IMMAGINAZIONE E LAVORO

INDICE argomenti

Cos'è un database
DBMS
Il Modello RELAZIONALE
Normalizzazione
Tipi di dato
SQL
  DDL
    1 parte
    2 parte
  DCL
  DML
  Query
Integrità referenziale
Union e JOIN
Funzioni
Raggruppamenti
Viste
Sub Query
Indici
Triggers
Stored Function
Events
Stored Procedures
Transaction
Installazione MySQL
Accesso DBMS (DBA)
Accesso DBMS (user)
Backup/Restoring (shell)

IMMAGINAZIONE E LAVORO

Strumenti

- MySQL Community Server 8.0.x
- Terminale/Shell
  - ambiente a riga di comando per interagire con il database tramite istruzioni SQL
- MySQLWorkbench: ambiente grafico per interagire con il database
- Visual Studio Code o altro editor (Sublime Text, Atom, Brackets…)
- Documentazione ufficiale MySQL:
  - https://dev.mysql.com/doc/refman/8.0/en

IMMAGINAZIONE E LAVORO

DATABASE

uso dei database relazionali

IMMAGINAZIONE E LAVORO

Cos’è un Database

È una raccolta di dati archiviati in un formato facilmente accessibile.

- I **database** o banche dati o base dati sono **collezioni di dati**, tra loro correlati, utilizzati per rappresentare una porzione del mondo reale. Sono strutturati in modo tale da consentire la gestione dei dati stessi in termini di inserimento, aggiornamento, ricerca e cancellazione delle informazioni.

- Il termine database in informatica sta ad indicare una **struttura organizzata di dati**.
  I database (o, più brevemente, DB), quindi, *sono gli archivi dove gli applicativi* (nel senso più ampio del termine) *memorizzano dei dati in modo persistente** al fine di poterli leggere successivamente e, se necessario, modificare o cancellare.

* *Nella programmazione informatica, la persistenza si riferisce in particolare alla possibilità di far sopravvivere delle strutture dati all'esecuzione di un singolo programma*

IMMAGINAZIONE E LAVORO

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),*
- ...;

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.

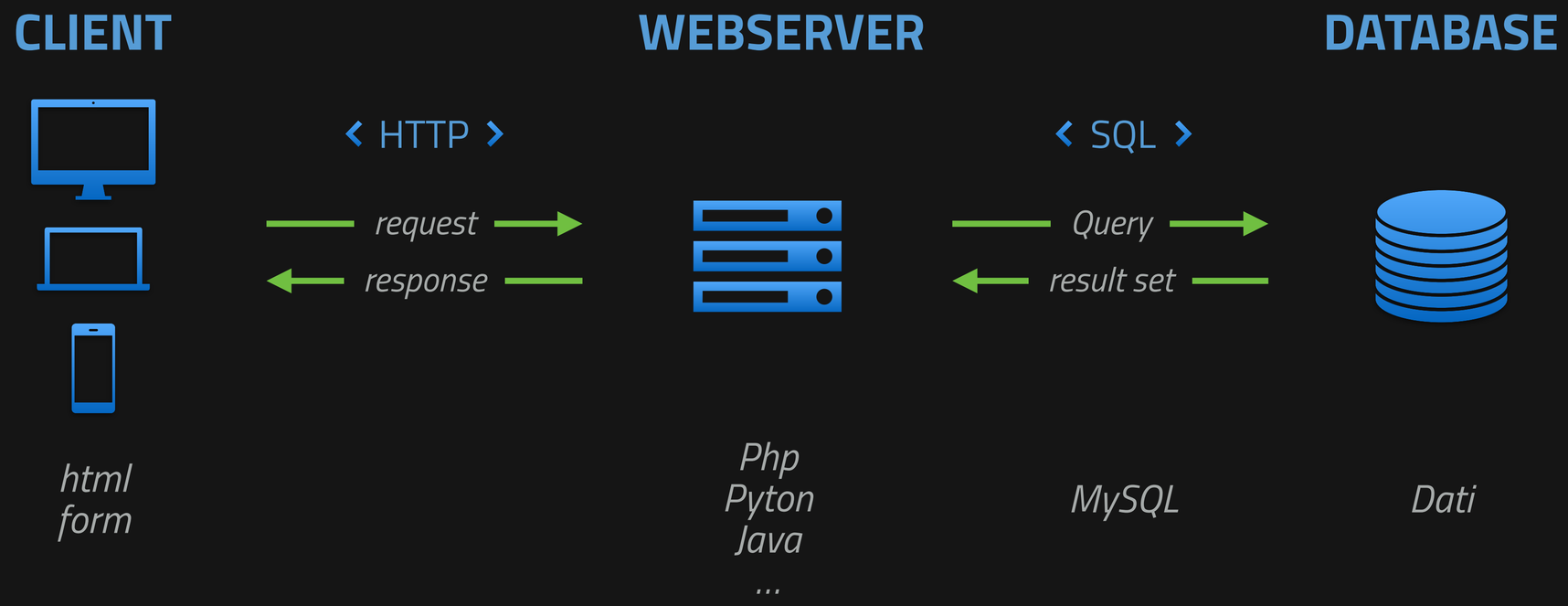
3. Il motore del database è in grado di gestire tutte le **connessioni simultanee** da parte degli utenti, ed utilizzare al meglio le prestazioni dell'hardware.

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**.

client-server

Esempio di richiesta dati attraverso un form via https



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**

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 tabelle 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.

IMMAGINAZIONE E LAVORO

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 tabelle, 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.

Esempio: una determinata tabella utilizza il *character set latin1* (quello maggiormente usato in Europa Occidentale) e la *collation latin1_general_cs*.
Tale collation è multilingue e "case sensitive" *(_cs)*, cioè tiene conto della differenza fra maiuscole e minuscole nell'ordinare o confrontare le stringhe.

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*.

