un singolo posto
- Ogni singolo posto può *essere occupato* solo da uno spettatore

Liceo/Scuola superiore

Relazione 1,N (uno a molti)



- Ad ogni classe *appartiene* più di un alunno
- Un alunno *appartiene* ad una singola classe

Università

Relazione N,N (molti a molti)



- Uno studente *frequenta* più corsi
- Ogni corso è *frequentato* da molti studenti

Cardinalità delle relazioni

La relazione R che lega due entità **E1** ed **E2** può essere classificata in base alla sua **cardinalità** (quante istanze delle due entità sono coinvolte nella relazione):

- **1, 1 (uno a uno)** se *ad un elemento di E1 può corrispondere un solo elemento di E2*
- **1,N (uno a molti)** se *ad un elemento di E1 possono corrispondere più di un elemento di E2 , ad un elemento di E2 può corrispondere un solo elemento di E1*
- **N,N (molti a molti)** se *ad ogni elemento di E1 possono corrispondere molti elementi di E2 e viceversa*

Esempi



Esempio

Database per una applicazione web che gestisce l'acquisto/iscrizione dei/ai corsi.

Disegnare una base dati per la gestione dell'acquisto di corsi offerti da una piattaforma web.

Gli utenti/studenti devono essere registrati sulla piattaforma, per registrarsi occorre nome, cognome ed email. Viene memorizzata anche la data di registrazione.

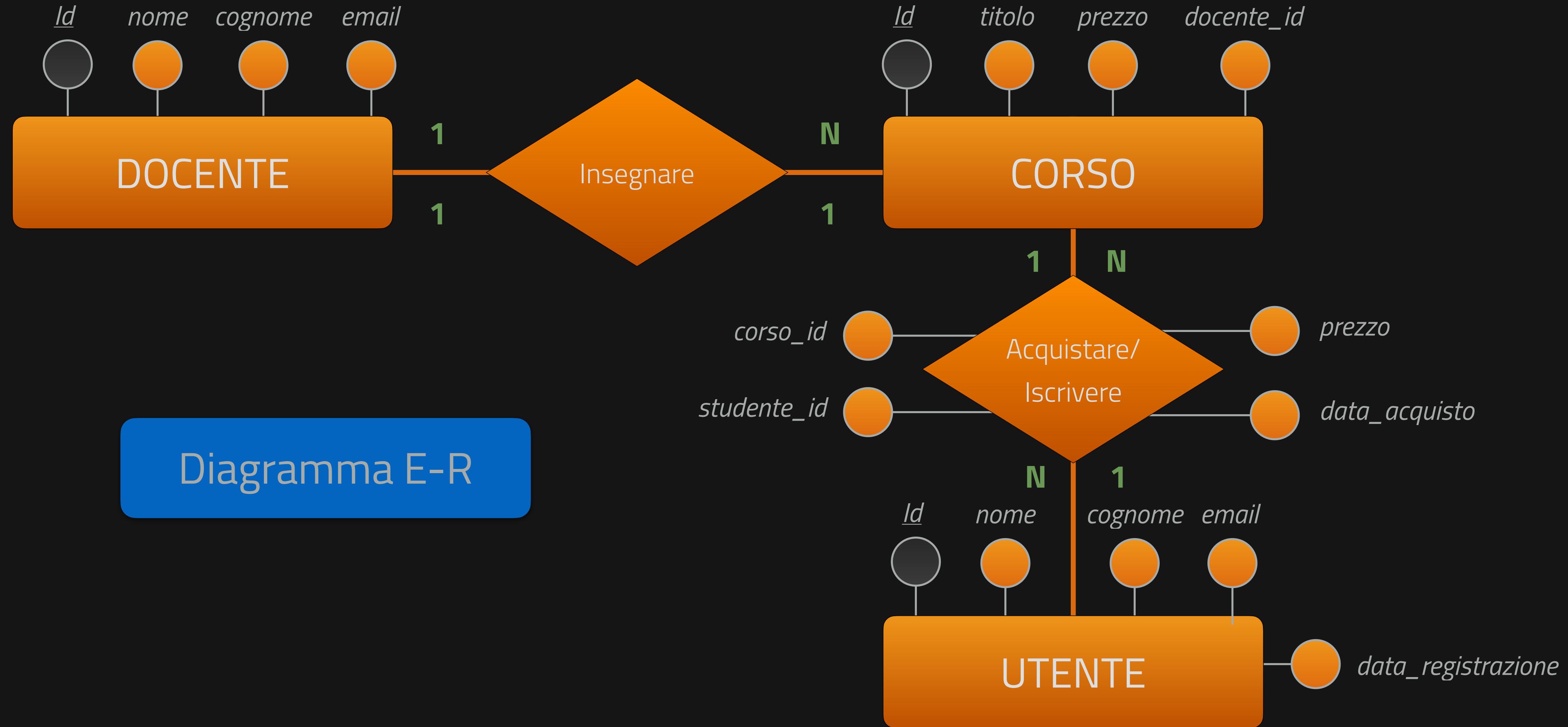
Gli studenti/utenti possono acquistare molti corsi. I corsi sono quindi a pagamento. Viene memorizzata anche la data di acquisto.

Il prezzo del corso può variare nel tempo.

I corsi si riferiscono a svariate materie quali: Java, Base di programmazione, Html, CSS, CMS, Javascript, React, PHP...

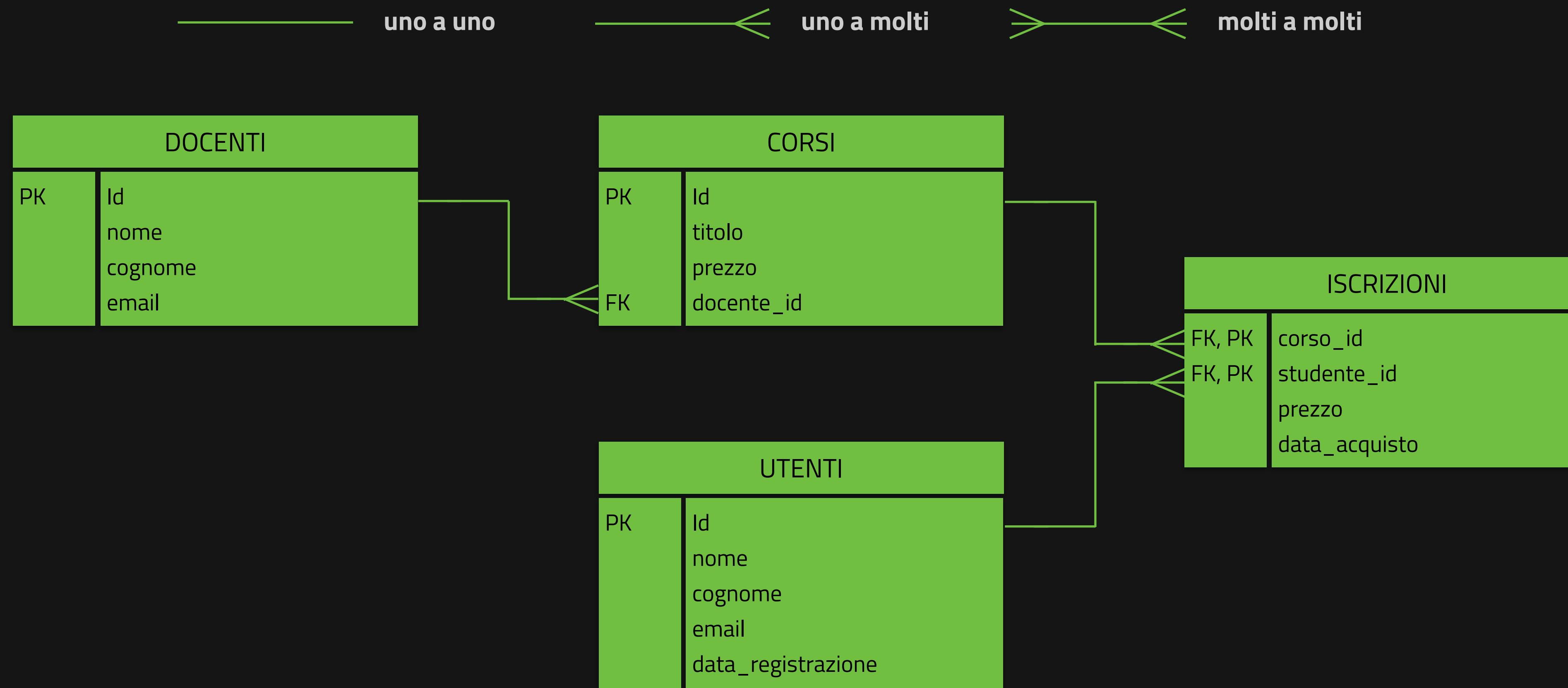
Ogni corso viene tenuto da un docente. Un docente può insegnare in molti corsi.

Modello Concettuale



Modello logico

Simbologia delle relazioni



Rappresentazione dell'informazione in tabelle nel database secondo il modello relazionale dell'esempio relativo alla piattaforma corsi:

UTENTI

• id	cognome	nome	email
1	Esposito	Giuseppe	esposito.g@icloud.com
2	Pautasso	Carlo	cpautasso@gmail.com
3	Piras	Efisio	efpiras@gmail.com
4	Conti	Renzo	conti.renzo@icloud.com
5	Rigon	Paolo	prigon@gmail.com

ISCRIZIONI

corso_id	studente_id	prezzo	data
1	1	180.00	22/9/2022
1	2	180.00	23/9/2022
3	1	300.00	25/9/2022
2	3	120.00	28/9/2022

DOCENTI

• id	cognome	nome	email
1	Rossi	Mario	rossi.mario@icloud.com
2	Verdi	Paola	verdi.paola@gmail.com
3	Bruni	Marco	bruni.marco@gmail.com
4	Bianchi	Daniele	bianchi.daniele@icloud.com

CORSI

• id	titolo	prezzo	docente_id
1	Php	200.00	4
2	Javascript	120.00	2
3	Java	300.00	4

Legame logico tra tabelle

CORSI				<i>Chiave Esterna</i>	<i>Campo Chiave</i>	DOCENTI			
•	titolo	prezzo	docente_i	• id	cognome	nome	email		
1	Php	200.00	4	1	Rossi	Mario	rossi.mario@icloud.com		
2	Javascript	120.00	2	2	Verdi	Paola	verdi.paola@gmail.com		
3	Java	300.00	4	3	Bruni	Marco	bruni.marco@gmail.com		
				4	Bianchi	Daniele	bianchi.daniele@icloud.com		

Join tra tabelle

• id	titolo	prezzo	docente_id	• id	cognome	nome	email
1	Php	200.00	4	4	Bianchi	Daniele	bianchi.daniele@icloud.com
2	Javascript	120.00	2	2	Verdi	Paola	verdi.paola@gmail.com
3	Java	300.00	4	4	Bianchi	Daniele	bianchi.daniele@icloud.com

Normalizzazione del database

Regole da rispettare nel definire le tabelle

La prima forma normale (1NF)

Si dice che una database è in 1NF (prima forma normale) se per ogni tabella/relazione contenuta nella base dati:

- *non presenta gruppi di attributi che si ripetono* (ossia ciascun attributo è definito su un dominio con valori atomici)
- tutti i valori di un attributo sono dello *stesso tipo* (appartengono allo stesso dominio)
- *esiste una chiave primaria* (ossia esiste un attributo o un insieme di attributi che identificano in modo univoco ogni tupla della relazione)
- *l'ordine delle righe è irrilevante* (non è portatore di informazioni)

Facciamo un esempio di una tabella che, seppur munita di una chiave primaria, non può essere considerata in forma normale:

● Codice Fiscale	Nome	Dettagli
LBRRSS79Y12T344A	Alberto	età: 30; professione: Impiegato
GNNBNCT84A11L61B	Gianni	età: 24; professione: Studente

La tabella qui sopra NON è in 1NF in quanto, pur avendo una chiave primaria, presenta un attributo (dettagli) che non contiene dati in forma atomica, ma un gruppo di attributi.

Non è solo mancanza di atomicità, ma anche violazione del principio “un attributo = un significato”.

● Codice Fiscale	Nome	Dettagli
LBRRSS79Y12T344A	Alberto	età: 30; professione: Impiegato
GNNBNCT84A11L611B	Gianni	età: 24; professione: Studente

Modifichiamo la tabella aggiungendo gli opportuni attributi:

● Codice Fiscale	Nome	eta	professione
LBRRSS79Y12T344A	Alberto	30	Impiegato
GNNBNCT84A11L611B	Gianni	24	Studente

Altro esempio

Id	Nome	Corsi
1	Maria	Matematica, Fisica
2	Giovanni	Biologia

La tabella di fianco NON è in 1NF in quanto, ogni colonna deve assumere un solo valore, ovvero non può essere una matrice o un'array di valori.

Id	Corso	Nome
1	Fisica	Maria
1	Matematica	Maria
2	Biologia	Giovanni

In questo caso la normalizzazione consiste nel riportare le celle che originariamente erano raggruppate in una unica colonna in più righe replicando gli altri valori

La seconda forma normale (2NF)

Perché una base dati possa essere in 2NF è necessario che:

- si trovi già in 1NF;
- tutti gli *attributi non chiave dipendano dall'intera chiave primaria* (e non solo da una parte di essa).

La 2NF elimina le dipendenze parziali e quindi riduce la ridondanza.

NOTA: La 2NF si applica solo se la PK è composta da più attributi.

Se la PK è semplice, 2NF = 1NF automaticamente.

Ora la tabella precedente che rispetta la 1FN non rispetta però la seconda forma normale.

Id	Corso	Nome
1	Fisica	Maria
1	Matematica	Maria
2	Biologia	Giovanni

La chiave primaria è la combinazione (Id, Corso).

L'attributo "Nome" dipende solo da Id, non da tutto l'insieme (Id, Corso).

Quindi c'è una dipendenza parziale \Rightarrow violazione della 2NF.

Soluzione: scomposizione in due tabelle

Id_insegnante	Nome
1	Maria
2	Giovanni

Id_corso	Corso	Id_insegnante
1	Fisica	1
2	Matematica	1
3	Biologia	2

Altro esempio: si supponga di avere a che fare con il database di una scuola con una chiave primaria composta dai campi "Codice Matricola" e "Codice Esame":

• Codice Matricola	• Codice Esame	Nome Matricola	Voto Esame
1234	M01	Rossi Alberto	6
1234	L02	Rossi Alberto	7
1235	L02	Verdi Mario	8

Il database qui sopra si trova in 1NF ma non in 2NF

Perché il campo "Nome Matricola" non dipende dall'intera chiave ma solo da una parte di essa ("Codice Matricola").

• Codice Matricola	• Codice Esame	Nome Matricola	Voto Esame
1234	M01	Rossi Alberto	6
1234	L02	Rossi Alberto	7
1235	L02	Verdi Mario	8

Per rendere il nostro database in 2NF dovremo scomporlo in due tabelle:

• Codice Matricola	• Codice Esame	Voto Esame	• Codice Matricola	Nome Matricola
1234	M01	6	1234	Rossi Alberto
1234	L02	7	1235	Verdi Mario
1235	L02	8		

La terza forma normale (3NF)

Un database è in 3NF se:

- è già in 2NF (e quindi, necessariamente, anche 1NF);
- tutti gli attributi non chiave dipendono direttamente dalla chiave,
quindi *non ci sono attributi "non chiave" che dipendono da altri attributi "non chiave"*.

In sintesi: un attributo non chiave non deve dipendere da un altro attributo non chiave.

Supponiamo di avere una base dati di una palestra in cui il codice fiscale dell'iscritto al corso frequentato è associato all'insegnante di riferimento.

Si supponga che il nostro DB abbia un'unica chiave primaria ("Codice Fiscale") e sia così strutturato:

• Codice Fiscale	Codice Corso	Insegnante
LBRRSS79Y12T344A	BB01	Marco
GNNBNCT84A11L61B	BB01	Marco
LBRMNN79E64A112A	BB01	Marco
GLSTM59U66P109B	AE02	Federica

Il nostro database non è in 3NF in quanto il campo "insegnante" non dipende dalla chiave primaria ma dal campo "Codice Corso" (che non è chiave).

• Codice Fiscale	Codice Corso	Insegnante
LBRRSS79Y12T344A	BB01	Marco
GNNBNCT84A11L611B	BB01	Marco
LBRMNN79E64A112A	BB01	Marco
GLSTMT59U66P109B	AE02	Federica

Per normalizzare il nostro DB in 3NF dovremo scomporlo in due tabelle:

• Codice Fiscale	Codice Corso	• Codice Corso	Insegnante
LBRRSS79Y12T344A	BB01	BB01	Marco
GNNBNCT84A11L611B	BB01	AE02	Federica
LBRMNN79E64A112A	BB01		
GLSTMT59U66P109B	AE02		

Tipi di dato

In una tabella MySQL per ciascuna colonna possiamo definire diversi tipi di dato (dominio):

- **Numerics** (numeri interi e a virgola mobile)
- **String** (stringa)
- **Date , Time** (data e ora)
- **JSON**

Dati numerici: interi

Tipo	Intervallo di valori	Solo se positivi (UNSIGNED)
BIT(M)	Da 1 a 64	/
TINYINT (1 byte)	da -128 a +127	da 0 a 255
SMALLINT (2 byte)	da -32 768 a +32 767	da 0 a 65 535
MEDIUMINT (3 byte)	da -8 388 608 a +8 388 607	da 0 a 16 777 215
INT (4 byte)	da -2 147 483 648 a +2 147 483 647	da 0 a 4 294 967 295
BIGINT (8 byte)	da -9 223 372 036 854 775 808 a +9 223 372 036 854 775 807	da 0 a 18 446 744 073 709 550 615

E' importante precisare che se all'interno di un campo di tipo numerico si cerca di inserire un valore maggiore di quanto ammesso dal tipo prescelto, MySQL produrrà un errore.

Tipi di numeri in virgola mobile (valore approssimativo)

Tipo	Tipo (sintassi)*	Tipo (sintassi deprecata)**	spazio
FLOAT	FLOAT[(p)], con p compreso tra 0 e 23	FLOAT, FLOAT(M,D)*	4byte
DOUBLE	FLOAT[(p)], con p compreso tra 24 e 53	DOUBLE, DOUBLE(M,D)*	8byte
DECIMAL	DECIMAL(M,D)		Dipende da M*

Nel caso di FLOAT, puoi specificare opzionalmente la precisione, usando una sintassi tipo FLOAT(p), dove "p" è la precisione in bit.

Se la precisione è tra 0 e 23, la colonna sarà considerata di tipo FLOAT e userà 4 byte.

Se la precisione è tra 24 e 53, sarà considerata di tipo DOUBLE e userà 8 byte.

MySQL permette anche una sintassi non standard, del tipo FLOAT(M,D) (o DOUBLE(M,D)).

M è il numero totale di cifre che puoi memorizzare (prima e dopo il punto decimale).

D è il numero di cifre dopo il punto decimale.

Per esempio, se dichiari una colonna come FLOAT(7,4), puoi memorizzare fino a 7 cifre in totale, con 4 cifre dopo il punto decimale.

Quindi i numeri vanno da -999.9999 a 999.9999.

* La sintassi FLOAT(p) è supportata ma ignorata: non influisce realmente sul tipo

** <https://dev.mysql.com/doc/refman/8.4/en/precision-math-decimal-characteristics.html>

Dati stringa

Tipo e lunghezza massima consentita

nome **VARCHAR(20)**

codiceFiscale **CHAR(16)**

titolo **TINYTEXT**

messaggio **TEXT**

Tipo	Lunghezza massima
CHAR(n)	255 caratteri
VARCHAR(n)	65.535 byte*
BINARY(b)	255 byte
VARBINARY(b)	65.535 byte
TINYTEXT	255 caratteri
TINYBLOB	255 byte
TEXT	65.535 caratteri
BLOB	65.535 byte
MEDIUMTEXT	16.777.215 caratteri
MEDIUMBLOB	16.777.215 byte
LONGTEXT	4.294.967.295 caratteri
LONGBLOB	4.294.967.295 byte
ENUM('value1','value2',...)	ENUM usa 1 o 2 byte a seconda del numero di valori
SET('value1','value2',...)	64 valori distinti

*Il limite è 65.535 byte perché dipende dal tipo di codifica adottata. Con utf8mb4 il massimo è ~16383 caratteri

I tipi **CHAR** e **VARCHAR** sono sicuramente i tipi più utilizzati.

La differenza tra questi due tipi è data dal fatto che CHAR ha *lunghezza fissa*, VARCHAR ha *lunghezza variabile*.

Questo significa che in una colonna CHAR(10) tutti i valori memorizzati occuperanno lo spazio massimo anche se costituiti da 3 soli caratteri.

I tipi **TEXT** e **BLOB** (Binary Large OBject) consentono di memorizzare grandi quantità di dati:

- TEXT è utilizzato per dati di tipo testuale,
- BLOB è utilizzato per ospitare dati binary (ad esempio il sorgente di un'immagine)

BINARY e VARBINARY

I tipi **BINARY** e **VARBINARY** sono simili a CHAR e VARCHAR, tranne per il fatto che *memorizzano stringhe binarie* anziché stringhe non binarie: memorizzano stringhe di byte anziché stringhe di caratteri.

Per questi campi il *set di caratteri* e *la collation*, il *confronto* e *l'ordinamento* si basano sui valori numerici dei byte memorizzati.

TEXT vs VARCHAR()

TEXT

- dimensione massima fissa di 65535 caratteri (non è possibile limitare la dimensione massima)
- prende $2 + c$ byte di spazio su disco, dove c è la lunghezza della stringa memorizzata.
- indice: può essere indicizzato solo con un indice: prefix index.

VARCHAR (M)

- dimensione massima variabile di byte M
- M deve essere compreso tra 1 e 65535
- prende $1 + c$ byte (per $M \leq 255$) o $2 + c$ (per $256 \leq M \leq 65535$) byte di spazio su disco dove c è la lunghezza della stringa memorizzata
- può essere parte di un indice

Se è necessario memorizzare stringhe più lunghe di circa 64 KB, utilizzare MEDIUMTEXT o LONGTEXT.

VARCHAR non supporta la memorizzazione di valori così grandi.

Tipi ENUM e SET

I tipi ENUM e SET sono un tipo di dato di testo in cui le colonne possono avere solo dei valori predefiniti.

ENUM: Tipo di dato ENUMerazione.

Contiene un insieme di valori prefissati tra cui scegliere: **si può inserire solamente uno dei valori previsti.**

I valori sono inseriti tra parentesi(elenco separato da virgola) dopo la dichiarazione ENUM.

```
genere ENUM('F', 'M', 'NB')
```

La colonna genere accetterà solamente i valori F , M o NB. Se proviamo a mettere un valore diverso con il comando INSERT, MYSQL restituirà errore.

SET: è una estensione di ENUM.

```
interessi SET('a', 'b', 'c', 'd')
```

Come per ENUM i valori sono fissi e disposti dopo la dichiarazione SET; tuttavia, le colonne SET possono assumere più di un valore tra quelli previsti.

DateTime

Tali tipi di dati sono molto utili quando si ha a che fare con informazioni riguardanti la data e l'orario.

Di seguito una tabella riepilogativa

Tipo	Formato	Intervallo
DATETIME	YYYY-MM-DD HH:MM:SS	'1000-01-01 00:00:00' a '9999-12-31 23:59:59'
DATE	YYYY-MM-DD	1000-01-01' a '9999-12-31'
TIME	HH:MM:SS	-838:59:59' a '838:59:59
YEAR	YYYY	un anno compreso fra 1901 e 2155, oppure 0000.
TIMESTAMP	YYYY-MM-DD HH:MM:SS	'1970-01-01 00:00:01'UTC' a '2038-01-19 03:14:07' UTC'

I campi di tipo DATETIME contengono sia la data che l'orario. I valori all'interno di questi campi possono essere inseriti sia sotto forma di stringhe che di numeri.

Sia *DATETIME* sia *TIMESTAMP* possono memorizzare in automatico la data.

Per ottenere ciò in fase di definizione del campo bisogna impostare il valore di *default di memorizzazione* (es):

```
ins TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
```

```
CURRENT_TIMESTAMP
```

```
data DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
```

```
CURRENT_TIMESTAMP
```

DATETIME o TIMESTAMP?

TIMESTAMP

Il tipo TIMESTAMP memorizza data e ora utilizzando il fuso orario del server per le conversioni automatiche da e verso UTC.

Quando memorizzi un valore in una colonna TIMESTAMP, MySQL lo converte automaticamente nel fuso orario UTC (Coordinated Universal Time) e lo memorizza in quel formato. Quando lo leggi, il valore viene riconvertito nel fuso orario del server o in quello del client (se configurato).

Questo rende il TIMESTAMP utile quando devi mostrare gli stessi dati di data e ora a utenti in fusi orari diversi, poiché MySQL gestisce automaticamente la conversione da e verso UTC.

TIMESTAMP copre solo il range di date da 1970-01-01 00:00:01 UTC fino a 2038-01-19 03:14:07 UTC.

Se hai bisogno di memorizzare date al di fuori di questo intervallo, non è adatto.

DATETIME o TIMESTAMP?

DATETIME

Il tipo DATETIME memorizza la data e l'ora così come sono, senza conversioni di fuso orario. Il valore viene memorizzato esattamente come viene inserito, senza alcuna considerazione del fuso orario del server o del client.

Questo significa che DATETIME è utile quando vuoi che la data e l'ora restino fisse, indipendentemente dal fuso orario dell'utente o del server.

È una buona scelta se i dati di data e ora devono essere gli stessi per tutti, ovunque si trovino.

DATETIME ha un range molto più ampio, da 1000-01-01 00:00:00 a 9999-12-31 23:59:59, quindi può essere usato per date molto più lontane nel passato o nel futuro rispetto a TIMESTAMP.

JSON

MySQL supporta JSON nativo come tipo di dati per gli oggetti nella notazione JSON.

Rende facile l'archiviazione, l'interrogazione e il recupero di documenti di tipo JSON piuttosto che archiviarli come stringhe di testo o BLOB binari (vedi MariaDB).

Per fare ciò mette a disposizione una serie di funzioni*

Sintassi per la definizione di un attributo di tipo JSON

columnName **JSON**

* <https://dev.mysql.com/doc/refman/8.0/en/json-function-reference.html>

Attributi

Per i campi numerici

Si tratta di un istruzione ulteriore che viene passata al DBMS in fase di creazione (o modifica) di una colonna di tabella.

- **AUTO_INCREMENT** - Aumenta automaticamente il valore di una colonna aggiungendo 1 al valore massimo già registrato nella tabella. E' opportuno utilizzarlo in combinazione con NOT NULL. Può essere applicato a tutti i tipi numerici interi.
- **UNSIGNED*** - un campo contrassegnato con UNSIGNED non accetta valori negativi.
- **ZEROFILL**** - viene utilizzato per memorizzare i numeri con degli zeri davanti nel caso in cui la lunghezza sia inferiore a quella massima prevista.

Se per esempio si dichiara un campo INT ZEROFILL e si desidera memorizzare il valore "1234", esso verrà visualizzato come "00000001234" con M(11) - sempre UNSIGNED.

*A partire da MySQL 8.0.17 è deprecato per FLOAT, DOUBLE e DECIMAL; **A partire da MySQL 8.0.17 è deprecato; [vedi documentazione mysql](#).

AUTO_INCREMENT (sequence)

In MySQL, una *sequenza* è un elenco di interi generati nell'ordine crescente, vale a dire 1,2,3...

Impostare l'attributo AUTO_INCREMENT in una colonna, tipicamente una colonna chiave primaria, crea automaticamente una sequenza in MySQL.

Le seguenti regole vengono applicate quando si utilizza l'attributo AUTO_INCREMENT:

- Ciascuna tabella ha **solo una colonna** AUTO_INCREMENT il cui tipo di dati è "**intero**".
- La colonna AUTO_INCREMENT **deve essere indicizzata**, il che significa che può essere:
 - ▶ PRIMARY KEY o UNIQUE.
- La colonna AUTO_INCREMENT deve avere un vincolo NOT NULL.
Quando si imposta l'attributo AUTO_INCREMENT in una colonna, **MySQL aggiunge automaticamente il vincolo NOT NULL** alla colonna implicitamente.

Attributi per i campi di tipo stringa

BINARY:

- L'unico vincolo/opzione che può essere utilizzato per i campi destinati ad ospitare dati stringa è **BINARY** il quale può essere utilizzato con **CHAR** o **VARCHAR** qualora questi campi siano destinati ad ospitare dati binari (pur non rendendosi necessario utilizzare un campo della famiglia **BLOB**)
- **CHAR(n) BINARY** e **VARCHAR(n) BINARY** differiscono dal *tipo di dato* **BINARY** e **VARBINARY** per la codifica e la collation utilizzata*

* <https://dev.mysql.com/doc/refman/8.0/en/binary-varbinary.html>

Attributi universali (sia per campi numerici sia per quelli testuali)

Possono essere utilizzati tanto con campi numerici quanto con campi di tipo stringa.

- **DEFAULT** - Può essere utilizzato con tutti i tipi di dati ad eccezione di TEXT e BLOB.
Serve per indicare un valore di default per il campo qualora questo venga lasciato vuoto.
- **NULL / NOT NULL** - Può essere utilizzato con tutti i tipi di campi e serve per definire se un dato campo può avere un valore NULL oppure no.

Vincoli

- **CHECK** (expression) - consente di imporre un vincolo al dato da inserire.
- **FOREIGN KEY** - consente di imporre un vincolo riferito alla chiave esterna.

Attributi/Indici

- **UNIQUE** - Con UNIQUE si imposta una regola di unicità, questo significa che nessun dato contenuto nella colonna può essere ripetuto: ogni dato deve, quindi, essere unico e se si cerca di inserire un dato duplicato si riceve un errore. Può essere nullo.
- **PRIMARY KEY** - Può essere utilizzato con tutti i tipi di dati (numerici e stringa) ed è una sorta di variante di UNIQUE che consente di creare un indice primario sulla tabella (campo chiave).
- **INDEX [KEY] (colonne)** - E' utilizzato per creare un'indice nella tabella ai fini di migliorare le performances di accesso ai dati.

INDEX (indici)

(introduzione al concetto, verranno trattati in modo approfondito in questa sezione)

- Servono ad ottimizzare le performance del database.
- Un indice è una struttura dati ausiliaria che consente di recuperare più velocemente i dati di una tabella, evitandone la lettura dell'intero contenuto (full table scan), tramite una selezione più mirata.
- devono essere usati consapevolmente per non ottenere l'effetto contrario ovvero rallentare il db.

SQL - Structured Query Language - introduzione

SQL - Structured Query Language, è il linguaggio che permette di effettuare le operazioni per estrarre e manipolare i dati da un database.

E' lo standard tra i sistemi relazionali: viene usato in tutti i prodotti DBMS come set di comandi per l'utente della base di dati

Tipi di istruzioni SQL

- **DCL** (Data control language): permette di gestire il controllo degli accessi e i permessi per gli utenti
- **DDL** (Data Definition Language): permette di definire la struttura del database
- **DML** (Data manipulation language): permette di modificare i dati contenuti nel db, con le operazioni di inserimento, variazione e cancellazione
- **TCL** (Transaction Control Language): queste operazioni gestiscono le transazioni nel database
- **Query Language**: permette di porre interrogazioni al db

DCL

gestire il controllo degli accessi e i permessi per gli utenti:

```
CREATE USER 'vecchione'@'host' IDENTIFIED BY 'password';
```

```
GRANT ALL  
ON nomedb.*  
TO 'user'@'host';
```

DDL

permette di definire la struttura del database

```
CREATE DATABASE databaseName; -- crea un nuovo database  
DROP DATABASE databaseName; -- cancella il database  
CREATE TABLE tableName(...); -- crea una nuova tabella nel DB  
ALTER TABLE tableName ... ; -- modifica la struttura di una tabella  
DROP TABLE tableName ... ; -- cancella una tabella dal DB  
CREATE INDEX indexName ... ; -- crea un indice su una certa tabella  
ALTER TABLE tableName DROP INDEX indexName ... ; -- elimina l'indice specificato
```

DML

permette di *modificare i dati contenuti nel db*, con le operazioni di inserimento, variazione e cancellazione

- Inserimento:

```
INSERT INTO tableName(field1, field2, ...)  
VALUES ('value1', 'value2', '...');
```

- Cancellazione:

```
DELETE FROM tableName  
WHERE column_name = some_value;
```

- Aggiornamento:

```
UPDATE tableName  
SET column_name = new_value  
WHERE column_name = some_value;
```

TCL

gestiscono le transazioni nel database

```
COMMIT; -- rende definitive le operazioni sul database  
ROLLBACK; -- ripristina i dati eliminando le modifiche temporanee  
SAVEPOINT save_point_name(...); -- crea un punto di salvataggio
```

Query Language

permette di porre interrogazioni al db

```
SELECT field(s)
FROM table(s)
WHERE condition(s);
```

Attraverso **SELECT** vengono selezionati dei campi (*attributi*) da una o più tabelle e restituiti all'utente sotto forma di una "nuova tabella" (*resultset*)

Attraverso la clausola **WHERE** è possibile filtrare il *resultset* sulla base di alcune regole

Installazione MySQL 8

Dal sito di mysql scaricate il pacchetto relativo al vostro sistema operativo (Linux, Windows o MacOS).

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

...

Il *client mysql* verrà installato nella cartella...



▶ C:\Program Files\MySQL\MySQL Server 8.0\bin



▶ /usr/local/mysql-8.0.26-macos10.15-x86_64/bin

dove si trova l'eseguibile di mysql

Avvio del servizio

MySQL viene eseguito come servizio o come *daemon*.

Un servizio o demone è un *programma in esecuzione continua* nel sistema operativo, il cui compito è quello di rimanere in attesa di richieste per la fruizione di specifiche funzionalità.

Il demone si chiama *mysqld*:

mysqld viene avviato in automatico all'avvio del sistema
(a seconda della configurazione applicata durante l'installazione)

Possiamo verificare che il servizio sia attivo con **mysqladmin**

```
mysqladmin -u root -p ping
```

Fornendo i corretti dati di autenticazione, se il DBMS è attivo, sarà stampato il messaggio “mysql is alive”;

```
mysqld is alive
```

se invece non è attivo, verrà mostrato un errore di connessione.

```
mysqladmin: connect to server at 'localhost' failed  
error: 'Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)'  
Check that mysqld is running and that the socket: '/tmp/mysql.sock' exists!
```

Avvio/Arresto del servizio

MacOSX

Su mac è installato un controllo tra le Preferenze di sistema per gestire il servizio di MySQL.

Qui avviate/arrestate il servizio cliccando sul pulsante:

Stop / Start MySQL Server

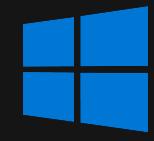
Windows

Apri la finestra *Esegui/Start* usando il tasto: Win + R

Digita *services.msc*

Ora cerca il servizio MySQL in base alla versione installata.

Fare clic su *interrompi, avvia* o *riavvia* il servizio.



PATH MySQL

Il *client mysql* si trova nella cartella...

- ▶ C:\Program Files\MySQL\MySQL Server 8.0\bin

Ogni volta che di deve accedere al servizio, una volta aperto il terminale, bisogna portarsi nel punto in cui è stato installato mysql:

```
cd C:\Program Files\MySQL\MySQL Server 8.x\bin
```

A questo punto possiamo accedere al servizio avviando mysql.

Per evitare di digitare ogni volta il percorso bisogna creare una scorciatoia per la shell

In Windows bisogna *aggiungere una variabile di ambiente* relativa al path di MySQL.



PATH MySQL

Il *client mysql* si trova nella cartella...

- ▶ /usr/local/mysql-8.0.23-macos10.15-x86_64/bin

Ogni volta che di deve accedere al servizio, una volta aperto il terminale, bisogna portarsi nel punto in cui è stato installato mysql:

```
cd /usr/local/mysql-8.0.26-macos10.15-x86_64/bin/
```

A questo punto possiamo accedere al servizio avviando mysql (vedi slide 70)

Per creare una scorciatoia per la shell (impostare la shell bash) su mac bisogna creare un file nascosto ".bash_profile" con la scorciatoia scritta in questo modo:

```
export PATH=${PATH}:/usr/local/mysql/bin
```

Accesso al DBMS

Amministratore del servizio (DBA - Data Base Administrator)

Da terminale accedere a MySql:

```
mysql -u root -p
```

Vi verrà chiesto di inserire la password (è quella creata in fase di installazione)

```
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 16  
Server version: 8.0.23 MySQL Community Server - GPL
```

```
Copyright (c) 2000, 2021, Oracle and/or its affiliates.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql>
```

Visualizzate elenco database disponibili:

```
mysql> show databases;
```

Essendo una prima installazione dovreste vedere i seguenti db:

```
+-----+  
| Database |  
+-----+  
| information_schema |  
| mysql |  
| performance_schema |  
| sys |  
+-----+  
4 rows in set (0.01 sec)  
  
mysql>
```

I database elencati sono funzionali al servizio, sono disponibili solo all'amministratore del sistema;

information_schema e *performance_schema* sono disponibili anche all'utente per ottenere informazioni sul proprio database e sulle prestazioni delle query.

DDL

Data Definition Language (1parte)

Creare un database

Una volta effettuato l'accesso possiamo eseguire l'istruzione CREATE DATABASE (CREATE SCHEMA) seguita dal nome del database da creare.

```
CREATE DATABASE databaseName;
```

Se il database è già presente mysql ve lo segnala attraverso un messaggio di errore:

```
ERROR 1007 (HY000): Can't create database 'nomedatabase'; database exists
```

Usando la sintassi seguente

```
CREATE DATABASE IF NOT EXISTS databaseName;
```

mysql verifica l'esistenza del db: se non esiste lo crea, se esiste vi da ok ma segnala un *warning*.

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

In fase di creazione di un database con MySQL è anche possibile specificare charset e collation; ad esempio:

```
CREATE DATABASE IF NOT EXISTS dbName
CHARACTER SET utf8
COLLATE utf8_general_ci;
```

Specificando questi valori è possibile "sovrascrivere" quelli impostati di default a livello server.

Per visualizzare come è stato creato il database

```
SHOW CREATE DATABASE dbName;
```

Per elencare rispettivamente i set di caratteri disponibili e le "collezioni" disponibili.

```
SHOW CHARACTER SET; SHOW COLLATION;
```

Cancellare un database

Per eliminare un DB basta scrivere l'istruzione DROP DATABASE (DROP SCHEMA) seguita dal nome del database da rimuovere.

```
DROP DATABASE databaseName;  
DROP DATABASE IF EXISTS databaseName;
```

Se si usa l'istruzione opzionale **IF EXISTS** si evita di ricevere l'errore qualora il database sia già stato eliminato.

DCL

Data Control Language

DCL

gestire il controllo degli accessi e i permessi per gli utenti:

Istruzione **CREATE USER**: crea l'utente e assegna una password

```
CREATE USER 'app_tsiw'@'localhost' IDENTIFIED BY 'tsiw2025!';
```

password : password associata all'utente che stiamo creando.

La password va scritta "in chiaro".

GRANT

Assegna i permessi e/o privilegi

```
GRANT --istruzioni_consentite  
ON namedatabase.*  
TO 'nameuser'@'hostuser';
```

istruzioni consentite: CREATE, SELECT, UPDATE, DELETE, ALTER, DROP,...

Per dare all'utente permessi completi utilizzare la parola chiave ALL.

database.* : nome del database sul quale l'utente potrà eseguire le istruzioni consentite.

Per tutte le tabelle del db: .* . Si può specificare il nome di una o più tabelle. Per tutti i database: *.* .

user : specifica il nome dell'utente che vogliamo creare o al quale vogliamo assegnare nuovi permessi.

host : specifica il/gli host da cui è ammessa la connessione. Se voglio indicare più IP devo usare la wild card %: 130.192.200.%

REVOKE

```
REVOKE --istruzioni_revocate  
ON databaseName.*  
FROM 'user'@'host';
```

per la quale valgono le stesse regole sopra viste per **GRANT**.

Per eliminare tutti i privilegi:

```
REVOKE ALL PRIVILEGES, GRANT OPTION -- istruzioni_revocate  
FROM 'user'@'host';
```

Quest'ultima sintassi elimina ogni permesso dell'utente su qualunque database del sistema.

Cambiare/aggiornare la password MySQL degli utenti

Per cambiare una normale password utente devi digitare:

- *Cambia password per l'utente (da root):*

```
ALTER USER 'userName'@'host' IDENTIFIED BY 'newpass';
```

- Cambiare la propria password:

```
ALTER USER USER() IDENTIFIED BY 'newpass';
```

Verificare i permessi utente

Verificare i privilegi di uno specifico utente:

```
SHOW GRANTS FOR 'user'@'localhost';
```

Verificare i privilegi dell'utente attualmente loggato a MySQL:

```
SHOW GRANTS FOR CURRENT_USER;
```

Eliminare un utente da MySQL

```
DROP USER 'user'@'localhost';
```

questo comando rimuove l'utente e i suoi permessi.

Visualizzare elenco utenti mysql (solo utente root)

```
SELECT user, host FROM mysql.user;
```

Accesso al DBMS

Utente

Da terminale accedere a MySql utilizzando le credenziali dell'utente creato:

```
mysql -u nomeUser -p
```

Vi verrà chiesto di inserire la password (quella assegnata all'utente)

```
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 16  
Server version: 8.0.23 MySQL Community Server - GPL
```

```
Copyright (c) 2000, 2021, Oracle and/or its affiliates.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql>
```

```
mysql> show databases;
```

Accesso al DBMS - Utente

A questo punto dovete prendere possesso del database per poterci lavorare:

L'istruzione è **USE** più nome del database assegnato (senza ; a chiusura del comando)

```
use nomeDatabase
```

Da terminale potete accedere *anche direttamente* al database per i quali avete i privilegi.

Dovete passare a mysql tutti i parametri (tranne la password) più il nome del database:

```
mysql -u nomeUser -p nomeDatabase
```

Non dovete più usare **USE**, siete nel database e potete lavorarci*

*le applicazioni, di solito, hanno un file dedicato alla connessione al db in cui vengono passati tutti i parametri di connessione compreso l'host e in alcuni casi anche la porta (dipende dal provider del servizio).

DDL

Data Definition Language (2 parte)

Creare le tabelle

Per creare una tabella usiamo il comando `CREATE TABLE tableName()`.

Quando creiamo una tabella dobbiamo definire anche tutti i campi ad essa associati, argomenti della parentesi.

- Per ogni campo verrà definito il dominio che indica al sistema quale tipo di dati verrà memorizzato nel campo.
- Per ogni campo definiamo anche gli eventuali attributi

```
CREATE TABLE IF NOT EXISTS nome_tabella(  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    fieldName1 VARCHAR(60) NOT NULL,  
    fieldName2 DATE,  
    fieldName3 TINYINT  
)  
[ENGINE InnoDB CHARACTER SET utf8 COLLATE utf8_general_ci];
```

Esempio di creazione di una tabella denominata studente, con il campo id come chiave primaria.

```
CREATE TABLE IF NOT EXISTS studenti(
    id INT AUTO_INCREMENT,
    nome VARCHAR(20),
    cognome VARCHAR(30) NOT NULL,
    genere ENUM('m','f'),
    indirizzo VARCHAR(100),
    citta VARCHAR(30),
    provincia CHAR(2) DEFAULT 'To',
    regione VARCHAR(30) DEFAULT 'Piemonte',
    email VARCHAR(100) NOT NULL UNIQUE,
    data_nascita date,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY(id)
);
```

Verificare se e come è stata creata la tabella:

- Per verificare se la nostra tabella è stata creata si usa l'istruzione che mostra le tabelle presenti:

```
SHOW TABLES;
```

- Per verificare che la struttura della tabella sia corretta:

```
SHOW CREATE TABLE tableName;
```

- Per visualizzare come è stata creata una tabella:

```
DESCRIBE tableName; DESC tableName;
```

Rinominare una tabella:

```
ALTER TABLE tableName RENAME newtableName;
```

```
RENAME TABLE tableName TO newtableName;
```

INFORMAZIONI SULLE TABELLE

Per conoscere la struttura della tabella con più o meno informazioni (valore dell'auto_increment, data di creazione, collation)

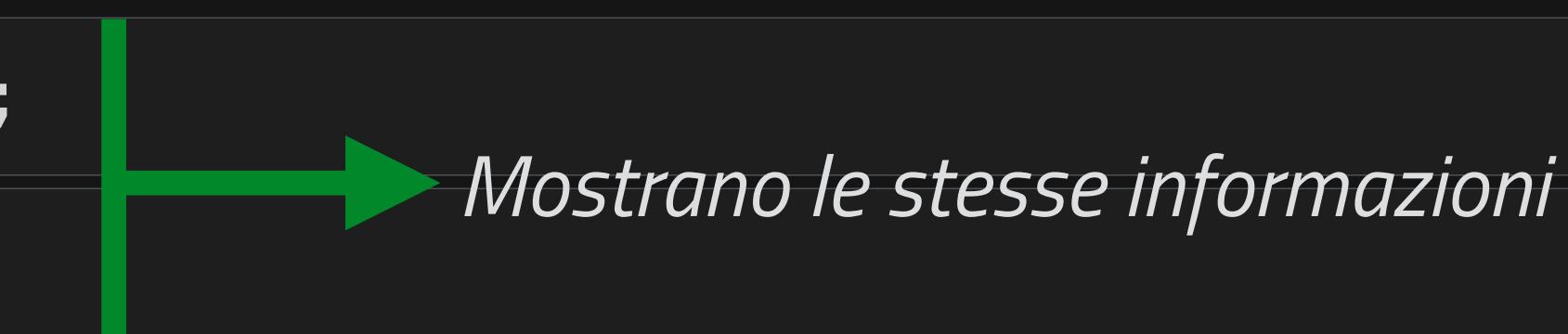
```
DESCRIBE tableName; DESC tableName;
```

```
SHOW COLUMNS FROM tableName;
```

```
SHOW FULL COLUMNS FROM tableName;
```

```
SHOW INDEX FROM tableName;
```

```
SHOW TABLE STATUS LIKE 'tableName'1; -- mostra valore auto_increment
```



Mostra ulteriori informazioni (privilegi e commenti)

Per conoscere solo il valore dell'auto_increment di una tabella possiamo interrogare anche il db INFORMATION_SCHEMA

```
SELECT table_name, auto_increment  
FROM information_schema.tables  
WHERE table_schema = 'databaseName'1;  
-- AND TABLE_NAME = 'tableName';
```

1) Le statistiche della tabella vengono memorizzate nella cache. Per disabilitare la cache e avere sempre l'ultima versione è necessario modificare la variabile del server che indica la durata del cache-clear a 0:

```
SET PERSIST information_schema_stats_expiry = 0 -- bisogna usare l'utente DBA (root)
```

INFORMATION_SCHEMA

INFORMATION_SCHEMA fornisce l'accesso ai metadati del database, le informazioni sul server MySQL come il nome di un database o una tabella, il tipo di dati di una colonna, o privilegi di accesso...