Diagramma E-R, simboli

Entità

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

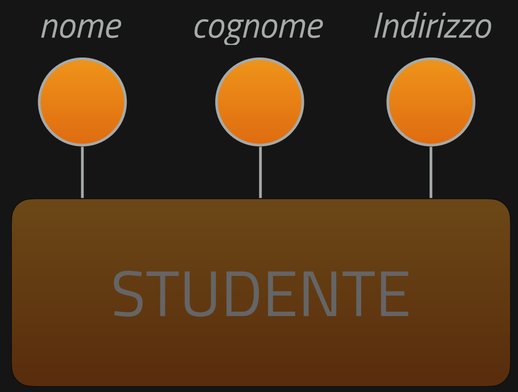
Rappresenta *classi di oggetti* (fatti, cose, persone, ...) che hanno *proprietà comuni* ed *esistenza autonoma* ai fini dell'applicazione di interesse.

- Identificata da un rettangolo

Attributi:

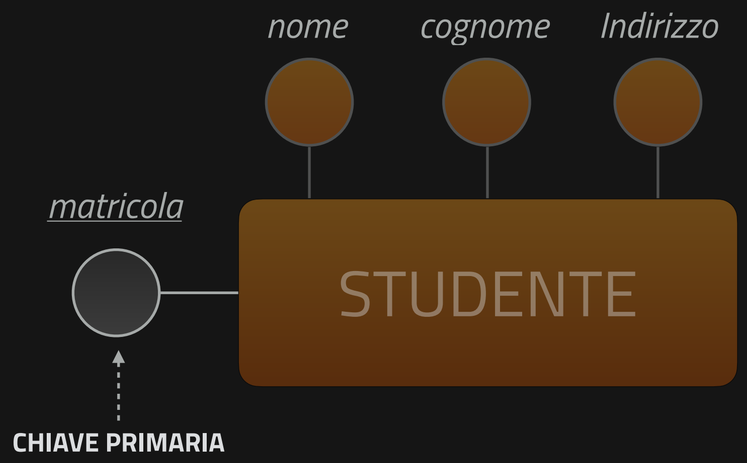
Caratteristiche specifiche di un'entità, utili (o necessarie) nella realtà da modellare

- 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

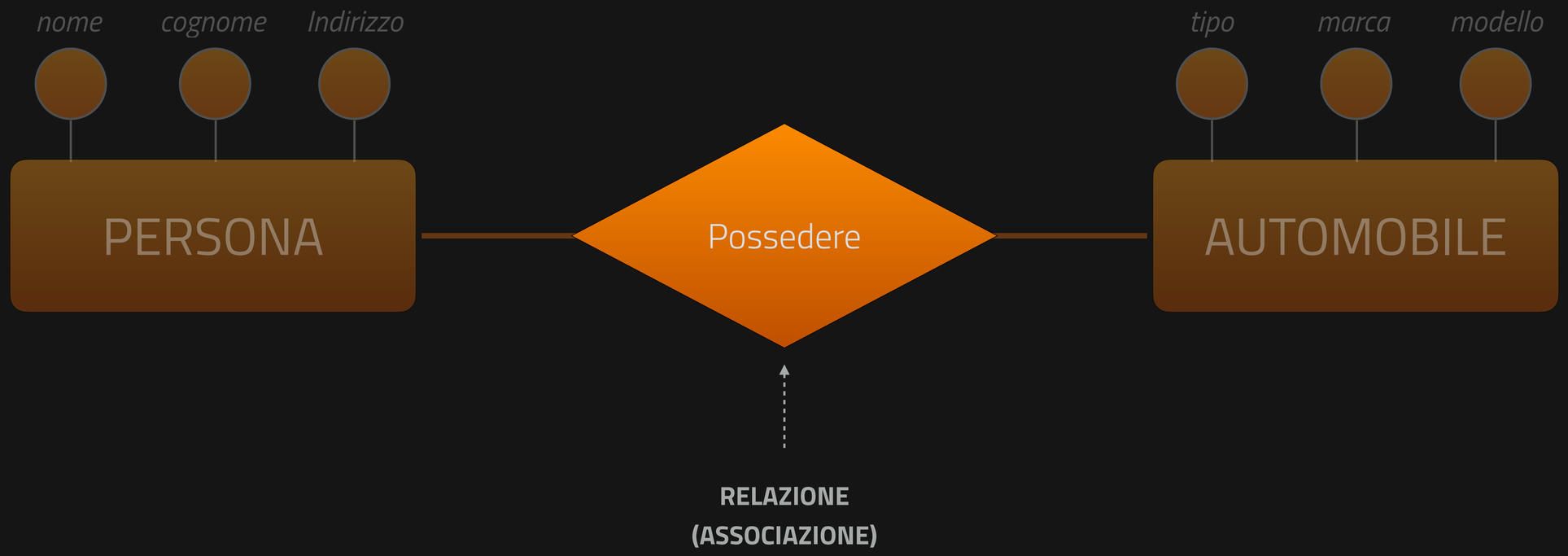
- non sono rappresentate nel Diagramma E-R ma si intendono contenute in ogni entità:
- *Carlo Rossi, via Verdi* è un'*istanza* dell'**entità** *ALUNNO* (**attributi**: *Nome*, *Cognome*, *Indirizzo*)

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:



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

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**)

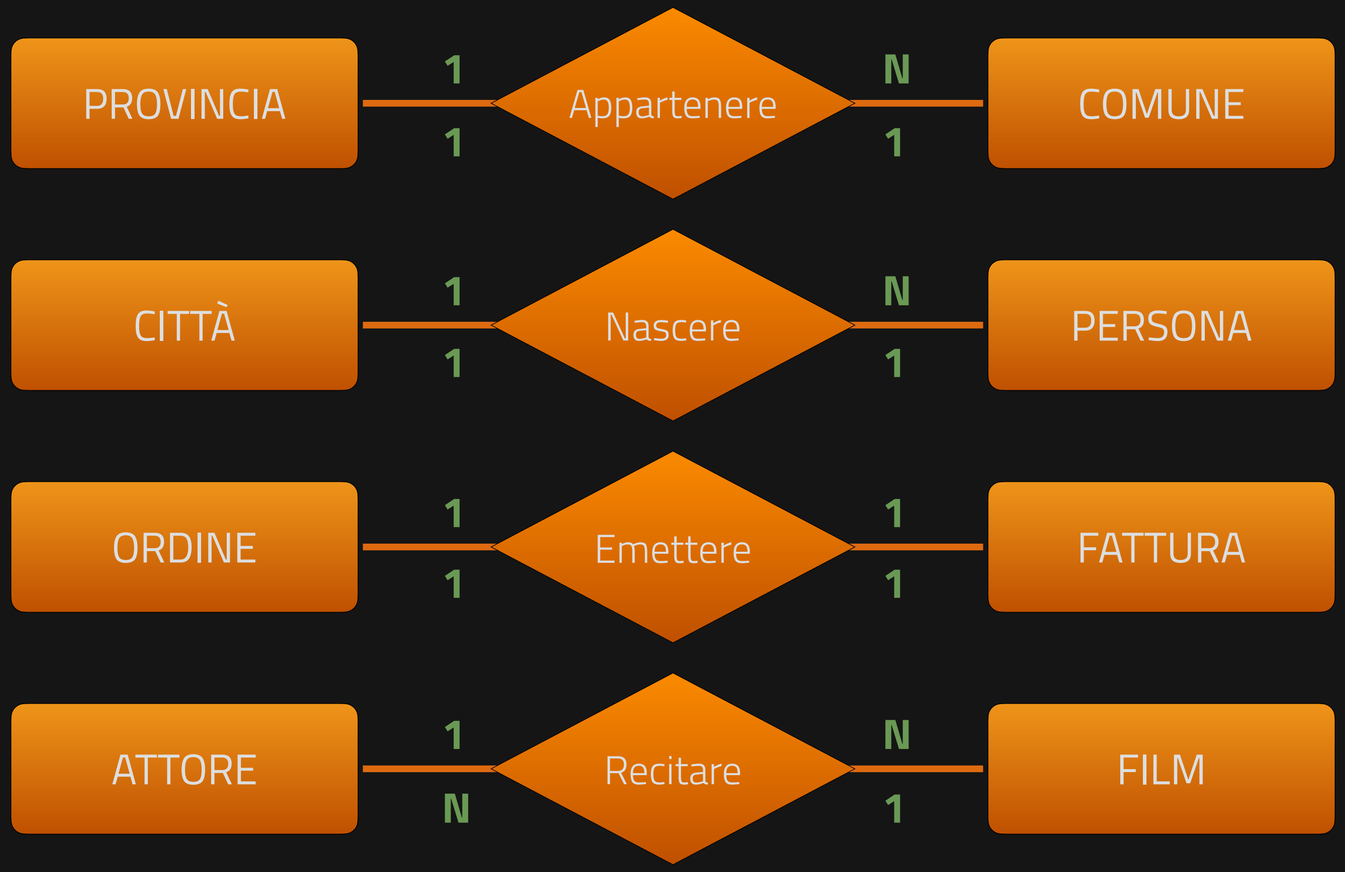
- Ad ogni classe *appartiene* più di un alunno
- Un alunno *appartiene* ad una singola classe

Università

Relazione N,N (**molti a molti**)

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

Esempi


PROVINCIA
1
1
Appartenere
N
1
COMUNE
CITTÀ
1
1
Nascere
N
1
PERSONA
ORDINE
1
1
Emettere
1
1
FATTURA
ATTORE
1
N
Recitare
N
1
FILM

Esempio

Database per la catalogazione e la gestione dei libri di una libreria.

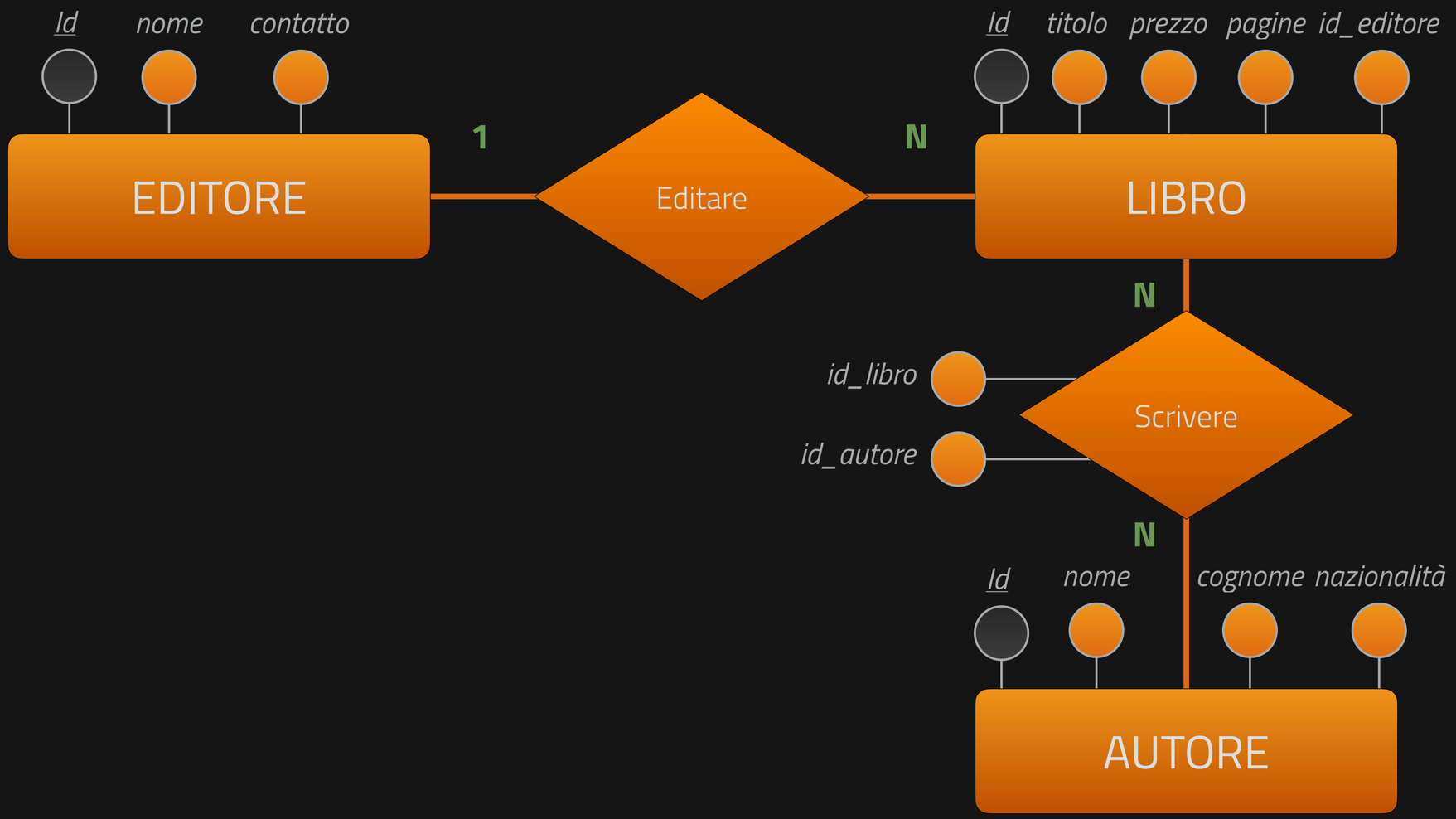
L'elemento centrale del nostro database è il libro che avrà alcune caratteristiche: titolo, prezzo, numero di pagine, la casa editrice, l'autore…