Note sull'utilizzo INFORMATION_SCHEMA

- INFORMATION_SCHEMA è una banca dati all'interno di ciascuna istanza di MySQL, il **luogo che memorizza le informazioni su tutti gli altri database che il server MySQL mantiene.**
Il database INFORMATION_SCHEMA contiene diverse tabelle di sola lettura (in realtà viste).
- Anche se è possibile selezionare INFORMATION_SCHEMA come database predefinito con un'istruzione USE, *è possibile SOLO leggere il contenuto delle tabelle*, non eseguire INSERT, UPDATE o DELETE.

INFORMATION_SCHEMA

Vediamo un esempio di utilizzo di INFORMATION_SCHEMA

```
SELECT table_name, table_type, engine, table_collation  
FROM information_schema.tables  
WHERE table_schema = 'databaseName'  
ORDER BY table_name;
```

Questa select ci restituirà l'elenco delle tabelle del nostro database indicando il tipo di tabella e l'engine associato.

Modificare le tabelle

L'istruzione **ALTER TABLE** viene utilizzata per aggiungere, eliminare o modificare le colonne di una tabella esistente.

Per *aggiungere un campo a una tabella*, utilizzare la seguente sintassi:

```
ALTER TABLE tableName  
ADD fieldName [DATATYPE];
```

Per modificare *nome* e *datatype* di un campo di una tabella

```
ALTER TABLE tableName  
CHANGE fieldName newFieldName [DATATYPE];
```

Per modificare *solo datatype* di un campo di una tabella

```
ALTER TABLE tableName  
MODIFY fieldName [DATATYPE];
```

Per modificare *solo nome* di un campo di una tabella

```
ALTER TABLE tableName  
RENAME COLUMN oldName TO newName;
```

Per cancellare un campo di una tabella

```
ALTER TABLE tableName  
DROP fieldName;
```

Potete combinare le istruzioni separandole con una virgola

```
ALTER TABLE tableName  
CHANGE fieldName newFieldName [DATATYPE],  
ADD fieldName3 [DATATYPE] AFTER fieldName2,  
DROP fieldName4;
```

Per aggiungere un campo in una data posizione

```
ALTER TABLE tableName  
ADD fieldName2 [DATATYPE]  
AFTER fieldName;
```

Usando *FIRST* al posto di *AFTER* si aggiunge il campo in prima posizione, *FIRST* non vuole il nome del campo

Per spostare un campo in una tabella

```
ALTER TABLE tableName  
MODIFY fieldName2 [DATATYPE]  
AFTER fieldName;
```

Per spostare un campo in prima posizione

```
ALTER TABLE tableName  
MODIFY fieldName2 [DATATYPE]  
FIRST;
```

se il campo è una chiave primaria non dovete indicarlo nel datatype altrimenti ricevete un errore di mysql

- Aggiungere alla tabella la PRIMARY KEY (se non già impostata):

```
ALTER TABLE tableName ADD PRIMARY KEY (field1[, field2, ...]);
```

- Aggiungere alla tabella la PRIMARY KEY aggiungendo campo nuovo apposito:

```
ALTER TABLE tableName  
ADD id INT AUTO_INCREMENT PRIMARY KEY;
```

- Eliminare la PRIMARY KEY

```
ALTER TABLE tableName DROP PRIMARY KEY;
```

Attenzione, per eliminare la primary key di un campo id con auto_increment, bisogna prima eliminare l'attributo auto_increment

```
ALTER TABLE tableName MODIFY id INT;
```

Duplicare tabelle

Se abbiamo necessità di duplicare una tabella possiamo utilizzare l'istruzione **CREATE TABLE** combinata con l'istruzione **LIKE**.

Per duplicare una tabella possiamo scrivere:

```
CREATE TABLE tableNameCopy LIKE tableName;
```

Questa istruzione duplica solo la struttura della tabella.

Cancellare una tabella dal database MySQL

Vediamo l'operazione inversa alla creazione di tabelle, la loro eliminazione.

- Per eliminare una tabella utilizzeremo il comando `DROP TABLE`.

```
DROP TABLE tableName;
```

L'eliminazione di una tabella, come per il database, è un'operazione irreversibile

- È possibile eliminare più di una tabella contemporaneamente:

```
DROP TABLE tableName, tableName2, tableName3;
```

DML

Data Manipulation Language

Creazione, lettura, aggiornamento e eliminazione dei record (CRUD)

Una volta creata la struttura del nostro database ci ritroveremo, ovviamente, con una serie di tabelle vuote.

Prima di aggiungere record a una tabella bisogna conoscere il tipo di dati previsto per ogni campo, quali campi non possono avere valore nullo, quali campi hanno l'incremento automatico...

Quando si inseriscono i dati bisogna usare le *virgolette* o gli *apici* per i dati *tipo stringa* (compresa la data), *senza virgolette o apici* per i dati di *tipo numerico*.

Non si inseriscono i valori per i campi definiti con l' *auto_increment*.

Consistenza dei dati

La consistenza dei dati è una proprietà fondamentale nei database che garantisce che i dati rispettino regole e vincoli predefiniti, mantenendo uno stato valido e coerente.

In altre parole, **i dati devono sempre essere corretti e rispettare le regole definite dal database.**

```
CREATE TABLE studenti (
    id INT PRIMARY KEY,
    nome VARCHAR(50),
    età TINYINT UNSIGNED CHECK (età >= 18) -- Vincolo di consistenza
);
```

La consistenza fa parte delle proprietà ACID (Atomicità, Consistenza, Isolamento, Durabilità), fondamentali per garantire la corretta esecuzione delle transazioni nei database relazionali.

INSERT INTO

INSERT INTO è l'istruzione utilizzata per inserire nuovi record in una tabella. Ha due parti

INSERT INTO seleziona la tabella e i campi per i quali effettuare l'inserimento

VALUE/VALUES elenca i valori dei campi da inserire

```
INSERT INTO tableName(field1, field3)
VALUES(value1, value3);
```

È possibile inserire più record con un solo INSERT separando l'elenco dei valori di ogni record con la: ,

```
INSERT INTO tableName(field1, field2, field3,...)
VALUES(r1_value1, r1_value2, r1_value3, ...),(r2_value1, r2_value2, r2_value3, ...);
```

Altra sintassi per singolo record con istruzione SET:

```
INSERT INTO tableName
SET field1 = 'value1', field2 = 'value2', field3 = 'value2';
```

INSERT INTO

È possibile usare il comando **INSERT INTO** senza l'uso di nomi di campo se si inserisce un record rispettando l'ordine dei campi della tabella

```
INSERT INTO tableName  
VALUES(value1, value2, value3);
```

in questo caso devono essere inseriti i valori di tutti i campi, anche i valori *AUTO_INCREMENT* o *TIMESTAMP* (passare default per inserimento automatico).
Per i campi che accettano i valori nulli potete passare `null`.

```
INSERT INTO studente  
VALUES(default, 'fabio', 'rossi', 'fbr@gmail.com', null, default);
```

Mostrare i record di una tabella

È possibile visualizzare i record di una tabella utilizzando l'istruzione **SELECT**.

Per visualizzare tutti i record da una tabella si usa il carattere jolly *.

Dobbiamo anche utilizzare l'istruzione **FROM** per identificare la tabella che vogliamo interrogare:

```
SELECT * FROM tableName;
```

Di solito si visualizzano campi specifici, piuttosto che l'intera tabella.

Dopo l'istruzione SELECT elencare i campi che interessano, separati da una virgola.

```
SELECT fieldName, fieldName2, fieldName3 FROM tableName;
```

INSERT INTO ... SELECT

L'istruzione INSERT INTO ... SELECT consente di inserire automaticamente dati in una tabella copiandoli da un'altra:

```
INSERT INTO amici(nome, cognome)
SELECT nome, cognome
FROM studenti;
```

Ad esempio, possiamo popolare la tabella *amici* copiando i dati già presenti nella tabella *studenti*.

- La tabella di destinazione (*amici*) deve esistere.
- I tipi di dato tra i campi selezionati devono essere compatibili.
- L'ordine dei campi nel SELECT deve corrispondere all'ordine dei campi specificati in INSERT.

Duplicare tabelle e suoi contenuti

Se abbiamo necessità di copiare il contenuto di una tabella in altra tabella, possiamo utilizzare l'istruzione **CREATE TABLE** combinata con **LIKE** e le istruzioni **SELECT**.

Per duplicare esattamente una tabella (con indici e chiavi) e i suoi contenuti bisogna usare due istruzioni separate:

```
CREATE TABLE studenti_bk LIKE studenti;  
INSERT INTO studenti_bk SELECT * FROM studenti;
```

Si può usare anche un'istruzione sola: in questo caso gli indici non vengono ricreati, cioè le strutture delle tabelle sono diverse:

```
CREATE TABLE studenti_bk2 [AS]  
SELECT * FROM studenti;
```

UPDATE

Aggiornamento dei record in una tabella.

Questa istruzione modifica il valore presente in una colonna di un record già esistente.

Viene utilizzata insieme all'istruzione **SET**:

```
UPDATE tableName  
SET field1 = value1, field2 = value2  
WHERE field3 = value3;
```

- dopo **UPDATE** indichiamo quale tabella è interessata
- con **SET** specifichiamo quali colonne modificare e quali valori assegnare
- con **WHERE** (opzionale) stabiliamo le condizioni che determinano quali righe saranno interessate dalle modifiche (Attenzione: **se WHERE è omesso, tutti i record saranno aggiornati** per le colonne indicate).

Per operare simultaneamente su più campi è sufficiente suddividere le coppie chiave/valore con una virgola.

Quando si inseriscono i dati in una tabella rammmentate sempre come sono state definite le colonne per evitare errori di inserimento.

Se si inserisce un valore *troppo lungo*, o *non compreso* dalla definizione della colonna, MySQL restituisce un errore* e non effettua alcuna modifica.

```
UPDATE studenti SET genere = 's' WHERE id = 1;  
ERROR 1265 (01000): Data truncated for column 'genere' at row 1
```

Il campo *genere* della tabella *studenti* è un campo definito come:

ENUM('f', 'm', 'nb'), accetta quindi solo i valori *f*, *m* o *nb*.

In questo caso stiamo tentando di inserire un valore non ammesso.

*dipende dall'impostazione della variabile globale @@sql_mode: di default mysql lavora in strict mode.

SQL Mode: STRICT MODE

Il server MySQL può funzionare in diverse modalità SQL e può applicare queste modalità in modo diverso per client diversi, a seconda del valore della variabile di sistema: *SQL_MODE*.

I DBA possono impostare la modalità SQL globale in modo che corrisponda ai requisiti operativi del server del sito e ogni applicazione può impostare la modalità SQL della sessione in base ai propri requisiti.

Le modalità influiscono sulla sintassi SQL supportata da MySQL e dai controlli di convalida dei dati che esegue. Ciò semplifica l'utilizzo di MySQL in ambienti diversi e l'utilizzo di MySQL insieme ad altri server di database.

```
SELECT @@SQL_MODE;
+-----+
| @@SQL_MODE |
+-----+
| NO_ZERO_IN_DATE,NO_ZERO_DATE,NO_ENGINE_SUBSTITUTION |
+-----+
```

Questo significa che se volete operare con un sessione che lavori in STRICT MODE (che attivi i controlli sui campi per esempio) dovete impostare la variabile ad inizio sessione di connessione:

```
SET SQL_MODE = 'TRADITIONAL';
```

<https://dev.mysql.com/doc/refman/8.0/en/sql-mode.html>

Eliminazione dei record in una tabella

L' istruzione **DELETE** viene utilizzata per eliminare gruppi di record in una tabella.

È necessario utilizzare la parola chiave condizionale **WHERE** per scegliere quali record si desidera eliminare, altrimenti si eliminano tutti i record.

La sintassi di base per l'istruzione è la seguente :

```
DELETE  
FROM tableName  
WHERE field = value;
```

esempio:

```
DELETE  
FROM studenti  
WHERE genere = 'm';
```

Eliminare tutti i record della tabella

Per svuotare una tabella si usa l'istruzione **TRUNCATE**

```
TRUNCATE [TABLE] tableName;
```

Questa soluzione è la più veloce perché elimina la struttura della tabella per poi ricrearne una uguale vuota azzerando il valore di eventuali campi AUTO_INCREMENT.

Usando **DELETE** si eliminano tutti i record presenti nella tabella specificata *record per record*.

```
DELETE FROM tableName;
```

Un simile modo di operare, seppur assolutamente funzionante, è poco efficiente perché dipende dalla quantità di righe presenti in tabella.

Inoltre usando DELETE il valore di un eventuale AUTO_INCREMENT rimane inalterato; per azzerarlo:

```
ALTER TABLE tableName AUTO_INCREMENT = 1;
```

INFORMAZIONI SULLE TABELLE

Per conoscere la struttura della tabella con più o meno informazioni (valore dell'auto_increment, data di creazione, collation)

```
DESCRIBE tableName; DESC tableName;
```

```
SHOW COLUMNS FROM tableName;
```

```
SHOW FULL COLUMNS FROM tableName;
```

```
SHOW INDEX FROM tableName;
```

```
SHOW TABLE STATUS LIKE 'tableName'^1; -- mostra valore auto_increment
```

Per conoscere solo il valore dell'auto_increment di una tabella possiamo interrogare anche il db INFORMATION_SCHEMA

```
SELECT table_name, auto_increment  
FROM information_schema.tables  
WHERE table_schema = 'databaseName'^1;  
-- AND TABLE_NAME = 'tableName';
```

1) Le statistiche della tabella vengono memorizzate nella cache. Per disabilitare la cache e avere sempre l'ultima versione è necessario modificare la variabile del server che indica la durata del cache-clear a 0:

```
SET PERSIST information_schema_stats_expiry = 0 -- bisogna usare l'utente DBA (root)
```

Commenti

MySQL Server supporta tre stili di commento:

- `#` da questo simbolo a fine riga.
- `--` da questo simbolo a fine riga.
Lo stile di commento (doppio trattino) richiede che il secondo trattino sia seguito da almeno uno spazio bianco o un carattere di controllo (come uno spazio, tab, nuova riga e così via).
- `/* commento */` come nel linguaggio C.

Questa sintassi consente un commento su più righe e l'inserimento del commento *inline*.

L'esempio seguente mostra tutti e tre gli stili di commento:

```
SELECT 1 + 1; # Questo commento continua fino alla fine della riga
SELECT 1 + 1; -- Questo commento continua fino alla fine della riga
SELECT 1 /* questo è un commento in linea */ + 1;
SELECT 1 +
/*
questo è un commento
su più linee
*/
1;
```

Query

Query Language - interrogazione dei dati

Interrogazione dei dati

Creazione di query di base

Abbiamo già introdotto il comando SELECT per visualizzare i record inseriti in una tabella.

```
SELECT field1, field2, field3 FROM tableName;
```

Utilizzando il carattere jolly * selezioniamo tutte le colonne dei campi di dati da visualizzare.

```
SELECT * FROM tableName;
```

ORDER BY

Consente di ordinare i risultati di una query.

L'istruzione *ORDER BY* è seguita dal(dai) campo(i) su cui si basa l'ordinamento.

L'ordine predefinito è crescente (ASC).

```
SELECT *
FROM studenti
ORDER BY cognome;
```

Usando l'istruzione **DESC** si ordina in modo decrescente:

```
SELECT *
FROM studenti
ORDER BY cognome DESC;
```

Se si vuole ordinare per nome con ordinamento *DESC* e per eta con ordinamento *ASC* dobbiamo scrivere:

```
SELECT *
FROM studenti
ORDER BY cognome DESC, eta;
```

LIMIT

Consente di definire il numero dei record massimo da visualizzare.

Accetta due argomenti: [offset] è opzionale, specifica l'indice da cui partire per mostrare i record.

Il secondo argomento indica la quantità di record da mostrare:

```
SELECT *
FROM tableName
ORDER BY field
LIMIT {[offset,] row_count | row_count OFFSET offset};
```

La query seguente mostra solo i primi 10 record della tabella *studente*:

```
SELECT *
FROM studenti
ORDER BY cognome
LIMIT 10;
```

La query seguente mostra 10 record della tabella *studente* presi a partire dall'*undicesimo record*.

Ricordate che l'indice parte da 0, per cui l'*undicesimo record* è l'indice numero 10 .

```
SELECT *
FROM studenti
ORDER BY cognome
LIMIT 10 , 10; -- LIMIT 10 OFFSET 10, alternativa più leggibile
```

SELECT e WHERE

WHERE consente di filtrare i risultati di una query, *mostrando solo i record che soddisfano la condizione imposta.*

Se vogliamo selezionare il nome e cognome degli studenti solo di genere maschile possiamo applicare un filtro sull'attributo genere come condizione del **WHERE** grazie all'uso degli operatori.

```
SELECT nome, cognome  
FROM studenti  
WHERE genere = 'm';
```

Operatori

- operatori di confronto

=, <>, !=, >, >=, <, <=

- operatori logici

AND, OR, NOT

- operatori di confronto avanzato

IN, NOT IN, BETWEEN, NOT BETWEEN, IS NULL, IS NOT NULL,
LIKE, NOT LIKE, REGEXP

- operatori matematici

+, -, *, /, %

Operatori di confronto

Operatore	Descrizione
=	Equal
<>	Not Equal
!=	Not Equal
>	Greater Than
>=	Greater Than or Equal
<	Less Than
<=	Less Than or Equal

operatori confronto

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE genere = 'f'  
ORDER BY cognome, nome;
```

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE citta <> 'torino'  
ORDER BY cognome, nome;
```

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE data_nascita >= '1989-12-31'  
ORDER BY data_nascita;
```

operatori logici: **AND** e **OR**

Quando si utilizza l'operatore **AND** **tutte le condizioni** devono essere vere.

```
SELECT field(s)
FROM tableName
WHERE condition1 AND condition2 AND condition3;
```

```
SELECT nome, cognome, email, data_nascita
FROM studenti
WHERE genere = 'm' AND provincia = 'to';
```

Vengono filtrati i record degli studenti maschi provenienti dalla provincia di Torino.

Quando si utilizza l'operatore **OR** **almeno una delle condizioni** deve essere vera.

```
SELECT nome, cognome, email, data_nascita
FROM studenti
WHERE genere = 'm' OR provincia = 'to';
```

Vengono filtrati i record degli studenti maschi e degli studenti che provengono dalla provincia di Torino (tra cui potrebbero esserci delle studentesse).

Attenzione: l'operatore AND ha precedenza su OR, quindi è consigliabile usare le parentesi per una migliore leggibilità.

Esempio senza parentesi (potenzialmente ambiguo):

```
SELECT * FROM studenti  
WHERE cognome = 'verdi' OR cognome = 'rossi'  
AND provincia = 'to'; -- AND ha precedenza
```

Meglio con le parentesi per chiarire l'ordine della ricerca:

```
SELECT * FROM studenti  
WHERE cognome = 'verdi' OR (cognome = 'rossi'  
AND provincia = 'to'); -- AND ha precedenza
```

Esempio con più condizioni:

```
SELECT * FROM studenti  
WHERE (cognome = 'verdi' OR cognome = 'rossi')  
AND (provincia = 'to' OR provincia = 'al');
```

operatori logici: **NOT**

L'operatore **NOT nega** le condizioni poste.

Questa query seleziona alcuni attributi degli studenti il cui genere **non** è 'm'

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE NOT genere = 'm';
```

Questa query seleziona alcuni attributi degli studenti il cui genere non è 'm' e la cui provincia è 'to'

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE NOT genere = 'm' AND provincia = 'to';
```

Questa query seleziona alcuni attributi degli studenti il cui genere **non** è 'm' e la cui provincia **non** è 'to'

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE NOT genere = 'm' AND NOT provincia = 'to';
```

Se volete negare una o più condizioni è preferibile usare l'operatore `<>` o `!=`, il NOT può confondere e rendere meno leggibile la query

Operatori di confronto avanzati: **IN**, **NOT IN**

L'operatore **IN** ci consente di selezionare i record indicando più valori di campo.

Possiamo farlo con l'operatore OR, ma può diventare complicato quando confrontiamo molti valori.

Prendiamo ad esempio l'esecuzione di una query in cui cerchiamo solo gli studenti di alcune province:

```
SELECT nome, cognome, email, data_nascita, provincia  
FROM studenti  
WHERE provincia IN ('to', 'cn', 'at', 'no');
```

L'operatore **NOT IN** funziona in modo simile a **IN**, mostra i record che NON contengono i valori selezionati.

Così la seguente query mostrerà tutti i record di studenti che *non appartengono alle province in elenco*.

```
SELECT nome, cognome, email, data_nascita, provincia  
FROM studenti  
WHERE provincia NOT IN ('to', 'cn', 'at', 'no');
```

ATTENZIONE: *NOT IN* non considera i valori *NULL*, se si cercano anche quelli bisogna aggiungere ulteriore condizione.

BETWEEN, NOT BETWEEN

Utilizzando l'operatore BETWEEN possiamo selezionare un intervallo di valori. I valori di inizio e fine dell'intervallo sono inclusi. I valori dell'intervallo possono essere numeri, testo o date.

ATTENZIONE: NOT BETWEEN non considera i valori NULL.

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE data_nascita BETWEEN '1985-01-01' AND '1994-12-31';
```

IS NULL e IS NOT NULL

L'operatore **IS NULL** viene utilizzato per visualizzare i record che non hanno un valore impostato per un campo.

Viceversa **IS NOT NULL** mostra i record che hanno un valore impostato per un campo.

```
SELECT nome, cognome, email, data_nascita  
FROM studenti  
WHERE data_nascita IS NULL;
```

Questi operatori possono essere utilizzati per trovare i record che hanno bisogno di informazioni aggiuntive.

LIKE, NOT LIKE

L'operatore LIKE permette di effettuare *ricerche testuali con pattern matching*.

ATTENZIONE: NOT LIKE non considera i valori NULL.

```
SELECT * FROM studenti WHERE cognome LIKE 'v%';
```

```
SELECT * FROM studenti WHERE nome LIKE '%a';
```

```
SELECT * FROM studenti WHERE indirizzo LIKE 'via %';
```

```
SELECT * FROM studenti WHERE email LIKE '%gmail.com';
```

```
SELECT * FROM studenti WHERE nome LIKE 'paol_';
```

```
SELECT * FROM studenti WHERE nome LIKE '_a%';
```

La differenza è data dalla posizione del carattere percentuale (%) che sta ad indicare "qualsiasi carattere dopo" e "qualsiasi carattere prima".

L'uso di underscore (_) indica un solo carattere.

- non abusarne.
- i criteri di ricerca che iniziano con caratteri jolly sono quelli con i tempi di elaborazione più lunghi.

Wildcard	Descrizione
%	sostituisce zero o più caratteri
-	sostituisce un singolo carattere
[elenco caratteri]	pattern di caratteri da trovare (con wildcard)

REGEXP_LIKE() (REGEXP e RLIKE sono sinonimi di REGEXP_LIKE()).

Un operatore più potente di LIKE è **REGEXP** il quale consente di utilizzare molti più simboli per ricerche più complesse.

```
SELECT * FROM studenti WHERE REGEXP LIKE(nome,'ra'); SELECT * FROM studenti WHERE nome REGEXP 'ra';  
SELECT * FROM studenti WHERE nome REGEXP '^mar';  
SELECT * FROM studenti WHERE nome REGEXP 'co$';  
SELECT * FROM studenti WHERE nome REGEXP 'mar|ara|ola';
```

Potete combinare i simboli

```
SELECT * FROM studenti WHERE nome REGEXP '^mar|ara|co$';  
SELECT * FROM studenti WHERE nome REGEXP '^^(mar|ara)|co$'; -- comincia per 'mar' o 'ara' o finisce con 'co'
```

Creare combinazioni di pattern

```
SELECT * FROM studenti WHERE nome REGEXP '[mcp]a';  
SELECT * FROM studenti WHERE nome REGEXP 'a[ero]';  
SELECT * FROM studenti WHERE nome REGEXP 'l[ao]$';  
SELECT * FROM studenti WHERE nome REGEXP '^[a-m]'; -- '^ [n-z]'
```

Approfondimento:

https://dev.mysql.com/doc/refman/8.0/en/regexp.html#operator_regexp

<https://dev.mysql.com/doc/refman/8.0/en/regexp.html#regexp-syntax>

Operatori matematici

MySQL supporta i classici operatori matematici tradizionali, cioè:

- + (addizione)
- (sottrazione)
- * (moltiplicazione)
- / (divisione)
- % (modulo - il resto della divisione tra due numeri)

Questi operatori risultano molto utili quando, ad esempio, si devono svolgere dei calcoli all'interno di una SELECT.

Per fare un esempio si supponga di voler restituire il valore dato dalla sottrazione di due campi:

```
SELECT (field1 - field2)
FROM tableName
[WHERE condition(s)];
```

Potete eseguire dei calcoli matematici con SELECT:

```
SELECT 6 / 2 ; ##3
```

```
SELECT 35 % 3 ; ##2
```

```
SELECT (35 / 3) * 2 ; ##11.6667
```

Calcolo campi al volo

Utilizzando il comando SELECT, possiamo aggiungere alla nostra query campi calcolati, cioè risultati di operazioni algebriche effettuate su campi esistenti.

Nell'esempio seguente, la query restituisce il titolo del libro, il prezzo corrente e il prezzo aumentato del 10%:

```
SELECT titolo, prezzo, prezzo * 1.1  
FROM libro;
```

L'uso degli operatori aritmetici consente di eseguire calcoli su ogni riga del risultato.

Colonne generate (virtuali e persistenti / memorizzate)

Una **colonna generata** è una colonna in una tabella che non può essere impostata esplicitamente su un valore specifico in una query DML.

Il suo valore *viene invece generato automaticamente in base a un'espressione*. Questa espressione potrebbe generare il valore in base ai valori di altre colonne nella tabella oppure potrebbe generare il valore chiamando funzioni incorporate o funzioni definite dall'utente (UDF) .

Esistono due tipi di colonne generate:

- **PERSISTENT (STORED)**: il valore di questo tipo è effettivamente memorizzato nella tabella.
- **VIRTUAL**: Il valore di questo tipo non viene memorizzato affatto. Il valore viene invece generato dinamicamente quando viene eseguita una query sulla tabella. Questo tipo è l'impostazione predefinita.

Le colonne generate sono talvolta chiamate anche colonne calcolate o colonne virtuali.

Sintassi

```
column_name data_type [GENERATED ALWAYS] AS ( <expression> )
[VIRTUAL | STORED] [NOT NULL | NULL] [UNIQUE [KEY]] [PRIMARY KEY] [COMMENT <text>]
```

Esempio sulla tabella *studente*. Creiamo un campo generato al volo che contenga il nome e il cognome dello studente:

```
ALTER TABLE studente ADD fullName VARCHAR(255) AS (CONCAT(nome, ' ', cognome))
AFTER cognome;
```

Esempio sulla tabella *libro*. Creiamo un campo generato al volo che calcoli il prezzo del libro compreso di IVA:

```
ALTER TABLE libro
ADD prezzo_iva DECIMAL(6,2) GENERATED ALWAYS AS (prezzo * 1.10) STORED
AFTER prezzo;
```

approfondimenti: <https://dev.mysql.com/doc/refman/8.0/en/create-table-generated-columns.html>

Limiti nella scrittura dell'espressione

- Sono consentiti valori letterali, funzioni integrate deterministiche e operatori. Una funzione è deterministica se, dati gli stessi dati nelle tabelle, più invocazioni producono lo stesso risultato, indipendentemente dall'utente connesso.
Esempi di funzioni che non sono deterministiche e non soddisfano questa definizione:
`CONNECTION_ID()`, `CURRENT_USER()`, `NOW()`, `CURDATE()`...
- Non sono consentite subquery.
- Una definizione di colonna generata può fare riferimento ad altre colonne generate, ma solo a quelle presenti in precedenza nella definizione di tabella.
Una definizione di colonna generata può fare riferimento a qualsiasi colonna di base (non generata) nella tabella indipendentemente dal fatto che la sua definizione sia precedente o successiva.
- L'attributo `AUTO_INCREMENT` non può essere utilizzato in una definizione di colonna generata.
- Non è possibile utilizzare una colonna `AUTO_INCREMENT` come colonna di base in una definizione di colonna generata.
- Se la valutazione dell'espressione provoca il troncamento o fornisce un input errato a una funzione, l'istruzione `CREATE TABLE` termina con un errore e l'operazione DDL viene rifiutata.

Limiti nella scrittura dell'espressione

La maggior parte delle espressioni deterministiche legali che possono essere calcolate sono supportate nelle espressioni per le colonne generate.

La maggior parte delle funzioni integrate è supportata nelle espressioni per le colonne generate.

Tuttavia, alcune funzioni integrate non possono essere supportate per motivi tecnici. Ad esempio, se si tenta di utilizzare una funzione non supportata in un'espressione, viene generato un errore simile al seguente:

```
ERROR 1901 (HY000): Function or expression 'dayname()' cannot be used in the GENERATED  
ALWAYS AS clause of `v`
```

Vediamo un esempio:

```
ALTER TABLE studenti ADD eta TINYINT AS (TIMESTAMPDIFF(YEAR,data_nascita,CURDATE()))  
VIRTUAL;
```

Non si può usare l'espressione, perché utilizza la funzione **CURDATE()** non deterministica

Interrogare più tabelle

STUDENTI

• id	cognome	nome	email
1	Esposito	Giuseppe	esposito.g@icloud.com
2	Pautasso	Carlo	cpautasso@gmail.com
3	Piras	Efisio	efpiras@gmail.com
4	Conti	Renzo	conti.renzo@icloud.com
5	Rigon	Paolo	prigon@gmail.com

ISCRIZIONI

corso_id	studente_id	prezzo	data
1	1	180.00	22/9/2022
1	2	180.00	23/9/2022
3	1	300.00	25/9/2022
2	3	120.00	28/9/2022

DOCENTI

• id	cognome	nome	email
1	Rossi	Mario	rossi.mario@icloud.com
2	Verdi	Paola	verdi.paola@gmail.com
3	Bruni	Marco	bruni.marco@gmail.com
4	Bianchi	Daniele	bianchi.daniele@icloud.com

CORSI

• id	titolo	prezzo	docente_id
1	Php	200.00	4
2	Javascript	120.00	2
3	Java	300.00	3

Legame logico tra tabelle

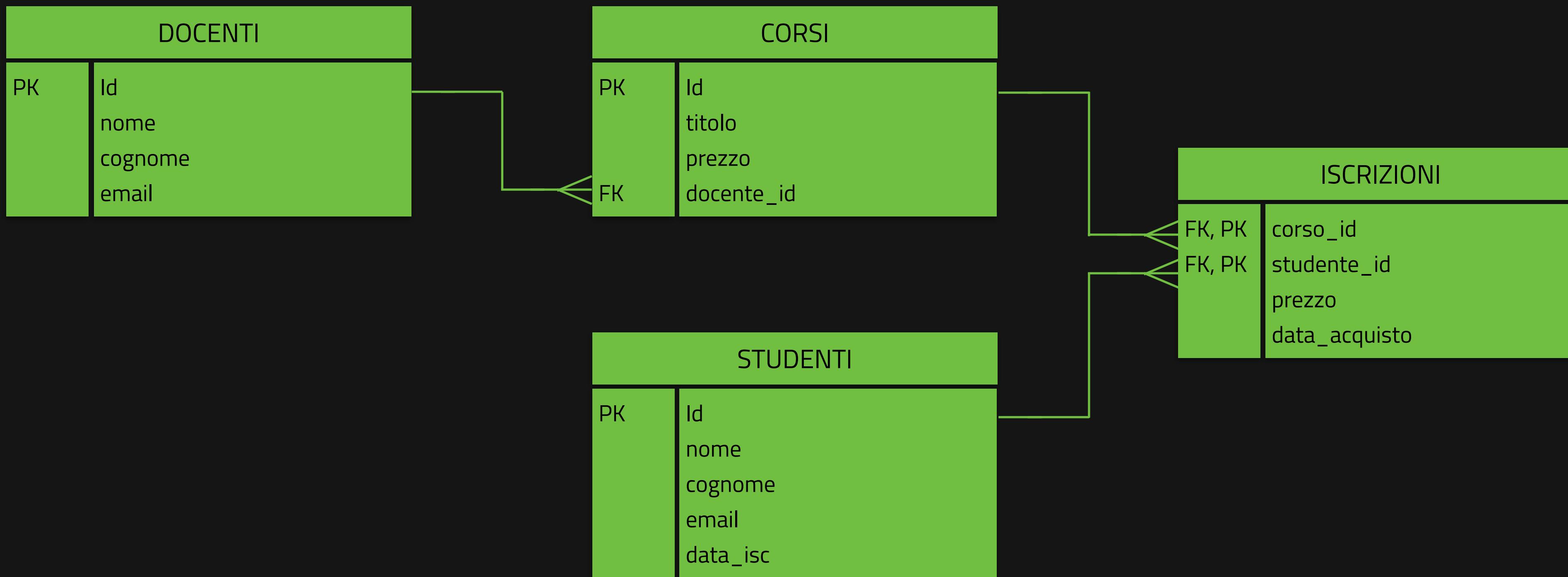
CORSI				DOCENTI			
				<i>Chiave Esterna</i> <i>Campo Chiave</i> ● ●			
● id	titolo	prezzo	docente_id	● id	cognome	nome	email
1	Php	200.00	4	1	Rossi	Mario	rossi.mario@icloud.com
2	Javascript	120.00	2	2	Verdi	Paola	verdi.paola@gmail.com
3	Java	300.00	3	3	Bruni	Marco	bruni.marco@gmail.com
				4	Bianchi	Daniele	bianchi.daniele@icloud.com

Join tra tabelle

● id	titolo	prezzo	docente_id	● id	cognome	nome	email
1	Php	200.00	4	4	Bianchi	Daniele	bianchi.daniele@icloud.com
2	Javascript	120.00	2	2	Verdi	Paola	verdi.paola@gmail.com
3	Java	300.00	3	3	Bruni	Marco	bruni.marco@gmail.com

Esercizio: creare le tabelle relative al database "Corsi"

Allo schema fin qui disegnato manca la definizione dei domini per ogni attributo,
bisogna cioè decidere il *data type*, eventuali *opzioni*, *gli indici*...



Riguardo alla tabella studenti possiamo partire da quella già creata, in modo da avere una anagrafica più completa (con più attributi rispetto allo schema qui sopra). Dobbiamo avere solo cura di controllare e adeguare la struttura a questo database.

Vediamo ora come interrogare più tabelle (*database corsi*):

Immaginiamo di voler estrarre l'elenco dei corsi e relativo docente assegnato.

Le informazioni richieste si trovano in due tabelle: *docenti* e *corsi*.

Le due tabelle sono in relazione *uno a molti* (un docente insegna in più corsi, un corso è insegnato da un solo docente).

La tabella *corsi* contiene l'attributo *docente_id* (chiave esterna) che la lega alla tabella *docenti*.

In *docente_id* viene infatti memorizzato l' *id* corrispondente al docente che insegna quel corso.

Vediamo la query:

```
SELECT nome, cognome, email, titolo  
FROM docenti, corsi  
WHERE docenti.id = corsi.docente_id;
```

Attraverso l'istruzione WHERE seleziono solo i record che hanno corrispondenza, quindi solo i corsi per i quali vi è associato un docente

Se volessimo estrarre l'elenco degli studenti e relativo corso a cui si sono iscritti dobbiamo interrogare tre tabelle per recuperare le informazioni necessarie:

il titolo si trova nella tabella *corsi*, il nome e il cognome degli studenti nella tabella *studenti*, l'iscrizione dello studente ad uno specifico corso nella tabella *iscrizioni* (associazione molti a molti).

Vediamo la query:

```
SELECT nome, cognome, email, titolo  
FROM studenti, iscrizioni, corsi  
WHERE studenti.id = iscrizioni.studente_id  
AND corsi.id = iscrizioni.corso_id;
```

Attraverso l'istruzione WHERE seleziono solo i record che hanno corrispondenza per le condizioni indicate. L'operatore AND assicura che entrambe siano vere.

Vogliamo estrarre l'elenco completo delle informazioni sui corsi (*database corsi*).

Le informazioni richieste si trovano in quattro tavole:

studenti, docenti, corsi e iscrizioni.

Vediamo la query:

```
SELECT studenti.nome, studenti.cognome, studenti.email, titolo, docenti.cognome,  
docenti.nome  
FROM studenti, iscrizioni, corsi, docenti  
WHERE studenti.id = iscrizioni.studente_id  
AND corsi.id = iscrizioni.corso_id  
AND docenti.id = corsi.docente_id;
```

Verranno estratti solo ed esclusivamente i valori che hanno una corrispondenza su tutte le tavole.

Uso degli alias

Gli alias sono utilizzati per rinominare temporaneamente una tabella o un'intestazione di campo.

Si usa l'istruzione **AS** (opzionale) quando si crea un alias¹.

- **Alias per le colonne:** è possibile utilizzare l'alias con GROUP BY, ORDER BY o HAVING per riferirsi alla colonna (vedremo in seguito l'uso di GROUP BY e HAVING).

```
SELECT data_nascita AS `Data di nascita`
FROM studente;
```

```
SELECT (campo1 - campo2) AS risultato
FROM nome_tabella
[WHERE condizione];
```

nota: AS si può anche omettere; SELECT data_nascita 'Data di nascita'..

¹⁾ se il nome alias è un nome composto (es: `Data di nascita`) o ha lettera accentata (es: `Età`) usate i backtick ` (apice retroverso) per wrappare il testo così da non avere risultati inattesi.

Approfondimento: <https://dev.mysql.com/doc/refman/8.0/en/problems-with-alias.html>

○ Alias per le tabelle

Gli **alias per le tabelle** permettono di abbreviare i nomi delle tabelle, rendendo le query più concise e leggibili.