- *Harry Potter e la Camera dei Segreti, di J. K. Rowling; ED: Salani.*

Definiamo lo schema concettuale per rappresentare graficamente e sinteticamente la realtà analizzata. Identifichiamo le entità coinvolte e gli attributi necessari per ciascuna entità e le relazioni che intercorrono tra le varie entità.

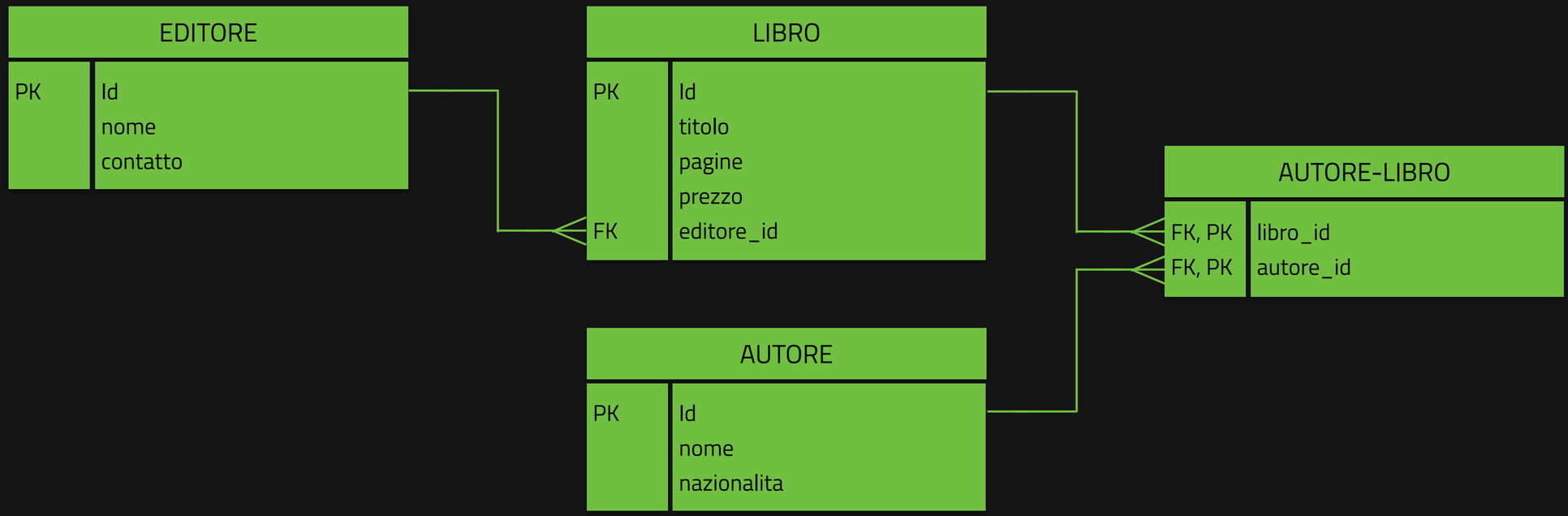
Nel database sono presenti diverse **entità** (libri, case editrici, autori) con *attributi specifici* (il titolo del libro, l'indirizzo della casa editrice, la biografia dell'autore) e *chiavi di identificazione univoche* (ad esempio un campo numerico incrementale, il codice ISBN del libro, il codice fiscale dell'autore…).

Diagramma E-R



Modello logico

Simbologia delle relazioni



Rappresentazione dell'informazione in tabelle nel database secondo il modello relazionale dell'esempio relativo alla catalogazione di libri:

LIBRO

• id	titolo	pagine	prezzo	editore_id
1	Il paradiso degli Orchi	128	6,00	2
2	Il gabbiano Jonathan Livingston	150	4,50	3
3	L'isola di Arturo	220	18,80	1
4	Caro Bogart	115	7,20	2

LIBRO-AUTORE

libro_id	autore_id
1	2
2	3
3	1
4	2

AUTORE

• id	cognome	Nome	Nazionalità
1	Morante	Elsa	ITA
2	Coe	Jonathan	ENG
3	Bach	Richard	USA
4	Pennac	Daniel	FRA

EDITORE

• id	nome	contatto
1	Einaudi	info@einaudi.it
2	Feltrinelli	amm@feltrinelli.i
3	BUR	contatti@bur.it

Legame logico tra tabelle

LIBRO

Chiave Esterna

id	titolo	editore_id
1	Harry Potter e la Camera dei Segreti	2
2	Guerra e Pace	1
3	Harry Potter e il Calice di Fuoco	2
4	Ulisse	3
5	Marcovaldo	2

Campo Chiave

EDITORE

id	nome
1	BUR
2	Mondadori
3	Sellerio
4	Salani

Join tra tabelle

id	titolo	editore_id	id	nome
1	Harry Potter e la Camera dei Segreti	2	2	Mondadori
2	Guerra e Pace	1	1	BUR
3	Harry Potter e il Calice di Fuoco	2	2	Mondadori
4	Ulisse	3	3	Sellerio
5	Marcovaldo	2	2	Mondadori

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:

- *tutte le tuple* della relazione *hanno lo stesso numero di attributi*
- *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.

• 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

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).

Per fare un 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
1234	M01	6
1234	L02	7
1235	L02	8

• Codice Matricola	Nome Matricola
1234	Rossi Alberto
1235	Verdi Mario

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"*.

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
GLSTMT59U66P109B	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
LBRRSS79Y12T344A	BB01
GNNBNCT84A11L611B	BB01
LBRMNN79E64A112A	BB01
GLSTMT59U66P109B	AE02

• Codice Corso	Insegnante
BB01	Marco
AE02	Federica

IMMAGINAZIONE E LAVORO

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** and **Time** (data e ora)
- **JSON**

Dati numerici: interi

Tipo	Intervallo di valori	Solo se positivi (UNSIGNED)
BIT[(M)]	Da 1 a 64	/
TINYINT[(M)] (1 byte)	da -128 a +127	da 0 a 255
SMALLINT[(M)] (2 byte)	da -32 768 a +32 767	da 0 a 65 535
MEDIUMINT[(M)] (3 byte)	da -8 388 608 a +8 388 607	da 0 a 16 777 215
INT[(M)] (4 byte)	da -2 147 483 648 a +2 147 483 647	da 0 a 4 294 967 295
BIGINT[(M)] (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.

L'indicazione del parametro M sugli interi *non influisce sui valori memorizzabili*, ma rappresenta la *lunghezza minima visualizzabile* per il dato.

Se il valore occupa meno cifre, viene riempito a sinistra con degli spazi, o con degli zeri nel caso di ZEROFILL.

Dati numerici a virgola mobile

Tipo	Tipo (sintassi deprecata)*	spazio
FLOAT	FLOAT(M,D)	4byte
DOUBLE	DOUBLE(M,D)	8byte
DECIMAL	DECIMAL(M,D)	il peso di M + 2byte

FLOAT garantisce precisione fino a circa 7 cifre dopo la virgola, *DOUBLE fino a 15* circa).

Come si vede dalla tabella questi tre tipi di dati prevedono un doppio valore tra parentesi (opzionale):

- la **M** indica il numero complessivo di cifre ammesse,
- la **D** il numero di Decimali.

Se ad es. passiamo il numero 12.345678 ad una colonna FLOAT(6,4), tale numero diverrà 12.3456

I dati *DECIMAL* rappresentano infine numeri "esatti", con M(65) cifre totali di cui D(30) decimali.

I valori di default sono 10 per M e 0 per D (da usare con i dati monetari o per tutti i casi in cui ci serve il numero esatto).

*A partire da MySQL 8.0.17 la sintassi FLOAT(M,D), DOUBLE(M,D) verrà deprecata

IMMAGINAZIONE E LAVORO

Dati stringa

Tipo e lunghezza massima consentita

```
nome VARCHAR(20)
codiceFiscale CHAR(16)
titolo TINYTEXT
messaggio TEXT
```

Tipo	Lunghezza massima
CHAR(n)	256 caratteri
VARCHAR(n)	65.536 caratteri
BINARY(b)	256 byte
VARBINARY(b)	65.536 byte
TINYTEXT	256 caratteri
TINYBLOB	256 caratteri
TEXT	65.536 caratteri
BLOB	65.536 caratteri
MEDIUMTEXT	16.777.216 caratteri
MEDIUMBLOB	16.777.216 caratteri
LONGTEXT	4.294.967.296 caratteri
LONGBLOB	4.294.967.296 caratteri
ENUM('value1','value2',...)	65.536 caratteri
SET('value1','value2',...)	64 gruppi massimo

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 caratteri M
- M deve essere compreso tra 1 e 65535
- prende *1* + *c* byte (per M ≤ 255) o *2* + *c* (per 256 ≤ M ≤ 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.

```
coniugato ENUM('S','N')
```

La colonna coniugato accetterà solamente i valori S o N. 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.

MySQL interpreta i valori dell'anno a 2 cifre utilizzando queste regole

- I valori dell'anno nell'intervallo 00-69 diventano 2000-2069.
- I valori dell'anno nell'intervallo 70-99diventano 1970-1999.

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
```

IMMAGINAZIONE E LAVORO

DATETIME o **TIMESTAMP**?

con TIMESTAMP, sei in grado di servire gli stessi dati di data e ora in fusi orari diversi, direttamente dal database.

Tieni presente che per impostazione predefinita il fuso orario applicato è il fuso orario del server.

Se vuoi servire i tuoi dati di data e ora allo stesso modo *indipendentemente dai fusi orari*, puoi usare il tipo DATETIME.

Altrimenti, puoi utilizzare TIMESTAMP e fornire i dati in base al fuso orario.

TIMESTAMP

Tipo	Formato
TIMESTAMP(14)	AAAAMMGGHHMMSS
TIMESTAMP(12)	AAMMGGHHMMSS
TIMESTAMP(10)	AAMMGGHHMM
TIMESTAMP(8)	AAAAMMGG
TIMESTAMP(4)	AAMM
TIMESTAMP(2)	AA

Questo campo ammette un intervallo da '1970-01-01 00:00:01'UTC a '2038-01-19 03:14:07' UTC.

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
```

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 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 VAR BINARY 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) - E' utilizzato per creare un'indice nella tabella ai fini di migliorare le performances di accesso ai dati.