```
SELECT d.nome, d.cognome, d.email, c.titolo AS `Titolo Corso`  
FROM docenti AS d, corsi AS c  
WHERE d.id = c.docente_id  
ORDER BY `Titolo Corso`;
```

✖ Una volta assegnato un alias a una tabella, in tutta la query deve essere usato solo l'alias, non il nome originale della tabella.

Per migliorare la leggibilità, è consigliato sempre specificare l'alias della tabella davanti ai nomi delle colonne, anche quando non ci sono ambiguità.

Nel caso di colonne con lo stesso nome in più tabelle, l'uso dell'alias scelto (se non definito uso del nome originario della tabella) è obbligatorio per evitare errori e specificare chiaramente da quale tabella proviene il dato.

Vediamo la query che estrae l'elenco completo delle informazioni sui corsi con l'uso degli *alias*.

Le informazioni richieste si trovano in quattro tabelle:

studenti, docenti, corsi e iscrizioni.

Vediamo la query:

```
SELECT
    s.nome `Nome studente`,
    s.cognome `Cognome studente`,
    s.email `Contatto studente`,
    c.titolo `Iscritto a`,
    d.cognome `Cognome docente`,
    d.nome `Nome docente`

FROM studenti s, corsi c, iscrizioni i, docenti d

WHERE s.id = i.studente_id
AND c.id = i.corso_id
AND d.id = c.docente_id;
```

Integrità referenziale

FOREIGN KEY: gestire le relazioni tra tabelle

L'integrità referenziale è un insieme di regole che garantiscono la validità delle relazioni tra i record di tabelle correlate. Serve a prevenire la modifica o l'eliminazione accidentale di dati collegati.

Le chiavi esterne (Foreign Key) possono essere utilizzate solo se:

- Entrambe le tabelle utilizzano il motore InnoDB.
- Il campo della tabella primaria è una chiave primaria o UNIQUE (MySQL richiede che il campo referenziato sia indicizzato e unico).
- I campi correlati hanno lo stesso tipo di dati, signedness, lunghezza e collation se la chiave è stringa.
- I campi coinvolti nella relazione hanno un indice (ad es. PRIMARY KEY, UNIQUE, o INDEX).
- Non si utilizzano campi di tipo BLOB o TEXT nelle chiavi coinvolte.

Le Foreign Key permettono di definire il comportamento del database quando si tenta di eliminare, inserire o modificare un record nella tabella primaria che è collegato a uno o più record nella tabella secondaria.

database "Corsi" - integrità dei dati

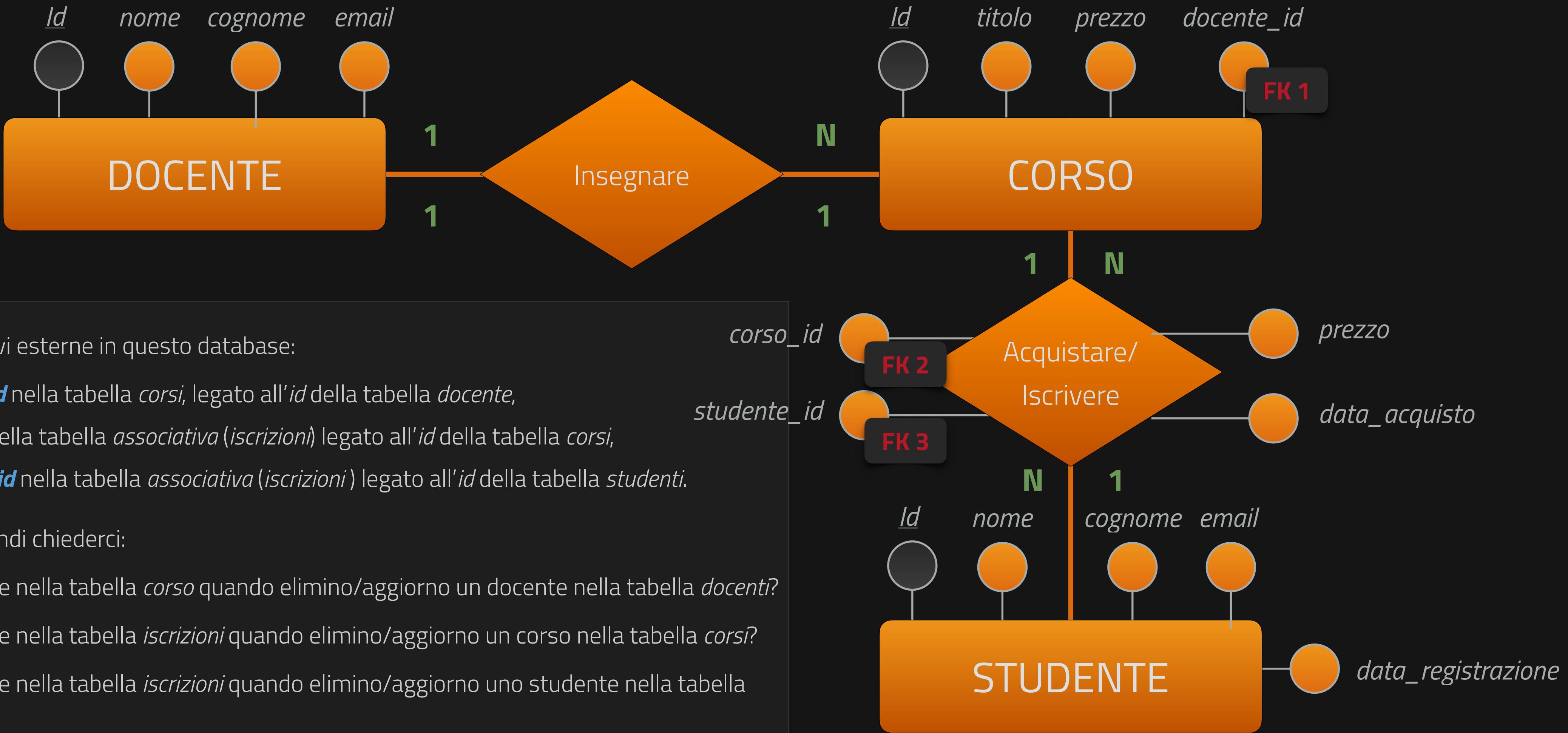
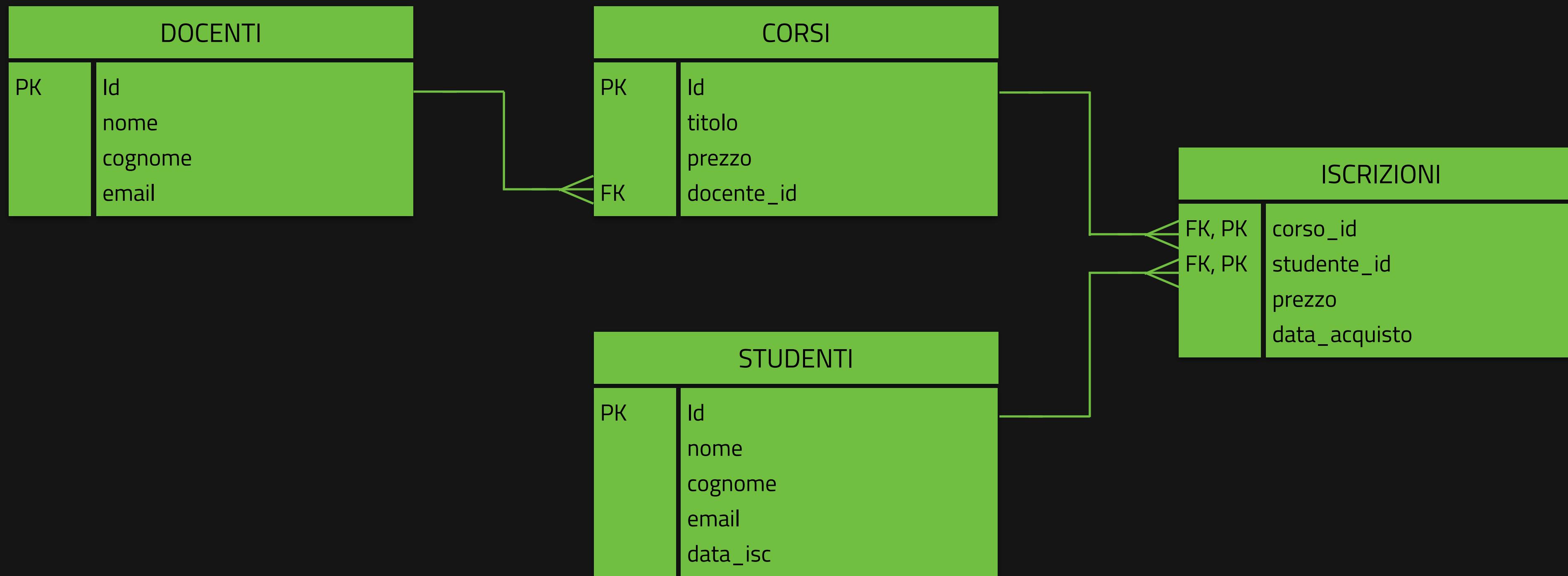


tabelle relative al database "Corsi"



Possiamo definire alcune azioni diverse da attivare in caso di cancellazione o modifica:

CASCADE

In questo caso la cancellazione o modifica di un record nella tabella primaria genererà la cancellazione o la modifica dei record collegati nella tabella secondaria.

SET NULL

In caso di eliminazione o modifica di un record nella tabella primaria i record collegati della tabella secondaria verranno modificati impostando il valore del campo chiave esterna = NULL.

Questa azione è attivabile solo se il campo interessato della tabella secondaria non è impostato a NOT NULL (non deve essere *required*).

RESTRICT o NO ACTION (In MySQL si comportano allo stesso modo)

Queste due azioni (alternative) impediscono direttamente la modifica o la cancellazione dei record della tabella primaria. Praticamente specificare una di queste due azioni equivale a non eseguire alcuna azione.

Valore di default in assenza di indicazioni diverse quando si costruisce il vincolo di chiave esterna.

In SQL STANDARD, RESTRICT e NO ACTION non sono equivalenti: RESTRICT (immediate constraint check). NO ACTION(deferred constraint check)

Vediamo un esempio. Creiamo un vincolo tra le due tabelle (*docenti*, *corsi*) per mezzo dei campi *docente.id* e *corsi.docente_id*

```
CREATE TABLE docenti (
    id INT AUTO_INCREMENT,
    nome VARCHAR(30),
    cognome VARCHAR(50),
    email VARCHAR(100),
    PRIMARY KEY(id));

CREATE TABLE corsi (
    id INT AUTO_INCREMENT,
    titolo VARCHAR(100),
    prezzo DECIMAL(6,2),
    docente_id INT,
    PRIMARY KEY (id),
    INDEX docente_key(docente_id), -- se non inserito viene aggiunto dal motore mysql
    CONSTRAINT fk_corsi_docenti -- diamo un nome alla FOREIGN kEY (opzionale, se non lo specifichiamo viene assegnato dal motore)
        FOREIGN KEY(docente_id) REFERENCES docenti(id)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT); -- possiamo anche omettere le azioni in questo caso perché RESTRICT o NO ACTION sono default
```

In questo modo

- abbiamo creato una chiave esterna nella tabella secondaria *corsi* riferita al campo *id* della tabella primaria *docenti* chiamata *fk_corsi_docenti*;
- abbiamo impostato le azioni da seguire in caso di eliminazione o aggiornamento di un record nella tabella editore.

Nell'esempio abbiamo stabilito che non possiamo eliminare un docente se il suo *id* è presente nella tabella *corsi*.

```
DELETE FROM docenti  
WHERE id = 1;
```

Questa query restituisce l'errore:

```
Cannot delete or update a parent row: a foreign key...
```

Nel caso specifico abbiamo stabilito che non si può eliminare un docente dalla tabella “docenti” se il suo id è presente nella tabella corsi, prima bisogna eliminare le dipendenze nella tabella corsi.

```
DELETE FROM corsi  
WHERE docente_id=1;
```

Ora posiamo eseguire la query precedente:

```
DELETE FROM docenti  
WHERE id = 1;
```

Se volessimo modificare la FOREIGN KEY dovremmo prima eliminarla:

```
ALTER TABLE corsi  
DROP FOREIGN KEY fk_corsi_docenti;
```

e poi ricrearla da capo con le nuove regole:

```
ALTER TABLE corsi ADD CONSTRAINT fk_corsi_docenti  
FOREIGN KEY(docente_id) REFERENCES docenti(id)  
ON DELETE CASCADE ON UPDATE CASCADE;
```

In questo modo abbiamo ricreato la chiave esterna con le nuove azioni da seguire in caso di eliminazione o aggiornamento di un record nella tabella *docenti*.

Ora la query seguente elimina a cascata tutti i record associati nella tabella figlia (*corsi*)

```
DELETE FROM docenti  
WHERE id = 1;
```

```
ALTER TABLE corsi  
DROP FOREIGN KEY fk_corsi_docenti;
```

nuove regole:

```
ALTER TABLE corsi ADD CONSTRAINT fk_corsi_docenti  
FOREIGN KEY(docente_id) REFERENCES docenti(id)  
ON DELETE SET NULL ON UPDATE CASCADE;
```

Nel caso specifico abbiamo stabilito che eliminando un docente dalla tabella “docenti”, impostiamo a *NULL* le dipendenze (campo *docente_id*) nella tabella *corsi*.

```
DELETE FROM docenti  
WHERE id = 1;
```

Questa query viene eseguita; contemporaneamente vengono aggiornate le righe con il campo *docente_id* = 1 della tabella *corsi*, impostando il valore del campo *docente_id* a *NULL*.

Possiamo definire anche una chiave esterna *riferita alla stessa tabella*.

Possiamo definire cioè una SELF-FOREIGN KEY.

Consideriamo come esempio una ipotetica tabella "impiegati" dove per ciascun impiegato registro anche l'attributo identificativo del suo impiegato responsabile:

```
CREATE TABLE IF NOT EXISTS impiegati (
    id int auto_increment,
    nome varchar(50),
    cognome varchar(50),
    ruolo varchar(50),
    responsabile_id int,
    stipendio decimal(6,2),
    FOREIGN KEY (responsabile_id) REFERENCES impiegati(id)
    ON DELETE SET NULL,
    PRIMARY KEY(id)
);
```

In questo caso il campo *responsabile_id* è chiave esterna riferita all' *id* di ciascun impiegato.

Controllo delle FOREIGN KEY in fase di INSERT

Quando una tabella contiene una chiave esterna (FOREIGN KEY), MySQL verifica che i valori inseriti rispettino i vincoli di integrità referenziale, anche durante un'operazione di [INSERT](#).

Non è possibile inserire un record nella tabella *corsi* specificando un valore per *docente_id* che non esiste nella tabella *docenti*.

Supponiamo che la tabella *docenti* contenga i seguenti ID: 1, 3, 4, 6, 7, 8, 10:

```
INSERT INTO Corsi(titolo, prezzo, docente_id)  
VALUES ('JQuery', 100.00, 12);
```

MySQL blocca l'operazione perché il valore 12 per *docente_id* non esiste nella tabella *docenti*, violando così il vincolo di chiave esterna.

Visualizzare la definizione della chiave esterna di una tabella:

```
SHOW CREATE TABLE nome_tabella;
```

Visualizzare tutte le chiavi esterne presenti in un database:

```
SELECT TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME, REFERENCED_TABLE_NAME, REFERENCED_COLUMN_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE TABLE_SCHEMA = 'nome_db'
/* AND TABLE_NAME = 'nome_tabella' */
AND referenced_column_name IS NOT NULL;
```

Visualizzare tutte le informazioni sui vincoli di chiave esterna di un database:

```
SELECT * FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
WHERE CONSTRAINT_SCHEMA = 'nome_db';
```

Questa vista può sostituire la query precedente basata su INFORMATION_SCHEMA.KEY_COLUMN_USAGE quando l'obiettivo è conoscere esattamente:

- quali Foreign Key esistono
- come sono configurate
- quali azioni eseguono in caso di UPDATE/DELETE

La query su KEY_COLUMN_USAGE è utile per vedere quali colonne partecipano alla relazione, ma non contiene le informazioni su CASCADE, SET NULL, RESTRICT ecc.

Disabilitare temporaneamente* i controlli sulle chiavi esterne attraverso FOREIGN_KEY_CHECKS che è una variabile di sistema:

```
SET FOREIGN_KEY_CHECKS = 0;
```

Ricordatevi di ripristinarle dopo un eventuale inserimento massiccio di record

```
SET FOREIGN_KEY_CHECKS = 1;
```

* disabilitare le chiavi esterne può essere utile durante il popolamento massivo del database, quando i dati vengono caricati senza rispettare l'ordine logico delle dipendenze tra tabelle:
es: con chiave esterna attiva non posso caricare record dei corsi prima di caricare i record dei docenti.

ATTENZIONE: NON è raccomandato in produzione!

CHECK (e altri vincoli)

I vincoli (CONSTRAINTS) impongono regole sui dati che possono essere inseriti in una tabella.

Servono a garantire l'integrità dei dati direttamente a livello di database, evitando di delegare tutto il controllo all'applicazione.

Quando un'istruzione viola un vincolo, MySQL genera un errore e impedisce l'esecuzione.

Tipi principali di vincoli:

- PRIMARY KEY

Identifica univocamente ogni riga della tabella. I valori devono essere unici e non nulli.

- FOREIGN KEY

Impone che i valori di una colonna (o gruppo di colonne) corrispondano ai valori di una chiave primaria in un'altra tabella, garantendo l'integrità referenziale.

- UNIQUE

Richiede che i valori in una colonna (o combinazione di colonne) non si ripetano all'interno della tabella.

- CHECK

Valida che i valori inseriti soddisfino una condizione specifica (es: CHECK (età >= 18)).

Se la condizione è falsa, l'inserimento viene bloccato.

CHECK

A partire da MySQL 8.0.16 vengono applicati i *CHECK*.

Prima di MySQL 8.0.16 le espressioni di vincolo erano accettate nella sintassi ma ignorate.

In MySQL puoi definire i vincoli in 2 modi diversi:

1. *[CONSTRAINT [constraint_name]] CHECK (expression)* dato come parte di una definizione di colonna;
2. *[CONSTRAINT [constraint_name]] CHECK (expression)* come vincolo di tabella (può riferirsi a più colonne).

Prima che una riga venga inserita o aggiornata, tutti i vincoli CHECK vengono valutati.

Se un'espressione di vincolo restituisce *false*, la riga non verrà inserita o aggiornata. È possibile utilizzare la maggior parte delle funzioni deterministiche in un vincolo.

Se non dai un nome al vincolo, il vincolo riceverà un nome generato automaticamente.

In modo da poter successivamente eliminare il vincolo con:

```
ALTER TABLE nome_tabella DROP CONSTRAINT nome_vincolo;
```

Uso CHECK alla creazione della tabella

```
CREATE TABLE libri2 (
    id int AUTO_INCREMENT,
    titolo varchar(255),
    prezzo decimal(6,2) NOT NULL,
    pagine smallint unsigned CHECK (pagine > 0) /* CONSTRAINT chk_pagine CHECK (pagine > 0) */,
    editore_id int,
    PRIMARY KEY (id),
    /* CHECK su due colonne */
    CONSTRAINT chk_prezzo_pagine CHECK ((prezzo > 0) AND (pagine > 0))
);
```

Aggiunta CHECK su tabella esistente con nome definito dall'utente:

```
ALTER TABLE studenti
ADD CONSTRAINT ck_eta CHECK(eta >= 18);
```

```
ALTER TABLE libri
ADD CONSTRAINT ck_prezzo CHECK(prezzo > 0);
```

Aggiunta CHECK su tabella esistente con nome definito dal motore ([nome_tabella]_chk_[numero sequenziale 1,2,...])

```
ALTER TABLE studenti
ADD CHECK(eta >= 18);
```

```
ALTER TABLE libri
ADD CHECK(prezzo > 0);
```

Visualizzare le CONSTRAINT CHECK (nome e vincolo) di una tabella con:

SHOW CREATE TABLE *nome_tabella*

```
SHOW CREATE TABLE libri2;
```

Visualizzare elenco CONSTRAINT CHECK di una tabella interrogando information_schema:

```
SELECT *
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
WHERE CONSTRAINT_TYPE = 'CHECK'
AND TABLE_NAME = 'libri';
```

Visualizzare elenco CONSTRAINT CHECK con definizione del vincolo di un database interrogando information_schema:

```
SELECT cc.CONSTRAINT_NAME, cc.CHECK_CLAUSE, tc.TABLE_NAME
FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS cc
JOIN INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
ON cc.CONSTRAINT_NAME = tc.CONSTRAINT_NAME
WHERE tc.CONSTRAINT_TYPE = 'CHECK'
AND tc.CONSTRAINT_SCHEMA = 'nome_database';
```

Combinare risultati: UNION, INTERSECT e EXCEPT¹

Le operazioni UNION, INTERSECT e EXCEPT combinano insiemi di risultati provenienti da due o più query separate.

- UNION unisce righe di più query in un unico risultato.
- INTERSECT restituisce solo le righe comuni tra query.
- EXCEPT restituisce solo le righe della prima query non presenti nella seconda.

Queste operazioni combinano risultati di query separate.

¹ ATTENZIONE: MySQL supporta gli operatori INTERSECT ed EXCEPT a partire da MySQL 8.0.31

L'operatore di MySQL **UNION** consente di combinare due o più set di risultati da più tabelle in un singolo set di risultati. La sintassi è la seguente:

```
SELECT COLUMN1, COLUMN2 FROM table1
UNION [DISTINCT | ALL]
SELECT COLUMN1, COLUMN2 FROM table2
UNION [DISTINCT | ALL]
SELECT COLUMN1, COLUMN2 FROM table3;
```

Affinché una **UNION** funzioni correttamente è necessario che:

- il *numero delle colonne selezionate sia uguale* in ciascuna delle query che si desidera unire;

Per impostazione predefinita, l'operatore **UNION** elimina le righe duplicate dal risultato anche se non si utilizza esplicitamente l'operatore **DISTINCT**.

Pertanto, si dice che la clausola **UNION** è la scorciatoia di **UNION DISTINCT**.

Se si utilizza **UNION ALL** esplicitamente, le righe duplicate, se disponibili, vengono mostrate. **UNION ALL** si esegue più velocemente di **UNION DISTINCT**.

Vediamo un esempio:

```
SELECT stato, capitale FROM europa
UNION
SELECT stato, capitale FROM africa
UNION
SELECT stato, capitale FROM americhe
ORDER BY stato;
```

Otteniamo l'elenco degli stati e delle capitali dei tre continenti

Altro esempio:

```
SELECT *, 'parente' FROM parenti
UNION ALL
SELECT *, 'amico' FROM amici
ORDER BY cognome, nome;
```

Consideriamo le possibili generazioni degli studenti:

- Generazione X (dal 1965 al 1980)
- Millenials (dal 1981 al 1996)
- Generazione Z (dal 1997 al 2012)

Scriviamo una query che ci mostri per ogni studente la generazione di appartenenza e con l'operatore UNION creiamo un unico result set:

```
SELECT cognome, data_nascita, 'X' Generazione
FROM studenti
WHERE data_nascita <= '1980-12-31'

UNION ALL

SELECT cognome, data_nascita, 'Millenials' Generazione
FROM studenti
WHERE data_nascita BETWEEN '1981-01-01' AND '1996-12-31'

UNION ALL

SELECT cognome, data_nascita, 'Z' Generazione
FROM studenti
WHERE data_nascita >= '1997-01-01'

ORDER BY data_nascita;
```

otteniamo l'elenco degli studenti divisi per generazione

INTERSECT

La query restituisce *solo* i valori comuni alle due tabelle:

```
SELECT nome, cognome FROM amici  
INTERSECT -- [DISTINCT | ALL]  
SELECT nome, cognome FROM parenti;
```

EXCEPT

La query restituisce i valori della prima tabella che non sono presenti nella seconda:

```
SELECT nome, cognome FROM amici  
EXCEPT -- [DISTINCT | ALL]  
SELECT nome, cognome FROM parenti;
```

Per impostazione predefinita, l'operatore `INTERSECT` e `EXCEPT` elimina le righe duplicate dal risultato anche se non si utilizza esplicitamente l'operatore `DISTINCT`.

Se si utilizza `INTERSECT ALL` o `EXCEPT ALL` esplicitamente, le righe duplicate, se disponibili, vengono mostrate.

`INTERSECT ALL` e `EXCEPT ALL` si eseguono più velocemente di `INTERSECT DISTINCT` e `EXCEPT DISTINCT`.

```
TABLE table1
UNION [DISTINCT | ALL]
SELECT COLUMN1, COLUMN2 FROM table2
UNION [DISTINCT | ALL]
TABLE table3;
```

Al posto della `SELECT` si può usare anche l'istruzione `TABLE` o un mix delle istruzioni¹:

```
TABLE europa
UNION ALL
SELECT id, stato, capitale FROM africa
UNION ALL
TABLE americhe
ORDER BY stato;
```

¹ Vale anche per `INTERSECT` e `EXCEPT`. Per la sintassi completa vedere la documentazione:
<https://dev.mysql.com/doc/refman/8.4/en/set-operations.html>

EQUIJOIN o JOIN semplice

```
SELECT corsi.id, titolo, prezzo, docente_id, docenti.id, nome, email  
FROM docenti, corsi  
WHERE docenti.id = corsi.docente_id;
```

La query per mostrare l'elenco dei corsi e relativo docente è un esempio di JOIN semplice.

La relazione tra le tabelle è data dall'uguaglianza della *chiave* della tabella padre e della *chiave esterna* della tabella figlia.

La query restituisce *solo* i valori corrispondenti. Se una riga non soddisfa la condizione non verrà mostrata:

• id	titolo	prezzo	docente_id	• id	cognome	nome	email
1	Php	200.00	4	4	Bianchi	Daniele	bianchi.daniele@icloud.com
2	Javascript	120.00	2	2	Verdi	Paola	verdi.paola@gmail.com
3	Java	300.00	3	3	Bruni	Marco	bruni.marco@gmail.com

Unire righe di tabelle: sintassi JOIN

Le JOIN permettono di combinare righe di due o più tabelle basandosi su chiavi comuni o altre condizioni.

INNER JOIN → restituisce solo le righe che hanno corrispondenza in tutte le tabelle.

LEFT / RIGHT JOIN → restituisce tutte le righe di una tabella anche se non ci sono corrispondenze nella tabella collegata.

CROSS JOIN / SELF JOIN / non-equi JOIN → altri modi di combinare righe.

Le JOIN operano riga per riga, a differenza di UNION che lavora su interi set di risultati.

JOIN è un costrutto del linguaggio SQL attraverso il quale vengono messe in relazione due tabelle, ed è alternativo al costrutto precedente che usa WHERE.

Lo scopo di JOIN è quello di unire le tabelle restituendo un risultato combinato sulla base di uno o più campi che trovano corrispondenza in tutte le tabelle coinvolte nella JOIN.

Il collegamento tra le tabelle viene effettuato mediante **INNER JOIN** e la relazione viene stabilita mediante la clausola **ON** che identifica i campi che, nelle tabelle, devono offrire l'egualanza:
verranno estratti, infatti, solo ed esclusivamente i valori che hanno una corrispondenza su tutte le tabelle.

Esempio di sintassi:

```
SELECT tabella1.campo1, tabella2.campo2  
FROM tabella1  
INNER JOIN tabella2 ON tabella1.key = tabella2.key
```

JOIN su più tabelle

Il join fra n tabelle implica un minimo di $n-1$ condizioni di join.

Per esempio, un join tra 3 tabelle implica almeno 2 condizioni di join.

Per le tabelle a, b, c :

Le condizioni di join saranno 2:

```
SELECT *
FROM a, b, c
WHERE a.key=b.key
AND b.key=c.key;
```

con la sintassi JOIN

```
SELECT *
FROM a
[INNER] JOIN b ON a.key=b.key
[INNER] JOIN c ON b.key=c.key;
```

Per esempio per estrarre i corsi e i relativi studenti iscritti le tabelle da interrogare sono tre.

Provate a costruire la query...

INNER JOIN

Vogliamo estrarre l'elenco dei corsi e relativo docente usando il costrutto INNER JOIN (INNER possiamo ometterlo).

Vediamo la query:

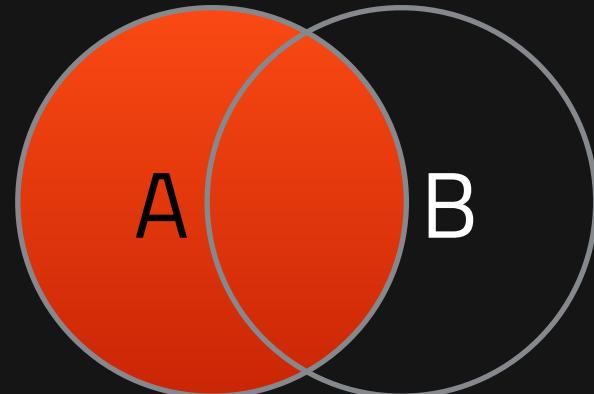
```
SELECT titolo AS Corso, cognome AS Cognome, nome AS Nome, email  
FROM docenti d  
JOIN corsi c  
ON c.docente_id = d.id;
```

Vogliamo estrarre l'elenco degli studenti e relativo corso usando il costrutto INNER JOIN.

```
SELECT cognome AS Cognome, nome AS Nome, email, titolo AS Corso  
FROM studenti s  
JOIN iscrizioni i ON s.id = i.studente_id  
JOIN corsi c ON c.id = i.corso_id;
```

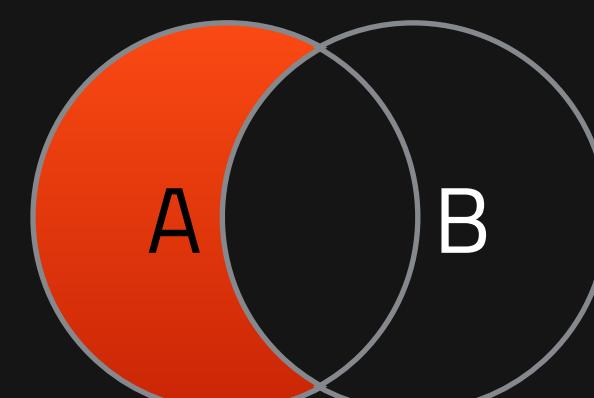
LEFT JOIN

Estrae tutti i valori della tabella a sinistra anche se non hanno corrispondenza nella tabella a destra



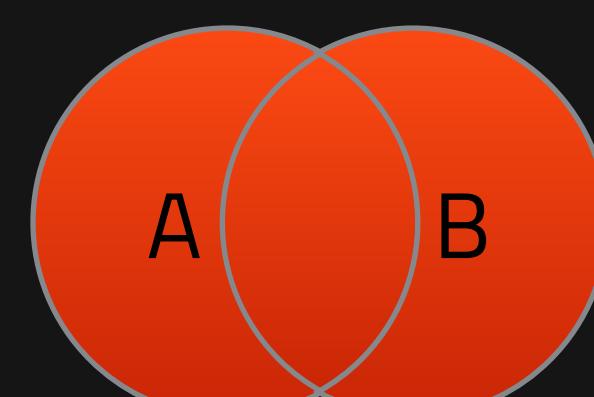
```
SELECT list
FROM tableA
LEFT JOIN tableB
ON A.key=B.key
```

Estrae solo i valori della tabella a sinistra che non hanno corrispondenza nella tabella a destra



```
SELECT list
FROM tableA
LEFT JOIN tableB
ON A.key=B.key
WHERE B.key
IS NULL
```

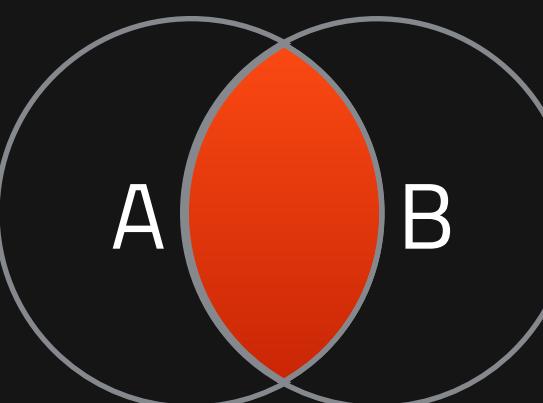
Estrae sia valori che hanno una corrispondenza su tutte le tabelle sia quelli che non ce l'hanno



```
SELECT list
FROM tableA
FULL OUTER JOIN tableB
ON A.key=B.key
```

INNER JOIN

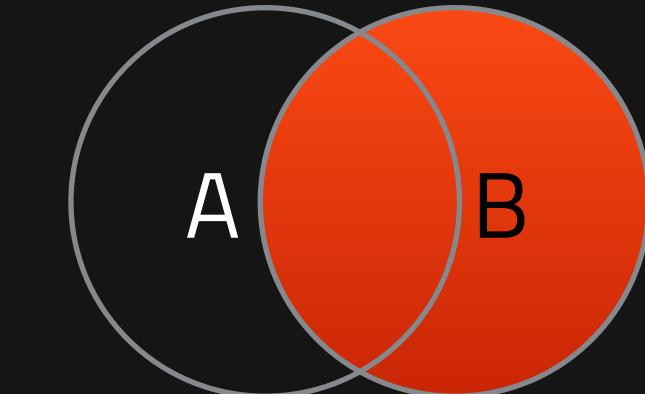
Estrae esclusivamente i valori che hanno una corrispondenza su tutte le tabelle



```
SELECT list
FROM tableA
INNER JOIN tableB
ON A.key=B.key
```

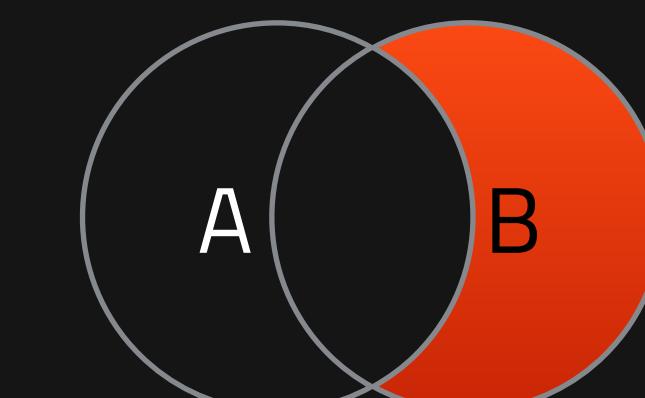
RIGHT JOIN

Estrae tutti i valori della tabella a destra anche se non hanno corrispondenza nella tabella a sinistra



```
SELECT list
FROM tableA
RIGHT JOIN tableB
ON A.key=B.key
```

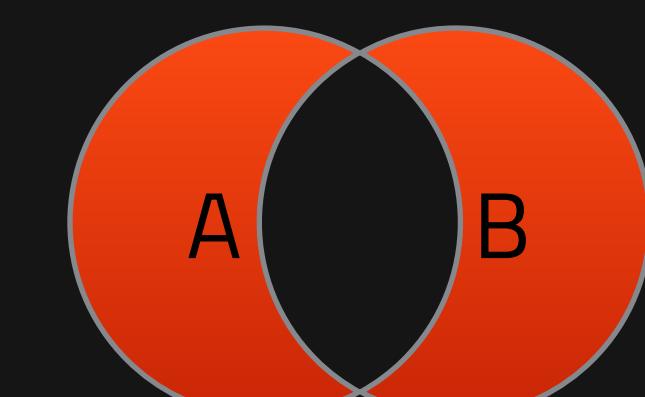
Estrae solo i valori della tabella a destra che non hanno corrispondenza nella tabella a sinistra



```
SELECT list
FROM tableA
RIGHT JOIN tableB
ON A.key=B.key
WHERE A.key
IS NULL
```

FULL OUTER JOIN

Estrae esclusivamente i valori che non hanno una corrispondenza su tutte le tabelle



```
SELECT list
FROM tableA
FULL OUTER JOIN tableB
ON A.key=B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

(Diagramma di Venn)

OUTER JOIN

A differenza delle INNER JOIN, le **OUTER JOIN** selezionano i risultati anche in assenza di una corrispondenza su entrambe le tabelle.

Più precisamente è possibile definire in MySQL due tipi di OUTER JOIN, cioè:

- **LEFT JOIN**: estrae tutti i valori della tabella a sinistra anche se non hanno corrispondenza nella tabella a destra;
- **RIGHT JOIN**: estrae tutti i valori della tabella a destra anche se non hanno corrispondenza nella tabella di sinistra.

Tabella di *sinistra* e tabella di *destra* si riferiscono all'ordine con cui compaiono le tabelle nella query.

OUTER JOIN

Vogliamo estrarre l'elenco dei docenti considerando anche quelli a cui non è stato associato un corso.

Usiamo **LEFT JOIN** per estrarre anche i record dalla tabella di sinistra (docenti -la prima che compare nella query) che non hanno corrispondenza con la tabella di destra (corsi).

Vediamo la query:

```
SELECT cognome, nome, email, titolo AS corso  
FROM docenti d  
LEFT JOIN corsi c  
ON d.id = c.docente_id;
```

Usando **RIGHT JOIN** otteniamo anche i record della tabella di destra(corsi) che non hanno corrispondenza con la tabella di sinistra (docenti).

Vediamo la query:

```
SELECT titolo AS corso, cognome, nome, email  
FROM docenti d  
RIGHT JOIN corsi c  
ON d.id = c.docente_id;
```

Vogliamo estrarre solo l'elenco dei docenti a cui non è stato assegnato alcun corso.

Usiamo sempre LEFT JOIN per estrarre i record dalla tabella di sinistra (docenti- la prima che compare nella query) che non hanno corrispondenza con la tabella di destra.

Tra questi selezioniamo solo quelli per cui il valore della chiave esterna della tabella B risulta NULL (vedi diagramma di Venn).

Vediamo la query:

```
SELECT cognome, nome, email  
FROM docenti d  
LEFT JOIN corsi c  
ON d.id = c.docente_id  
WHERE c.id IS NULL;
```

Vogliamo estrarre l'elenco solo dei corsi a cui non è stato associato alcun docente.

Vediamo la query:

```
SELECT titolo AS corso  
FROM docenti d  
RIGHT JOIN corsi c  
ON d.id = c.docente_id  
WHERE d.id IS NULL;
```

FULL OUTER JOIN

In mysql non sono implementate le *FULL OUTER JOIN*.

Ma con un escamotage si possono ottenere ugualmente:

attraverso l'unione (UNION) di due *OUTER JOIN*: *LEFT* e *RIGHT*.

```
SELECT cognome, nome, email, titolo AS corso
FROM docenti d
LEFT JOIN corsi c
ON d.id = c.docente_id
```

UNION

```
SELECT cognome, nome, email, titolo AS corso
FROM docenti d
RIGHT JOIN corsi c
ON d.id = c.docente_id

ORDER BY corso;
```

FULL OUTER JOIN

solo gli esclusi

```
SELECT cognome, nome, email, titolo AS corso
FROM docenti d
LEFT JOIN corsi c
ON d.id = c.docente_id
WHERE c.id IS NULL

UNION

SELECT cognome, nome, email, titolo AS corso
FROM docenti d
RIGHT JOIN corsi c
ON d.id = c.docente_id
WHERE d.id IS NULL

ORDER BY corso;
```

SELF JOIN

Immaginiamo di avere una tabella "impiegato":

All'interno della tabella registriamo, tra gli attributi (*nome*, *cognome*, *ruolo*, *id_resp*, *stipendio*), anche l'id del responsabile di ciascun impiegato in *id_resp*.

Se volessimo conoscere l'elenco degli impiegati e il loro responsabile possiamo usare una SELF JOIN grazie al meccanismo degli alias.

Vediamo la query:

```
SELECT i.cognome, i.nome, i.ruolo, r.cognome Responsabile
FROM impiegati i
JOIN impiegati r
ON i.id_responsabile = r.id
ORDER BY ruolo;
```

Vediamo la query che comprende anche gli impiegati responsabili:

```
SELECT i.cognome, i.nome, i.ruolo, r.cognome Responsabile
FROM impiegati i
LEFT JOIN impiegati r
ON i.id_responsabile = r.id
ORDER BY ruolo;
```

Scorciatoia JOIN

Se abbiamo usato la stessa label per i *campi chiave* e la *chiave esterna* (es: *docente_id* nella tabella *docente* ed *docente_id* nella tabella *corsi*) possiamo scrivere la join usando l'istruzione **USING**¹ in questo modo:

```
SELECT cognome, nome, email, titolo AS corso  
FROM docenti  
JOIN corsi  
USING (docente_id);
```

¹La clausola nomina un elenco di colonne che devono esistere in entrambe le tavelle.

CROSS JOIN

La CROSS JOIN restituisce il prodotto cartesiano di due tabelle: ogni riga della prima tabella viene combinata con tutte le righe della seconda.

Se abbiamo due tabelle — ad esempio:

- prodotti (elenco dei prodotti)
- colori (colori disponibili)

il risultato avrà $n \times m$ righe.

```
SELECT prodotto, colore
FROM prodotti
CROSS JOIN colori;
```

È possibile ottenere lo stesso risultato scrivendo.

```
SELECT prodotto, colore
FROM prodotti
INNER JOIN colori;
```

Usare CROSS JOIN in modo esplicito rende il codice più chiaro e leggibile, evitando l'ambiguità o l'idea che si sia dimenticata la condizione ON

USO della sintassi JOIN per UPDATE e DELETE

La sintassi JOIN si può usare anche per operazioni di aggiornamento dati.

Nell'esempio seguente riduciamo del 10% il prezzo dei corsi senza studenti iscritti.

```
UPDATE Corsi c
LEFT JOIN Iscrizioni d ON c.id = iscrizioni.corso_id
SET c.prezzo = c.prezzo * 0.90
WHERE iscrizioni.corso_id IS NULL;
```

La sintassi JOIN si può usare anche per operazioni di eliminazione dati.

Nell'esempio seguente eliminiamo i docenti che non hanno corsi assegnati:

```
DELETE d FROM docenti d
LEFT JOIN corsi c
ON d.id = c.docente_id
WHERE c.id IS NULL;
```

Non EQUI-JOIN

Un join non equi-join è un tipo di join che utilizza una condizione diversa dall'uguaglianza (come $<$, \leq , $>$, \geq , \neq , ecc.) per combinare righe da due tabelle.

Vediamo un esempio utilizzando due tabelle: *studenti* e *generazioni*.

La tabella *generazioni* [da creare nel nostro database, campi: id, generazione, anno inizio, anno fine] definisce gli intervalli degli anni che comprendono una specifica generazione:

- Boomers (dal 1946-01-01 al 1964-12-31)
- Generazione X (dal 1965-01-01 al 1980-12-31)
- Millenials (dal 1981-01-01 al 1996-12-31)
- Generazione Z (dal 1997-01-01 al 2012-12-31)

Questa tabella viene definita **tabella di lookup**.

Una tabella di lookup è una tabella che contiene un insieme di valori di riferimento utilizzati per:

- Classificare o categorizzare dati in altre tabelle
- Ridurre la ridondanza (normalizzazione)
- Centralizzare e standardizzare valori ricorrenti
- Rendere le query più chiare e manutenibili

Vogliamo associare ogni studente alla sua generazione basata sul suo anno di nascita.

Questo richiede un non equi-join.

Di seguito due esempi simili ma con operatori diversi (il primo esempio è più leggibile).

```
SELECT s.cognome, s.nome, s.data_nascita, g.generazione
FROM studenti s
JOIN generazioni g
ON s.data_nascita
BETWEEN g.anno_inizio AND g.anno_fine
ORDER BY s.data_nascita;
```

```
SELECT s.cognome, s.nome, s.data_nascita, g.generazione
FROM studenti s
JOIN generazioni g
ON s.data_nascita >= g.anno_inizio
AND s.data_nascita <= g.anno_fine
ORDER BY s.data_nascita;
```

Altro esempio con il database dei Corsi

```
SELECT nome, cognome, data_isc
FROM studenti s
JOIN iscrizioni i ON i.studente_id = s.id
AND i.data_isc >= '2025-03-01';
```

L' **AND** nella condizione di **JOIN** fa sì che il **JOIN** unisca solo le righe che corrispondono alla data di iscrizione indicata, diverso sarebbe scrivere:

```
SELECT nome, cognome, data_isc
FROM studenti s
JOIN iscrizioni i ON i.studente_id = s.id
WHERE i.data_isc >= '2025-03-01';
```

Perché il **JOIN** unisce tutte le righe e poi filtra sulla base del **WHERE**.

Funzioni

Funzioni di aggregazione

Possiamo eseguire calcoli su tutta la colonna utilizzando le funzioni di aggregazione di SQL.

Questo ci permetterà di guardare i dati nel loro insieme e ottenere calcoli come la media, il numero di valori, la somma totale di tutti i valori e altro ancora.

AVG() non considera i valori nulli

La funzione AVG() restituisce il valore medio (media) del l'intero set di dati.

```
SELECT AVG(campo)  
FROM tabella;
```

```
SELECT AVG(prezzo) AS 'prezzo medio'  
FROM corsi;
```

`COUNT(*)` considera i valori nulli, `COUNT[(colonna)]` non considera i valori nulli

La funzione `COUNT()` restituisce il numero di record trovati.

```
SELECT COUNT(*)
FROM tabella;
```

Possiamo anche filtrare il conteggio, vengono contati i record che soddisfano la condizione:

```
SELECT COUNT(*)
FROM tabella
WHERE condizione;
```

Per contare il numero delle studentesse donna:

```
SELECT COUNT(*)
FROM studenti
WHERE genere='f';
```

Per contare il numero degli studenti a cui non è stato assegnato il valore per l'attributo 'genere':

```
SELECT COUNT(*)
FROM studenti
WHERE genere IS NULL;
```

MAX() e MIN() non considera i valori nulli

Le funzioni MAX() e MIN() restituiscono il valore più alto e il valore più basso del campo richiesto nella query.

```
SELECT MAX(campo), MIN(campo)  
FROM tabella;
```

```
SELECT MAX(prezzo) AS 'più caro', MIN(prezzo) AS 'più economico'  
FROM corsi;
```

SUM() non considera i valori nulli

La funzione SUM() restituisce la somma dei valori di un dato campo.

```
SELECT SUM(campo)  
FROM tabella;
```

```
SELECT SUM(prezzo) AS 'Valore corsi'  
FROM corsi;
```

Funzioni matematiche

FLOOR() e CEILING()

Le funzioni FLOOR() e CEILING() arrotondano i numeri dopo la virgola rispettivamente verso l'intero più basso o più alto.

```
SELECT FLOOR(campo), CEILING(campo)
FROM tabella;
```

```
SELECT FLOOR(prezzo), CEILING(prezzo)
FROM libri;
```

ROUND()

La funzione ROUND() rispetta l'arrotondamento matematico.

Accetta un secondo argomento in cui indicare i decimali dopo la virgola.

```
SELECT ROUND(campo)
FROM tabella;
```

```
SELECT ROUND(prezzo,2)
FROM libri;
```

Funzioni sulle stringhe: LENGTH() e CHAR_LENGTH()

CHAR_LENGTH() restituisce il numero di caratteri (visibili), inclusi spazi.

LENGTH() restituisce il numero di byte, che può differire da CHAR_LENGTH() se ci sono caratteri speciali, accenti, emoji o alfabeti non latini.

Con caratteri ASCII semplici (es. solo lettere inglesi), le due funzioni danno lo stesso risultato.

```
SELECT
    nome,
    CHAR_LENGTH(nome) AS `Numero caratteri`,
    LENGTH(nome) AS `Numero byte`
FROM studenti;
```

Funzioni sulle stringhe: CONCAT() e CONCAT_WS()

CONCAT() unisce due o più stringhe in una sola, senza separatori.

CONCAT_WS (separatore, str1, str2, ...) fa lo stesso, aggiungendo un separatore tra gli elementi (WS = With Separator).

```
-- Concatena nome e cognome con uno spazio
SELECT CONCAT(nome, ' ', cognome) AS `Nome completo`
FROM studenti;

-- CONCAT_WS evita di scrivere manualmente lo spazio
SELECT CONCAT_WS(' ', nome, cognome) AS `Nome completo`
FROM studenti;
```

SUBSTRING(str, pos, len) o SUBSTRING(str FROM pos FOR len);

se non specifichiamo "len" vengono presi tutti i caratteri seguenti alla posizione indicata

```
SELECT nome, SUBSTRING(nome, 2, 3)
FROM studenti;
```

LEFT() / RIGHT():

mostra uno specifico numero di caratteri iniziando da sinistra(LEFT) o destra(RIGHT)

```
SELECT nome, LEFT(nome, 1)
FROM studenti;
```

```
SELECT CONCAT(LEFT(nome, 1), ' . ', cognome)
FROM studenti;
```

Uso di funzioni combinate

Lunghezza media in termini di caratteri considerando la stringa *nome + cognome* insieme:

```
SELECT AVG(LENGTH(nome)+LENGTH(cognome))  
FROM studenti;
```

```
SELECT AVG(LENGTH(CONCAT(nome, cognome)))  
FROM studenti;
```

Elimina i numeri dopo la virgola, FLOOR() arrotonda verso il basso, CEILING() verso l'alto;

```
SELECT FLOOR(AVG(LENGTH(CONCAT(nome, cognome))))  
FROM studenti;
```

```
SELECT CEILING(AVG(LENGTH(CONCAT(nome, cognome))))  
FROM studenti;
```

Funzioni informative¹

LAST_INSERT_ID().

LAST_INSERT_ID() restituisce un valore (a 64 bit) BIGINT UNSIGNED che rappresenta il primo valore generato automaticamente inserito correttamente per una colonna AUTO_INCREMENT come risultato dell'ultima istruzione INSERT eseguita.