IMMAGINAZIONE E LAVORO

INDEX

Indici

- 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

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
- **Query Language**: permette di porre interrogazioni al db

IMMAGINAZIONE E LAVORO

DCL

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

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

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

IMMAGINAZIONE E LAVORO

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
```

IMMAGINAZIONE E LAVORO

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;
```

IMMAGINAZIONE E LAVORO

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 sollevato 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 10

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 8.0\bin
```

A questo punto possiamo accedere al servizio avviando mysql (vedi slide 67)

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 67)

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 è disponibile anche all'utente per ottenere informazioni sul proprio database.

DDL

Data Definition Language (1parte)

Creare un database

Una volta effettuato l'accesso possiamo eseguire l'istruzione CREATE DATABASE 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 databaseName
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 databaseName;
```

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 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

IMMAGINAZIONE E LAVORO

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_user'@'localhost' IDENTIFIED BY 'user_2023!';
```

password : password associata all'utente che stiamo creando.

La password va scritta "in chiaro".

IMMAGINAZIONE E LAVORO

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;
```

IMMAGINAZIONE E LAVORO

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).

IMMAGINAZIONE E LAVORO

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 ZEROFILL
)
[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),
    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,
    ins TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY(id),
    KEY k_cogn(cognome)
);
```

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 TO newtableName;
```

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

IMMAGINAZIONE E LAVORO

INFORMAZIONI SULLE TABELLE

Per conoscere la struttura della tabella

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

```
SHOW COLUMNS FROM tableName;
```

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

Per avere maggiori dettagli sulla tabella, tra cui il valore dell'auto_increment, data di creazione, collation...

```
SHOW TABLE STATUS LIKE 'tableName';
```

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

```
SELECT AUTO_INCREMENT
FROM information_schema.tables
WHERE table_schema = 'databaseName'
AND table_name = 'tableName';
```

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.

IMMAGINAZIONE E LAVORO

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;
```

- Modificare la PRIMARY KEY(eliminare prima l'auto_increment se presente con ALTER TABLE…MODIFY…):

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

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

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*.

IMMAGINAZIONE E LAVORO

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*.

```
INSERT INTO studente
VALUES(1, 'fabio','rossi','fbr@gmail.com','1995-05-05','2021-10-27
12:10:53');
```

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

Inserire i dati prendendoli da un'altra tabella:

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

Nell'esempio qui sopra abbiamo immaginato di popolare di dati la tabella *amico* inserendo automaticamente i dati già presenti nella tabella *studente*.

È necessario che i campi nelle due tabelle contengano lo stesso tipo di dato e che la tabella *amico* esista.

IMMAGINAZIONE E LAVORO

CREATE TABLE ... SELECT

Si possono creare delle tabelle già popolate di dati mediante l'uso congiunto delle istruzioni **CREATE TABLE** e **SELECT**.

CREATE TABLE crea la nuova tabella,

SELECT carica i dati prelevandoli da un'altra tabella.

Il suo funzionamento, in pratica, è analogo a quello di **INSERT INTO** ... **SELECT**.

```
CREATE TABLE parenti (
  id INT AUTO_INCREMENT,
  nome VARCHAR(20),
  cognome VARCHAR(30),
  PRIMARY KEY(id)
) SELECT nome, cognome FROM amici;
```

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 (se non specifichiamo una condizione tutte le righe saranno modificate)

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

Quando si inseriscono i dati in una tabella rammentate sempre come sono stati definiti gli attributi per evitare errori di inserimento.

Se si inserisce un valore *troppo lungo*, o *non compreso* dalla definizione dell'attributo, 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('m','f')` , accetta quindi solo i valori *m* o *f*.

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://mariadb.com/kb/en/sql-mode/
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 isolare 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';
```

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

```
SELECT AUTO_INCREMENT
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'databaseName';
-- AND TABLE_NAME = 'tableName';
```

Commenti

MySQL Server supporta tre stili di commento:

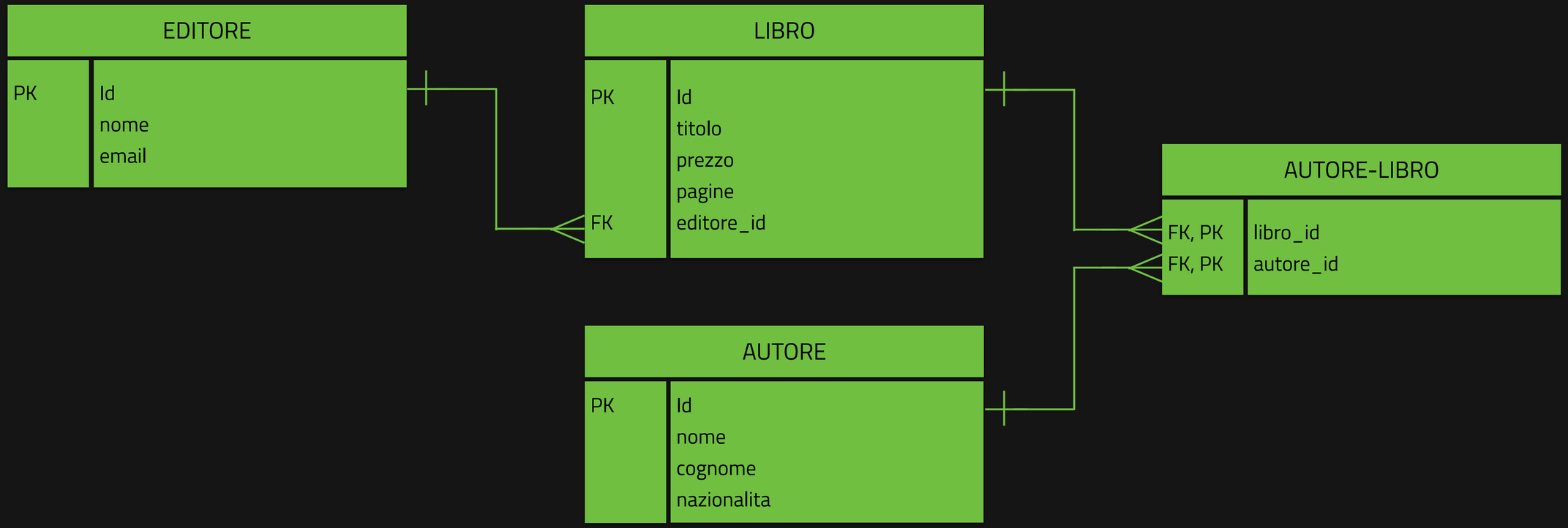
- `#` da questo simbolo a fine riga.
- `--` da questo simbolo a fine riga.
  lo stile di commento (doppio trattino) richiede che il secondo trattino sia seguito da almeno uno spazio bianco o un carattere di controllo (come uno spazio, tab, nuova riga e così via).
- `/* commento */` come nel linguaggio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