Il valore di LAST_INSERT_ID() rimane invariato se nessuna riga viene inserita correttamente.

Dopo aver inserito una riga che genera un AUTO_INCREMENT

```
INSERT INTO studenti(cognome, nome, email)
VALUE('rossi','marco','marco.rossi@gmail.com');
```

si può ottenere il valore in questo modo:

```
SELECT LAST_INSERT_ID();
-- 28 valore dell'auto increment inserito (il valore riportato è un esempio)
-- se si inseriscono più record viene sempre restituito il primo id inserito del gruppo
```

1) <https://dev.mysql.com/doc/refman/8.0/en/information-functions.html>

Aggiornamento / sostituzione

Se dobbiamo aggiornare parte del testo di un campo nella nostra tabella possiamo usare una delle funzioni stringa:

REPLACE¹

Vediamo ad esempio come aggiornare le informazioni presenti in un campo modificando solo una parte del suo valore

```
UPDATE studenti  
SET email = REPLACE(email, '.com', '.it');
```

La funzione utilizza tre argomenti racchiusi tra parentesi:

- il primo argomento è il campo di riferimento per la ricerca/sostituzione;
- il secondo argomento, racchiuso tra apici, rappresenta la stringa di testo da sostituire;
- il terzo argomento, racchiuso tra apici, rappresenta la stringa nuova.

Possiamo usarla anche in lettura se vogliamo sostituire al volo una sotto-stringa:

```
SELECT email, REPLACE(email, '.com', '.it')  
FROM studenti;
```

¹⁾ La funzione REPLACE è "case sensitive": *REPLACE* intercetta esattamente la stringa indicata, "Testo" è diverso da "testo".

funzioni data e ora

per impostare le informazioni temporali attuali, si possono usare le seguenti funzioni:

FUNZIONE	DESCRIZIONE
NOW()	restituisce data e ora attuali. Ammette i sinonimi CURRENT_TIMESTAMP() e CURRENT_TIMESTAMP
CURDATE()	restituisce data attuale. Ammette i sinonimi CURRENT_DATE() e CURRENT_DATE
CURTIME()	restituisce orario attuale. Ammette i sinonimi CURRENT_TIME() e CURRENT_TIME

Una volta impostate, le informazioni data/ora possono essere lette, in tutto o in parte, prelevandone solo alcuni elementi. Per queste operazioni, esistono apposite funzioni:

- giorni, mesi e anni: YEAR(), MONTH() e DAY() che, rispettivamente, restituiscono anno, mese e giorno;
- ore, minuti e secondi: HOUR(), MINUTE() e SECOND();
- giorno nella settimana o nell'anno: DAYOFWEEK() (restituisce: da 1 a 7); DAYOFYEAR() (restituisce: da 1 a 365)
- nome del giorno della settimana DAYNAME(), nome del mese MONTHNAME()

```
SELECT YEAR(CURDATE());
SELECT MONTH(CURDATE());
SELECT DAY(CURDATE());
SELECT HOUR(CURTIME());
SELECT MINUTE(CURTIME());
SELECT SECOND(CURTIME());
SELECT DAYNAME(CURDATE());
SELECT DAYNAME('vostra_data_nascita'); #se volete sapere il giorno i cui siete nati
SELECT MONTHNAME(CURDATE());
```

Per avere i nomi in lingua bisogna impostare la lingua italiana (potreste non avere i privilegi):

```
SET lc_time_names = 'it_IT'; #(https://dev.mysql.com/doc/refman/5.7/en/locale-support.html)
```

Per conoscere la lingua utilizzata: `SELECT @@lc_time_names;`

```
SELECT DAYOFWEEK(CURDATE()); #notate il numero restituito pur avendo impostato la lingua italiana
SELECT DAYOFMONTH(CURDATE());
SELECT DAYOFYEAR(CURDATE());
SELECT WEEKOFYEAR(CURDATE());
```

Formattare date e orari

DATE_FORMAT() e TIME_FORMAT().

La funzione DATE_FORMAT() permette di personalizzare il formato di una data usando appositi meta-caratteri:

METACARATTERE	DESCRIZIONE
%d	giorno del mese, %D numerale ordinale del giorno
%m	mese espresso in numero. La variante %M esprime il mese in parole
%Y	l'anno su quattro cifre. La variante %y esprime l'anno su due cifre
%H	indica le ore (da 0 a 24, in alternativa %h le mostra da 0 a 12)
%i	indica i minuti
%s	%S o %s per i secondi.
%p	indica AM o PM

NOTA: si applicano ad altre funzioni: STR_TO_DATE(), TIME_FORMAT(), UNIX_TIMESTAMP()

Elenco completo: https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_date-format

```
SELECT DATE_FORMAT('2017-02-28', '%d/%m/%Y'); #restituisce: 28/02/2017
```

```
SELECT DATE_FORMAT('2017-02-28', '%d/%m/%y'); #restituisce: 28/02/17
```

```
SELECT DATE_FORMAT('2017-02-28', '%d %M %Y'); #restituisce: 28 Febbraio 2017
```

```
SELECT TIME_FORMAT('17:25:34', '%H-%i'); #restituisce: 17-25
```

```
SELECT TIME_FORMAT('17:25:34', '%h:%i %p'); #restituisce: 05:25 PM
```

```
SELECT TIME_FORMAT(CURTIME(), 'sono le %H e %i minuti') AS 'che ore sono?';  
#restituisce: 'sono le 18 e 58 minuti'
```

Se interrogate una tabella la formattazione della data può essere:

```
SELECT nome, cognome, DATE_FORMAT(data_nascita, '%d %M %Y')  
FROM studenti;
```

il formato data impostato è tra apici, è quindi una stringa in cui potete inserire anche del testo: vedi esempio TIME_FORMAT().

STR_TO_DATE (*str, format*)

Prende una stringa (*str*, ad esempio '`01-01-2001`') e un formato (*format*, ad esempio '`%d-%m-%Y`').

`STR_TO_DATE()` restituisce sia un valore DATETIME, se specifichiamo nel formato data e ora, sia un valore DATE o TIME, se specifichiamo solo data o solo ora.

Se la data, l'ora o il valore DATETIME estratti da *str* è illegale, `STR_TO_DATE()` restituisce NULL e genera un *warning*.

Conversione della stringa in data, esempi:

```
INSERT INTO studenti (nome, cognome, email, data_nascita)
VALUES ('marco','allegri','marco.allegri@gmail.com', STR_TO_DATE('05,10,1969','%d,%m,%Y'));
```

```
INSERT INTO studenti (nome, cognome, email, data_nascita)
VALUES ('marco','allegri','marco.allegri@gmail.com', STR_TO_DATE('1 February 2017','%d %M %Y'));
```

```
SELECT STR_TO_DATE(CONCAT_WS(' ','',"05","10","1969"),'%d,%m,%Y');
```

Calcoli con date e orari

Per sommare un periodo di tempo ad una data o un orario si possono usare le funzioni **ADDDATE()** e **ADDTIME()**.

```
SELECT ADDDATE('2017-03-01', INTERVAL 5 DAY); #restituisce: 2017-03-06
```

```
SELECT ADDDATE('2017-03-01', 5); #restituisce 2017-03-06
```

Quando si usa ADDDATE(data, numero), il numero viene interpretato come giorni.

Se si vuole specificare mesi, anni o altri intervalli temporali, è necessario usare INTERVAL.

```
SELECT ADDDATE('2017-03-01', INTERVAL 5 YEAR); restituisce: 2022-03-01
```

Discorso analogo vale per ADDTIME. Ecco direttamente qualche esempio:

```
SELECT ADDTIME('17:25', '05:05'); restituisce: 22:30:00
```

```
SELECT ADDTIME('17:25', '00:05:05'); restituisce: 17:30:05
```

Nel caso di ADDTIME, si può indicare direttamente il lasso di tempo da sommare.

Sottrazione: **SUBDATE()**/**DATE_SUB()** il cui funzionamento è speculare a **ADDDATE()**/**(DATE_ADD)**:

```
SELECT SUBDATE('2015-03-01', INTERVAL 5 DAY); restituisce: 2015-02-24
```

Sottrazione ore/minuti/secondi **SUBTIME()**:

```
SELECT SUBTIME(CURTIME(), '03:03');
```

DATEDIFF().

Questa funzione calcola il numero di giorni che intercorrono tra due date.

Questa funzione accetta due argomenti ed il risultato sarà un numero positivo se la prima data è più recente della seconda, altrimenti il risultato sarà un numero negativo:

```
SELECT DATEDIFF('2017-03-01','2017-02-10');
```

```
SELECT DATEDIFF('2017-01-01','2017-02-10');
```

```
SELECT DATEDIFF(CURDATE(),'2020-04-10');
```

Se volessimo conoscere i giorni trascorsi da una determinata data dovremmo impostare una query tipo:

```
SELECT DATEDIFF(CURDATE(),data) FROM nome_tabella  
WHERE condizioni;
```

TIMESTAMPADD(*unità*, *intervallo*, *espr_datetime*).

Aggiunge l'espressione intera *intervallo* all'argomento *espr_datetime* DATE o DATETIME.

L'argomento *unità* rappresenta l'unità di misura dell'argomento intervallo, che dovrebbe essere uno dei seguenti valori:

MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER o YEAR.

```
SELECT TIMESTAMPADD(MINUTE, 1, '2003-01-02 00:01:00');
```

```
SELECT TIMESTAMPADD(WEEK, 1, '2003-01-02');
```

TIMESTAMPDIFF(*unità*, *espr1*, *espr2*)

Restituisce il risultato di *espr2* - *espr1*, che sono entrambe di tipo DATETIME.

Una espressione potrebbe essere una data e un'altra un datetime; in questo contesto, un valore DATE viene considerato come un DATETIME che ha come orario '00:00:00'.

Il *risultato* è un *intero* e l'*unità* di misura è quella specificata dall'argomento *unità*.

I valori ammessi sono gli stessi che si possono usare per la funzione TIMESTAMPADD().

TIMESTAMPDIFF() può essere utilizzato per calcolare l'età:

```
SELECT TIMESTAMPDIFF(MONTH, '2003-02-01', '2003-05-01');
```

```
SELECT TIMESTAMPDIFF(YEAR, '2002-05-01', '2001-01-01');
```

```
SELECT TIMESTAMPDIFF(MINUTE, '2003-02-01', '2003-05-01 12:05:55');
```

Calcolare età

```
SELECT NOME, COGNOME, TIMESTAMPDIFF(YEAR, data_nascita, CURDATE()) AS `Età`  
FROM studenti;
```

Vediamo come sfruttare questo calcolo in fase di UPDATE e di INSERT.

```
UPDATE studenti  
SET eta = TIMESTAMPDIFF(YEAR, data_nascita, CURDATE());
```

```
INSERT INTO studenti  
(nome, cognome, genere, data_nascita, email, eta)  
VALUES ('paperina', 'duck', 'f', '1999-01-01', 'paperina_d@gmail.com',  
TIMESTAMPDIFF(YEAR, data_nascita, CURDATE()));
```

Anche se, attraverso la registrazione della data di nascita, siamo sempre in grado di ricavare l'età corretta.

Giorni al compleanno

```
SELECT nome, cognome, DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita) AS Giorni  
FROM studenti  
ORDER BY Giorni;
```

Posso aggiungere una condizione per intercettare quelli che ricadono nel prossimo mese:

```
SELECT nome, cognome, data_nascita, DAYOFYEAR(CURDATE())-  
DAYOFYEAR(data_nascita) AS Giorni  
FROM studenti  
WHERE (DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita)) BETWEEN -31 AND 0;
```

Se uso WHERE l'alias della colonna non viene intercettata (vedi slide successiva)

```
SELECT nome, cognome, (DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita)) AS Giorni  
FROM studenti  
WHERE Giorni BETWEEN -31 AND 0  
ORDER BY Giorni;  
/* seleziono quelli a cui al compleanno mancano meno di 31 giorni */
```

HAVING

WHERE viene utilizzato per selezionare i dati nelle tabelle originali in elaborazione.

HAVING viene utilizzato per filtrare i dati nel set di risultati prodotto dalla query.

Ciò significa che HAVING può fare riferimento a valori aggregati (vedere [Raggruppamenti](#)) e alias nella clausola SELECT.

Ricordiamo l'ordine di scrittura delle istruzioni in mysql:

```
SELECT  
FROM  
WHERE  
GROUP BY  
HAVING  
ORDER BY  
LIMIT
```

```
SELECT nome, cognome, data_nascita, (DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita)) AS Giorni  
FROM studenti  
HAVING Giorni BETWEEN -31 AND 0  
ORDER BY Giorni;  
/* seleziono quelli a cui al compleanno mancano meno di 31 giorni */
```

JSON* function

Inserire i dati in formato JSON, per un attributo definito in questo modo, richiede la notazione JSON

```
INSERT INTO articoli(descrizione, specifiche)
VALUES(
    'TV SAMSUNG 32" SMART TV',
    '{
        "marca": "SAMSUNG",
        "pesoKg": "5.12",
        "schermo": "LCD",
        "pollici": 32,
        "uscite": ["HDMI", "USB"]
    }'
);
```

Per gestire il tipo di dati JSON mysql mette a disposizione una serie di funzioni: ne analizzeremo alcune*.

* Approfondimento

** JSON function: [elenco completo](#)

JSON_OBJECT([key, val [, key, val] ...])

La funzione JSON_OBJECT accetta un elenco di coppie chiave/valore JSON_OBJECT(key1, value1, key2, value2, ... key(n), value(n)) e restituisce un oggetto JSON.

Si verifica un errore se il nome di una chiave dispari è NULL o il numero di argomenti è dispari.

```
INSERT INTO articoli(descrizione, specifiche)
VALUES(
    'TV SONY 32" SMART TV',
    JSON_OBJECT(
        "marca", "SONY",
        "pesoKg", 6.5,
        "schermo", "LED",
        "pollici", 32,
        "uscite", "HDMI"
    )
);
```

JSON_ARRAY(([val [, val] ...]))

La funzione JSON_ARRAY accetta in argomento una lista di valori e crea una array in formato JSON.

I valori dell'elenco possono essere di diverso tipo (numeri, stringhe, null e booleani)

```
INSERT INTO articoli(descrizione, specifiche)
VALUES(
    'TV PHILIPS 55" SMART TV',
    JSON_OBJECT(
        "marca", "PHILIPS",
        "pesoKg", 9.5,
        "schermo", "LED",
        "pollici", 55,
        "uscite", JSON_ARRAY('HDMI', 'RCA', 'USB', 'COAXIAL', 'SCART')
    )
);
```

JSON_EXTRACT(json_doc, path[, path] ...)

Restituisce i dati da un documento JSON, selezionati dalle parti del documento corrispondenti agli argomenti *path*.

Restituisce NULL se un argomento è NULL o nessun percorso individua un valore nel documento.

Si verifica un errore se l'argomento *json_doc* non è un documento JSON valido o qualsiasi argomento *path* non è un'espressione di percorso valida.

Il valore restituito è costituito da tutti i valori che corrispondono agli argomenti *path*.

Se è possibile che tali argomenti possano restituire più valori, i valori corrispondenti vengono inseriti automaticamente in un array, nell'ordine corrispondente ai percorsi che li hanno prodotti.

In caso contrario, il valore restituito è il singolo valore corrispondente.

Gli attributi (key) del documento JSON si possono intercettare in base al percorso (path) all'interno del JSON.

```
SELECT JSON_EXTRACT(column, '$.key')
FROM tableName;
```

```
SELECT JSON_EXTRACT(specifiche, '$.uscite')
FROM articoli;
```

Questa funzione ha un alias che ne semplifica la scrittura.

```
SELECT column -> '$.key'
FROM tableName;
```

```
SELECT specifiche -> '$.uscite'
FROM articoli;
```

Vediamo l'uso della funzione con gli array.

```
SELECT JSON_EXTRACT(' [10, 20, [30, 40]] ', '$[1]');--20
```

```
SELECT JSON_EXTRACT(' [10, 20, [30, 40]] ', '$[0]', '$[1]');--[10, 20]
```

```
SELECT JSON_EXTRACT(' [10, 20, [30, 40]] ', '$[2]');--[30, 40]
```

```
SELECT JSON_EXTRACT(' [10, 20, [30, 40]] ', '$[2][1]');--[40]
```

Vediamo l'uso della funzione con gli array nella tabella articolo.

```
SELECT JSON_EXTRACT(specifiche, '$.uscite[2]')
FROM articoli;
```

```
SELECT specifiche -> '$.uscite[2]'
FROM articoli;
```

`JSON_SET(json_doc, path, val[, path, val] ...)`

La funzione `JSON_SET` sostituisce i valori esistenti e aggiunge valori inesistenti.

Restituisce `NULL` se un argomento è `NULL` o se il path fornito non individua un oggetto.

Si verifica un errore se l'argomento `json_doc` non è un documento JSON valido.

```
UPDATE articoli
SET specifiche =
    JSON_SET(specifiche,
              '$.marca', 'LG',
              '$.uscite', JSON_ARRAY('HDMI', 'SCART', 'S/PDIF'),
              '$.ingressi', JSON_ARRAY('ETHERNET', 'USB'))
WHERE id = 1;
```

JSON_INSERT(json_doc, path, val[, path, val] ...)

La funzione JSON_INSERT inserisce i valori senza sostituire i valori esistenti.

```
UPDATE articoli
    SET specifiche = JSON_INSERT(specifiche, '$.uscite[2]', 'RGB')
WHERE id = 1;
-- non produce risultato perché la posizione nell'array è occupata
```

```
UPDATE articoli
    SET specifiche = JSON_INSERT(specifiche, '$.uscite[3]', 'RGB')
WHERE id = 1;
-- aggiunge il nuovo elemento
```

`JSON_REPLACE(json_doc, path, val[, path, val] ...)`

La funzione `JSON_REPLACE` sostituisce solo i valori esistenti.

Se la chiave non esiste il documento non viene modificato.

```
UPDATE articoli
SET specifiche = JSON_REPLACE(specifiche, '$.marca', 'SAMSUNG')
WHERE id = 1;
```

```
UPDATE articoli
SET specifiche = JSON_REPLACE(specifiche, '$.marca', 'SHARP', '$.schermo', 'PLASMA')
WHERE id = 1;
```

Vediamo l'uso della funzione per modificare un elemento di un array.

```
UPDATE articoli
SET specifiche = JSON_REPLACE(specifiche, '$.uscite[1]', 'HDMI2')
WHERE id = 1;
```

JSON_REMOVE(json_doc, path[, path] ...)

La funzione JSON_REMOVE elimina i valori in una colonna di tipo JSON.

Se la proprietà non esiste il documento non viene modificato.

```
UPDATE articoli
  SET specifiche = JSON_REMOVE(specifiche, '$.profondita')
WHERE id = 1; -- non produce risultato perché la proprietà non esiste
```

```
UPDATE articoli
  SET specifiche = JSON_REMOVE(specifiche, '$.uscite[1]')
WHERE id = 1; -- elimina il secondo elemento della proprietà uscite
```

Control Flow Function (funzioni per il controllo del flusso)*:

IF(expr1, expr2, expr3), CASE

Se *expr1* è vera viene usata *expr2*, se *expr1* è falsa viene usata *expr3*

```
SELECT IF ( 1 > 2 , 0 , 1 ); -- 1
```

```
SELECT IF ( 1 < 2 , 'yes' , 'no' ); -- 'yes'
```

Interroghiamo la tabella studente. Se uno studente proviene dalla provincia di Torino lo consideriamo uno studente *in sede*, altrimenti *fuori sede*:

```
SELECT cognome,  
       IF( provincia = 'to' , 'sede' , 'fuori sede' ) sede  
  FROM studenti  
 ORDER BY sede DESC, cognome;
```

*per approfondimento: <https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html>

CASE

```
CASE value WHEN [compare_value]
THEN result [WHEN [compare_value]
THEN result ...]
[ELSE result] END
```

```
CASE WHEN [condition]
THEN result [WHEN [condition]
THEN result ...][ELSE result] END
```

Esempio

```
SELECT
    provincia,
    CASE provincia
    WHEN 'to' THEN 'Torino'
    WHEN 'at' THEN 'Asti'
    WHEN 'no' THEN 'Novara'
    WHEN 'al' THEN 'Alessandria'
    WHEN 'cn' THEN 'Cuneo'
    ELSE 'Vercelli' END 'Provincia completa'
FROM studenti;
```

```
SELECT
    titolo,
    prezzo,
    CASE
        WHEN prezzo < 5 THEN 'economico'
        WHEN prezzo >= 5 AND prezzo <= 10 THEN 'medio'
        WHEN prezzo > 10 THEN 'caro'
    END valutazione
FROM libri;
```

Esempio

```
SELECT
    cognome,
    CASE WHEN genere = 'f' THEN 'Donna' ELSE 'Uomo' END Genere
from studenti;
```

Esempio

Consideriamo la query della slide relativa alla istruzione UNION, utilizzata per suddividere gli studenti sulla base della generazione di appartenenza (**slide**), con la funzione CASE potremmo riscriverla così:

```
SELECT cognome, data_nascita `Data di nascita` ,  
CASE WHEN year(data_nascita) <= 1980 THEN 'X'  
WHEN year(data_nascita) > 1980  
      AND year(data_nascita) < 1997 THEN 'millenials'  
WHEN year(data_nascita) >= 1997 THEN 'Z'  
WHEN year(data_nascita) IS NULL THEN 'Manca data nascita'  
END Generazione  
FROM studenti  
ORDER BY Generazione;
```

Raggruppamenti

Selezionare Valori Unici con DISTINCT in MySQL

Per visualizzare i risultati senza ripetizioni, possiamo usare l'istruzione DISTINCT.

DISTINCT su una singola colonna

Se vogliamo ottenere un elenco di valori unici per una sola colonna, possiamo usare:

```
SELECT DISTINCT cognome  
FROM studenti;
```

Questa query restituisce tutti i cognomi distinti presenti nella tabella studenti.

Se un cognome compare più volte, verrà mostrato solo una volta.

DISTINCT su più colonne

Quando si usa DISTINCT su più colonne, la funzione si applica all'intera combinazione dei valori.

Ad esempio:

```
SELECT DISTINCT cognome, nome  
FROM studenti  
ORDER BY cognome;
```

In questo caso, verranno eliminate solo le righe in cui sia *cognome* sia *nome* sono identici.

Se due studenti hanno lo stesso nome ma cognomi diversi, entrambe le righe saranno mantenute nel risultato.

Utilizzo di WHERE con DISTINCT

Possiamo combinare DISTINCT con la clausola WHERE per filtrare i dati prima di rimuovere i duplicati:

```
SELECT DISTINCT cognome  
FROM studenti  
WHERE cognome LIKE 'V%'  
ORDER BY cognome;
```

Questa query seleziona solo i cognomi che iniziano con la lettera "V", eliminando i duplicati e ordinando il risultato in ordine alfabetico.

GROUP BY

La clausola GROUP BY è un elemento opzionale che può essere utilizzato all'interno di una SELECT.

Il suo scopo è quello di raggruppare i dati in base ai valori di una o più colonne.

Se vogliamo raggruppare i record della tabella studenti in base al cognome, possiamo usare:

```
SELECT cognome  
FROM studenti  
GROUP BY cognome;
```

In questo caso, il risultato sarà lo stesso di una query con DISTINCT:

```
SELECT DISTINCT cognome  
FROM studenti;
```

Entrambe le query restituiscono un elenco di cognomi univoci.

Tuttavia, GROUP BY viene tipicamente utilizzato in combinazione con funzioni di aggregazione (COUNT(), SUM(), AVG(), ecc.), mentre DISTINCT serve solo per rimuovere duplicati.

Raggruppare i risultati di funzioni aggregate

Le potenzialità di GROUP BY emergono:

- quando si utilizza una funzione di aggregazione {COUNT(), AVG(), MAX(), MIN(), SUM()};

Per contare il numero degli studenti raggruppati per genere possiamo scrivere:

```
SELECT genere, COUNT(genere)
FROM studente
GROUP BY genere;
```

Per contare il numero dei corsi raggruppati per docente possiamo scrivere:

```
SELECT cognome, nome, count(*) quanti
FROM docenti d
JOIN corsi c
ON c.docente_id = d.id
GROUP BY d.id
ORDER BY quanti DESC;
```

Per conoscere l'età media degli studenti raggruppati per genere possiamo scrivere:

```
SELECT genere,  
       FL00R(AVG(TIMESTAMPDIFF(YEAR,data_nascita,CURDATE()))) `età media`  
FROM studenti  
GROUP BY genere;
```

Per conoscere il valore dei corsi raggruppati per docente possiamo scrivere:

```
SELECT cognome, nome, sum(prezzo) valore  
FROM docenti d  
JOIN corsi c  
ON d.id = c.docente_id  
GROUP BY d.id  
ORDER BY valore;
```

Per conoscere per ciascuno studente quanto mediamente ha speso per i corsi, a quanti corsi è iscritto, quanto ha speso in totale, il valore del corso più economico a cui si è iscritto e il valore del corso più caro a cui si è iscritto possiamo scrivere:

```
SELECT
    cognome,
    nome,
    round(avg(i.prezzo),2) `Spesa media`,
    count(*) `Iscrizioni`,
    sum(i.prezzo) `Totale speso`,
    min(prezzo) `Minimo speso`,
    max(prezzo) `Massimo speso`
FROM studenti s
JOIN iscrizioni i
ON s.id = i.studente_id
GROUP BY s.id;
```

- quando è utilizzata insieme ad HAVING che aggiunge un filtro su valori raggruppati:

```
SELECT cognome, nome, count(*) quanti
FROM docenti d
JOIN corsi c
ON c.docente_id = d.id
GROUP BY d.id
ORDER BY quanti DESC;
HAVING quanti > 1
ORDER BY quanti;
```

```
SELECT cognome, COUNT(cognome) AS `numero`
FROM studenti
GROUP BY cognome
HAVING numero > 1
ORDER BY cognome;
```

Mediante questa query abbiamo filtrato il resultset mostrando unicamente quei cognomi per i quali è possibile rinvenire più di una occorrenza all'interno della tabella.

```
SELECT provincia, genere, COUNT(*) AS `numero`
FROM studenti
GROUP BY provincia, genere
HAVING `numero` > 1
ORDER BY provincia, genere;
```

Mediante questa query abbiamo raggruppato per provincia e genere gli studenti, contandone il numero; la seconda istruzione filtra ulteriormente il gruppo considerando i valori contati maggiori di 1.

```
SELECT provincia, genere, COUNT(cognome) `numero`  
FROM studenti  
WHERE provincia != 'to'  
GROUP BY provincia, genere  
HAVING numero > 1  
ORDER BY provincia;
```

WHERE può essere usato per ridurre il result set usato, ma prima del GROUP BY:

WHERE filtra le righe;

per un filtro ulteriore *sul gruppo* usate HAVING.

```
SELECT  
    provincia,  
    genere,  
    FLOOR(AVG(TIMESTAMPDIFF(YEAR, data_nascita, curdate()))) `età media` ,  
    count(*) `numero`  
FROM studenti  
GROUP BY provincia, genere  
HAVING `numero` > 1  
ORDER BY provincia, genere
```

GROUP BY ... WITH ROLLUP (istruzione non standard SQL):

```
SELECT provincia, count(*) AS `numero`
FROM studenti
GROUP BY provincia WITH ROLLUP;
```

Mediante questa query abbiamo contato gli studenti divisi per provincia.

Con l'aggiunta dell'istruzione (non SQL) WITH ROLLUP siamo in grado di produrre una riga in più con il totale

```
SELECT provincia, genere, count(*) AS `numero`
FROM studenti
GROUP BY provincia, genere WITH ROLLUP;
```

Mediante questa query abbiamo contato gli studenti divisi per provincia e per genere.

Con l'aggiunta dell'istruzione (non SQL) WITH ROLLUP siamo in grado di produrre le righe in più con i sub totali e il totale.

GROUP BY ... WITH ROLLUP e GROUPING

Con l'aggiunta dell'istruzione **WITH ROLLUP** si produce una riga in più con i subtotali delle province, il valore del superaggregato `province` è `NULL`

Per aggiungere un etichetta e non riportare il valore `NULL` possiamo usare la funzione `grouping()` che distingue le righe del superaggregato dalle righe regolari.

```
SELECT
    provincia,
    count(*) AS `numero`,
    grouping(provincia)
FROM studenti
GROUP BY provincia WITH ROLLUP;
```

`GROUPING` restituisce 0 per le righe regolari e 1 per il superaggregato.

Con l'istruzione `IF` possiamo quindi assegnare un'etichetta al superaggregato:

```
SELECT
    IF(GROUPING(provincia), 'tutte le province', provincia) AS provincia,
    count(*) AS `numero`
FROM studenti
GROUP BY provincia WITH ROLLUP;
```

```
SELECT
    provincia,
    genere,
    count(*) AS `numero`,
    grouping(provincia),
    grouping(genere)
FROM studenti
GROUP BY provincia, genere WITH ROLLUP;
```

```
SELECT
    IF(GROUPING(provincia), 'tutte', provincia) `provincia` ,
    IF(GROUPING(genere), 'totale generi', genere) genere,
    count(*) AS `numero`
FROM studenti
GROUP BY provincia, genere WITH ROLLUP;
```

WINDOWS function

Le [window function](#)¹ (disponibili a partire da MySQL 8) eseguono operazioni aggregate su un insieme di righe specificato, ma a differenza delle funzioni aggregate utilizzate con la clausola [GROUP BY](#), producono un risultato per ogni riga della query.

- Ogni riga su cui viene valutata la funzione è definita come riga corrente
- le righe della query coinvolte nella valutazione della funzione costituiscono la finestra per la riga corrente.

Le funzioni aggregate con [GROUP BY](#) riducono il numero totale di righe restituite dalla query, le [window function](#) operano su un sottoinsieme di righe *senza ridurre il numero di righe restituite dalla query stessa*.

Forniscono un'analisi dettagliata dei dati in base alle condizioni specificate nella finestra di valutazione.

¹ approfondimento: <https://dev.mysql.com/doc/refman/8.0/en/window-functions.html>

La maggior parte delle funzioni aggregate possono essere utilizzate anche come funzioni finestra:

SUM(), AVG(), COUNT(), MAX(), MIN().

Tabella WINDOWS FUNCTION

Function	Descrizione
CUME_DIST()	Valore della distribuzione cumulativa
DENSE_RANK()	Rango della riga corrente all'interno della sua partizione, senza spazi vuoti.
FIRST_VALUE()	Valore dell'argomento dalla prima riga del frame della finestra.
LAG()	Valore dell'argomento dalla riga in ritardo rispetto alla riga corrente all'interno della partizione.
LAST_VALUE()	Valore dell'argomento dall'ultima riga del frame della finestra.
LEAD()	Valore dell'argomento dalla riga in anticipo rispetto alla riga corrente all'interno della partizione.
NTH_VALUE()	Valore dell'argomento dalla N-esima riga del frame della finestra.
NTILE()	Numero del bucket della riga corrente all'interno della sua partizione.
PERCENT_RANK()	Valore del rango percentuale.
RANK()	Rango della riga corrente all'interno della sua partizione, con spazi vuoti.
ROW_NUMBER()	Numero della riga corrente all'interno della sua partizione.

Sintassi:

```
window_function_name(expression) OVER (
    [partition_definition]
    [order_definition]
    [frame_definition]
)
```

`window_function_name`: specifica il nome della funzione finestra seguito da un'espressione.

L'istruzione OVER ha tre possibili elementi:

- definizione della partizione: `PARTITION BY <expression>`

Suddivide le righe in blocchi o partizioni . Due partizioni sono separate da un confine di partizione.

La funzione finestra viene eseguita all'interno delle partizioni e reinizializzata quando si attraversa il confine della partizione.

- definizione dell'ordine: `ORDER BY <expression>`

specificata come vengono ordinate le righe all'interno di una partizione. È possibile ordinare i dati all'interno di una partizione su più chiavi, ciascuna chiave è specificata da un'espressione. Ha senso utilizzare la `ORDER BY` solo per le funzioni della finestra sensibili all'ordine.

- definizione del frame.

Un frame è un sottoinsieme della partizione corrente.

Le parentesi di apertura e chiusura, che compaiono dopo la clausola OVER, sono obbligatorie, anche senza espressione.

[frame_definition]: frame_unit {<frame_start>|<frame_between>}

il **frame unit** specifica *quali righe devono essere incluse nel calcolo della funzione per ogni riga corrente*. Le opzioni principali per il frame unit sono:

ROWS: Specifica un numero fisso di righe da includere nel frame, a partire dalla riga corrente.

Esempio: ROWS 3 PRECEDING includerà le tre righe precedenti alla riga corrente,

Esempio: ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING includerà la riga corrente e le due righe adiacenti (precedente e successiva).

RANGE: Specifica un intervallo di valori da includere nel frame, invece di un numero fisso di righe

Questo può essere utile quando si lavora con dati ordinati in base a un certo criterio, come le date o i valori numerici.

Esempio: RANGE BETWEEN INTERVAL 1 DAY PRECEDING AND CURRENT ROW includerà tutte le righe con date che sono entro un giorno prima o uguale alla data della riga corrente.

Per quanto riguarda *frame_start* e *frame_between*, sono parte della sintassi per specificare il **frame unit**.

frame_start: specifica l'inizio del frame

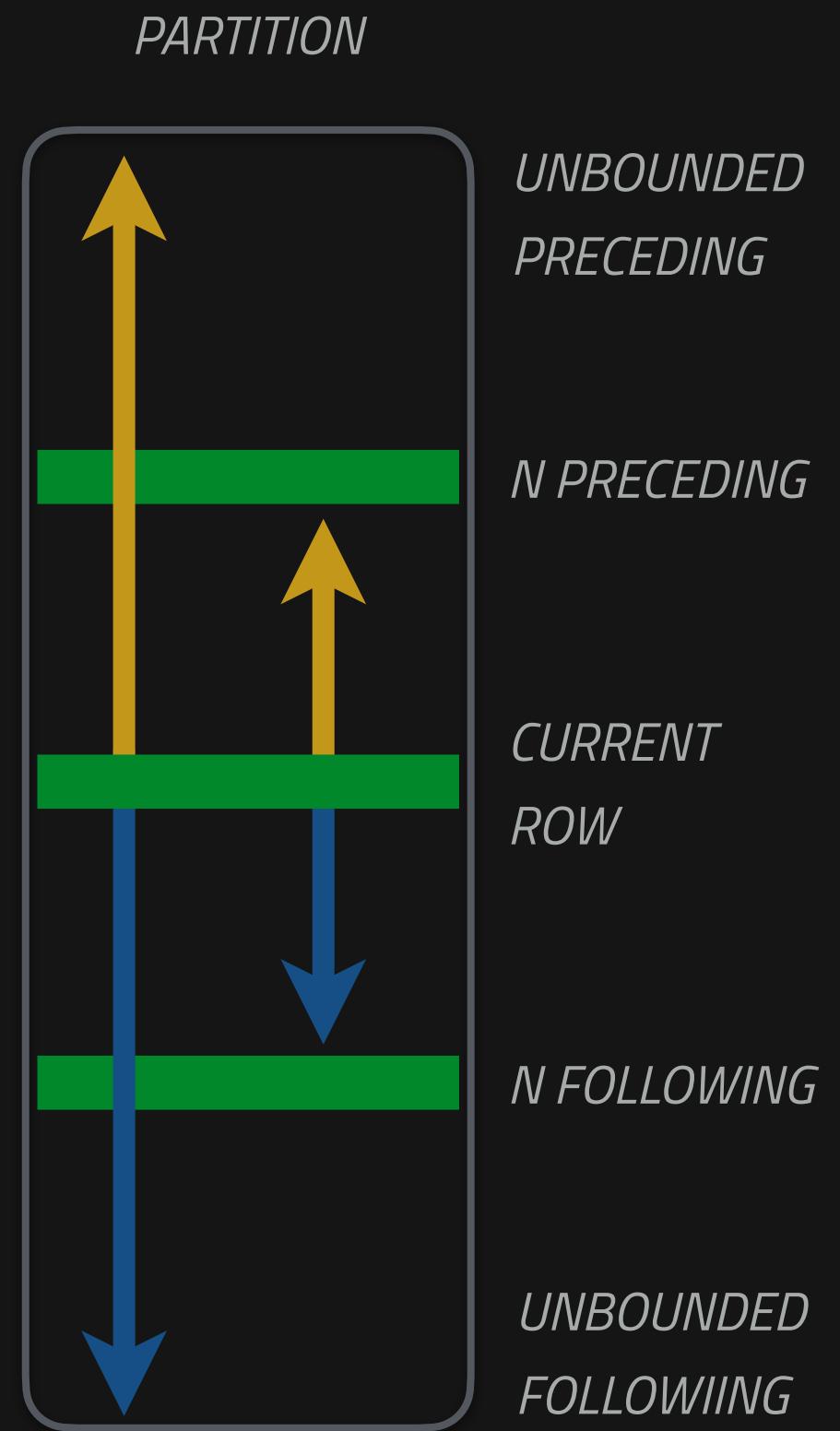
frame_between: specifica l'intervallo del frame.

Questi possono essere utilizzati per definire il frame unit in modo più dettagliato rispetto a quanto fatto utilizzando solo ROWS o RANGE.

Esempio: ROWS BETWEEN 3 PRECEDING AND CURRENT ROW definisce un frame unit che include le tre righe precedenti alla riga corrente e la riga corrente stessa.

Esempio: RANGE BETWEEN INTERVAL 1 DAY PRECEDING AND CURRENT ROW definisce un frame unit che include tutte le righe con date entro un giorno prima o uguale alla data della riga corrente.

Valore di default: RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW



Esempio: AVG()

Voglio ottenere la differenza di stipendio per ciascun impiegato rispetto allo stipendio medio:

```
SELECT
    nome,
    cognome,
    stipendio,
    ROUND(AVG(stipendio) OVER(),2) `Stipendio medio`,
    stipendio - ROUND(AVG(stipendio) OVER(),2) Differenza
FROM impiegati
ORDER BY stipendio DESC;
```

Esempio: AVG()

Voglio ottenere la differenza di stipendio per ciascun impiegato rispetto allo stipendio medio del dipartimento(ufficio) di appartenenza:

```
SELECT
    nome,
    cognome,
    ufficio_id,
    stipendio,
    ROUND(
        AVG(stipendio) OVER(PARTITION BY ufficio_id)
        ,2) `Salario medio`,
    stipendio - ROUND(
        AVG(stipendio) OVER(PARTITION BY ufficio_id)
        ,2) `Differenza`
FROM impiegati
ORDER BY stipendio DESC;
```

Esempio: SUM()

Supponiamo di avere una tabella vendite che registra le vendite degli impiegati per anno fiscale. Nella tabella vendite abbiamo un riferimento all'id dell'impiegato, l'anno fiscale e la somma delle vendite per quell'anno.

Voglio ottenere le vendite totali `SUM(vendita)` per anno fiscale `OVER (PARTITION BY anno)`

```
SELECT
    anno,
    cognome,
    nome,
    totale,
    SUM(totale) OVER (
        PARTITION BY anno -- crea le partizioni considerando gli anni
        -- default, prende tutte le righe della partizione
        -- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        -- prendo riga precedente e successiva
        -- ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING -- prendo riga precedente e successiva
        -- prendo le 2 righe precedenti
        -- ROWS 2 PRECEDING
    ) vendite_totali
FROM vendite
JOIN impiegati
ON impiegati.id = vendite.impiegato_id;
```

Esempio: SUM()

Voglio ottenere le vendite totali **SUM(totale)** per anno fiscale **OVER (PARTITION BY anno)** e il valore percentuale:

```
SELECT
    anno,
    cognome,
    nome,
    totale,
    SUM(totale) OVER (
        PARTITION BY anno
    ) `vendite totali`,
    ROUND(totale / SUM(totale) OVER (
        PARTITION BY anno
    ) * 100, 2) `Percentuale`
FROM vendite
JOIN impiegati
ON impiegati.id = vendite.impiegato_id;
```

LAG(), LEAD()

La funzione LAG() restituisce il valore di una colonna dalla riga precedente rispetto alla riga corrente all'interno di una finestra.

```
LAG(expression [, offset] [, default]) OVER (
    [PARTITION BY partition_expression, ...]
    ORDER BY order_by_expression
)
```

La funzione LEAD() restituisce il valore di una colonna dalla riga successiva rispetto alla riga corrente all'interno di una finestra.

```
LEAD(expression [, offset] [, default]) OVER (
    [PARTITION BY partition_expression, ...]
    ORDER BY order_by_expression
)
```

Esempio: LAG()

La seguente query utilizza la funzione LAG() per confrontare le vendite di un anno con quello precedente:

```
SELECT
    cognome,
    nome,
    anno,
    totale,
    LAG(totale, 1 , 0) OVER (
        -- la funzione LAG() restituisce le vendite dell'anno precedente
        -- (o zero) dalla riga corrente, senza 0 il valore è NULL
        PARTITION BY impiegato_id
        -- divide le righe nella tabella delle vendite
        -- in partizioni in base agli addetti alle vendite
        ORDER BY anno
        -- le righe in ciascuna partizione vengono ordinate
        -- in base alla colonna dell'anno fiscale
        ) 'anno precedente'
FROM vendite
JOIN impiegati
ON impiegati.id = vendite.impiegato_id;
```

Esempio: LAG()

La seguente query utilizza due volte la funzione LAG().

Per confrontare le vendite di un anno con quello precedente, e con i due anni precedenti:

```
SELECT
    cognome,
    nome,
    anno,
    totale,
    LAG(totale, 1, 0) OVER (
        PARTITION BY impiegato_id
        ORDER BY anno
    ) 'anno precedente',
    LAG(totale, 2, 'nessun riferimento') OVER (
        PARTITION BY impiegato_id
        ORDER BY anno
    ) '2 anni precedente'
FROM vendite
JOIN impiegati
ON impiegati.id = vendite.impiegato_id;
```

Esempio: LEAD()

L'esempio seguente utilizza la funzione LEAD() per inserire le vendite della riga successiva nella riga corrente:

```
SELECT
    cognome,
    nome,
    anno,
    totale,
    LEAD(totale, 1 , 0) OVER (
        PARTITION BY impiegato_id
        ORDER BY anno
    ) AS 'anno successivo'
FROM vendite
JOIN impiegati
ON impiegati.id = vendite.impiegato_id;
```

ROW_NUMBER(), RANK(), DENSE_RANK()

La funzione ROW_NUMBER() restituisce il numero di riga corrente all'interno di una finestra.

È utile per ottenere informazioni sulla posizione relativa di una riga rispetto alle altre all'interno della finestra.

```
ROW_NUMBER() OVER (
    [PARTITION BY partition_expression, ...]
    ORDER BY order_by_expression
)
```

La funzione RANK() assegna un valore di rango a ciascuna riga all'interno di una finestra, assegnando lo stesso valore a righe con valori uguali e saltando i valori successivi. Se ci sono più righe con lo stesso valore ordinato, ottengono lo stesso valore di rango e il successivo viene saltato.

```
RANK() OVER (
    [PARTITION BY partition_expression, ...]
    ORDER BY order_by_expression
)
```

La funzione DENSE_RANK() è simile a RANK(), ma non salta i valori successivi se ci sono valori duplicati. Assegna un valore di rango univoco a ciascuna riga all'interno della finestra, anche se ci sono valori uguali.

```
DENSE_RANK() OVER (
    [PARTITION BY partition_expression, ...]
    ORDER BY order_by_expression
)
```

Esempio: ROW_NUMBER()

```
SELECT  
    id,  
    cognome,  
    data_nascita,  
    UPPER(provincia) `Provincia`,  
    ROW_NUMBER() OVER( PARTITION BY provincia ORDER BY data_nascita ) `Posizione nella partizione`,  
    ROW_NUMBER() OVER() `Posizione sul totale`  
FROM studenti  
ORDER BY provincia, data_nascita;
```

Nota: la prima funzione mostra il numero di riga per ciascuna provincia, mentre la seconda mostra il numero progressivo assoluto nel set risultante.

Esempio: ROW_NUMBER(), RANK() e DENSE_RANK()

```
SELECT
    cognome,
    data_nascita,
    UPPER(provincia) `Provincia`,
    ROW_NUMBER() OVER(PARTITION BY provincia ORDER BY data_nascita, cognome ) `Riga`,
    RANK() OVER(PARTITION BY provincia ORDER BY data_nascita ) `Rank`,
    DENSE_RANK() over(PARTITION BY provincia ORDER BY data_nascita ) `Dense rank`
FROM studenti;
```

```
SELECT
    i.nome,
    cognome,
    stipendio,
    u.nome `Dipartimento`,
    ROW_NUMBER() OVER(PARTITION BY ufficio_id ORDER BY stipendio) `Riga`,
    RANK() OVER(PARTITION BY ufficio_id ORDER BY stipendio ) `Rank`,
    DENSE_RANK() OVER(PARTITION BY ufficio_id ORDER BY stipendio ) `Dense rank`
FROM impiegati i
JOIN uffici u
ON i.ufficio_id = u.id
ORDER BY ufficio_id;
```

VIEW (Viste)

Una vista è una tabella logica basata su una o più query SELECT.

Non contiene dati propri, ma espone il risultato di una query definita su:

- tabelle fisiche (dette base table),
- oppure altre viste.

Vantaggi delle viste

- Limitano l'accesso ai dati sensibili: puoi mostrare solo alcune colonne o righe di una tabella.
- Mascherano la complessità del database: l'utente non deve conoscere i dettagli delle join o dei filtri.
- Riduzione dell'impatto dei cambiamenti: se cambia la struttura delle tabelle, puoi aggiornare solo la vista.
- Semplificano le query: permettono di ottenere risultati complessi usando una SELECT semplice.

Esempio: una vista può fornire i dati da più tabelle collegate, senza che l'utente sappia come scrivere il JOIN.

VIEW (Viste)

Vantaggi

Semplificano le query

Riducono l'impatto dei cambiamenti

Limitano accesso ai dati

```
CREATE VIEW nome_vista AS  
SELECT nome_campi  
FROM nome_tabella  
WHERE condizioni;
```

Crea una vista basata su una query.

Se la vista esiste già, il comando produce un errore.

Le viste si comportano come tabelle virtuali, ma non memorizzano dati: ogni accesso esegue la query definita.

Creare una vista equivale a salvare una query con un nome, che può poi essere richiamata come una tabella.

Le viste create possono essere *semplici*:

- deriva da una sola tabella;
- non contiene funzioni di aggregazione;

o *complesse*:

- deriva da più tabelle in join;
- può contenere funzioni di aggregazione;

Modificare una view

`CREATE OR REPLACE VIEW` sovrascrive la vista se esiste ricreandola da capo.

```
CREATE OR REPLACE VIEW nome_vista AS  
SELECT nome_campi  
FROM nome_tabella  
WHERE condizioni;
```

`ALTER VIEW` modifica la vista.

```
ALTER VIEW nome_vista AS  
SELECT nome_campi  
FROM nome_tabella  
WHERE condizioni;
```

Per rinominare la view, modificarne solo il nome, potete scrivere:

```
RENAME TABLE nome_vista TO nuovo_nome_vista;
```

Viste aggiornabili e non aggiornabili

Una VIEW si dice aggiornabile quando consente di modificare i dati nella tabella sottostante.

Per essere *aggiornabile* la Vista deve possedere un rapporto uno ad uno con la tabella sottostante, *quindi la SELECT che genera la View...*:

- **NON** può utilizzare DISTINCT;
- **NON** può far ricorso a funzioni di aggregazione (SUM, MIN, MAX ...);
- **NON** può utilizzare GROUP BY / HAVING;
- **NON** può contenere UNION;

Le viste con JOIN possono essere aggiornabili *solo in casi specifici*, quando l'aggiornamento impatta una sola tabella in modo non ambiguo.

Una vista con JOIN è aggiornabile solo se il DB può identificare esattamente una riga modificabile in una sola tabella.

In presenza di JOIN complessi o relazioni molti-a-molti, una vista è spesso non aggiornabile, a meno che l'aggiornamento sia limitato a una sola tabella e non introduca ambiguità.

VISTA semplice

```
CREATE VIEW studenti_contatto AS  
SELECT id, nome, cognome, email  
FROM studenti;
```

VISTA complessa

```
CREATE VIEW iscritti AS  
SELECT cognome, nome, email, titolo AS corso, i.prezzo, data_isc  
FROM studenti s  
JOIN iscrizioni i ON s.id = i.studente_id  
JOIN corsi c ON c.id = i.corso_id;
```

Anche se una vista può includere ORDER BY, l'ordinamento non è garantito quando si fa SELECT dalla vista.

Per ottenere un ordine specifico, è meglio usare ORDER BY nella query esterna, quella che richiama la vista.

NOTA: MySQL ignora ORDER BY all'interno della definizione di una vista, salvo che sia usato insieme a LIMIT.

Interrogare una view

Accedere ad una vista è semplicissimo: funzionando quest'ultima esattamente come una comune tabella, sarà sufficiente effettuare una SELECT.

```
SELECT * FROM nome_vista ORDER BY colonna;
```

Per elencare le tabelle e verificare quali tra queste sono in realtà viste:

```
SHOW FULL TABLES WHERE Table_type = 'VIEW';
```

Potete interrogare anche *information_schema* (che è una vista) per ottenere l'elenco delle vostre tabelle con l'indicazione del tipo: *base table* o *view*:

```
SELECT table_name, table_type  
FROM information_schema.tables  
WHERE table_schema = 'nome_db'  
ORDER BY table_name;
```

Eliminare una view

```
DROP VIEW nome_vista [, nome_vista];
```

- L'istruzione DROP rimuove la definizione della vista dal database;
- cancellando la vista non ci sono effetti sulla *base table*;
- viste o altre applicazioni basate sulla vista cancellata diventano invalide;

mostrare il codice della view

```
SHOW CREATE VIEW nome_vista;
```

Vantaggi delle VIEW

Semplificare le query

```
CREATE VIEW iscritti AS
SELECT
    cognome,
    nome,
    email,
    titolo AS `Corso`,
    i.prezzo AS `Prezzo pagato`,
    data_isc AS `Data iscrizione`
FROM studenti s
JOIN iscrizioni i
ON s.id = i.studente_id
JOIN corsi c
ON c.id = i.corso_id;
```

Ora basta una query semplice:

```
SELECT * FROM iscritti; -- e aggiungere eventuali filtri con il WHERE, o GROUP BY...
```

Nota: Semplifica molto le interrogazioni frequenti ed evita ripetizioni.

Vantaggi delle VIEW

Ridurre l'impatto dei cambiamenti

Immagina di voler rinominare una colonna (`c.titolo → c.nome_corso`) e cambiare il nome di una tabella (es. `Corsi → CatalogoCorsi`): o uno dei due casi. Tutte le query esistenti che usano nella SELECT

la colonna *titolo* e il nome tabella *Corsi*

non funzionerebbero più, perché la struttura del database è cambiata.

Se le query sono scritte ovunque nel codice, dovrai modificare tutte le query manualmente.

Ma se usi una vista intermedia, puoi semplicemente aggiornare solo la definizione della vista:

```
ALTER VIEW VistaCorsi AS  
SELECT id, nome_corso AS titolo, prezzo, docente_id  
FROM CatalogoCorsi;
```

E tutte le query che puntano a `VistaCorsi` continueranno a funzionare anche se la tabella cambia.

NOTA: Le viste fanno da “strato di astrazione” e isolano il codice dai cambiamenti nella struttura sottostante.

Vantaggi delle VIEW

Limitano l'accesso ai dati

Supponiamo che un assistente didattico debba vedere solo le iscrizioni con nome e corso degli studenti, ma non i prezzi o le email degli studenti.

Puoi creare una vista sicura:

```
CREATE VIEW VistaIscrizioniLimitata AS
SELECT
    s.nome,
    s.cognome,
    c.titolo AS corso,
    i.data_isc
FROM Iscrizioni i
JOIN Studenti s ON i.studente_id = s.id
JOIN Corsi c ON i.corso_id = c.id;
```

Il DBA concede poi i permessi di lettura solo su questa vista:

```
GRANT SELECT ON VistaIscrizioniLimitata TO assistente_didattico;
```

VIEW con WITH CHECK OPTION

```
CREATE VIEW studenti_v AS  
SELECT id, nome, cognome, email, provincia  
FROM studenti  
WHERE provincia = 'to'  
WITH CHECK OPTION;
```

Specifica che solo le righe accessibili dalla vista possono essere inserite o modificate.

Quindi **INSERT** e **UPDATE** effettuate sulla vista, non possono creare o modificare i dati di cui la vista non ha visibilità.

```
UPDATE studenti_v  
SET provincia = 'cn'  
WHERE id = 1; -- studente che ha provincia uguale a 'T0'
```

```
INSERT INTO studenti_v(nome, cognome, email, provincia)  
VALUES('paolo','picchio','ppicchio89@msn.com','al');
```

L'update e l'insert restituiscono l'errore:

```
ERROR 1369 (HY000): CHECK OPTION failed 'studente_v'
```

Il **CHECK OPTION** è utile per garantire che gli utenti non possano 'uscire' dal perimetro della vista.

È possibile togliere il check option ridefinendo la vista con **ALTER VIEW**.

tabelle temporanee

In MySQL, una *tabella temporanea* è un *tipo speciale di tabella* che *consente di memorizzare un set di risultati temporanei*, che è possibile riutilizzare più volte in una singola sessione.

Una tabella temporanea di MySQL ha le seguenti caratteristiche:

- Viene creata una tabella temporanea utilizzando l'istruzione [CREATE TEMPORARY TABLE](#).
- MySQL rimuove automaticamente la tabella temporanea quando la sessione termina o la connessione viene terminata. Naturalmente, è possibile utilizzare l'istruzione [DROP TEMPORARY TABLE](#) per rimuovere una tabella temporanea esplicitamente quando non la si usa più ([TEMPORARY](#) è opzionale, ma è meglio usarlo).
- Una tabella temporanea è disponibile e accessibile solo al client che lo crea. I diversi clienti possono creare tabelle temporanee con lo stesso nome senza causare errori perché solo il client che crea la tabella temporanea può vederla. Tuttavia, nella stessa sessione, due tabelle temporanee non possono condividere lo stesso nome.
- Una tabella temporanea può avere lo stesso nome di una tabella normale in un database. Ma è *caldamente sconsigliato* poiché in caso di interruzione forzata della sessione al successivo accesso si potrebbe creare confusione tra le due tabelle.

Esempio di tabella temporanea:

```
CREATE TEMPORARY TABLE giovane  
SELECT nome, cognome, genere, eta  
FROM studente  
WHERE eta < 31;
```

Questa tabella temporanea contiene tutti gli studenti con meno di 31 anni.

È ora possibile interrogarla come una normale tabella.

```
SELECT * FROM giovane;  
  
SELECT genere, count(*)  
FROM giovane  
GROUP BY genere;
```

Rimozione di una tabella temporanea di MySQL

È possibile utilizzare l'istruzione **DROP TABLE** per rimuovere tabelle temporanee tuttavia è buona norma aggiungere la parola chiave **TEMPORARY** come segue:

```
DROP TEMPORARY TABLE table_name;
```

L'istruzione **DROP TEMPORARY TABLE** rimuove solo una tabella temporanea, non una tabella permanente.

Aiuta a evitare il rischio di cancellare per errore una tabella permanente quando si denombra la tabella temporanea uguale al nome della tabella permanente (sconsigliato, come già detto - vedi slide precedente)

Creiamo una tabella temporanea che riporti la generazione di appartenenza di ciascuno studente:

```
CREATE TEMPORARY TABLE generazioni
SELECT
    cognome,
    data_nascita `Data di nascita`,
    CASE
        WHEN year(data_nascita) <= 1965 THEN 'Boomer'
        WHEN year(data_nascita) >= 1966
            AND year(data_nascita) <= 1980 THEN 'X'
        WHEN year(data_nascita) >= 1981
            AND year(data_nascita) <= 1995 THEN 'millenials'
        WHEN year(data_nascita) >= 1996 THEN 'Z'
        WHEN year(data_nascita) IS NULL THEN 'Manca data nascita'
    END Generazione
FROM studenti
ORDER BY Generazione;
```

Estrai solo gli studenti appartenenti alla generazione dei millennial

```
SELECT * FROM generazioni
WHERE Generazione = 'millenials';
```

Creiamo una tabella temporanea che riporti il riepilogo di ciascuno studente:

```
CREATE TEMPORARY TABLE RiepilogoComplessivo AS
SELECT
    s.id AS studente_id,
    s.nome,
    s.cognome,
    COUNT(i.id) AS num_corsi,
    SUM(i.prezzo) AS totale_speso
FROM Studenti s
JOIN Iscrizioni i ON s.id = i.studente_id
GROUP BY s.id;
```

Uso la tabella per effettuare filtri e ordinamenti

```
SELECT * FROM RiepilogoComplessivo
WHERE totale_speso > 200 -- WHERE cognome LIKE 'v%' o WHERE num_corsi > 1
ORDER BY totale_speso DESC; -- ORDER BY cognome...
```

Esercizio: Gestione ordini per shop hardware/software

Vogliamo gestire gli ordini dei clienti di uno shop di hardware e software, tenere traccia degli articoli e gestire il magazzino.

- Per ogni ordine registriamo la *data*, il *cliente* che lo effettua, gli *articoli* richiesti e la *quantità* per ciascuno.
- Gli ordini hanno uno *stato* e *informazioni* per la spedizione.
- Ogni ordine fa maturare al cliente un *credito*, calcolato come 1 punto per ogni euro speso.
- Gli articoli sono categorizzati in *hardware* o *software* (categoria fissa).
- I dipendenti appartengono a uffici specifici (*amministrazione*, *logistica*, *vendita*, *assistenza*) e hanno ruoli diversi (*tecnico*, *amministrativo*, *venditore*, *magazziniere*).

Realizzare il diagramma relativo al *progetto concettuale* (entità/associazioni) e definite lo *schema logico* (tabelle e attributi), quindi:

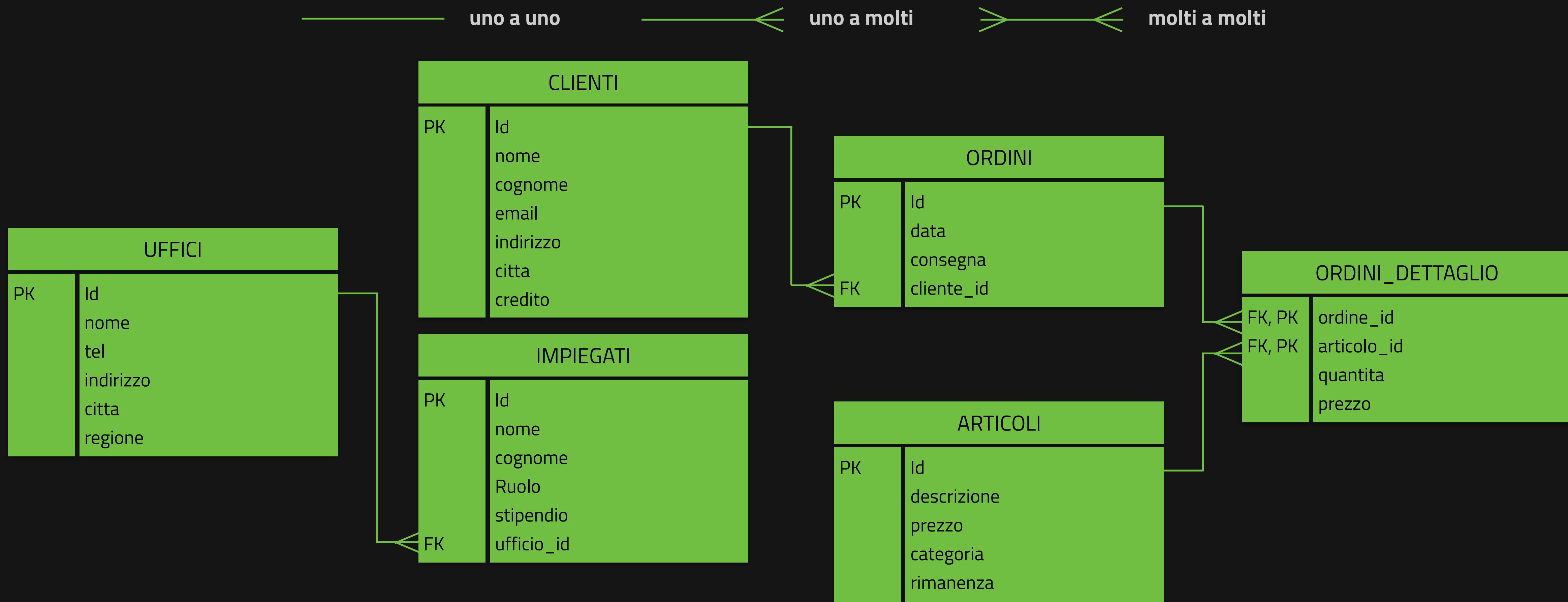
- Create un nuovo database, chiamatelo “*gestionale*”;
- Assegnate i privilegi all’utente attuale per accedere al database;
- Create le tabelle necessarie per questo specifico database;
- Analizzate le regole di integrità referenziale, definite le foreign key necessarie (scrivete solo uno schema, poi prima di applicarle lo vediamo insieme)

Diagramma E-R



Modello logico

Simbologia delle relazioni



Assegnazione accesso a nuovo database all'utente attuale

DCL: gestire il controllo degli accessi e i permessi per gli utenti: **GRANT**

```
GRANT ALL #istruzioni consentite  
ON database.*  
TO 'user'@'localhost';
```

Si possono assegnare privilegi, ad un utente, per più database, solo se i database condividono un prefisso comune (es: `sql%` [%,_] sono wildcard per la sostituzione rispettivamente di più caratteri o di un solo carattere -vedi operatore LIKE).

```
GRANT ALL  
ON `sql%`.*  
TO 'user'@'localhost';
```

Subquery

Una subquery è un'istruzione **SELECT** all'interno di un'altra istruzione SQL(**SELECT** , **INSERT** , **UPDATE** , **DELETE** ...).

Una subquery MySQL può essere nidificata all'interno di un'altra subquery.

Una subquery viene in genere aggiunta all'interno della condizione **WHERE** di un'altra istruzione **SELECT** .

È possibile utilizzare gli operatori di confronto, come **>** , **<** , **=** ...

L'operatore di confronto può anche essere un operatore a più righe, ad esempio

IN , **NOT IN** , **ANY** , **SOME** o **ALL** .

La subquery è anche chiamata *query interna*, mentre la query che contiene la subquery si chiama *query esterna*.

La *subquery* (*query interna*) viene eseguita prima della *query esterna*, a meno che non sia una *subquery correlata*.

Tipi di subquery:

- subquery scalare
- subquery con operatori di confronto
- subquery con operatori di confronto avanzato **ALL** , **ANY** , **IN** , **NOT IN**
- row subquery con costruttore di righe, istruzione **ROW()**
- subquery correlate
- subquery con **EXISTS** o **NOT EXISTS**
- subquery nella clausola **FROM**

vantaggi:

- Consentono query strutturate in modo che sia possibile isolare ogni parte di una dichiarazione.
- Forniscono metodi alternativi per eseguire operazioni che altrimenti richiederebbero **UNION** e **JOIN** complesse.
- Molti trovano le subquery più leggibili rispetto a **UNION** o **JOIN** complesse.

Vediamo un esempio di sintassi:

```
SELECT elenco_campi  
FROM tabella  
WHERE espressione operatore (SELECT elenco_campi FROM tabella);
```

Query interna. Query esterna.

Una subquery può restituire un risultato *scalare* (un singolo valore), *una singola riga*, *una singola colonna* o *una tabella* (una o più righe di una o più colonne). Queste sono chiamate subquery scalari, a colonne, a righe e a tabelle.

Vediamo un esempio: vogliamo l'elenco degli impiegati, pagati più del impiegato "Barba" il cui id è 6;

Possiamo ottenere l'elenco in due passaggi:

```
SELECT stipendio FROM impiegati WHERE id = 6; ##[ r:1500.00 ]
```

```
SELECT nome, cognome, stipendio  
FROM impiegati  
WHERE stipendio > 1500.00  
ORDER BY cognome;
```

Oppure unire le due query nidificandone una nell'altra:

```
SELECT nome, cognome, stipendio  
FROM impiegati  
WHERE stipendio >  
(SELECT stipendio FROM impiegati WHERE id = 6)  
ORDER BY stipendio;
```

Abbiamo utilizzato l'id dell'impiegato perché la sub query deve restituire una sola riga, il risultato della condizione è un valore solo (subquery scalare);

Se avessimo utilizzato il cognome saremmo potuti incorrere nell'errore come da esempio seguente, dal momento che di impiegati con cognome uguale a "Barba" ce n'è più d'uno:

```
SELECT nome, cognome, stipendio  
FROM impiegati  
WHERE stipendio > (SELECT stipendio FROM impiegati WHERE cognome = 'Barba')  
ORDER BY cognome;
```

```
ERROR 1242 (2100): Subquery returns more than 1 row
```

La subquery scalare.

Vediamo altro esempio:

Selezioniamo dalla tabella *corsi*, i *corsi* il cui prezzo è maggiore del prezzo medio del nostro catalogo corsi.

```
SELECT titolo, prezzo  
FROM corsi  
WHERE prezzo > (SELECT AVG(prezzo) FROM corsi)  
ORDER BY prezzo DESC;
```

Notate la **SELECT** utilizzata per ottenere il valore del prezzo medio come valore di confronto nella condizione **WHERE**.

La seguente query non ha senso:

```
SELECT prezzo FROM corsi WHERE prezzo > AVG(prezzo);
```

```
ERROR 1111 (HY000): Invalid use of group function
```

La subquery scalare.

Vediamo altro esempio:

Selezioniamo dalla tabella *corsi*, i corsi che costano di più nel nostro catalogo.

```
SELECT titolo, prezzo  
FROM corsi  
WHERE prezzo = (SELECT max(prezzo) FROM corsi)  
ORDER BY prezzo DESC;
```

Notate la **SELECT** utilizzata per ottenere il valore del prezzo massimo come valore di confronto nella condizione **WHERE**.

La subquery scalare.

Vediamo altro esempio:

Selezioniamo le info del cliente che ha eseguito l'ultimo ordine.

```
SELECT c.cognome, c.nome, c.email  
FROM clienti c  
JOIN ordini o  
ON c.id=o.cliente_id  
WHERE o.id = (SELECT MAX(id) FROM ordini);
```

Selezioniamo le info del cliente che ha eseguito l'ultimo ordine con più query nidificate (senza join)

```
SELECT cognome, nome, email  
FROM clienti  
WHERE id = (  
    SELECT cliente_id  
    FROM ordini  
    WHERE id = (  
        SELECT MAX(id)  
        FROM ordini  
    )  
);
```

Subquery con operatori di confronto.

Una Subquery può essere utilizzata insieme a uno qualsiasi degli operatori di confronto.

La Subquery può restituire al massimo un valore. Il valore può essere il risultato di un'espressione aritmetica o di una funzione di colonna.

SQL confronta quindi il valore risultante dalla subquery con il valore sull'altro lato dell'operatore di confronto.

Vediamo due esempi

```
SELECT nome, cognome, stipendio  
FROM impiegati  
WHERE stipendio < (SELECT AVG(stipendio) FROM impiegati)  
ORDER BY stipendio DESC;
```

```
SELECT titolo, prezzo  
FROM corsi  
WHERE prezzo > (SELECT AVG(prezzo) FROM corsi);
```

Subquery con: ALL

È possibile utilizzare dopo un operatore di confronto, l'operatore di confronto avanzato:
ALL, ANY [SOME] prima della subquery.

L'operatore **ALL** confronta ogni valore restituito dalla subquery.

Pertanto, l'operatore **ALL** (che deve seguire un operatore di confronto: `=, >, < ...`) restituisce VERO se il confronto è VERO **per TUTTI i valori** nella colonna restituiti dalla subquery.

La seguente query seleziona i corsi con più iscritti.

La subquery conta gli studenti raggruppati per ciascun corso, quindi la query principale seleziona tra questi i corsi con più iscritti.

```
SELECT c.titolo, COUNT(i.corso_id) AS `quanti`
FROM corsi c
JOIN iscrizioni i ON c.id = i.corso_id
GROUP BY c.id, c.titolo
HAVING quanti >= ALL (SELECT COUNT(studente_id) FROM iscrizioni GROUP BY corso_id)
ORDER BY `quanti` DESC;
```

Nota: qui è stata utilizzata la parola chiave ALL per questa subquery poiché l'ufficio selezionato dalla query deve avere uno stipendio medio superiore o uguale allo stipendio medio degli altri uffici.

La seguente query seleziona l'ufficio i cui impiegati hanno il salario medio più alto.

La subquery trova lo stipendio medio per ciascun ufficio, quindi la query principale seleziona l'ufficio con lo stipendio medio più alto.

```
SELECT u.nome, AVG(stipendio) `Stipendio medio`
FROM impiegati i
JOIN uffici u
ON u.id = i.ufficio_id
GROUP BY i.ufficio_id
HAVING `Stipendio medio` >= ALL
    ( SELECT AVG( stipendio ) FROM impiegati GROUP BY
ufficio_id );
```

Nota: qui è stata utilizzata la parola chiave ALL per questa subquery poiché l'ufficio selezionato dalla query deve avere uno stipendio medio superiore o uguale allo stipendio medio degli altri uffici.

Subquery con: ANY(SOME)

Le subquery che usano la parola chiave **ANY [SOME]** restituiscono TRUE se la comparazione restituisce TRUE **per almeno una delle righe** restituite dalla subquery.

Se utilizzato con una subquery, la parola **IN** è un alias per **= ANY**.

Quindi, queste due istruzioni sono uguali.

La seguente query seleziona gli impiegati che lavorano in una data regione, es: '*piemonte*'.

La subquery trova l'ID degli uffici che si trovano in '*piemonte*', quindi la query principale seleziona gli impiegati che lavorano in uno di questi uffici.

```
SELECT cognome, nome  
FROM impiegati  
WHERE ufficio_id = ANY  
(SELECT id FROM uffici WHERE regione = 'piemonte');
```

abbiamo utilizzato la parola chiave **ANY** in questa query perché è probabile che la subquery troverà più di un ufficio nella regione Piemonte.

Se si utilizza la parola chiave **ALL** anziché la parola chiave **ANY**, nessun dato viene selezionato perché nessun dipendente lavora in tutti gli uffici che si trovano in Piemonte.

Subquery con: ANY(SOME)

Posso contare gli impiegati che lavorano in una data regione

```
SELECT 'Piemonte', COUNT(*)
FROM impiegati
WHERE ufficio_id = ANY -- IN
(SELECT id FROM uffici WHERE regione = 'piemonte');
```

Otteniamo la stessa cosa con la JOIN

```
SELECT regione, COUNT(*)
FROM impiegati JOIN uffici
ON impiegati.ufficio_id = uffici.id
AND regione = 'piemonte';
```

Subquery con: IN (= ANY)

Se una subquery restituisce più di un valore si possono effettuare confronti utilizzando, all'interno della clausola WHERE gli operatori avanzati: IN, NOT IN.

Vediamo l'esempio con IN: selezioniamo i docenti che *hanno corsi*.

```
SELECT cognome, nome, email  
FROM docenti  
WHERE id IN (SELECT docente_id FROM corsi); -- WHERE ID = ANY
```

di seguito lo stesso esempio con una JOIN.

```
SELECT DISTINCT cognome, nome, email  
FROM docenti d  
JOIN corsi c  
ON d.id = c.docente_id;
```

Vediamo l'esempio con **IN**: selezioniamo i clienti che *hanno effettuato ordini*.

```
SELECT cognome, telefono, citta  
FROM clienti  
WHERE id IN (SELECT DISTINCT cliente_id FROM ordini); — WHERE  
ID = ANY
```

di seguito lo stesso esempio con una JOIN.

```
SELECT DISTINCT cognome, telefono, citta  
FROM clienti  
INNER JOIN ordini ON clienti.id=ordini.cliente_id;
```

Subquery con: NOT IN

Vediamo un esempio con NOT IN: selezioniamo i *docenti che non hanno corsi*.

```
SELECT cognome, nome, email  
FROM docenti  
WHERE id NOT IN (SELECT docente_id FROM corsi); -- equivalente a ID <> ALL
```

Alternativa più sicura e robusta con LEFT JOIN:

```
SELECT cognome, nome, email  
FROM docenti d  
LEFT JOIN corsi c ON d.id=c.docente_id  
WHERE c.id IS NULL;
```

Attenzione: NOT IN è sensibile ai valori *NULL*.

Se la subquery restituisce anche un solo valore *NULL*, il confronto fallisce e nessuna riga viene restituita.

È più sicuro usare la JOIN con *IS NULL*, che non soffre di questo problema. Nel nostro caso ci possono essere *docente_id = NULL*, diversamente potremmo usare tranquillamente NOT IN.

Oppure aggiungere alla subquery: WHERE *docente_id IS NOT NULL*

Vediamo un esempio funzionante con NOT IN: selezioniamo i *clienti che non hanno effettuato ordini*.

```
SELECT cognome, telefono, citta
FROM clienti
WHERE id NOT IN (SELECT DISTINCT cliente_id FROM ordini); --
equivalente a ID <> ALL
```

In questo caso sappiamo che *cliente_id* non può essere *NULL* quindi la query funziona anche con NOT IN.

Alternativa con LEFT JOIN:

```
SELECT cognome, telefono, citta
FROM clienti c
LEFT JOIN ordini o ON c.id=o.cliente_id
WHERE o.id IS NULL;
```

Vediamo altro esempio funzionante con NOT IN: selezioniamo gli *articoli che non sono presenti negli ordini*.

```
SELECT descrizione  
FROM articoli a  
WHERE a.id NOT IN (SELECT DISTINCT articolo_id FROM ordini_dettaglio);
```

In questo caso sappiamo che *articolo_id* non può essere *NULL* quindi la query funziona anche con NOT IN.

Alternativa con LEFT JOIN:

```
SELECT descrizione  
FROM articoli a  
LEFT JOIN ordini_dettaglio od ON a.id=od.articolo_id  
WHERE od.articolo_id IS NULL;
```

Row subquery: ROW(field1, field2, [field3],...)

Una subquery di riga è una subquery che restituisce **una singola riga e più di un valore di colonna**.

Quando una subquery restituisce una singola riga, può essere usata per fare confronti attraverso i *costruttori di righe*:

L'espressione **ROW(nome, cognome)** è un costruttore di riga, che può essere espresso anche come **(nome, cognome)**.

Vediamo un esempio:

```
SELECT * FROM amici  
WHERE ROW( nome, cognome ) = ( SELECT nome, cognome FROM studenti WHERE id = 4 );
```

Questa query confronta le righe dalla tabella amici per i campi nome e cognome con la riga estratta nella subquery finché non trova una corrispondenza.

```
SELECT * FROM amici  
WHERE ROW( nome, cognome ) = ('[nome]', '[cognome]');
```

Questa query confronta le righe dalla tabella amici per i campi nome e cognome con la riga specificata, finché non trova una corrispondenza.

```
SELECT * FROM amici  
WHERE (cognome, nome) IN (SELECT cognome, nome FROM parenti);
```

Questa query risponde alla richiesta " trova tutte le righe nella tabella amici che esistono anche nella tabella parenti.

documentazione: <https://dev.mysql.com/doc/refman/8.0/en/row-subqueries.html>

Subquery correlate

Le subquery correlate contengono un riferimento ad una delle tabelle che fanno parte della query esterna, quindi non sono indipendenti:

Vediamo esempio:

```
UPDATE articoli a
SET rimanenza = 100 -
  (SELECT SUM(quantita)
   FROM ordini_dettaglio od
   WHERE od.articolo_id = a.id
  );
```

Questa query aggiorna la tabella articoli sulla base degli ordini effettuati.

Notare che se un articolo non è mai stato ordinato la rimanenza verrà impostata a *NULL*; di conseguenza dovremmo aggiornare tutti i valori *NULL* al valore del magazzino = 100.

```
UPDATE articoli SET rimanenza = 100 WHERE rimanenza IS NULL;
```

Riprendendo l'esempio precedente relativo all'aggiornamento del magazzino, grazie alla funzione IFNULL tutto si può scrivere in una sola query (IFNULL non è standard SQL, è funzione di MySQL):

```
UPDATE articoli a
SET rimanenza = 100 -
IFNULL(
(
    SELECT SUM(quantita)
    FROM ordini_dettaglio od
    WHERE od.articolo_id = a.id)
,
0
);
```

NOTA: L'efficienza delle subquery correlate dipende dal numero di righe e dagli indici disponibili. È quindi da valutare caso per caso.

Su dataset piccoli o medi è efficiente e chiara, ma su dati molto grandi conviene valutare query con JOIN e aggregazioni per migliorare le prestazioni. Vedi slide "subquery nella clausola FROM".

Subquery per aggiornare il credito di tutti i clienti (esempi con COALESCE - standard SQL):

```
UPDATE clienti c
SET credito = COALESCE(
    (
        SELECT SUM(od.prezzo * od.quantita)
        FROM ordini o
        JOIN ordini_dettaglio od ON o.id = od.ordine_id
        WHERE o.cliente_id = c.id
    ),
    0
);
```

Subquery per aggiornare il credito di un cliente:

```
UPDATE clienti c
SET credito = COALESCE(
    (
        SELECT SUM(od.prezzo * od.quantita)
        FROM ordini o
        JOIN ordini_dettaglio od ON o.id = od.ordine_id
        WHERE o.cliente_id = c.id
    ),
    0
)
WHERE c.id = 3;
```

Subquery con EXISTS o NOT EXISTS

L'operatore **EXISTS** verifica l'esistenza di righe nel set di risultati della subquery.

Se viene trovato un valore di riga, la subquery **EXISTS** è *TRUE* e in questo caso la subquery **NON EXISTS** è *FALSE*.

Questa query estrae i nomi dei docenti che hanno almeno un corso assegnato nella tabella *corsi*.

```
SELECT cognome, nome
FROM docenti d
WHERE EXISTS (
    SELECT 1 -- la subquery non usa i valori
    FROM corsi c
    WHERE c.docente_id = d.id
);
```

Vediamo la stessa cosa con una JOIN

```
SELECT DISTINCT cognome, nome
FROM docenti d
JOIN corsi c ON c.docente_id = d.id;
```

NOTA: La subquery non ci serve per sapere quale *docente*, ma solo se ce n'è almeno uno.

Per questo scriviamo **SELECT 1**: è una convenzione per dire 'non mi interessa il contenuto, mi interessa solo l'esistenza di righe'."

Questa query estrae i nomi dei docenti che non hanno corsi assegnati nella tabella *corsi*.

```
SELECT cognome, nome
FROM docenti d
WHERE NOT EXISTS (
    SELECT 1
    FROM corsi c
    WHERE c.docente_id = d.id
);
```

Vediamo la stessa cosa con una OUTER JOIN

```
SELECT cognome, nome
FROM docenti d
LEFT JOIN corsi c
ON d.id = c.docente_id
WHERE c.id IS NULL;
```

Questa query estrae i nomi dei clienti che hanno almeno un ordine registrato nella tabella *ordini*.

```
SELECT cognome, nome  
FROM clienti c  
WHERE EXISTS  
(SELECT 1 FROM ordini o WHERE o.cliente_id = c.id);
```

Vediamo la stessa cosa con una JOIN

```
SELECT DISTINCT cognome, nome  
FROM clienti c  
INNER JOIN ordini o  
ON c.id = o.cliente_id;
```

NOTA: La subquery non ci serve per sapere quale *ordine*, ma solo se ce n'è almeno uno.

Per questo scriviamo SELECT 1: è una convenzione per dire 'non mi interessa il contenuto, mi interessa solo l'esistenza di righe'."

Questa query estrae i nomi dei clienti che non hanno ordini registrati nella tabella *ordini*.

```
SELECT cognome, nome
FROM clienti c
WHERE NOT EXISTS
  (SELECT 1 FROM ordini o
 WHERE o.cliente_id = c.id);
```

Vediamo la stessa cosa con una OUTER JOIN

```
SELECT cognome, nome
FROM clienti c
LEFT JOIN ordini o
ON c.id = o.cliente_id
WHERE o.id IS NULL;
```

Subquery nella clausola FROM

Le subquery possono essere inserite anche nella istruzione `FROM`.

Ricordiamoci delle viste, che sono i realtà query memorizzate nel database!

Consideriamo la vista `studenti_giovani` in cui mostriamo gli studenti che hanno meno di 31 anni.

```
CREATE OR REPLACE VIEW studenti_giovani AS
SELECT cognome, nome, email, timestampdiff(YEAR, data_nascita, curdate()) `età`
FROM studenti
WHERE timestampdiff(YEAR, data_nascita, curdate()) < 30;
```

Quando interroghiamo la vista la SELECT è la seguente:

```
SELECT * FROM studenti_giovani;
```

Siccome la vista è una query memorizzata è come se scrivessimo:

```
SELECT * FROM (
    SELECT cognome, nome, email, timestampdiff(YEAR, data_nascita, curdate()) `età`
    FROM studenti
    WHERE timestampdiff(YEAR, data_nascita, curdate()) <= 30
) AS tbl
ORDER BY `età` DESC;
```

NOTA: Ogni tabella derivata deve avere un suo nome (alias)

*: attenzione verificate sempre la subquery quando questo è possibile, cioè in caso di subquery indipendente.

Prendiamo in considerazione la query che aggiorna le quantità in magazzino sulla base degli articoli ordinati.

```
UPDATE articoli a
SET rimanenza = 100 -
IFNULL(
(
    SELECT SUM(quantita)
    FROM ordini_dettaglio od
    WHERE od.articolo_id = a.id
)
,
0
);
```

Questa query esegue il calcolo della *subquery* per ogni riga della tabella articoli.

Se ci sono 1000 articoli la *subquery* viene eseguita 1000 volte!

Possiamo tentare di ottimizzare la query facendo il **JOIN** tra la tabella *articoli* e una tabella temporanea aggregata:

```
UPDATE articoli a
LEFT JOIN (
    SELECT articolo_id, SUM(quantita) AS totale
    FROM ordini_dettaglio
    GROUP BY articolo_id
) AS od_sum
ON a.id = od_sum.articolo_id
SET a.rimanenza = 100 - IFNULL(od_sum.totale, 0);
```

Prendiamo in considerazione la query che aggiorna il credito dei clienti sulla base degli ordini eseguiti.

```
UPDATE clienti c
SET credito = COALESCE(
    (SELECT SUM(od.prezzo * od.quantita)
     FROM ordini o
     JOIN ordini_dettaglio od ON o.id = od.ordine_id
     WHERE o.cliente_id = c.id
    )
    , 0
);
```

Questa query esegue il calcolo della *subquery* per ogni riga della tabella *clienti*.

Se ci sono 1000 clienti la *subquery* viene eseguita 1000 volte!

Possiamo tentare di ottimizzare la query facendo il **JOIN** tra la tabella *clienti* e una tabella temporanea aggregata:

```
UPDATE clienti c
LEFT JOIN (
    SELECT o.cliente_id, SUM(od.prezzo * od.quantita) AS totale
    FROM ordini o
    JOIN ordini_dettaglio od ON o.id = od.ordine_id
    GROUP BY o.cliente_id
) totali ON c.id = totali.cliente_id
SET c.credito = COALESCE(totali.totale, 0);
```

Prendiamo in considerazione la tabella ordini_dettaglio:

Vogliamo ricavare il numero massimo, il numero minimo e la media di articoli venduti rispetto agli ordini:

```
SELECT  
    MAX(q_articoli),  
    MIN(q_articoli),  
    ROUND(AVG(q_articoli))  
FROM  
(  
    SELECT /* ordine_id */ SUM(quantita) AS q_articoli  
    FROM ordini_dettaglio  
    GROUP BY ordine_id  
) AS tbl;
```

In questo caso la subquery seleziona e somma:

`SUM(quantita)`

la quantità di articoli presenti in ciascun ordine:

`GROUP BY ordine_id`

e la passa alla query principale, sotto forma di tabella virtuale:

`... FROM (SELECT...) AS tbl;`

che ricava il numero massimo, il numero minimo e la media di articoli venduti.

Backup (dump) e restoring

Il termine "dump" nel contesto di un database si riferisce a una copia completa o a uno snapshot dei dati presenti nel database in un determinato momento.

Questo processo di creazione di una copia completa dei dati del database è noto come "[dumping del database](#)".

Il termine "dump" deriva dalla parola inglese "dump", che significa scaricare o gettare in modo informale.

In questo caso, il database "dump" è una rappresentazione completa dei dati, spesso sotto forma di un file o di un insieme di file, che può essere archiviato o trasferito in altri sistemi per scopi di backup, ripristino o migrazione. Il dump quindi, è un backup logico, non fisico.

Un dump di database può contenere tutte le tabelle, gli indici, le viste e altri oggetti di database, insieme a tutti i dati in essi contenuti.

Questo snapshot rappresenta una "fotografia" del database in un momento specifico e può essere utilizzato per ripristinare il database in caso di perdita di dati o per clonare il database su un altro sistema.

In breve, il termine "dump" nel contesto dei database indica la creazione di una copia completa e statica dei dati del database, ed è una pratica comune nell'amministrazione dei database per scopi di backup e manutenzione.

Backup (dump) e restoring

Backup di un DB MySQL (da interfaccia grafica: *MySQLWorkbench* o *PhpMyAdmin*)

Normalmente l'operazione di backup (dump) di un DB MySQL si esegue attraverso un software con un'interfaccia grafica.

Ci sono vari software che vi consentono di gestire il database:

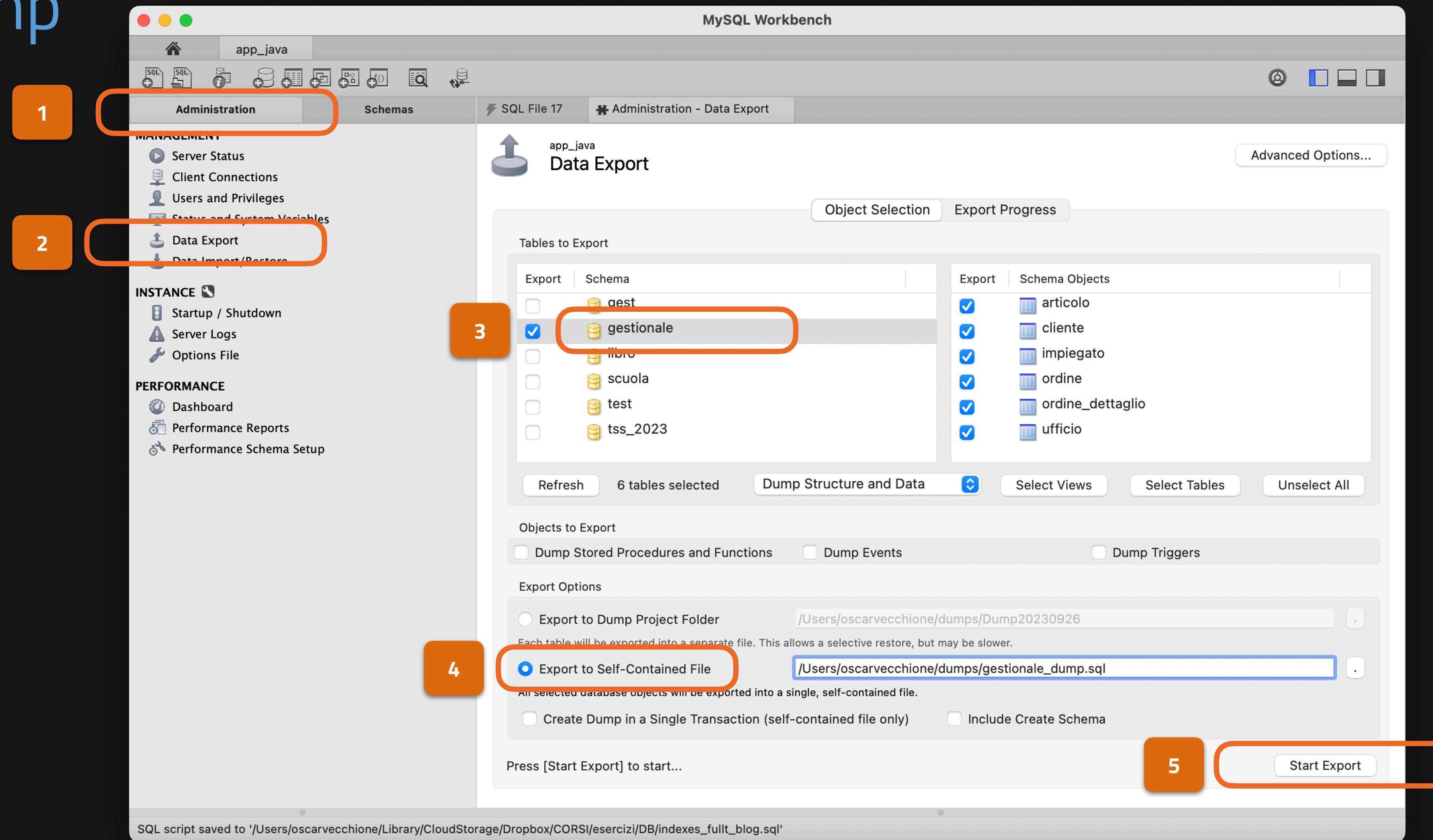
sul web il più diffuso è sicuramente PhpMyAdmin, software scritto in php.

in locale (sul proprio PC) il più diffuso è sicuramente MySQLWorkbench.