Esercizio: creare le tabelle relative al database "Catalogazione libri"

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



Scrivere e utilizzare uno script .sql

L'istruzione **SOURCE**

Mysql può leggere ed eseguire istruzioni SQL salvate in un file di testo con estensione sql;

Per farlo si usa l'istruzione SOURCE seguita dal path del file
(attenzione: senza `;` per chiudere l'istruzione):

```
SOURCE path file.sql
```

Se voglio caricare un file dalla scrivania scriverò:

```
SOURCE C:\Users\anskat_PC\Desktop\script.sql
```

Se ricevete errore: `ERROR: Unknown command '\U'. …'\a' …'\D'.` (dove '\[*lettera*]' è l'iniziale della directory nel path inserito), potete:

- invertire nel path la backslash "\" con forwardslash "/",

```
SOURCE C:/Users/anskat_PC/Desktop/script.sql
```

- provare ad effettuare l'escape inserendo la il backslash '\/'.

```
SOURCE C:\/Users\/anskat_PC\/Desktop\/script.sql
```

- Se ci sono spazi nel nome del path, racchiudetelo tra *virgolette*.

```
SOURCE "C:\Users\anskat PC\Desktop\script.sql"
```

IMMAGINAZIONE E LAVORO

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: [i] , 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 [i],10;
```

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;
```

IMMAGINAZIONE E LAVORO

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';
```

IMMAGINAZIONE E LAVORO

Operatori

- operatori di confronto

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

- operatori logici

  `AND, OR`

- 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

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

```sql
SELECT nome, cognome, email, data_nascita
FROM studenti
WHERE citta != 'torino'
ORDER BY cognome, nome;
```

```sql
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).

Attraverso l'uso delle parentesi potete creare e combinare i vostri criteri di ricerca.

I risultati saranno diversi:

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

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

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
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
FROM studenti
WHERE provincia NOT IN ('to','cn','at','no');
```

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.

```
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.

IMMAGINAZIONE E LAVORO

LIKE, NOT LIKE

Un operatore piuttosto "particolare" è **LIKE** il quale consente di effettuare dei "paragoni di somiglianza".

```
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)

IMMAGINAZIONE E LAVORO

REGEXP

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

```
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$';
```

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-h]o';
```

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

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 campi alla nostra query che rappresentano calcoli algebrici di base su un campo esistente.

Nell'esempio che segue la query restituirà il prezzo corrente e il prezzo con il 10% in più.

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

Posiamo usare un alias per creare etichette più chiare nei risultati della query. Ecco un esempio dello stesso codice con l'alias:

```
SELECT titolo, prezzo AS 'prezzo senza IVA', prezzo * 1.1 AS 'con IVA 10%'
FROM libro;
```

Usare gli operatori permette di fare un calcolo per ogni campo

Interrogare più tabelle

Rappresentazione dell'informazione in tabelle nel database secondo il modello relazionale dell'esempio relativo alla catalogazione di libri:

LIBRO

• id	titolo	pagine	prezzo	editore_id
1	Il paradiso degli Orchi	128	6,00	2
2	Il gabbiano Jonathan Livingston	150	4,50	3
3	L'isola di Arturo	220	18,80	1
4	Caro Bogart	115	7,20	2

LIBRO-AUTORE

libro_id	autore_id
1	2
2	3
3	1
4	2

AUTORE

• id	cognome	Nome	Nazionalità
1	Morante	Elsa	ITA
2	Coe	Jonathan	ENG
3	Bach	Richard	USA
4	Pennac	Daniel	FRA

EDITORE

• id	nome	contatto
1	Einaudi	info@einaudi.it
2	Feltrinelli	amm@feltrinelli.it
3	BUR	contatti@bur.it

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

Immaginiamo di voler estrarre l'elenco dei libri (titolo e prezzo) e relativo editore.

Le informazioni richieste si trovano in due tabelle: *libro* ed *editore*.

Vediamo la query:

```
SELECT libro.titolo, libro.prezzo ,editore.nome
FROM libro, editore
WHERE libro.editore_id = editore.id;
```

Se volessimo estrarre l'elenco dei libri (titolo) e relativo autore le informazioni richieste si trovano in tre tabelle: il titolo in *libro,* nome e cognome dell'autore in *autore,* l'associazione tra libro e chi l'ha scritto in *autore_libro*.

Vediamo la query:

```
SELECT libro.titolo, autore.nome ,autore.cognome, autore.nazionalita
FROM libro, autore_libro, autore
WHERE libro.id = autore_libro.libro_id AND autore.id = autore_libro.autore_id;
```

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 alias[1].

- **Alias per le colonne**: è possibile utilizzare l'alias con GROUP BY, ORDER BY o HAVING per riferirsi alla colonna[2].

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

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

- **Alias per le tabelle**

I nomi abbreviati delle tabelle vengono utilizzati per rendere la query più semplice da scrivere.

```
SELECT titolo AS Titolo, prezzo AS Prezzo, nome AS `Edito da`
FROM libro AS l, editore AS e
WHERE l.editore_id=e.id
ORDER BY l.Titolo;
```

nota: AS si può anche omettere; SELECT titolo Titolo, nome 'Edito da'...

1) se il nome alias è un nome composto (es: `Edito da`) usare i backtick ` (apice retroverso) per wrappare il testo così da non avere risultati inattesi.

2) è possibile utilizzare l'alias SOLO con GROUP BY, ORDER BY o HAVING
https://dev.mysql.com/doc/refman/8.0/en/problems-with-alias.html

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

Le informazioni richieste si trovano in quattro tabelle:

libro, *editore*, *autore* e *autore_libro*.