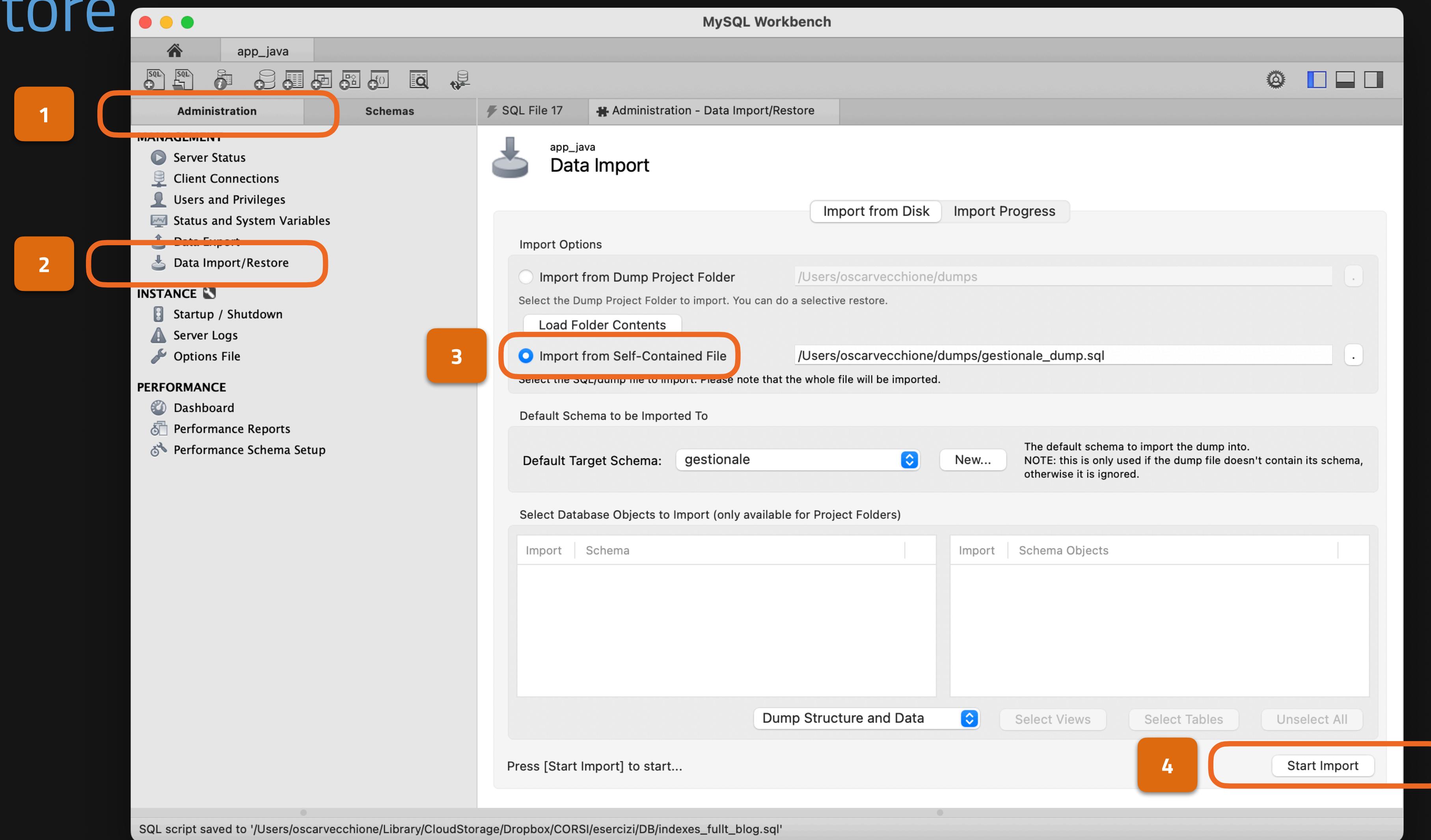
Le operazioni di esportazione e importazione si trovano in apposite sezioni di questi software:

- Sezione "Administration", item "Data Export" per l'esportazione; item "Data Import/Restore" per l'importazione (MySQLWorkbench)
- Sezione (tab) "Esporta" per l'esportazione; sezione (tab) "Importa" per il ripristino (PhpMyAdmin);

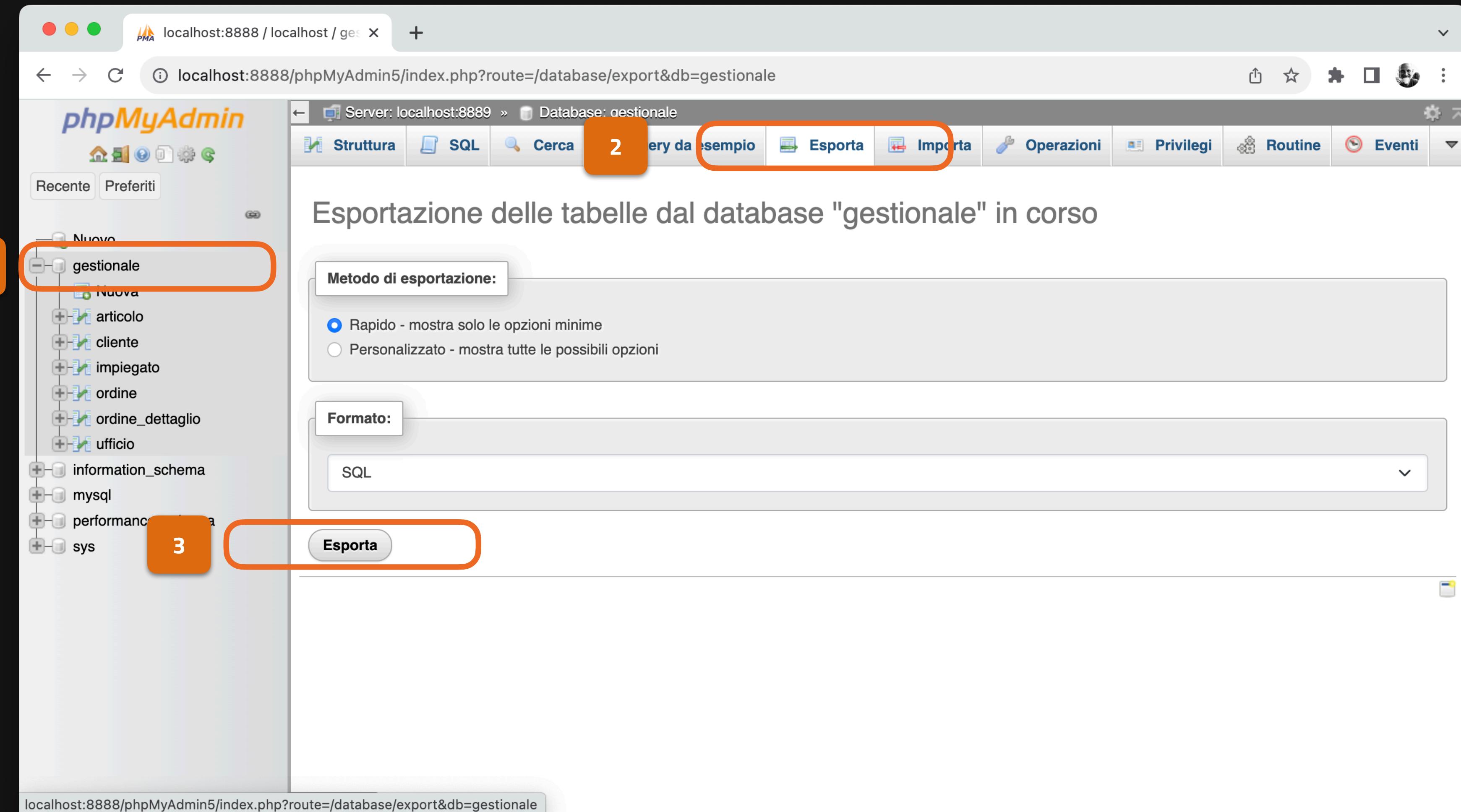
Dump



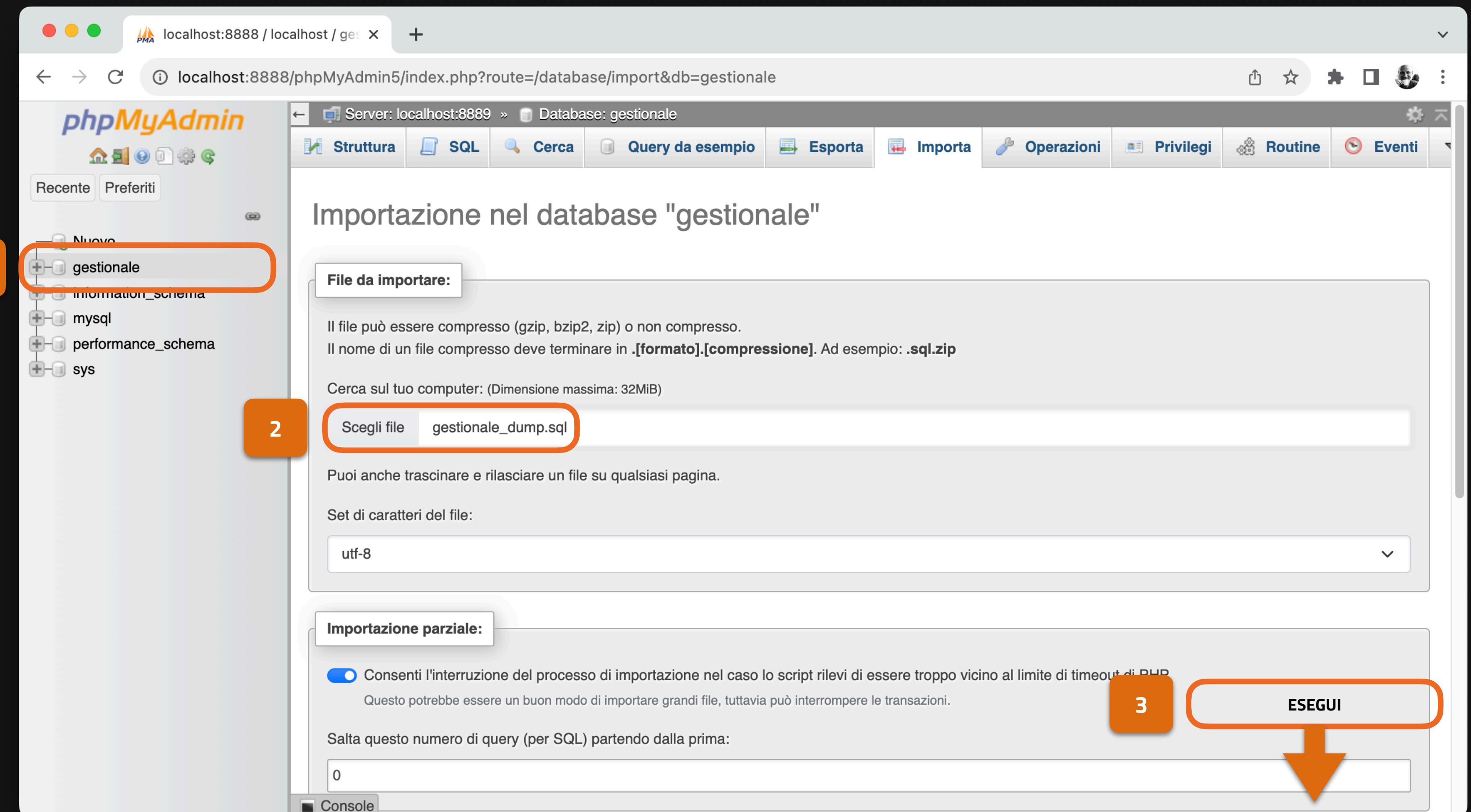
Restore



Dump



Restore



Backup (dump) e restoring

Backup di un DB MySQL (da shell)

Privilegi da amministratore: *utente root*

L'operazione di backup (dump) di un DB MySQL, normalmente, si esegue attraverso il comando `mysqldump` prima di collegarsi al db.

Nella sua versione base la sintassi è la seguente:

```
mysqldump -u root* -p nome_database > nomefile.sql**  
mysqldump -u root* -p --databases db_1 db_2 db_3 > nomefile.sql**  
mysqldump -u root -p --all-databases > nomefile.sql**
```

Nel primo caso stiamo esportando un database (`nome_database`);

Nel secondo caso stiamo esportando tre database: `db_1`, `db_2`, `db_3`;

Nel terzo caso stiamo esportando tutti i database, utenti e privilegi;

L'opzione `--databases` scrive l'istruzione: `CREATE DATABASE IF NOT EXIST e USE nomedb.`

* = nome dell'utente, in questo caso l'utente con privilegi massimi

** = percorso del file in cui scrivere le istruzioni sql (es: C:/Users/anskat_PC/Desktop/), se si specifica solo il nome del file, il file viene copiato nella directory corrente (nel caso di xampp c:\xampp)

Backup (dump) e restoring

Backup di un DB MySQL (da shell)

Privilegi da utente: *app_java*

L'operazione di backup (dump) di un DB MySQL, normalmente, si esegue attraverso il comando `mysqldump` prima di collegarsi al db.

Nella sua versione base la sintassi è la seguente:

```
mysqldump -u user1 -p2 --no-tablespaces3 mio_db > mio_db.sql4
```

¹ nome dell'utente

² per sicurezza la password viene digitata successivamente e non *passata in chiaro*

³ <https://anothercoffee.net/how-to-fix-the-mysqldump-access-denied-process-privilege-error/>

l'opzione --no-tablespaces prima di -u user è obbligatoria a partire da mysql 5.7.31 e 8.0.21)

<https://dev.mysql.com/doc/refman/5.6/en/innodb-system-tablespace.html>

L'opzione --no-tablespaces evita gli errori legati ai permessi sui metadati avanzati, perché i tablespace sono configurazioni di storage avanzate che, se non usate esplicitamente, non influenzano i dati esportati.

⁴ percorso del file in cui scrivere il dump sql (es: C:/Users/anskat_PC/Desktop/), se si specifica solo il nome del file, il file viene copiato nella directory corrente.

```
mysqldump -u user1 -p --no-tablespaces mio_db nome_tabella > nomefile.sql2
mysqldump -u user1 -p --no-tablespaces mio_db nome_tabella01 nome_tabella02
nome_tabella03 > nomefile.sql2
```

nel primo caso esportiamo solo la tabella *nome_tabella*;

nel secondo caso esportiamo tre tabelle: *nome_tabella01 nome_tabella02 nome_tabella03*

Esportazione della **sola struttura del database** (definizione delle tabelle)

```
mysqldump -u user1 -p --no-tablespaces -d mio_db > nomefile.estensione2
```

Esportazione dei **soli dati del database** (contenuti)

```
mysqldump -u user1 -p --no-tablespaces -t mio_db > nomefile.estensione2
```

Una volta premuto invio possiamo verificare se il file è stato creato correttamente nella directory indicata.

¹ nome dell'utente

² percorso del file in cui scrivere le istruzioni sql (es: C:/Users/anskat_PC/Desktop/), se si specifica solo il nome del file, il file viene copiato nella directory corrente

Restoring di un DB MySQL

Il restoring del database di solito si esegue per spostare un database in altro database che qualcun altro ha creato per voi (DBA).

Il DBA vi fornisce le credenziali di accesso: *nome_database, host, user e password*.

Questa la sintassi per il restoring

```
mysql -u user -p mio_db < mio_db.sql
```

Se da un backup contenente una pluralità di DB ne vogliamo ripristinare uno solo, possiamo farlo utilizzando l'opzione --one-database in questo modo:

```
mysql -u user -p --one-database nome_del_db < backup.sql
```

Il db deve essere presente; bisogna avere i privilegi.

INDICI

Quando il database cresce in dimensioni per rendere più veloce ed efficiente la ricerca si usano gli **INDICI** per i campi usati di frequente nel WHERE, JOIN, GROUP BY e ORDER BY.

L'aggiornamento, l'inserimento e la cancellazione dei record nella tabella sarà leggermente più lento perché anche l'indice dovrà essere aggiornato.

```
CREATE INDEX indexName  
ON tableName(fieldName);
```

```
ALTER TABLE tableName  
ADD INDEX indexName(fieldName);
```

STANDARD SQL

Per eliminare l'indice creato:

```
DROP INDEX indexName ON tableName;
```

```
ALTER TABLE tableName DROP INDEX indexName;
```

Per rinominare l'indice:

```
ALTER TABLE tableName RENAME INDEX oldName TO newName;
```

Per mostrare gli indici di una tabella

```
SHOW INDEX FROM tableName; SHOW INDEX IN tableName;
```

Nella visualizzazione della struttura di una tabella (`DESC nome_tabella`) l'indice viene indicato con la sigla: *MUL*, che sta per "multiple" perché sono consentite diverse occorrenze dello stesso valore.

Approfondimento sugli indici

Capire cosa sono gli indici è importante per riuscire a scrivere le query in modo performante.

Un *indice* è una sorta di schedario che tiene traccia su dove sono posizionati i dati all'interno delle tabelle nel database.

Gli indici possiamo considerarli come delle tabelle speciali associate alle tabelle dati consultabili dal database.

Ogni operazione che tenta il recupero di dati da qualsiasi tabella del database, in assenza di un indice, costringe il database a leggere l'intera tabella, eseguendo quello che in gergo viene definito **Table Scan** (lettura record per record di tutta tabella).

Attraverso l'INDICE il database identifica la posizione esatta dalle informazioni che usa per recuperare i dati necessari.

- si evita il Table Scan,
- il recupero dei dati su cui le query devono lavorare avviene in modo più veloce;
- la query stessa è più performante.

In InnoDB, un indice secondario memorizza i valori della colonna indicizzata (ordinati) insieme alla chiave primaria (PK) della riga corrispondente. Durante una ricerca:

- Se la query utilizza solo colonne presenti nell'indice (covering index), MySQL legge direttamente dall'indice senza accedere alla tabella.
- Se la query richiede colonne non presenti nell'indice, MySQL usa la PK trovata per recuperare i dati aggiuntivi dal clustered index (tabella stessa).



Gli indici sono fondamentalmente strutture di dati, utilizzate dai motori di database per trovare rapidamente i dati.

MySQL utilizza come struttura dati per gli indici B+Tree.

Le strutture B+Tree sono mantenute in parte in memoria (nel buffer pool di InnoDB) per velocizzare l'accesso. Se non presenti in memoria, vengono lette da disco.

Accedere prima agli indici è più performante e veloce nel recupero dei record piuttosto che leggere tutta la tabella (full table scan).

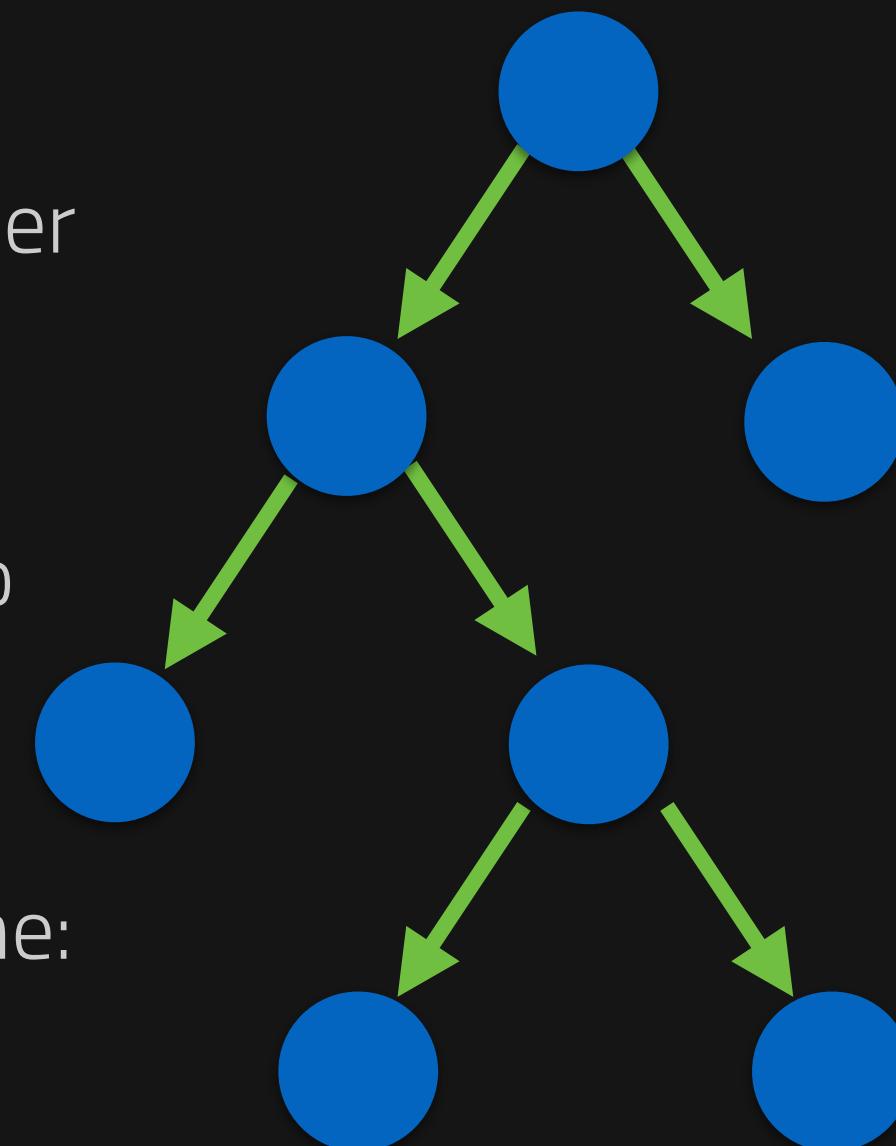
Se gli indici sono fondamentali per le performance delle query, bisogna considerare che:

- *Aumentano le dimensioni del database*, vengono memorizzati con le tabelle;
- *Rallentano la scrittura*, gli indici devono essere aggiornati;

È quindi fondamentale **creare gli indici sulla base delle query** e non in base alle tabelle.

1 <https://it.wikipedia.org/wiki/B-albero>

B+Tree



INDICE

Provincia

AL

AL

MI

MI

TO

TO

TO

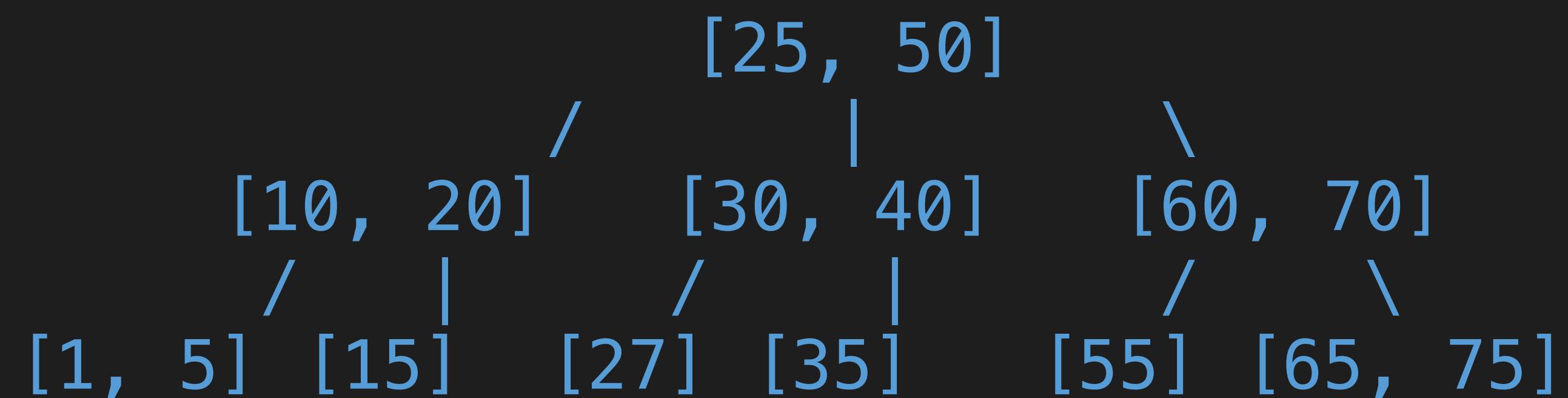
Cercare il valore 5:

1. Radice [25,50]: $5 < 25 \rightarrow$ vai al primo ramo ([10,20]).
2. Nodo [10,20]: $5 < 10 \rightarrow$ vai al primo ramo ([1,5]).
3. Nodo [1,5]: 5 è presente \rightarrow valore trovato!

Perché è efficiente?

Il B+tree riduce i confronti necessari. Con 8 valori nell'esempio, servono solo 3 passaggi (livelli).

In un indice reale con milioni di valori, i livelli sono tipicamente 4-5, rendendo le ricerche istantanee.



ANALYZE

Aggiorna le statistiche degli indici e della tabella per ottimizzare l'esecuzione delle query.

- Statistiche degli indici:
 - Distribuzione dei valori (cardinality: numero di valori unici presenti in quella colonna (o combinazione di colonne) indicizzata).
 - Densità dei dati (quanto è "affollato" un indice: quanto i valori in un indice sono ripetuti o distribuiti).
- Statistiche della tabella:
 - Numero di righe (rows).
 - Dimensione fisica della tabella.

Quando eseguirlo:

- Dopo modifiche massive ai dati.
- Quando le query diventano lente.
- Dopo modifiche strutturali importanti

```
ANALYZE TABLE studenti; -- Forza l'aggiornamento immediato
```

ANALYZE, esempio d'uso

```
ANALYZE TABLE studenti; -- Forza l'aggiornamento immediato
```

Restituisce:

Table	Op	Msg_type	Msg_text
corsi.studenti	analyze	status	OK

Quando Eseguire ANALYZE TABLE?

- Dopo aver modificato >10% dei dati (es. inserimenti/cancellazioni massive).
- Se le query diventano improvvisamente lente senza motivo apparente.
- Dopo la creazione di nuovi indici.

EXPLAIN

L'istruzione **EXPLAIN** fornisce informazioni su come MySQL esegue le query:

EXPLAIN funziona con le istruzioni **SELECT**, **DELETE**, **INSERT** e **UPDATE**.

Quando viene utilizzato **EXPLAIN** con un'istruzione spiegabile, MySQL visualizza le informazioni dell'ottimizzatore sul piano di esecuzione dell'istruzione.

MySQL mostra come elaborerebbe l'istruzione, comprese le informazioni su come vengono unite le tabelle e in quale ordine.

Con l'aiuto di **EXPLAIN**, puoi vedere dove aggiungere indici alle tabelle in modo che l'istruzione venga eseguita più velocemente usando gli indici per trovare le righe.

```
SELECT nome, cognome, email, credito FROM clienti  
WHERE provincia = "To";
```

Se eseguiamo questa query in assenza di indici la tabella *clienti* verrà letta record per record fino a trovare le corrispondenze.

Il database inoltre accede al disco per la lettura della tabella (*non sempre*).

Verifichiamola con l'istruzione EXPLAIN

```
EXPLAIN SELECT nome, cognome, email, credito FROM clienti  
WHERE provincia = "To";
```

Eseguita la query con `EXPLAIN`, mysql restituisce il piano di esecuzione risultante:

<code>id</code>	<code>select_type</code>	<code>table</code>	<code>partitions</code>	<code>type</code>	<code>possible_keys</code>	<code>key</code>	<code>key_len</code>	<code>ref</code>	<code>rows</code>	<code>filtered</code>	<code>Extra</code>
1	SIMPLE	cliente	NULL	ALL	NULL	NULL	NULL	NULL	7	14.29	Using where

Osservate la colonna `type`, la colonna `key`, la colonna `rows` e la colonna `Extra`:

In assenza di indici vengono letti tutti i record.

Ora creiamo un indice sulla tabella `clienti` per l'attributo `provincia`

```
CREATE INDEX k_prov ON clienti(provincia);
```

<code>id</code>	L'identificatore SELECT
<code>select_type</code>	Tipo di operazione di selezione che viene eseguita.
<code>table</code>	La tabella a cui si riferisce l'operazione.
<code>partitions</code>	Le partizioni corrispondenti, se ci sono. Altrimenti <code>NULL</code> .
<code>type</code>	Metodo di accesso utilizzato per recuperare le righe dalla tabella.
<code>possible_keys</code>	I possibili indici tra cui scegliere
<code>key</code>	L'indice effettivamente scelto
<code>key_len</code>	La lunghezza dell'indice scelto in byte.
<code>ref</code>	Le colonne rispetto all'indice
<code>rows</code>	Stima delle righe da esaminare
<code>filtered</code>	Percentuale di righe filtrate in base alla condizione della tabella
<code>Extra</code>	Informazioni aggiuntive

Eseguite nuovamente la query con EXPLAIN:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cliente	NULL	ref	k_prov	k_prov	8	const	3	100.00	Using index condition

Come si può notare, type = ref indica che l'accesso è basato su una colonna indicizzata con valore costante, che è stato usato l'indice creato e le righe lette in totale sono 3.

Il filtro sui record ha la massima efficacia.

Consideriamo altro esempio:

```
SELECT nome, cognome, email, credito FROM clienti  
WHERE provincia = "To" and credito > 100;
```

In questo caso la query utilizza l'indice.

```
EXPLAIN SELECT nome, cognome, email, credito FROM cliente  
WHERE provincia = "To" and credito > 100;
```

+-----+ <th>id</th> <th>select_type</th> <th>table</th> <th>partitions</th> <th>type</th> <th>possible_keys</th> <th>key</th> <th>key_len</th> <th>ref</th> <th>rows</th> <th>filtered</th> <th>Extra</th>	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
+-----+	1	SIMPLE	cliente	NULL	ref	k_prov	k_prov	8	const	3	33.33	Using index condition; Using where

Ma il valore di *filtered* è basso, si può quindi ottimizzare ulteriormente?

Creiamo un indice composto, cioè un indice su più campi

```
CREATE INDEX k_prov_credito ON clienti(provincia, credito);
```

Eseguiamo la query con EXPLAIN

```
EXPLAIN SELECT nome, cognome, email, credito FROM cliente  
WHERE provincia = "To" and credito > 100;
```

+-----+ <th>id</th> <th>select_type</th> <th>table</th> <th>partitions</th> <th>type</th> <th>possible_keys</th> <th>key</th> <th>key_len</th> <th>ref</th> <th>rows</th> <th>filtered</th> <th>Extra</th>	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
+-----+	1	SIMPLE	cliente	NULL	range	k_prov,k_prov_credito	k_prov_credito	13	NULL	1	100.00	Using index condition

Quando si deve decidere l'ordine degli attributi nella creazione dell'indice bisogna considerare le query che verranno utilizzate e la *cardinalità* (numero di valori distinti in una colonna) degli attributi scelti:

```
SELECT nome, cognome, email, indirizzo, citta FROM clienti  
WHERE provincia = "To";
```

```
SELECT nome, cognome, email, indirizzo, citta, credito FROM clienti  
WHERE provincia = "To" and credito > 100;
```

L'indice composto più adatto è

```
CREATE INDEX k_prov_credito ON clienti(provincia, credito);
```

Una linea guida suggerisce di mettere per prima la colonna con più *alta cardinalità* nell'indice composto, perché riduce meglio il numero di righe da analizzare.

Tuttavia, l'ordine corretto dipende sempre dalle query effettive eseguite.

È sempre consigliabile creare l'indice sulla base delle query reali, e verificare con EXPLAIN ANALYZE se la scelta è efficace.

EXPLAIN - sintesi

Cosa fa EXPLAIN

- Mostra il piano di esecuzione della query.
- Aiuta a identificare colli di bottiglia.

Quando usarlo

- Per ottimizzare query lente.
- Per verificare l'uso degli indici.

Output chiave da guardare

- type: Tipo di accesso (es. ALL = scansione completa, index = lettura di tutto l'indice, range = legge più valori nell'indice, ref = uso indice)...
- keys: Indice usato.
- rows: record letti.
- Extra: Operazioni aggiuntive (es. Using filesort = ordinamento lento).

Esiste anche una versione di EXPLAIN che mostra più dettagli: [EXPLAIN FORMAT= JSON](#)

EXPLAIN ANALYZE (MySQL 8+) – Analisi Reale delle Query

- EXPLAIN mostra il piano stimato.
- EXPLAIN ANALYZE mostra anche tempi e righe reali.

EXPLAIN ANALYZE

```
SELECT nome, cognome, email FROM impiegati_record -- tabella con oltre 300.000 record
WHERE provincia = 'TO';

-> Index lookup on impiegati_record using k_prov (provincia='TO'),
  with index condition: (impiegati_record.provincia = 'TO')
  (cost=6864 rows=54456) -- stima iniziale di righe lette
  (actual time=1.06..35.8 rows=27403 loops=1) -- righe effettivamente lette
```

Vantaggi:

- Mostra tempo reale di esecuzione (actual time)
- Mostra righe effettivamente lette (rows)
- Aiuta a verificare se l'indice è realmente usato e utile

NOTA: Le stime possono essere sbagliate. EXPLAIN ANALYZE conferma cosa accade davvero.

Riassumendo

Istruzioni per analizzare le query

- EXPLAIN (testo o tabellare)

- * Mostra solo stime.
- * È il "piano previsto" dall'ottimizzatore.

- EXPLAIN FORMAT= JSON

Mostra ancora stime, ma molto più dettagliate, come:

- * costi stimati
- * condizioni pushdown
- * dettagli delle join
- * ottimizzazioni interne
- * uso degli hint
- * stima su righe filtrate, gruppi, file sort, temp table

- EXPLAIN ANALYZE

è il migliore strumento in assoluto per sapere come MySQL esegue davvero la query.

- Esegue veramente la query
- Riporta tempi reali per ogni step
- Riporta righe realmente lette, non stimate
- Confronta stime vs valori reali
- Indica se l'indice è stato davvero usato
- Mostra loop e iterazioni

Quando NON è utile creare un indice

Gli indici non sono sempre utili. In alcuni casi peggiorano le prestazioni.

Casi in cui evitarli:

- Colonne con pochi valori distinti

Esempio: *genere, attivo, is_admin* (Alta probabilità di leggere molte righe comunque)

- Tabelle molto piccole

Se la tabella ha poche centinaia di righe, il full table scan è più veloce

- Campi non usati in query

Non indicizzare colonne mai usate in WHERE, JOIN, ORDER BY, GROUP BY

- Scritture frequenti

Troppe scritture → overhead per aggiornare l'indice

Regola pratica:

Crea indici in base alle query, non in base alla struttura della tabella.

Indice FULLTEXT

MySQL supporta l'indicizzazione e la ricerca full-text:

Gli indici full-text possono essere utilizzati solo con tabelle InnoDB e MyISAM.

MySQL supporta indici FULLTEXT in InnoDB da MySQL 5.6.

Possono essere creati solo per colonne CHAR, VARCHAR o TEXT (nelle diverse lunghezze).

L'indice FULLTEXT può essere definito quando viene creata una tabella o aggiunto in seguito utilizzando ALTER TABLE o CREATE INDEX.

La definizione di indice **FULLTEXT** in **CREATE TABLE** è meno costosa rispetto alla creazione di un indice **FULLTEXT** su una tabella che è già caricata con i dati.

```
-- Creazione con CREATE TABLE
CREATE TABLE posts
(
    id int NOT NULL AUTO_INCREMENT,
    titolo varchar(255) NOT NULL,
    testo text NOT NULL,
    data_pubblicazione datetime DEFAULT NULL,
    FULLTEXT INDEX fullk_titolotesto(titolo, testo)
    PRIMARY KEY(id)
);

-- Se non creato insieme alla tabella aggiunta con ALTER TABLE
ALTER TABLE posts ADD FULLTEXT INDEX fkt_titolo_testo(titolo, testo);
```

Se l'indice **FULLTEXT** viene definito su una tabella prima del caricamento dei dati, non è necessario ricostruire la tabella e i relativi indici per aggiungere la nuova colonna.

Gli indici **FULLTEXT** utilizzano un indice invertito (inverted index):

1. Memorizzano tutte le parole uniche presenti nel testo (es. "database", "MySQL").
2. Per ogni parola, salvano:
 - Le righe (documenti) in cui appare.
 - Le posizioni nel testo (per ricerche di prossimità, come "MySQL database").

In sintesi:

Un indice **FULLTEXT** funziona come un dizionario che, per ogni parola, elenca tutte le pagine (righe) del libro (tabella) dove compare, segnando anche la posizione esatta.

Ricerca

La ricerca full-text viene eseguita utilizzando la sintassi: MATCH() e AGAINST().

MATCH() prende un elenco separato da virgole che denombra le *colonne* da cercare.

AGAINST() accetta *una stringa* da cercare e un *modificatore* facoltativo che indica il tipo di ricerca da eseguire.

La stringa di ricerca deve essere un valore stringa costante durante la valutazione della query.

Ciò esclude, ad esempio, una colonna di tabella perché può differire per ogni riga.

La lunghezza minima predefinita delle parole trovate dalle ricerche full-text è di 3 caratteri per gli indici InnoDB o di 4 caratteri per MyISAM.

Sintassi:

MATCH (col1,col2,...) AGAINST (expr [search_modifier])

search_modifier:

{

 IN NATURAL LANGUAGE MODE

 | IN BOOLEAN MODE

}

IN NATURAL LANGUAGE MODE: Una ricerca in linguaggio naturale interpreta la stringa di ricerca come una frase nel linguaggio umano naturale (una frase in testo libero).

Non ci sono operatori speciali, ad eccezione dei caratteri virgolette ("").

Le ricerche full-text sono ricerche in linguaggio naturale se viene fornito il modificatore **IN NATURAL LANGUAGE MODE** o se non viene fornito alcun modificatore.

IN BOOLEAN MODE: Una ricerca *booleana* interpreta la stringa di ricerca utilizzando le regole di un linguaggio di query speciale.

La stringa contiene le parole da cercare. Può anche contenere operatori che specificano requisiti tali che una parola debba essere presente o assente nelle righe corrispondenti o che debba avere un peso maggiore o minore del normale.

IN NATURAL LANGUAGE MODE

```
SELECT titolo, testo, data_pubblicazione FROM posts  
WHERE MATCH(titolo, testo) AGAINST('react redux');
```

L'esempio mostra come utilizzare la funzione MATCH() in cui le righe vengono restituite in ordine di pertinenza decrescente.

```
SELECT titolo, testo, data_pubblicazione, MATCH(titolo, testo) AGAINST('react redux')  
AS peso  
FROM posts ;
```

L'esempio mostra come recuperare i valori di pertinenza in modo esplicito. Le righe restituite non sono ordinate perché l'istruzione SELECT non include le clausole WHERE né ORDER BY.

```
SELECT titolo, testo, data_pubblicazione, MATCH(titolo, testo) AGAINST('react redux')  
FROM posts  
WHERE MATCH(titolo, testo) AGAINST('react redux');
```

La query restituisce i valori di pertinenza e ordina anche le righe in ordine di pertinenza decrescente.

Per ottenere questo risultato, bisogna specificare MATCH() due volte: una volta nell'elenco SELECT e una volta nella clausola WHERE. Ciò non causa alcun sovraccarico aggiuntivo, poiché l'ottimizzatore MySQL rileva che le due chiamate MATCH() sono identiche e richiama il codice di ricerca full-text solo una volta.

```
SELECT titolo, testo, data_pubblicazione, MATCH(titolo, testo)
AGAINST('"gestione dello stato è un problema")')
FROM posts
WHERE MATCH(titolo, testo) AGAINST('"gestione dello stato è un problema"');
```

Una frase racchiusa tra virgolette doppie (" ") corrisponde solo alle righe che contengono la frase letteralmente, così come è stata digitata.

Il motore full-text suddivide la frase in parole ed esegue una ricerca FULLTEXT nell'indice delle parole.

Non è necessario che i caratteri non di parola corrispondano esattamente: la ricerca di frasi richiede solo che le corrispondenze contengano esattamente le stesse parole della frase e nello stesso ordine.

Ad esempio, "test phrase" corrisponde a "test, phrase". La punteggiatura viene ignorata.

Se la frase non contiene parole presenti nell'indice, il risultato è vuoto. Ad esempio, se tutte le parole sono stopword¹ o più corte della lunghezza minima delle parole indicizzate, il risultato è vuoto.

¹⁾ Una stopword è una parola come "li" o "con" che è così comune che si ritiene abbia valore semantico zero.

Esiste un elenco di parole non significative integrato, ma può essere sovrascritto da un elenco definito dall'utente.

IN BOOLEAN MODE

Attraverso il boolean mode possiamo usare degli operatori¹ per escludere o includere un termine (-, +, "", [per altri operatori vedi il link in nota])

```
SELECT *, MATCH(titolo, testo) AGAINST('react -redux' IN BOOLEAN MODE)
      peso
   FROM posts
 WHERE MATCH(titolo, testo) AGAINST('react -redux' IN BOOLEAN MODE);
```

```
SELECT *, MATCH(titolo, testo) AGAINST('redux -react +stato' IN BOOLEAN
      MODE) peso
   FROM posts
 WHERE MATCH(titolo, testo) AGAINST('redux -react +stato' IN BOOLEAN
      MODE);
```

¹⁾ <https://dev.mysql.com/doc/refman/5.6/en/fulltext-boolean.html>

(nessun operatore)

Per impostazione predefinita (quando né + né - viene specificato), la parola è facoltativa, ma le righe che la contengono hanno un punteggio più alto. Questo imita il comportamento MATCH() AGAINST() senza il modificatore IN BOOLEAN MODE.

Operatore +

Un segno più iniziale o finale indica che questa parola **deve** essere presente in ogni riga restituita.

InnoDB supporta solo i segni più iniziali (direttamente davanti alla parola).

Operatore -

Un segno meno iniziale o finale indica che questa parola **non deve** essere presente in nessuna delle righe restituite.

InnoDB supporta solo i segni meno iniziali (direttamente davanti alla parola).

Nota: l'operatore - agisce solo per escludere le righe che altrimenti corrispondono ad altri termini di ricerca.

Pertanto, una ricerca in modalità *booleana* che contiene solo termini preceduti da - restituisce un risultato vuoto.

Non restituisce "tutte le righe tranne quelle che contengono uno qualsiasi dei termini esclusi."

Ottimizzazione Query in MySQL

1. Introduzione all'ottimizzazione delle query
2. Ordine di esecuzione delle query in MySQL
3. Strategie di ottimizzazione basate sull'ordine di esecuzione

1. Introduzione all'ottimizzazione delle query

L'ottimizzazione delle query è cruciale per migliorare le prestazioni delle applicazioni database.

Un'ottima comprensione dell'ordine di esecuzione delle query SQL in MySQL è fondamentale per ottimizzare efficacemente le query.

2. Ordine di esecuzione delle query in MySQL

L'ordine reale con cui MySQL esegue i componenti di una query SQL:

FROM e JOIN:

Recupera le righe dalle tabelle di origine. Effettua le unioni specificate.

WHERE:

Filtra le righe in base alle condizioni specificate. Rimuove le righe che non soddisfano le condizioni.

GROUP BY:

Raggruppa le righe per una o più colonne. Aggrega i dati per i gruppi specificati.

HAVING:

Filtra i gruppi creati dal GROUP BY basandosi su condizioni di aggregazione.

SELECT:

Seleziona le colonne specificate per l'output finale.

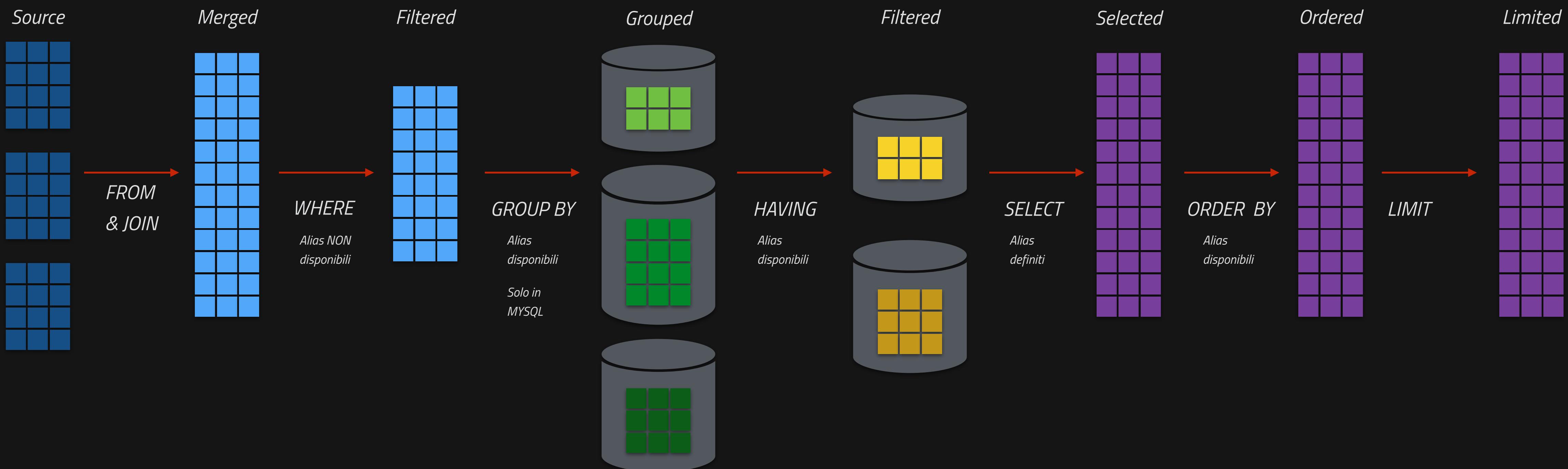
ORDER BY:

Ordina il risultato in base ai criteri specificati.

LIMIT:

Limita il numero di righe restituite dal risultato finale.

2. SCHEMA - Ordine di esecuzione delle query in MySQL



3. Strategie di ottimizzazione basate sull'ordine di esecuzione

Ottimizzazione di `FROM` e `JOIN`

Utilizzo di Indici: assicurati che le colonne utilizzate nelle condizioni di join siano indicizzate.

Riduzione delle tavole: unisci solo le tavole necessarie per ridurre il carico di elaborazione.

3. Strategie di ottimizzazione basate sull'ordine di esecuzione

Ottimizzazione di WHERE

Filtraggio precoce: applica condizioni di filtro il prima possibile per ridurre il numero di righe da elaborare.

Condizioni di filtraggio ottimali: *Utilizza indici sulle colonne utilizzate nelle condizioni WHERE.*

Evita funzioni su colonne indicizzate (perché rompono l'uso dell'indice)

-- Evita:

WHERE YEAR(data_nascita) = 2000 -- Non usa indice

-- Meglio:

WHERE data_nascita BETWEEN '2000-01-01' AND '2000-12-31' -- Usa indice

3. Strategie di ottimizzazione basate sull'ordine di esecuzione

Ottimizzazione di GROUP BY e HAVING

Riduzione dei gruppi: raggruppa solo quando necessario e cerca di ridurre il numero di gruppi.

Utilizzo di indici: indici su colonne di raggruppamento possono migliorare le prestazioni.

3. Strategie di ottimizzazione basate sull'ordine di esecuzione

Ottimizzazione di SELECT

Selezione delle colonne necessarie: **evita SELECT *** e seleziona *solo le colonne necessarie*.

Eliminazione delle duplicazioni: Utilizza DISTINCT solo quando necessario.

DISTINCT crea overhead se applicato su grandi dataset.

Verifica sempre se è realmente necessario.

3. Strategie di ottimizzazione basate sull'ordine di esecuzione

Ottimizzazione di ORDER BY e LIMIT

Indici per ordinamento: utilizza indici sulle colonne utilizzate per l'ordinamento.

Utilizzo di LIMIT: limita il numero di righe restituite per migliorare le prestazioni.

Transazioni

Consideriamo il caso di un caricamento di un ordine nel gestionale.

Le operazioni da eseguire sono: inserimento del record nella tabella *ordine* e inserimento dei dettagli nella tabella *ordine_dettaglio*. Ma se il sistema va in crash prima di eseguire la seconda query?

- ✓ `INSERT INTO ordini...`
- ✗ `INSERT INTO ordini_dettaglio...`

L'ordine risulterà incompleto e il dato non si troverà in uno stato consistente.

Le transazioni assicurano che le query che modificano valori in relazione tra loro ma appartenenti a più di una tabella del database siano eseguite come se si trattasse di un'unica query.

Se una query termina in errore allora tutte le tabelle coinvolte nell'operazione verranno ripristinate allo stato antecedente l'inizio della transazione. Diversamente le modifiche vengono registrate.

L'operazione di annullare le modifiche alle tabelle è detta **ROLLBACK**.

Si può richiamare il Roll-Back di una transazione *quando si trovano errori oppure in seguito a un'istruzione.* Altrimenti si esegue il **COMMIT**, e le modifiche verranno registrate in modo permanente.

Le transazioni, usate in tabelle InnoDB hanno le seguenti proprietà (**ACID**):

- **Atomicità.** Le transazioni sono operazioni indivisibili che riescono o falliscono nel loro insieme.
- **Consistenza.** Le tabelle del data base si trovano in uno stato consistente sia prima che dopo le transazioni.
- **Isolamento.** Le transazioni avvengono separatamente tra loro. InnoDB usa un bloccaggio a livello di record, in modo che i dati non siano modificati fino a quando l'operazione non è completa.
Ciò significa che queste transazioni vengono isolate o protette l'una dall'altra quando si tenta di modificare gli stessi dati. Le altre transazioni dovranno aspettare che la transazione sia completa.
- **Durabilità.** Al termine completo della transazione i dati vengono memorizzati permanentemente nel database.

AUTO COMMIT

Ogni volta che eseguiamo una query (SELECT, INSERT, UPDATE e DELETE) mysql esegue il **COMMIT**, e le modifiche vengono registrate in modo permanente, a meno che non venga rilevato un errore.

Ogni query è *wrappata* in una transaction con il commit implicito.

Questo comportamento di mysql dipende da una variabile chiamata AUTOCOMMIT:

```
SHOW VARIABLES LIKE 'autocommit'; -- ON
```

Quindi quando si esegue una istruzione mysql *wrappa* questa istruzione in una transaction ed esegue il commit a meno che non venga rilevato un errore.

LAST_INSERT_ID()

Questa funzione fa parte delle funzioni *informative* di mysql e restituisce il valore della colonna AUTOINCREMENT per l'ultimo INSERT¹:

```
INSERT INTO ordini_dettaglio  
VALUES(LAST_INSERT_ID(),9,10,120.00);
```

1) ATTENZIONE: Se inserisci più righe utilizzando un'unica istruzione INSERT, LAST_INSERT_ID() restituisce solo il valore generato per la prima riga inserita.

Vediamo la sintassi:

```
START TRANSACTION;  
{ blocco di istruzioni... }  
  
COMMIT / ROLLBACK;
```

Vediamo un esempio partendo dal caricamento di un ordine nel gestionale.

```
START TRANSACTION;  
  
INSERT INTO ordini(cliente_id, `data`, consegna)  
VALUE(7, '2022-05-03', 'da spedire');  
  
INSERT INTO ordini_dettaglio  
VALUES(LAST_INSERT_ID(), 9, 10, 120.00);  
  
COMMIT;
```

Una volta eseguito il COMMIT, le operazioni sono registrate in modo permanente.

Riproduciamo l'esempio caricando altro ordine nel gestionale.

Ora eseguiamo il codice riga per riga, e prima di eseguire la seconda query chiudiamo la connessione al database:

```
START TRANSACTION;
```

```
INSERT INTO ordini(cliente_id, `data`, consegna)
VALUE(7, '2022-05-03', 'da spedire');
```

SERVER CRASH

interruzione della connessione

```
ordini_dettaglio
INSERT_ID( ), 9, 10, 120.00;
```

```
COMMIT;
```

I dati non sono state modificati in modo permanente, il *crash* ha causato il ROLLBACK.

Operazioni concorrenti

Di solito molti user interagiscono contemporaneamente con la nostra applicazione e di conseguenza con i nostri dati.

La CONCORRENZA (Concurrency) può diventare un problema quando un utente modifica i dati che altri utenti stanno cercando di recuperare o modificare.

Vediamo come mysql gestisce la concorrenza per impostazione predefinita:

Attiviamo due connessioni (due shell, o due istanze di workbench) e creiamo due TRANSACTION

Connessione 1

```
START TRANSACTION;  
  
UPDATE clienti  
SET credito = credito + 100  
WHERE id = 1;  
  
COMMIT;
```

Connessione 2

```
START TRANSACTION;  
  
UPDATE clienti  
SET credito = credito + 100  
WHERE id = 1;  
  
COMMIT;
```

Eseguiamo riga per riga le istruzioni senza eseguire il commit.

La seconda TRANSACTION viene messa in attesa della conclusione della prima TRANSACTION, mysql blocca le righe interessate sino a conclusione dell'operazione o in seguito ad un *time-out*.

InnoDB offre 4 livelli di isolamento*:

- **READ UNCOMMITTED**: Una transazione può vedere modifiche alle righe fatte da un'altra transazione persino prima che sia avvenuto il commit.
- **READ COMMITTED**: Una transazione può vedere modifiche alle righe fatte da altre transazioni solo dopo che il commit di queste è avvenuto prima che la transazione iniziasse.
- **REPEATABLE READ** (default): Non considera nessuna delle modifiche prodotte da altre transazioni a prescindere dal fatto che queste ultime siano state terminate con il commit. Se una transazione esegue un'istruzione di selezione due volte, viene restituito lo stesso risultato in entrambi i casi, purché la transazione non modifichi essa stessa i dati .
- **SERIALIZABLE**: Questo livello di isolamento è simile a Repeatable Read ma isola le transazioni. Le righe che devono essere viste da una transazione non sono visibili da un'altra transazione fino a quando la transazione è completa.

```
SHOW VARIABLES LIKE 'transaction_isolation';
```

* per un approfondimento vedere la documentazione ufficiale mysql:

<https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html>

consultare il paragrafo 5 di questo pdf:

<http://users.dimmi.uniud.it/~angelo.montanari/SQL-Transazioni.pdf>

Problemi di concorrenza gestiti dai diversi livelli di isolamento:

	aggiornamenti persi	lettura sporche	lettura non ripetute	lettura fantasma
READ UNCOMMITTED				
READ COMMITTED		✓		
REPEATABLE READ	✓	✓	✓	
SERIALIZABLE	✓	✓	✓	✓

L'isolamento di default di InnoDB è **REPEATABLE READ**. Il livello di isolamento può essere impostato.

È possibile impostare le caratteristiche della transazione a livello *globale*, per la *sessione corrente* o solo per la *transazione successiva*:

```
SET TRANSACTION ISOLATION LEVEL nuovo_livello
```

Per usare efficacemente le transazioni bisogna inserirle in un programma in cui data una condizione viene eseguito il **COMMIT** altrimenti il **ROLLBACK**.

- Senza **SESSION** o **GLOBAL** si applica solo alla successiva transazione eseguita all'interno della sessione
- se aggiungiamo **SESSION** dopo l'istruzione **SET** il livello di isolamento resterà impostato a tutte le transazioni successive eseguite nella sessione corrente; es:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

- se aggiungiamo **GLOBAL** dopo l'istruzione **SET** il livello di isolamento riguarderà il server e verrà applicato a tutte le connessioni successive*.

```
SET GLOBAL TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

* dipende dai privilegi assegnati, le impostazioni a livello di server dipendono dal DBA

LETTURE SPORCHE

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

Per ogni shell create una transazione:

```
START TRANSACTION;
```

Nella prima shell effettuate l'aggiornamento del credito del cliente con id = 1, senza eseguire il COMMIT

```
UPDATE clienti SET credito = 200 WHERE id = 1;
```

Nella seconda shell effettuate una select sulla tabella cliente

```
SELECT credito FROM clienti WHERE id = 1;
```

Si crea una lettura sporca della seconda transazione se la prima transazione fallisce.

Il livello di isolamento READ COMMITTED risolve questo problema;

LETTURE RIPETUTE

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Per ogni shell create una transazione:

```
START TRANSACTION;
```

Nella prima shell effettuate una prima select del credito del cliente con id = 1

```
SELECT credito FROM clienti WHERE id = 1;
```

Nella seconda shell effettuate l'aggiornamento del credito del cliente con id = 1 ed eseguite il COMMIT

```
UPDATE clienti SET credito = 20 WHERE id = 1;  
COMMIT;
```

Nella prima shell effettuate una seconda select del credito del cliente con id = 1

```
SELECT credito FROM clienti WHERE id = 1;
```

La seconda lettura è *inconsistente*.

Il livello di isolamento REPEATABLE READ risolve questo problema;

LETTURE FANTASMA

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Per ogni shell create una transazione:

```
START TRANSACTION;
```

Nella prima shell effettuate una prima select

```
SELECT * FROM clienti WHERE provincia = 'mi';
```

Nella seconda shell effettuate l'aggiornamento della provincia del cliente con id = 1

```
UPDATE clienti SET provincia = 'mi' WHERE id = 1;
```

Nella prima shell effettuate una seconda select

```
SELECT * FROM clienti WHERE provincia = 'mi';
```

La seconda lettura non tiene conto (indipendentemente dal commit) del cambiamento della seconda transazione, provocando una lettura fantasma.

Il livello di isolamento SERIALIZABLE risolve questo problema;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Per ogni shell create una transazione:

```
START TRANSACTION;
```

Nella prima shell effettuate l'aggiornamento della provincia del cliente con id = 1

```
UPDATE clienti SET provincia = 'mi' WHERE id = 1;
```

Nella seconda shell effettuate una prima select

```
SELECT * FROM clienti WHERE provincia = 'mi';
```

La seconda lettura rimane in attesa che la prima transazione si concluda, prevenendo la lettura fantasma.

Il livello di isolamento SERIALIZABLE esegue ogni transazione una alla volta, isolando le transazioni.

I **SAVEPOINT** nelle transazioni, sono posizioni a cui il ROLLBACK si arresterà. Le operazioni della transazione che precedono i savepoint comporteranno un aggiornamento del database anche se la transazione fallisce: `SAVEPOINT nome_savepoint;`

```
START TRANSACTION;  
...primo blocco di istruzioni...  
SAVEPOINT sp1;  
...secondo blocco di istruzioni...  
ROLLBACK TO SAVEPOINT sp1;  
...terzo blocco di istruzioni...  
COMMIT;
```

In questo caso, dopo avere avviato la transazione abbiamo eseguito un primo blocco di aggiornamenti, seguito dalla creazione del **SAVEPOINT** col nome '**sp1**' ;

in seguito abbiamo eseguito un secondo blocco di aggiornamenti;

l'istruzione `ROLLBACK TO SAVEPOINT sp1` fa sì che "ritorniamo" alla situazione esistente quando abbiamo creato il **SAVEPOINT** : in pratica solo il secondo blocco di aggiornamenti viene annullato, e la transazione rimane aperta; una semplice `ROLLBACK` invece avrebbe annullato tutto e chiuso la transazione.

La `COMMIT` effettuata dopo il terzo blocco consolida gli aggiornamenti effettuati nel primo e nel terzo blocco.

Programmazione SQL

Le slide successive trattano gli argomenti relativi al miglioramento e ottimizzazione del database attraverso oggetti quali:

- Stored function
- Stored procedure
- Triggers

Prima dobbiamo però approfondire l'uso delle *variabili*, *dei cicli* e altre istruzioni SQL necessarie alla creazione degli oggetti sopra citati.

Programmazione SQL

Le slide successive si concentrano sulla programmazione SQL in MySQL, con l'obiettivo di migliorare e ottimizzare il database utilizzando strumenti avanzati.

Argomenti trattati

Impareremo a utilizzare i seguenti oggetti:

- Stored Function: funzioni memorizzate che restituiscono un valore e possono essere utilizzate nelle query.
- Stored Procedure: procedure memorizzate che eseguono operazioni complesse e possono restituire più risultati o effettuare modifiche.
- Triggers: eventi associati a operazioni DML (INSERT, UPDATE, DELETE) per automatizzare comportamenti specifici.

Fondamenti

Prima di affrontare questi argomenti, approfondiremo:

- L'uso delle variabili SQL (locali e di sistema) per memorizzare e gestire dati temporanei.
- I cicli (LOOP, WHILE, REPEAT) per eseguire operazioni ripetitive.
- Altre istruzioni SQL essenziali come:
 - Condizioni (IF, CASE) per prendere decisioni.
 - Gestione degli errori (SIGNAL, RESIGNAL) per controllare comportamenti anomali.

Variabili

MySQL può utilizzare le variabili in tre modi diversi, che sono riportati di seguito:

Variabile definita dall'utente:

- La variabile definita dall'utente ci consente di memorizzare un valore in un'istruzione e in seguito possiamo riferirlo a un'altra istruzione.
MySQL fornisce un'istruzione SET e SELECT per dichiarare e inizializzare una variabile.
Il nome della variabile definita dall'utente inizia con il simbolo @ .
- Le variabili definite dall'utente non fanno distinzione tra maiuscole e minuscole come @name e @NAME; entrambi sono uguali.
- Una variabile definita dall'utente dichiarata in una sessione non può essere vista da un'altra sessione.
- Le variabili utente non hanno tipo fisso: il tipo dipende automaticamente dal valore assegnato.
- La variabile definita dall'utente può avere una lunghezza massima di 64 caratteri

Esempio di USER VARIABLE

Il seguente esempio illustra il funzionamento delle variabili definite dall'utente:

```
SET @mediaPrezzo = (SELECT avg(prezzo) FROM libro);
SELECT * FROM libri WHERE prezzo > @mediaPrezzo;
```

L'esempio serve ad illustrarne il funzionamento, attraverso una subquery avremmo potuto semplicemente scrivere:

```
SELECT * FROM libri WHERE prezzo > (SELECT avg(prezzo) FROM libro)
```

Per vedere le variabili definite dall'utente bisogna interrogare il database performance_schema, se si hanno i privilegi.

```
SELECT * FROM performance_schema.user_variables_by_thread;
```

Variabili LOCALI

Una variabile è un oggetto dati denominato il cui valore può cambiare.

È necessario dichiarare una *variabile locale*, dentro uno Stored Program, prima di poterla utilizzare.

```
DECLARE variable_name datatype DEFAULT default_value;
```

- *variable_name*: il nome della variabile deve seguire le regole di denominazione di MySQL dei nomi delle colonne della tabella.
- *datatype*: Il tipo di dati della variabile e la sua dimensione. Una variabile può avere qualsiasi tipo di dati MySQL, come INT, VARCHAR, DATETIME, etc.

Quando si dichiara una variabile, il suo valore iniziale è NULL.

È possibile assegnare alla variabile un valore predefinito utilizzando DEFAULT.

Ad esempio, possiamo dichiarare una variabile denominata *total_sale* con il tipo di dati INT e il valore di default 0 come segue:

```
DECLARE total_sale INT DEFAULT 0;
```

MySQL consente di dichiarare due o più variabili che condividono lo stesso tipo di dati utilizzando una singola istruzione:

```
DECLARE x, y INT DEFAULT 0;
```

Abbiamo dichiarato due interi variabili *x* e *y*, e impostato i valori di default a zero.

Assegnazione di variabili

Una volta dichiarata una variabile, è possibile iniziare ad usarla.

Per assegnare a una variabile un valore, si utilizza l'istruzione SET:

```
DECLARE TOTAL_COUNT INT DEFAULT 0;  
SET TOTAL_COUNT = 10;
```

Il valore della variabile dopo l'assegnazione è TOTAL_COUNT=10.

Oltre l'istruzione SET, è possibile utilizzare l'istruzione SELECT INTO per assegnare a una variabile il risultato di una query che restituisce un valore scalare.

Vedere il seguente esempio:

```
DECLARE total_products INT DEFAULT 0;  
SELECT COUNT(*) INTO total_products  
FROM prodotti;
```

Abbiamo dichiarato una variabile denominata *total_products* e inizializzato il suo valore a 0.

Poi, abbiamo usato l'istruzione SELECT INTO per assegnare a *total_products* il numero di prodotti che abbiamo contato dalla tabella prodotti.

Ambito variabili (scope)

Una variabile ha il suo ambito di applicazione che definisce la sua durata.

- Se si dichiara una variabile in una *stored procedure/trigger/stored function* all'interno delle istruzioni BEGIN e END, la variabile non sarà più raggiungibile dopo l'istruzione END.
- È possibile dichiarare due o più variabili con lo stesso nome in ambiti diversi, perché una variabile è efficace solo nel proprio ambito.

Però la dichiarazione delle variabili con lo stesso nome in ambiti diversi non è una buona pratica di programmazione.

- Una variabile che inizia con il simbolo @ è una variabile di sessione dichiarata dallo user. È disponibile e accessibile fino al termine della sessione.
- Una variabile che inizia con il simbolo @@ è una variabile del sistema.

Variabile di sistema:

- MySQL contiene varie variabili di sistema che ne configurano il funzionamento e ogni variabile di sistema contiene un valore predefinito.
- Possiamo modificare alcune variabili di sistema in modo dinamico utilizzando l' istruzione SET in fase di esecuzione. Ci consente di modificare il funzionamento del server senza fermarlo e riavviarlo. La variabile di sistema può essere utilizzata anche nelle espressioni.
- Il server MySQL fornisce un sacco di variabili di sistema come i tipi GLOBAL e SESSION. Possiamo vedere la variabile GLOBAL durante tutto il ciclo di vita del server, mentre la variabile SESSION rimane attiva solo per una particolare sessione.

(es: @@lc_time_names, @@foreign_key_checks, @@sql_mode, @@log_bin_trust_function_creators, @@event_scheduler...)

Per visualizzare i valori correnti utilizzati dal server in esecuzione:

```
SHOW VARIABLES;
```

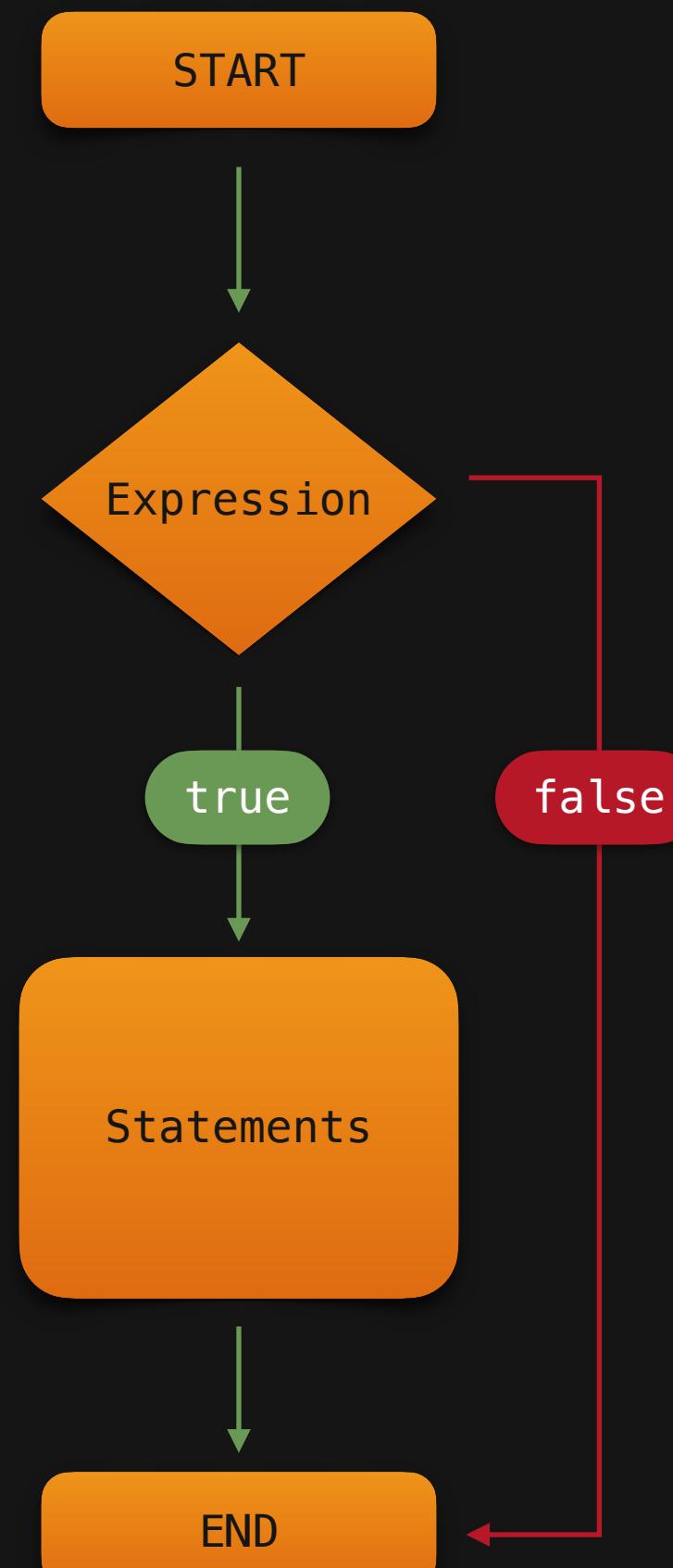
```
SHOW VARIABLES LIKE '%event_scheduler%';
```

istruzione IF

`IF expression THEN
statements;
END IF;`

Se l'espressione è TRUE, allora saranno eseguite le istruzioni.

In caso contrario, il controllo passa alla successiva istruzione che segue l'END IF.

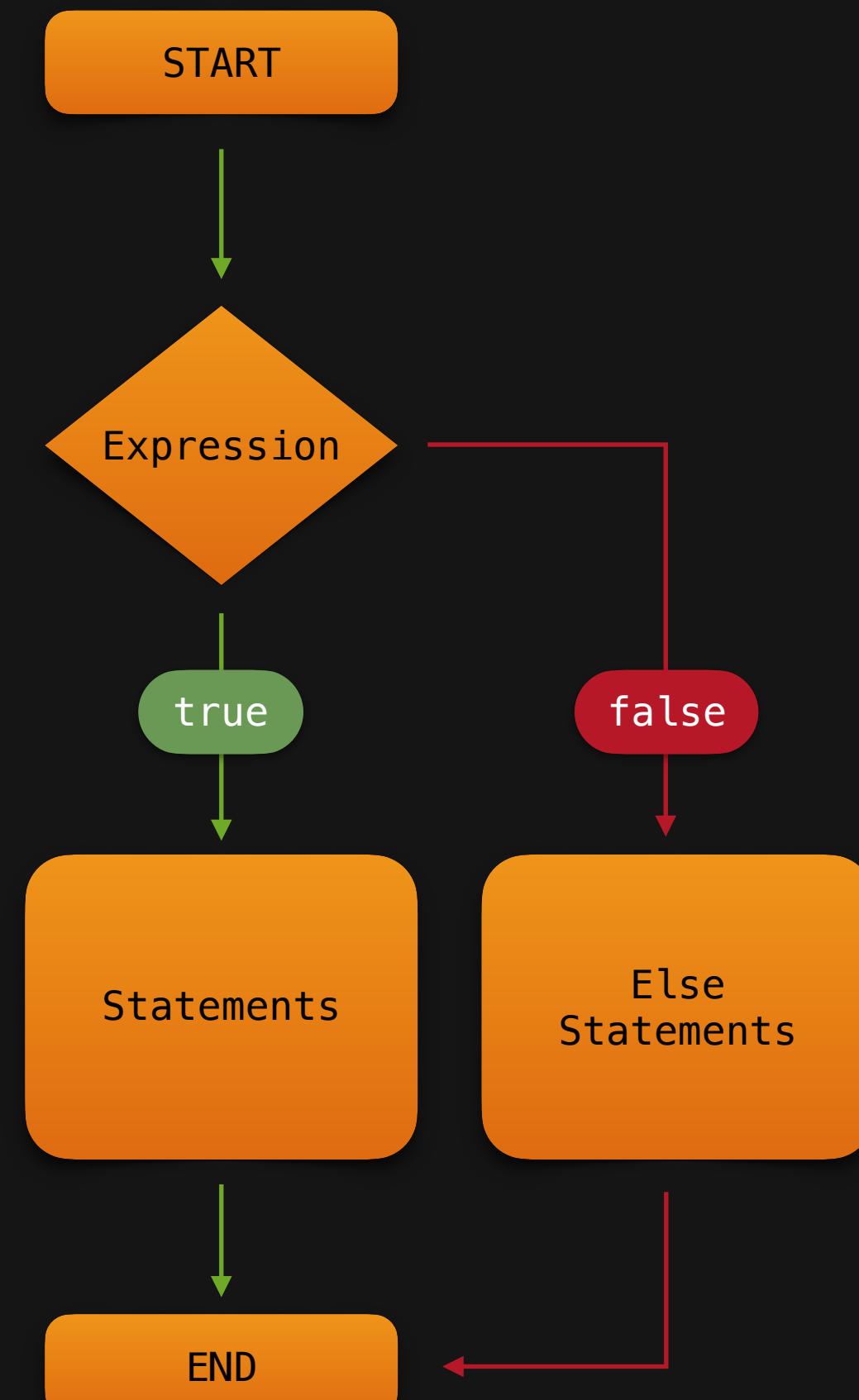


`IF expression THEN
statements;
ELSE
else-istruzioni;
END IF;`

Nel caso in cui si desidera eseguire le istruzioni quando l'espressione restituisce FALSE, si utilizza l'istruzione IF... ELSE

Se l'espressione è TRUE, allora saranno eseguite le istruzioni.

In caso contrario saranno eseguite le istruzioni del blocco ELSE.



istruzione IF

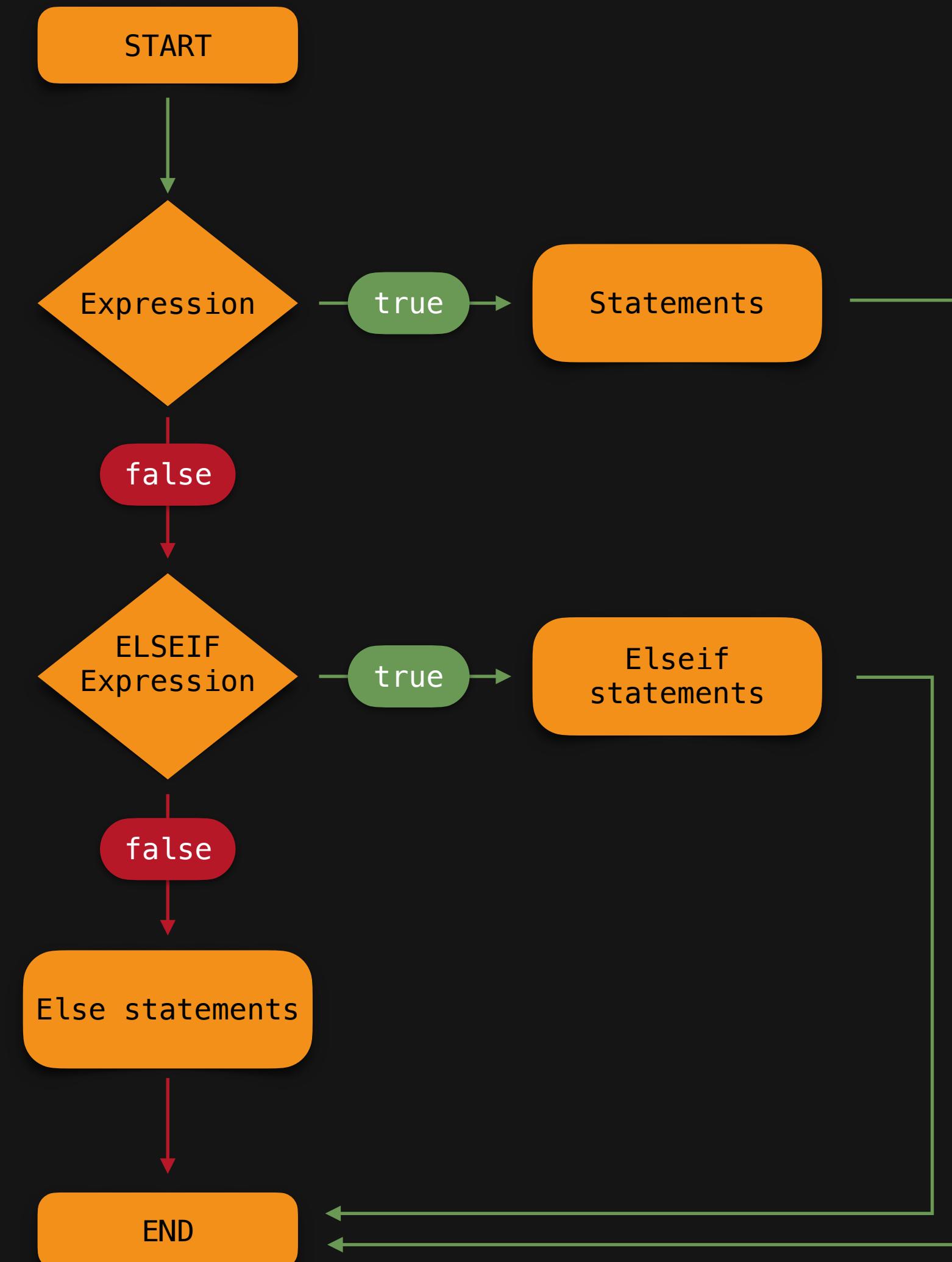
Se si desidera eseguire istruzioni condizionale sulla base di molteplici espressioni, si utilizza l'istruzione IF ... ELSEIF... ELSE

IF expression **THEN**
statements;

ELSEIF elseif-expression **THEN**
elseif-statements;

ELSE
else-statements;

END IF;



istruzione **CASE** semplice

Oltre **IF**, MySQL offre una dichiarazione condizionale alternativa chiamata **CASE**.

```
CASE case_expression
  WHEN when_expression_1 THEN statements
  WHEN when_expression_2 THEN statements
  ...
  ELSE statements
END CASE;
```

Si utilizza la semplice istruzione **CASE** per confrontare il valore di un'espressione con un insieme di valori unici.

La *case_expression* può essere qualsiasi espressione valida. **WHEN** confronta il valore della *case_expression* con *when_expression* per ogni clausola: *when_expression_1*, *when_expression_2*, ecc.

Quando il valore della *case_expression* e *when_expression_n* sono uguali, vengono eseguiti i comandi definiti dopo il **THEN**.

Quando nessuno dei *when_expression* nella clausola **WHEN** corrisponde al valore della *case_expression*, verranno eseguiti i comandi nella clausola **ELSE**.

ELSE è opzionale: se la si omette e non viene trovata alcuna corrispondenza, MySQL genererà un errore.

```
CASE regione_cliente #è una variabile dichiarata
  WHEN 'Piemonte' THEN tempo_cons = '1 giorno';
  WHEN 'Lombardia' THEN tempo_cons = '2 giorni';
  ELSE
    SET tempo_cons = '5 giorni';
END CASE;
```

istruzione **CASE** *searched*

L'istruzione **CASE** *simple* consente solo di abbinare un valore di un'espressione in confronto con un insieme di valori distinti.

Al fine di svolgere confronti più complessi come gli intervalli, si utilizza l'istruzione **CASE** *searched*.

L'istruzione **CASE** *searched* è equivalente alla istruzione **IF**, tuttavia, la sua costruzione è molto più leggibile.

```
CASE
    WHEN condition_1 THEN statements
    WHEN condition_2 THEN statements
    ...
    ELSE statements
END CASE;
```

MySQL valuta le condizioni nella clausola **WHEN** finché non trova una condizione il cui valore è vero, allora eseguirà i comandi corrispondenti nella clausola **THEN**.

Se nessuna condizione è **TRUE**, verrà eseguito il comando nella clausola **ELSE**.

Se non si specifica la clausola **ELSE** e nessuna condizione è vera, MySQL restituirà un messaggio di errore. Nell'esempio sotto non serve: le casistiche sono tutte comprese.

```
CASE
    WHEN credito > 2000 THEN
        SET livello_cliente = 'PLATINUM';

    WHEN (credito <= 2000 AND credito >= 1000) THEN
        SET livello_cliente = 'GOLD';

    WHEN credito < 1000 THEN
        SET livello_cliente = 'SILVER';

END CASE;
```

MySQL mette a disposizione le istruzioni `IF` e `CASE` per eseguire un blocco di codice SQL in base a determinate condizioni.

Quale utilizzare? Per la maggior parte degli sviluppatori, scegliere tra `IF` e `CASE` è solo una questione di preferenze personali.

Tuttavia, quando si decide di utilizzare `IF` o `CASE`, bisogna considerare i seguenti punti:

- *Una semplice dichiarazione CASE è più leggibile di IF quando si confrontano valori unici.* Inoltre, la semplice istruzione CASE è più efficiente di IF.
- *Quando si utilizzano espressioni complesse in base a più valori, l'istruzione IF è più facile da capire.*
- *Se si sceglie di utilizzare l'istruzione CASE, è necessario fare in modo che almeno una delle condizioni CASE sia vera.* In caso contrario, è necessario gestire gli errori.
- In alcune situazioni, utilizzare entrambi IF e CASE rende il codice più leggibile ed efficiente.

Altre istruzioni di ciclo:

WHILE

MySQL fornisce istruzioni di ciclo che consentono di eseguire un blocco di codice SQL più volte in base a una condizione. Ci sono tre istruzioni di ciclo in MySQL: **WHILE**, **REPEAT** e **LOOP**

```
WHILE expression DO  
    statements  
END WHILE
```

REPEAT

```
REPEAT  
    statements;  
UNTIL expression  
END REPEAT
```

LOOP, istruzioni LEAVE e ITERATE

L'istruzione **LEAVE** consente di uscire immediatamente dal loop senza attendere il controllo della condizione. L'istruzione LEAVE funziona come l'istruzione *break* in altre lingue come PHP, C / C ++, Java, etc.

L'istruzione **ITERATE** permette di saltare l'intero codice sotto di essa e iniziare una nuova iterazione.

L'istruzione **ITERATE** è simile alla istruzione *continue* in PHP, C / C ++, Java, etc.

SIGNAL

SIGNAL¹ è il modo per "restituire" un errore.

Questa istruzione può essere utilizzata ovunque, ma è generalmente utile se utilizzata all'interno di un *programma memorizzato* (triggers, stored function, stored procedure).

SIGNAL fornisce informazioni di errore a un gestore, a una parte esterna dell'applicazione o al client.

Inoltre, fornisce il controllo sulle caratteristiche dell'errore (numero errore, valore di SQLSTATE², messaggio).

Per segnalare un valore generico di SQLSTATE *bisogna utilizzare il codice '45000'*, che significa "*eccezione definita dall'utente, non gestita.*"

¹ <https://dev.mysql.com/doc/refman/8.0/en/signal.html>

² <https://dev.mysql.com/doc/refman/8.0/en/error-message-elements.html>

SIGNAL

SIGNAL¹ è il modo per "restituire" un errore personalizzato in MySQL. È particolarmente utile all'interno di programmi memorizzati (come trigger, stored procedure e stored function) per segnalare errori a un gestore interno o al client.

SIGNAL consente di:

1. personalizzare l'errore specificando:
 - Un codice SQLSTATE (ad esempio, '45000' per un'eccezione generica definita dall'utente).
 - Un messaggio di errore personalizzato.
 - Altri attributi come MYSQL_ERRNO, MESSAGE_TEXT, ecc.
2. Migliorare il controllo del flusso in programmi complessi, bloccando l'esecuzione con un errore quando vengono rilevate condizioni non valide.

Casi d'uso

- Validazione di parametri.
- Segnalazione di stati inconsueti o proibiti.
- Interruzione del flusso di esecuzione in presenza di errori logici.

¹ <https://dev.mysql.com/doc/refman/8.0/en/signal.html>

² <https://dev.mysql.com/doc/refman/8.0/en/error-message-elements.html>

Sintassi

```
SIGNAL SQLSTATE '45000'  
    SET MESSAGE_TEXT = 'Messaggio di errore personalizzato';
```

SQLSTATE '45000': indica un'eccezione definita dall'utente.

MESSAGE_TEXT: il testo descrittivo dell'errore.

Altri attributi opzionali:

MYSQL_ERRNO: un codice numerico personalizzato (facoltativo).

CONDITION: ulteriori dettagli sull'errore.

STORED FUNCTION

Le STORED FUNCTION definiscono vere e proprie funzioni, come quelle già fornite da MySQL.

Restituiscono un valore, e non possono quindi restituire result set, al contrario delle STORED PROCEDURES.

Sintassi:

```
CREATE FUNCTION nome_funzione(param1, param2,...)
    RETURNS datatype
    [NOT] DETERMINISTIC
BEGIN
    STATEMENTS
END$$
```

nota: x problemi di privilegi, da utente root: SET GLOBAL log_bin_trust_function_creators = 1;

1) Specificare il nome della funzione dopo l'istruzione: **CREATE FUNCTION**

2) Elencare tutti i parametri da passare alla funzione all'interno delle parentesi.

Per impostazione predefinita, tutti i parametri sono di tipo **IN**

3) Specificare il tipo di dati del valore restituito nell'istruzione: **RETURNS**

Può essere qualsiasi tipo di dati MySQL valido.

4) Decidere se una funzione memorizzata è deterministica o meno.

Una funzione **DETERMINISTIC** restituisce sempre gli stessi risultati se vengono forniti gli stessi valori di input.

Una funzione **NOT DETERMINISTIC** può restituire risultati diversi ogni volta che viene chiamata, anche quando vengono forniti gli stessi valori di input.

5) Scrivere il codice nel corpo (tra **BEGIN** e **END**) della funzione.

All'interno della sezione del corpo bisogna specificare almeno una dichiarazione **RETURN**: l' istruzione **RETURN** restituisce un valore al chiamante.

Ogni volta che viene raggiunta l' istruzione **RETURN** , l'esecuzione della funzione viene immediatamente interrotta.

CREAZIONE DI UNA STORED FUNCTION: anni()

Calcolo età a partire dalla data di nascita¹⁾

```
DELIMITER $$  
CREATE FUNCTION anni(p_data_nascita date) RETURNS tinyint  
NOT DETERMINISTIC  
BEGIN  
    DECLARE eta tinyint;  
    SET eta = TIMESTAMPDIFF(YEAR,p_data_nascita,CURDATE());  
    RETURN (eta);  
END$$  
DELIMITER ;
```

Richiamo la funzione

```
SELECT nome Nome, cognome Cognome, data_nascita 'Data di nascita', anni(data_nascita) `Età`  
FROM studenti  
ORDER BY cognome;
```

1) Questa funzione non è deterministica, vedi: <https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html>

CREAZIONE DI UNA STORED FUNCTION: ClientLevel()

Calcolo livello del cliente sulla base del campo credito

```
DELIMITER $$  
CREATE FUNCTION clientLevel(p_credit_value SMALLINT) RETURNS VARCHAR(8)  
DETERMINISTIC  
BEGIN  
    DECLARE client_level varchar(8);  
    CASE  
        WHEN p_credit_value >= 3000 THEN SET client_level = 'PLATINUM';  
        WHEN (p_credit_value < 3000 AND p_credit_value >= 2000) THEN client_level = 'GOLD';  
        WHEN (p_credit_value < 2000 AND p_credit_value >= 1000) THEN client_level = 'SILVER';  
        WHEN (p_credit_value < 1000 AND p_credit_value >= 1) THEN client_level = 'BASIC';  
        ELSE client_level = 'NONE';  
    END CASE;  
    RETURN (client_level);  
END$$  
DELIMITER ;
```

Richiamo la funzione

```
SELECT cognome Cognome, credito, clientLevel(credito) Livello  
FROM clienti  
ORDER BY Livello desc;
```

CREAZIONE DI UNA STORED FUNCTION: clienteLeveIf()

Calcolo livello del cliente sulla base del campo credito

```
DELIMITER $$  
CREATE FUNCTION clientLevelIf(p_credit_value SMALLINT) RETURNS VARCHAR(8)  
DETERMINISTIC  
BEGIN  
    DECLARE client_level varchar(8);  
    IF p_credit_value >= 3000 THEN  
        SET client_level = 'PLATINUM';  
    ELSEIF (p_credit_value < 3000 AND p_credit_value >= 2000) THEN  
        SET client_level = 'GOLD';  
    ELSEIF (p_credit_value < 2000 AND p_credit_value >= 1000) THEN  
        SET client_level = 'SILVER';  
    ELSEIF (p_credit_value < 1000 AND p_credit_value >= 1) THEN  
        SET client_level = 'BASIC';  
    ELSE  
        SET client_level = 'NONE';  
    END IF;  
    RETURN (client_level);  
END$$  
DELIMITER ;
```

Richiamo la funzione

```
SELECT cognome Cognome, clientLevelIf(credito) Livello  
FROM clienti  
ORDER BY Livello desc;
```

Per mostrare le funzioni personalizzate:

```
SHOW FUNCTION STATUS WHERE db = 'databasename';
```

```
SHOW FUNCTION STATUS WHERE db = 'miodb_1' OR db = 'miodb_2';
```

```
SHOW FUNCTION STATUS WHERE definer LIKE 'username%';
```

Per rimuovere la FUNCTION:

```
DROP FUNCTION nome_function;
```

Per vedere come è stata creata la FUNCTION:

```
SHOW CREATE FUNCTION nome_function;
```

Trigger

Un **trigger** è un insieme di istruzioni che si eseguono, o vengono attivate, quando un dato evento avviene su una tabella.

L'evento può essere **INSERT**, **UPDATE** o **DELETE**¹.

Principalmente servono per:

- creare tabelle di auditing(log) per i record che vengono modificati o eliminati;
- monitorare i campi di una tabella ed eventualmente generare eventi ad hoc;

Quando definiamo un trigger, stabiliamo per quale evento deve essere attivato (*inserimento, aggiornamenti o cancellazioni*) e se deve essere eseguito *prima* o *dopo* tale evento;

Il trigger stabilirà un'istruzione (o una serie di istruzioni) che saranno eseguite per ogni riga interessata dall'evento.

BEFORE INSERT
attivato prima che i dati vengano inseriti nella tabella

BEFORE UPDATE
attivato prima che i dati nella tabella vengano aggiornati

BEFORE DELETE
attivato prima che i dati vengano rimossi dalla tabella

AFTER INSERT
attivato dopo che i dati sono stati inseriti nella tabella

AFTER UPDATE
attivato dopo che i dati nella tabella sono stati aggiornati

AFTER DELETE
attivato dopo dati vengono rimossi dalla tabella

¹ Sono stati introdotti a partire da MySQL 5.0.2. Prima di MySQL versione 5.7.2 (prima di MariaDB 10.2.3), era possibile definire un massimo di 6 trigger per ogni tabella.

nota: x problemi di privilegi sui triggers, da utente root: SET GLOBAL log_bin_trust_function_creators = 1;

Il trigger è associato ad una tabella, ma fa parte di un database, per cui il suo nome deve essere univoco all'interno del db stesso.

È consigliato nominare i trigger utilizzando la seguente convenzione di denominazione:

(BEFORE | AFTER)_tableName_(INSERT| UPDATE | DELETE)

"before_ordini_dettaglio_update" è un trigger invocato prima che una riga della tabella ordini venga aggiornata

I Trigger non possono:

- Utilizzare le istruzioni: **SHOW**, **LOAD DATA**, **FLUSH** .
- Utilizzare le istruzioni: **COMMIT**, **ROLLBACK**, **START TRANSACTION**, **LOCK / UNLOCK TABLES**, **ALTER**, **CREATE**, **DROP**, **RENAME**, ecc
- Utilizzare le istruzioni: **PREPARE**, **EXECUTE**, etc (Utilizzare istruzioni SQL dinamiche).

Attenzione: I trigger non vengono attivati da azioni di chiavi esterne.

nota: da MySQL versione 5.1.4, un trigger può chiamare una stored procedure o stored function; non era possibile nelle versioni precedenti.

Ora vediamo la sintassi per creare un trigger relativamente all'azione BEFORE UPDATE per un ipotetico tracciamento delle modifiche ad una tabella:

```
CREATE TRIGGER nome_trigger  
BEFORE UPDATE ON nome_tabella  
FOR EACH ROW  
    INSERT INTO tabella_audit  
    SET action = 'update',  
        nome_campo1 = OLD.nome_campo1,  
        nome_campo2 = OLD.nome_campo2,  
        ...  
        nome_campo_n = OLD.nome_campo_n;
```

corpo del trigger

OLD e NEW sono due parole chiave utilizzate per recuperare il *vecchio* e il *nuovo* valore del record

- In un trigger definito per *INSERT*, è possibile utilizzare *solo la parola chiave NEW*.
- Nel trigger definito per *DELETE*, non vi è alcuna nuova riga, si usa *solo OLD*.
- Nel trigger *UPDATE*, *OLD* si riferisce alla riga prima di venire aggiornata e *NEW* si riferisce alla riga dopo essere stata aggiornata.

La tabella "tabella_audit" ovviamente deve già esistere e contenere tutti i campi che vogliamo tracciare più altri campi in cui scrivere le info relative all'azione, al momento in cui è stata eseguita, la persona che la esegue...

Ora vediamo la sintassi per creare trigger più complessi:

se si stanno inserendo più istruzioni dalla riga di comando, è necessario impostare temporaneamente un *nuovo delimitatore* (DELIMITER), per poter utilizzare i punto e virgola come delimitatore per le istruzioni che compongono il trigger.

Le diverse istruzioni SQL (corpo del trigger) vengono racchiuse tra BEGIN e END.

```
DELIMITER $$  
CREATE TRIGGER nome_trigger  
BEFORE UPDATE ON nome_tabella  
FOR EACH ROW  
BEGIN  
    INSERT INTO tabella01  
    SET attribute01 = 'valore';  
  
    INSERT INTO tabella02  
    SET attribute02 = 'valore';  
  
END$$  
DELIMITER ;
```

corpo del trigger

Ora creiamo la tabella *studenti_audit* e poi il trigger relativamente all'azione BEFORE UPDATE:

```
CREATE TRIGGER before_studenti_update
BEFORE UPDATE ON studenti
FOR EACH ROW
    INSERT INTO studenti_audit
    SET action = 'update',
        studente_id = OLD.id,
        nome = OLD.nome,
        cognome = OLD.cognome,
        genere = OLD.genere,
        indirizzo = OLD.indirizzo,
        citta = OLD.citta,
        provincia = OLD.provincia,
        regione = OLD.regione,
        email = OLD.email,
        data_nascita = OLD.data_nascita,
        ins_record = OLD.ins;
```

corpo del trigger

Eseguiamo degli aggiornamenti dalla tabella *studenti* e verifichiamo se vengono tracciate nella tabella *studenti_audit*

Ora creiamo il primo trigger relativamente all'azione AFTER DELETE:

```
CREATE TRIGGER after_studenti_delete
AFTER DELETE ON studenti
FOR EACH ROW
    INSERT INTO studenti_audit
    SET action = 'delete',
        studente_id = OLD.id,
        nome = OLD.nome,
        cognome = OLD.cognome,
        genere = OLD.genere,
        indirizzo = OLD.indirizzo,
        citta = OLD.citta,
        provincia = OLD.provincia,
        regione = OLD.regione,
        email = OLD.email,
        data_nascita = OLD.data_nascita,
        ins_record = OLD.ins;
```

corpo del trigger

Eseguiamo delle cancellazioni dalla tabella studenti e verifichiamo se vengono tracciate nella tabella *studenti_audit*

Ora vediamo l'uso di Trigger per monitorare i campi di una tabella ed eventualmente generare eventi ad hoc.

Creiamo il trigger relativamente all'azione AFTER INSERT

Aggiornamento della tabella *articoli* relativamente all'attributo "rimanenza" mediante un trigger evocato dalla tabella *ordini_dettaglio*:

```
CREATE TRIGGER after_od_insert_rimanenza  
AFTER INSERT ON ordini_dettaglio  
FOR EACH ROW  
    UPDATE articoli  
    SET rimanenza = rimanenza - NEW.quantita  
    WHERE id = NEW.articolo_id;
```

corpo del trigger

Eseguiamo degli **INSERT** nella tabella *ordini_dettaglio* e verifichiamo se la rimanenza degli articoli registrati nella tabella articoli viene scalata.

Creiamo il trigger relativamente all'azione AFTER DELETE

Aggiornamento della tabella *articoli* mediante un trigger evocato dalla tabella *ordini_dettaglio*:

```
CREATE TRIGGER after_od_delete_rimanenza
AFTER DELETE ON ordini_dettaglio
FOR EACH ROW
    UPDATE articoli
        SET rimanenza = rimanenza + OLD.quantita
    WHERE id = OLD.articolo_id;
```

Eseguiamo dei **DELETE** nella tabella *ordini_dettaglio* e verifichiamo se la rimanenza degli articoli registrati nella tabella *articoli* viene aggiornata.

Creiamo il trigger relativamente all'azione BEFORE UPDATE

Aggiornamento della tabella *articoli* mediante un trigger evocato dalla tabella *ordini_dettaglio*¹:

```
CREATE TRIGGER before_od_update_rimanenza
BEFORE UPDATE ON ordini_dettaglio
FOR EACH ROW
    UPDATE articoli
    SET rimanenza = rimanenza-(NEW.quantita - OLD.quantita)
    WHERE id = OLD.articolo_id;
```

Eseguiamo degli **UPDATE** nella tabella *ordini_dettaglio* e verifichiamo se la rimanenza degli articoli registrati nella tabella *articoli* viene aggiornata.

¹⁾ Attenzione: se *quantita* è definito unsigned produciamo errore nel caso di una riduzione (valore negativo di *quantita*).

Quindi eliminiamo unsigned su attributo *quantita* e, per mantenere il controllo sul valore positivo, aggiungiamo *constraint check quantita > 0* su *ordini_dettaglio*

Ora vediamo l'uso di Trigger per incrementare il credito del cliente quando inserisco un record in *ordini_dettaglio*.

Creiamo il trigger relativamente all'azione **AFTER INSERT**

Aggiornamento della tabella **clienti** relativamente all'attributo "*credito*" mediante un trigger evocato dalla tabella *ordini_dettaglio*:

```
CREATE trigger after_od_insert_credito
AFTER INSERT ON ordini_dettaglio
FOR EACH ROW

    UPDATE clienti c SET credito =
    credito + (NEW.quantita * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordini o
        WHERE NEW.ordine_id = o.id
    );
```

Eseguiamo degli **INSERT** nella tabella *ordini_dettaglio* e verifichiamo se il credito del cliente viene incrementato nella tabella *cliente*.

Ora vediamo l'uso di Trigger per ridurre il credito del cliente quando elimino un record in *ordine_dettaglio*.

Creiamo il trigger relativamente all'azione AFTER DELETE

Aggiornamento della tabella **clienti** relativamente all'attributo "*credito*" mediante un trigger evocato dalla tabella *ordini_dettaglio*:

```
CREATE trigger after_od_delete_credito
AFTER DELETE ON ordini_dettaglio
FOR EACH ROW

    UPDATE clienti c SET credito =
    credito - (OLD.quantita * OLD.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordini o
        WHERE OLD.ordine_id = o.id
    );
```

Eseguiamo il **DELETE** nella tabella *ordini_dettaglio* e verifichiamo se il credito del cliente viene ridotto nella tabella *cliente*.

Ora vediamo l'uso di Trigger per aggiornare il credito del cliente quando modifico un record in *ordine_dettaglio*¹.

Creiamo il trigger relativamente all'azione AFTER UPDATE

Aggiornamento della tabella **clienti** relativamente all'attributo "credito" mediante un trigger evocato dalla tabella *ordini_dettaglio*:

```
CREATE trigger after_od_update_credito
AFTER UPDATE ON ordini_dettaglio
FOR EACH ROW

    UPDATE clienti c SET credito =
    credito + ((NEW.quantita - OLD.quantita) * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordini o
        WHERE NEW.ordine_id = o.id
    );
```

Eseguiamo l'**UPDATE** di una quantità nella tabella *ordini_dettaglio* e verifichiamo se il credito del cliente viene aggiornato nella tabella *cliente*.

¹⁾ Attenzione: se *quantita* è definito unsigned produciamo errore nel caso di una riduzione (valore negativo di *quantita*).

Quindi eliminiamo unsigned su attributo *quantita* e, per mantenere il controllo sul valore positivo, aggiungiamo *constraint check quantita > 0* su *ordine_dettaglio*

Ora vediamo l'uso di Trigger per aggiornare il totale della fattura speso del cliente quando modifico un record in *ordine_dettaglio*¹.

TEST x aggiornare "totale" in ORDINI

Creiamo il trigger relativamente all'azione **AFTER UPDATE**

Aggiornamento della tabella **ordine** relativamente all'attributo "*credito*" mediante un trigger evocato dalla tabella *ordini_dettaglio*:

```
CREATE trigger after_od_update_tot_ordine  
AFTER UPDATE ON ordini_dettaglio  
FOR EACH ROW  
  
    UPDATE ordini o  
    SET totale = totale + ((NEW.quantita - OLD.quantita) * NEW.prezzo)  
    WHERE o.id = NEW.ordine_id;
```

Creare i trigger per
INSERT e DELETE

Eseguiamo l'**UPDATE** di una quantità nella tabella *ordini_dettaglio* e verifichiamo se il totale del ordine viene aggiornato nella tabella *ordini*.

¹⁾ Attenzione: se *quantita* è definito unsigned produciamo errore nel caso di una riduzione (valore negativo di *quantita*).

Quindi eliminiamo unsigned su attributo *quantita* e, per mantenere il controllo sul valore positivo, aggiungiamo *constraint check quantita > 0* su *ordine_dettaglio*

OTTIMIZZAZIONE TRIGGERS precedenti

I TRIGGERS creati per mantenere aggiornata la rimanenza e il credito cliente possono essere inseriti nello stesso TRIGGER.

Dovendo inserire più istruzioni SQL il trigger sarà complesso, quindi definisco un nuovo DELIMITER, e inserisco il corpo del TRIGGER tra BEGIN e END

Le tre slide successive mostrano il codice completo

TRIGGER unico per aggiornare rimanenza in tabella *articoli* e credito in tabella *clienti*

```
DELIMITER $$

CREATE TRIGGER after_od_insert_rimanenza_credito
AFTER INSERT ON ordini_dettaglio
FOR EACH ROW
BEGIN

    -- aggiorno rimanenza in articolo
    UPDATE articoli
    SET rimanenza = rimanenza - NEW.quantita
    WHERE id = NEW.articolo_id;

    -- aggiorno credito in cliente
    UPDATE clienti c SET credito =
    credito + (NEW.quantita * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordini o
        WHERE NEW.ordine_id = o.id
    );

END$$

DELIMITER ;
```

TRIGGER unico per aggiornare rimanenza in tabella *articoli* e credito in tabella *clienti*

```
DELIMITER $$

CREATE TRIGGER after_od_delete_rimanenza_credito
AFTER DELETE ON ordini_dettaglio
FOR EACH ROW
BEGIN

    -- aggiorno rimanenza in articolo
    UPDATE articoli
    SET rimanenza = rimanenza + OLD.quantita
    WHERE id = OLD.articolo_id;

    -- aggiorno credito in cliente
    UPDATE clienti c SET credito =
    credito - (OLD.quantita * OLD.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordini o
        WHERE OLD.ordine_id = o.id
    );

END$$

DELIMITER ;
```

TRIGGER unico per aggiornare rimanenza in tabella *articoli* e credito in tabella *clienti*

```
DELIMITER $$

CREATE TRIGGER before_od_update_rimanenza_credito
BEFORE UPDATE ON ordini_dettaglio
FOR EACH ROW
BEGIN

    -- aggiorno rimanenza in articolo
    UPDATE articoli
    SET rimanenza = rimanenza-(NEW.quantita - OLD.quantita)
    WHERE id = OLD.articolo_id;

    -- aggiorno credito in cliente
    UPDATE clienti c SET credito =
    credito + ((NEW.quantita - OLD.quantita) * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordini o
        WHERE NEW.ordine_id = o.id
    );

END$$

DELIMITER ;
```

TRIGGER unico per aggiornare rimanenza in tabella *articoli*, credito in tabella *clienti* e totale in tabella *ordini*

DELIMITER \$\$

```
CREATE TRIGGER before_od_update_rimanenza_credito_totale
BEFORE UPDATE ON ordini_dettaglio
FOR EACH ROW
BEGIN

    -- aggiorno rimanenza in articolo
    UPDATE articoli
    SET rimanenza = rimanenza-(NEW.quantita - OLD.quantita)
    WHERE id = OLD.articolo_id;

    -- aggiorno credito in cliente
    UPDATE clienti c SET credito =
    credito + ((NEW.quantita - OLD.quantita) * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordini o
        WHERE NEW.ordine_id = o.id
    );

    -- aggiorno totale in ordini
    UPDATE ordini o
    SET totale = totale + ((NEW.quantita - OLD.quantita) * NEW.prezzo)
    WHERE o.id = NEW.ordine_id;

END$$

DELIMITER ;
```

TEST totale

BEFORE INSERT: triggers che tiene conto delle quantità del magazzino. (TRIGGER complesso, uso di DELIMITER, BEGIN e END)

```
DELIMITER $$  
CREATE TRIGGER before_od_insert_rimanenza  
BEFORE INSERT ON ordini_dettaglio  
FOR EACH ROW  
BEGIN  
    DECLARE rimanenza_d TINYINT;  
    DECLARE msg VARCHAR(255);  
    SET rimanenza_d = (SELECT rimanenza FROM articoli WHERE id = NEW.articolo_id);  
    IF NEW.quantita <= rimanenza_d  
        THEN  
            UPDATE articoli  
            SET rimanenza = rimanenza_d - NEW.quantita  
            WHERE id = NEW.articolo_id;  
    ELSE  
        SET msg = CONCAT('Non ci sono abbastanza articoli in magazzino. Articoli in giacenza: ', rimanenza_d);  
        SIGNAL SQLSTATE VALUE '45000'  
        SET MESSAGE_TEXT = msg;  
    END IF;  
END$$  
DELIMITER ;
```

Vedere slide successiva per un trigger completo che tenga conto anche dello stato dell'ordine.

LIMITAZIONI del TRIGGER: i trigger non vengono attivati da azioni di chiavi esterne

Consideriamo la situazione: TRIGGER + chiave esterna ON DELETE CASCADE in *ordini_dettaglio*.

TRIGGER:

AFTER DELETE: triggers che aggiorna le quantità del magazzino in caso di eliminazione di un dettaglio in *ordini_dettaglio*.

```
CREATE TRIGGER after_od_delete_rimanenza
AFTER DELETE ON ordini_dettaglio
FOR EACH ROW

    UPDATE articoli
    SET rimanenza = rimanenza + OLD.quantita
    WHERE id = OLD.articolo_id;
```

...

CHAVE ESTERNA:

```
CONSTRAINT fk_od_ordine
FOREIGN KEY(ordine_id) REFERENCES ordini(id)
ON DELETE CASCADE
```

Quando *elimino una riga da ordini_dettaglio*, il trigger verrà attivato.

Quando *elimino una riga da ordini*, anche *le righe correlate in ordini_dettaglio verranno eliminate dalla chiave esterna*.

Ma **il TRIGGER non viene attivato**¹: MySQL esegue controlli di chiavi esterne PRIMA di richiamare qualsiasi trigger.

Quindi in questo caso le quantità di magazzino non vengono aggiornate

Il modo di risolvere questo problema è:

- *Modificare l'opzione ON DELETE CASCADE sulla chiave esterna: ON DELETE RESTRICT*
- *Aggiungere il trigger BEFORE DELETE nella tabella ordini che elimina le righe correlate in ordini_dettaglio*

Quando elimini una riga da ordine, il trigger verrà attivato e cancellerà le righe correlate *"direttamente"* in *ordini_dettaglio*.

Dal momento che i record vengono eliminati direttamente, il trigger su *ordini_dettaglio* verrà attivato.

¹ <https://dev.mysql.com/doc/mysql-reslimits-excerpt/5.7/en/stored-program-restrictions.html#stored-routines-trigger-restrictions>

Soluzione al problema della chiave esterna + trigger

1) modifico la FOREIGN KEY da CASCADE a RESTRICT:

```
ALTER TABLE ordini_dettaglio
DROP FOREIGN KEY fk_ordini_dettaglio_ordine;

ALTER TABLE ordini_dettaglio ADD CONSTRAINT fk_ordini_dettaglio_ordine
FOREIGN KEY(ordine_id) REFERENCES ordini(id)
ON DELETE NO ACTION ON UPDATE CASCADE;
```

l'eliminazione di un ordine sarà impedita se ci sono righe figlie in *ordini_dettaglio*.

Dovrò prima eliminare le righe in *ordini_dettaglio* (attivando così i trigger) e poi la riga in *ordini*.

Per fare questo creo un trigger che tenga conto anche dello stato dell'ordine: se l'ordine è *da spedire*, l'ordine potrà essere eliminato e di conseguenza le righe figli in *ordini_dettaglio* (tramite il trigger, (punto 2)).

Se l'ordine è stato spedito o consegnato l'operazione viene bloccata da trigger informando l'utente con un messaggio

2) Creo TRIGGER ad hoc:

Vedi slide successiva ->

BEFORE DELETE: trigger che tiene conto dello stato della spedizione prima di eliminare un ordine.

Chiave esterna su *ordini_dettaglio* riferita a *ordine* impostata a RESTRICT.

```
DELIMITER $$  
CREATE TRIGGER before_ordine_delete  
BEFORE DELETE ON ordini  
FOR EACH ROW  
BEGIN  
  
DECLARE spedizione ENUM('consegnato','da spedire','spedito');  
SET spedizione = OLD.consegna;  
IF spedizione = 'da spedire'  
    THEN  
        DELETE FROM ordini_dettaglio  
        WHERE ordine_id = OLD.id; #esegue l'eliminazione delle righe correlate  
    ELSE  
        SIGNAL SQLSTATE VALUE '45000'  
        SET MESSAGE_TEXT = "Non puoi eliminare l'ordine, gli articoli sono stati già spediti";  
    END IF;  
END$$  
DELIMITER ;
```

Eseguiamo dei DELETE nella tabella *ordini* e verifichiamo se la cancellazione avviene solo per gli ordini 'da spedire'.

Per visualizzare tutti i trigger nel database corrente, si utilizza*

```
SHOW TRIGGERS \G
```

Per visualizzare i trigger di uno specifico database

```
SHOW TRIGGERS FROM nome_database \G
```

Per visualizzare i trigger di una data tabella:

```
SHOW TRIGGERS FROM nome_database WHERE `table` = 'nome_tabella' \G
```

```
SHOW TRIGGERS FROM gestionale WHERE `table` = 'ordine_dettaglio' \G
```

```
SHOW TRIGGERS WHERE EVENT LIKE 'tipo_evento(INSERT,UPDATE,DELETE)' \G -- case sensitive
```

```
SHOW TRIGGERS WHERE EVENT LIKE 'UPDATE' \G
```

Per visualizzare come è stato creato un trigger:

```
SHOW CREATE TRIGGER nome_trigger \G
```

Per rimuovere un trigger:

```
DROP TRIGGER nome_trigger;
```

* = usate [\G] come delimiter per l'istruzione al posto di [;] in modo da farvi restituire le informazioni in formato scheda (verticali);
per altre opzioni della shell digitate \h

STORED PROCEDURES

Una STORED PROCEDURE (routine) è un set di istruzioni SQL memorizzate all'interno del database, un'oggetto del database che contiene un blocco di codice SQL.

Una stored procedure può essere invocata dai trigger, da altre stored procedure e da applicazioni scritte in Java, Python, PHP, ecc.

Sono disponibili a partire da mysql 5.0.

Si possono *richiamare* (CALL) le PROCEDURE per recuperare, aggiornare, inserire o cancellare i dati.

MySQL stored procedure **VANTAGGI**

- In genere le stored procedure contribuiscono ad **aumentare le prestazioni delle applicazioni.**

Una volta create, le stored procedure sono compilate e memorizzate nel database. Dopo la compilazione MySQL mette la stored procedure in una cache e la mantiene per ogni singola connessione.

Se un'applicazione utilizza una stored procedure più volte in una singola connessione, viene utilizzata la versione compilata, diversamente, la stored procedure funziona come una query.

- contribuiscono a **ridurre il traffico tra l'applicazione e il server** di database perché, invece di inviare più istruzioni SQL lunghe, l'applicazione deve inviare solo il nome e i parametri della stored procedure.
- Le stored procedure **sono riutilizzabili e trasparenti per tutte le applicazioni.** Le stored procedure espongono i dati del database in modo che gli sviluppatori non debbano sviluppare funzioni che sono già supportate nelle stored procedures.
- Le stored procedure **sono sicure.**

L'amministratore del database può concedere le autorizzazioni appropriate per le applicazioni che accedono alle stored procedure nel database senza dare autorizzazioni per le tabelle del database sottostante.

Memorizza e Organizza l'SQL

Esecuzione VELOCE

Maggior SICUREZZA

MySQL stored procedure: **Svantaggi**

- Utilizzare un **elevato numero** di stored procedure **aumenterà** notevolmente l'**utilizzo della memoria** di ogni connessione. Inoltre, aumenterà anche l'**utilizzo della CPU** perché il server di database non è progettato per le operazioni di business logic.
- Solo pochi sistemi di gestione di database consentono di eseguire il debug di stored procedure. Purtroppo, MySQL **non offre strutture per il debug di stored procedure**.
- Per sviluppare e mantenere le stored procedures è spesso necessario una **competenza specializzata** che non tutti gli sviluppatori di applicazioni possiedono. Questo può portare a problemi sia nella fase di sviluppo sia di manutenzione applicativa.

Scriviamo una semplice stored procedure denominata `getAllStudents()` per acquisire familiarità con la sintassi.

`getAllStudents()` seleziona tutti gli studenti dalla tabella studenti:

```
CREATE PROCEDURE getAllStudents()
    SELECT nome, cognome, genere, indirizzo, citta, provincia, regione, email, data_nascita
    FROM studenti ORDER BY cognome;
```

In questo caso, il nome della stored procedure è `getAllStudents`. Abbiamo messo le parentesi dopo il nome della stored procedure.

In questa stored procedure, si usa una semplice istruzione `SELECT` per ricavare i dati dalla tabella di studenti.

Per chiamare una stored procedure è possibile utilizzare la sintassi seguente:

```
CALL STORED_PROCEDURE_NAME();
```

Per chiamare la stored procedure `getAllStudents()`, è possibile utilizzare la seguente istruzione:

```
CALL getAllStudents();
```

Utente che può eseguire solo PROCEDURES

crea l'utente e assegna una password:

```
CREATE USER 'app_procedure'@'localhost' IDENTIFIED BY 'procedure';
```

GRANT EXECUTE

Assegna i privilegi che consentono l'utilizzo delle sole *stored procedure*. Non si accede alle tabelle.

```
GRANT EXECUTE  
ON gestionale.*  
TO 'app_procedure'@'localhost';
```

Accedi con questo utente, potrai richiamare solo le *procedure*. Non potrai usare altre istruzioni SQL.
La SELECT non è consentita.

Scriviamo altra semplice stored procedure denominata `getAllClients()`

`getAllClients()` seleziona tutti i clienti dalla tabella clienti:

```
CREATE PROCEDURE getAllClients()
    SELECT cognome, nome, telefono, email, indirizzo,
citta, provincia, regione, credito
    FROM clienti;
```

Ora possiamo chiamare la stored procedure:

```
CALL getAllClients();
```

MySQL stored procedures parametri

I parametri rendono le stored procedures più flessibili e utili.

IN: Quando si definisce un parametro IN in una stored procedure, l'applicativo deve passare un argomento per la stored procedure. Inoltre, il valore di un parametro IN è protetto. La stored procedures funziona solo sulla copia del parametro IN.

OUT: il valore di un parametro OUT può essere modificato all'interno della stored procedure e il nuovo valore viene passato all'applicativo.

INOUT: un parametro INOUT è la combinazione di parametri IN e OUT. Ciò significa che l'applicativo può passare l'argomento, e la stored procedure è in grado di modificare il parametro INOUT e restituire il nuovo valore all'applicativo.

La sintassi per definire un parametro nella stored procedure è la seguente:

`MODE [IN, OUT, INOUT] param_name PARAM_TYPE(PARAM_SIZE)`

Il `param_name` è il nome del parametro. Il nome del parametro deve seguire le regole di denominazione del nome della colonna in MySQL.

Il `PARAM_TYPE` è il tipo e la dimensione dei dati. Come una variabile, il tipo di dati del parametro può essere qualsiasi tipo di dati MySQL.

Ogni parametro è separato da una virgola [,] se la stored procedure ha più di un parametro.

MySQL stored procedure esempi di parametri: IN

Il seguente esempio illustra come utilizzare il parametro IN nella stored procedure *getClientOrders* che seleziona gli ordini di un determinato cliente.

```
CREATE PROCEDURE getClientOrders(IN p_id_cliente INT)
    SELECT c.nome, c.cognome, o.id, o.data
    FROM clienti c, ordini o
    WHERE c.id=o.cliente_id AND p_id_cliente=o.cliente_id;
```

Ora possiamo evocare la procedura attraverso la chiamata CALL e nome procedura per farci restituire l'elenco degli ordini di un determinato cliente:

```
CALL getClientOrders(1); restituisce elenco ordini cliente id = 1
CALL getClientOrders(3); restituisce elenco ordini cliente id = 3
```

Il seguente esempio illustra come utilizzare il parametro IN nella stored procedure getClientProv in base alla provincia.

```
CREATE PROCEDURE getClientProv(IN p_prov CHAR(2))
    SELECT *
    FROM clienti
    WHERE provincia = p_prov;
```

Ora possiamo evocare la procedura attraverso la chiamata CALL e nome procedura per farci restituire l'elenco dei clienti di una determinata provincia:

```
CALL getClientProv('TO'); restituisce elenco clienti di Torino
CALL getClientProv('MI'); restituisce elenco clienti di Milano
```

Il seguente esempio illustra come farsi restituisc un valore di DEFAULT in caso di parametro *null*.

```
DROP PROCEDURE IF EXISTS getClientProv;
DELIMITER $$

CREATE PROCEDURE getClientProv(IN p_prov char(2))
BEGIN
    IF p_prov IS NULL THEN
        SET p_prov = 'to';
    END IF;
    SELECT * FROM clienti WHERE provincia = p_prov;
END$$

DELIMITER ;
```

Ora possiamo testare la procedura attraverso la chiamata CALL e passare il valore *null*:

```
CALL getClientProv(null); restituisce elenco clienti della provincia di Torino
```

Potremmo però farci restituire tutti i record i caso di parametro null.

```
DROP PROCEDURE IF EXISTS getClientProv;
DELIMITER $$

CREATE PROCEDURE getClientProv(IN p_prov char(2))
BEGIN
    IF p_prov IS NULL THEN
        SELECT * FROM clienti;
    ELSE
        SELECT * FROM clienti WHERE provincia = p_prov;
    END IF;
END$$

DELIMITER ;
```

Ora possiamo testare la procedura attraverso la chiamata CALL e passare il valore *null*:

```
CALL getClientProv(null); restituisce tutti clienti
```

IFNULL(expr1, expr2)

Questa funzione fa parte delle funzioni di controllo di flusso come *case* e *if*:

IFNULL() restituisce *expr1* se l'*expr1* non è NULL,
altrimenti restituisce *expr2*.

```
SELECT
    cognome,
    IFNULL(data_nascita, 'manca data')
FROM studenti;
```

Ottimizzazione della procedura precedente: funzione IFNULL().

```
DROP PROCEDURE IF EXISTS getClientProv;
DELIMITER $$

CREATE PROCEDURE getClientProv(IN p_prov char(2))
BEGIN
    SELECT * FROM clienti
    WHERE provincia = IFNULL(p_prov, provincia);
END$$

DELIMITER ;
```

Ora possiamo testare la procedura attraverso la chiamata CALL e passare il valore *null*:

```
CALL getClientProv(null); restituisce tutti clienti
```

Ottimizzazione della procedura getClientOrders: funzione IFNULL().

```
DROP PROCEDURE IF EXISTS getClientOrders;

CREATE PROCEDURE getClientOrders(IN p_id_cliente INT)
    SELECT c.nome, c.cognome, o.id, o.data
    FROM clienti c, ordini o
    WHERE c.id=o.cliente_id
    AND o.cliente_id = IFNULL(p_id_cliente, o.cliente_id);
```

Ora possiamo testare la procedura attraverso la chiamata CALL e passare il valore *null*:

```
CALL getClientOrders(null); restituisce tutti gli ordini
```

Il seguente esempio illustra una procedura con due parametri di IN.

Questa PROCEDURE estraе uno specifico ordine di uno specifico cliente.

La PROCEDURE avrà due parametri di IN, uno per l'id del cliente e l'altro per l'id dell'ordine.

```
DROP PROCEDURE IF EXISTS getClientOrder;
CREATE PROCEDURE getClientOrder(
    p_cliente_id INT,
    p_ordine_id INT
)
SELECT cognome, o.id, `data`
FROM clienti c
    JOIN ordini o
    ON c.id=o.cliente_id
    AND o.cliente_id = p_cliente_id
    AND o.id = p_ordine_id
ORDER by cognome, o.id;
```

Ora possiamo testare la procedura attraverso la chiamata CALL e passare il valori:

```
CALL getClientOrder(3,2); restituisce l'ordine 2 del cliente con id 3
```

Ottimizziamo la PROCEDURE attraverso la funzione IFNULL() vista in precedenza.

```
DROP PROCEDURE IF EXISTS getClientOrder;
CREATE PROCEDURE getClientOrder(
    p_cliente_id INT,
    p_ordine_id INT
)
SELECT cognome, o.id, `data`
FROM clienti c
    JOIN ordini o
    ON c.id=o.cliente_id
    AND o.cliente_id = IFNULL(p_cliente_id, o.cliente_id)
    AND o.id = IFNULL(p_ordine_id, o.id)
ORDER by cognome, o.id;
```

Ora possiamo testare la procedura attraverso la chiamata CALL e passare il valori:

```
CALL getClientOrder(3,2); restituisce cliente con id = 3 e ordine con id = 2
```

```
CALL getClientOrder(3,null); restituisce tutti gli ordini del cliente con id = 3
```

```
CALL getClientOrder(null,null); restituisce tutti gli ordini
```

```
DROP PROCEDURE IF EXISTS getClientOrderDetails;
CREATE PROCEDURE getClientOrderDetails(
    p_cliente_id INT,
    p_ordine_id INT
)
SELECT c.cognome,
    o.id ordine,
    date_format(`data`, '%d/%m/%Y') `data emissione`,
    a.descrizione articolo,
    od.prezzo,
    od.quantita qt,
    od.prezzo*od.quantita `sub.totale`
FROM clienti c
    JOIN ordini o
    ON c.id=o.cliente_id
    JOIN ordini_dettaglio od
    ON o.id = od.ordine_id
    JOIN articoli a
    ON a.id = od.articolo_id
    AND o.cliente_id = IFNULL(p_cliente_id, o.cliente_id)
    AND o.id = IFNULL(p_ordine_id, o.id)
ORDER by cognome, a.descrizione;
```

Ora possiamo testare la procedura per ottenere il dettaglio di un particolare ordine di un dato cliente:

```
CALL getClientOrderDetails(3,4); restituisce cliente con id = 3 e ordine con id = 2
```

```
CALL getClientOrderDetails(3,null); restituisce tutti gli ordini del cliente con id = 3
```

```
CALL getClientOrderDetails(null,null); restituisce tutti gli ordini
```

Il seguente esempio illustra come utilizzare una PROCEDURE per l'UPDATE dei dati passando tre parametri.

Questa procedura aggiornerà il campo quantità della tabella *ordini_dettaglio* sulla base dell'id dell'ordine e dell'articolo_id passati dall'applicativo.

```
DROP PROCEDURE IF EXISTS updateOrderDetails;
CREATE PROCEDURE updateOrderDetails(
    p_ordine_id INT,
    p_articolo_id INT,
    p_quantita TINYINT
)
UPDATE ordini_dettaglio
    SET quantita = p_quantita
    WHERE ordine_id = p_ordine_id
    AND articolo_id = p_articolo_id;
```

Ora possiamo evocare la procedura attraverso la chiamata CALL e aggiornare la tabella ordine_dettaglio con la nuova quantità richiesta per uno specifico ordine esistente:

```
CALL updateOrderDetails(1,2,10);
## aggiorna la quantità (10) dell'articolo con id 2 dell'ordine con id 1
```

MySQL stored procedure esempi di parametri: OUT

Il seguente esempio illustra come utilizzare il parametro OUT nella stored procedure "countDelivered".

Questa procedura conta gli ordini sulla base dello stato di consegna:

Ha due parametri:

p_consegnato: il parametro **IN** è lo stato ordine;

p_total: il parametro **OUT** memorizza il numero di ordini per uno status specifico dell'ordine.

```
CREATE PROCEDURE countDelivered(IN p_consegnato VARCHAR(20), OUT p_total INT)
    SELECT COUNT(id)
    INTO p_total
    FROM ordini
    WHERE consegna=p_consegnato;
```

Ora possiamo evocare la procedura attraverso la chiamata CALL per farci restituire il totale dello stato degli ordini: consegnato, spedito, da consegnare:

```
CALL countDelivered('consegnato',@p_total); SELECT @p_total;
CALL countDelivered('spedito',@p_total); SELECT @p_total;
CALL countDelivered('da spedire',@p_total); SELECT @p_total;
```

MySQL Stored Procedures che restituiscono valori multipli

Per sviluppare stored procedures che restituiscono valori multipli, è necessario utilizzare i parametri **IN**, **INOUT** o **OUT**.

La seguente stored procedure restituisce il numero totale di ordini spedito, consegnato o da consegnare per un dato cliente.

```
DELIMITER $$  
CREATE PROCEDURE getDeliveryClient(  
    IN p_codice_cliente INT,  
    OUT p_consegnato INT,  
    OUT p_da_consegnare INT,  
    OUT p_spedito INT)  
  
BEGIN  
  
    -- consegnato  
    SELECT COUNT(*) INTO p_consegnato FROM ordini WHERE cliente_id = p_codice_cliente AND consegna = 'consegnato';  
  
    -- da consegnare  
    SELECT COUNT(*) INTO p_da_consegnare FROM ordini WHERE cliente_id = p_codice_cliente AND consegna = 'da consegnare';  
  
    -- spedito  
    SELECT COUNT(*) INTO p_spedito FROM ordini WHERE cliente_id = p_codice_cliente AND consegna = 'spedito';  
  
END $$  
DELIMITER ;
```

Oltre al parametro **IN**, la stored procedure prende 3 parametri **OUT** aggiuntivi: *consegnato*, *da_consegnare* e *spedito*.

All'interno della stored procedure utilizziamo un'istruzione **SELECT** con la funzione **COUNT** per ottenere il corrispondente totale degli ordini in base allo stato dell'ordine e assegnarlo al rispettivo parametro.

```
CALL getDeliveryClient(3,@p_consegnato,@p_da_consegnare,@p_spedito);  
SELECT @p_consegnato,@p_da_consegnare,@p_spedito;
```

```
DELIMITER $$

CREATE PROCEDURE getTimeShipping(
    IN p_codice_cliente INT,
    OUT p_tempo VARCHAR(50),
    OUT p_regione VARCHAR(50))
BEGIN
    DECLARE regione_cliente VARCHAR(50);

    SELECT regione INTO regione_cliente
    FROM clienti
    WHERE id = p_codice_cliente;

    CASE regione_cliente
        WHEN 'Piemonte' THEN SET p_tempo = '1 giorno', p_regione=regione_cliente;
        WHEN 'Lombardia' THEN SET p_tempo = '2 giorni', p_regione=regione_cliente;
        ELSE
            SET p_tempo = '5 giorni', p_regione=regione_cliente; /* o ='Fuori zona' */
    END CASE;

END$$

DELIMITER ;
```

Possiamo scrivere uno script di utilizzo della procedure come il seguente:

```
CALL getTimeShipping(3,@p_tempo,@p_regione);
SELECT @p_regione AS Regione, @p_tempo AS 'Tempo di spedizione';
```

Istruzioni per la gestione delle stored procedures

Elenco delle stored procedure:

```
SHOW PROCEDURE STATUS WHERE db = 'nome_db';
```

```
SHOW PROCEDURE STATUS WHERE name LIKE '%product%';
```

Visualizzazione del codice delle stored procedure:

```
SHOW CREATE PROCEDURE stored_procedure_name;
```

```
SHOW CREATE PROCEDURE getTimeShipping;
```

Eliminare una stored procedure:

```
DROP PROCEDURE IF EXISTS nome_procedura;
```

Pianificazione Eventi (Event Scheduler)

Un evento (programmato) è un oggetto le cui istruzioni vengono eseguite in risposta al trascorrere di un intervallo di tempo specificato.

Gli eventi sono simili al crontab di Unix (noto anche come " cron job ").

Gli eventi MySQL hanno le seguenti caratteristiche e proprietà principali:

- In MySQL, un evento è identificato in modo univoco dal suo nome e dal database a cui è assegnato.
- Un evento esegue un'azione specifica in base a una pianificazione. L'azione consiste in una o più istruzioni SQL.
Se ci sono più istruzioni SQL allora bisogna racchiudere le istruzioni tra `BEGIN` e `END`, come per i triggers.
- La tempistica di un evento può essere *una tantum* o *ricorrente*. Un evento occasionale viene eseguito una sola volta.
Alla pianificazione di un evento ricorrente possono essere assegnati un giorno e un'ora di inizio specifici, un giorno e un'ora di fine, entrambi o nessuno dei due.
- Come per `TRIGGERS`, `FUNCTION` e `PROCEDURE` non potete fare debug, dovete fare dei test.
Potete però *modificare* l'evento con l'istruzione `ALTER EVENT`
- L'istruzione di azione di un evento può includere la maggior parte delle istruzioni SQL consentite all'interno delle routine memorizzate.
Per le restrizioni, vedere la documentazione¹

¹ <https://dev.mysql.com/doc/refman/8.0/en/event-scheduler.html>

Verifichiamo se la variabile globale per creare gli eventi è abilitata o meno:

```
SELECT @@event_scheduler;
```

Se mysql risponde ON, possiamo creare gli eventi. Diversamente dovremmo digitare:

```
SET GLOBAL event_scheduler = ON;
```

La sintassi è la seguente:

```
CREATE EVENT nome_evento  
ON SCHEDULE  
    AT timestamp [+ INTERVAL interval] ...  
    | EVERY  
    interval  
        [STARTS timestamp [+ INTERVAL interval] ...]  
        [ENDS timestamp [+ INTERVAL interval] ...]  
    [COMMENT 'string']  
    DO istruzioni_SQL
```

interval:

quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE | WEEK | SECOND ...}

Approfondimento: <https://dev.mysql.com/doc/refman/8.0/en/events-overview.html>

Ora creiamo un evento, a solo dopo dimostrativo, che copia i record dalla tabella studente nella tabella bk_studente.

Ovviamente quest'ultima deve essere presente nel nostro database:

```
DELIMITER $$  
CREATE EVENT backup_studenti  
ON SCHEDULE  
AT CURRENT_TIMESTAMP + INTERVAL 1 MINUTE  
COMMENT 'Crea copia della tabella studente'  
DO BEGIN  
    TRUNCATE bk_studente;  
    INSERT INTO bk_studente  
    SELECT * FROM studente;  
END $$  
DELIMITER ;
```

Ora creiamo un evento che aggiorna le rimanenze nella tabella articolo sulla base delle quantità presenti in ordine_dettaglio (meglio TRIGGER):

L'evento aggiornerà il magazzino ogni giorno e partirà 1 minuto dopo la sua creazione.

```
CREATE EVENT update_magazzino
ON SCHEDULE
EVERY 1 DAY STARTS CURRENT_TIMESTAMP + INTERVAL 1 MINUTE
COMMENT 'Aggiorna le rimanenza nella tabella articolo'
DO
UPDATE articolo a
SET a.rimanenza = IF(
(SELECT sum(quantita)
FROM ordine_dettaglio od
WHERE od.articolo_id=a.id
GROUP BY a.id) > 0 ,
100 - (SELECT sum(quantita)
FROM ordine_dettaglio od
WHERE od.articolo_id=a.id
GROUP BY a.id),
100
);
```

Ora creiamo un evento che archivia gli ordini dell'anno passato spostandoli nelle tabelle *ordine_bk* e *od_bk*.

L'evento è riferito al database *gestionale* configurato con chiave esterna (*ordine_id*) eliminata da *ordine_dettaglio*.

Le tabelle *ordine_bk* e *od_bk* devono essere presenti nel nostro database:

```
DELIMITER $$  
CREATE EVENT archivia_ordini  
ON SCHEDULE  
EVERY 1 MINUTE  
COMMENT 'Sposta gli ordini vecchi'  
DO BEGIN  
  
    INSERT INTO ordine_bk  
    SELECT * from ordine WHERE year(data) != year(now());  
    INSERT INTO od_bk  
    SELECT * FROM ordine_dettaglio where ordine_id in (  
        SELECT id from ordine WHERE year(data) != year(now())  
    );  
    DELETE from ordine_dettaglio WHERE ordine_id in (  
        SELECT id from ordine WHERE year(data) != year(now())  
    );  
    DELETE from ordine WHERE year(data) != year(now());  
  
END $$  
DELIMITER ;
```

Per mostrare tutti gli eventi nel database, utilizzare la seguente istruzione.

```
SHOW EVENTS;  
SHOW EVENTS FROM database_name;
```

Se l'output non mostra tutti gli eventi è perché gli eventi vengono automaticamente eliminati quando scaduti.

Nei primo esempio (backup_studenti) abbiamo creato un evento permanente (EVERY 2 MINUTE)

Nel secondo esempio (archivia_ordini) si tratta di un evento occasionale (AT '2022-04-13 16:53:00') che scade quando la sua esecuzione è stata completata.

Per modificare gli eventi nel database, utilizzare:

```
ALTER EVENT ... seguito dalle nuove istruzioni
```

Per eliminare un evento nel database, utilizzare:

```
DROP EVENT [IF EXIST] event_name;
```

RISORSE

documentazione mysql: <http://dev.mysql.com/doc/>