Vediamo la query:

```
SELECT l.titolo, a.cognome Autore, e.nome `Edito da`, l.prezzo Prezzo
FROM libro l, editore e, autore a, autore_libro al
WHERE l.editore_id = e.id
AND l.id = al.libro_id
AND a.id = al.autore_id;
```

Se non è stato registrato l'autore o l'editore, il libro non comparirà in elenco, **verranno estratti**, infatti, **solo ed esclusivamente i valori che hanno una corrispondenza su tutte le tabelle**.

Integrità referenziale

FOREIGN KEY: gestire le relazioni tra tabelle

L'integrità referenziale è un insieme di regole usate per assicurare che le relazioni tra i record delle tabelle correlate siano valide e che non vengano eliminati o modificati per errore i dati correlati.

Le Foreign key sono utilizzabili quando:

- entrambe le tabelle utilizzano l'engine InnoDB
- il campo della tabella primaria è una chiave primaria.
- i campi correlati contengono lo stesso tipo di dati.
- i campi coinvolti nella relazione devono avere un indice di qualche tipo.
- non sono inclusi campi di tipo BLOB o TEXT nelle chiavi usate

Le Foreign key aiutano lo sviluppatore a stabilire i comportamenti che il database assumerà nel momento in cui si tenta di eliminare o modificare un record di una tabella primaria legato ad uno o più record nella tabella secondaria.

database "Catalogazione libri" - integrità dei dati

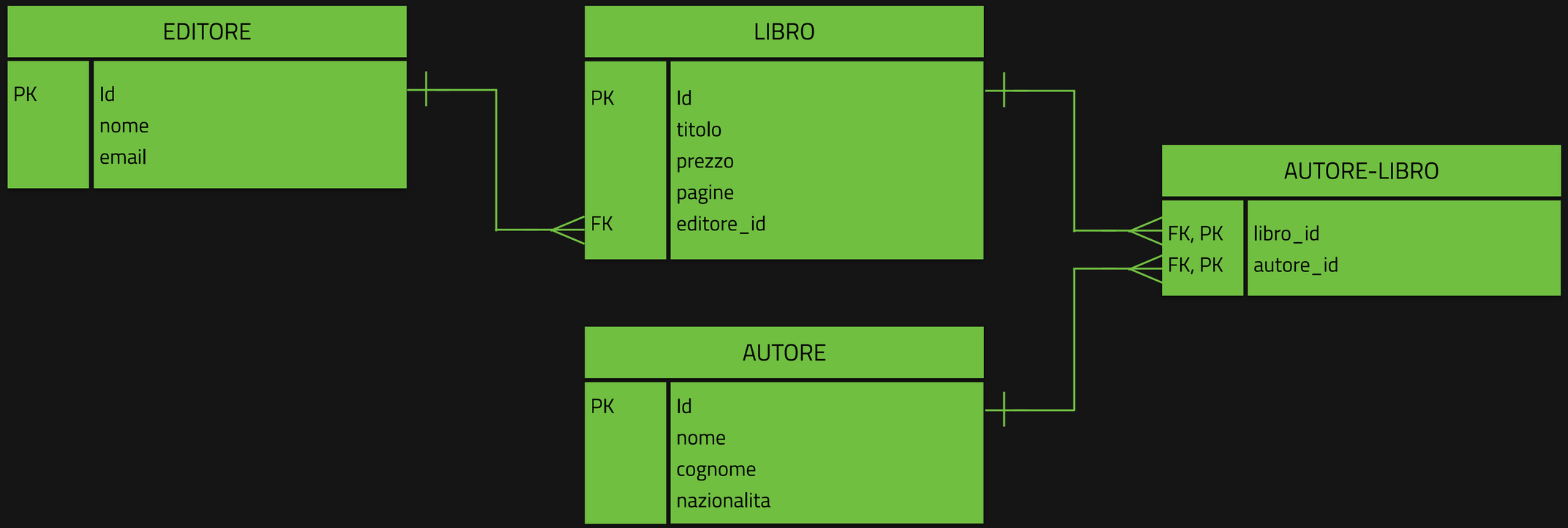
Ci sono 3 chiavi esterne in questo database:

- FK 1 ***editore_id*** nella tabella *libro*, legato all'*id* della tabella *editore*,
- FK 2 ***libro_id*** nella tabella *associativa* (*autore_libro*) legato all'*id* della tabella *libro*,
- FK 3 ***autore_id*** nella tabella *associativa* (*autore_libro*) legato all'*id* della tabella *autore*.

Dobbiamo quindi chiederci:

1. Cosa succede nella tabella *libro* quando elimino/aggiorno un editore dalla tabella *editore*?
2. Cosa succede nella tabella *autore_libro* quando elimino/aggiorno un libro dalla tabella *libro*?
3. Cosa succede nella tabella *autore_libro* quando elimino/aggiorno un autore dalla tabella *autore*?

tabelle relative al database "Catalogazione libri"



Possiamo definire alcune azioni diverse da far 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 in: NULL.
Questa azione è attivabile solo se il campo interessato della tabella secondaria non è impostato a NOT NULL (non deve essere required).

NO ACTION o **RESTRICT**

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

Vediamo un esempio. Creiamo un vincolo tra due tabelle (editore, libro) per mezzo dei campi *editore.id* e *libro.editore_id*

```
CREATE TABLE editore (
  id INT AUTO_INCREMENT,
  nome VARCHAR (200),
  PRIMARY KEY(id));

CREATE TABLE libro (
 id INT AUTO_INCREMENT,
 titolo VARCHAR (200),
 editore_id INT NOT NULL,
 PRIMARY KEY (id),
 INDEX editore_key(editore_id),
 CONSTRAINT fk_libro_editore /* opzionale[1] */
  FOREIGN KEY(editore_id) REFERENCES editore(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE);
```

In questo modo

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

Nell'esempio abbiamo stabilito che eliminando un editore dalla tabella editore vengano automaticamente eliminati tutti i record dalla tabella libro collegati all'editore eliminato.

* CONSTRAINT fk_libro_editore è opzionale, serve a dare un nome alla foreign key, altrimenti il nome viene assegnato dal motore del db: [*nome_tabella_ibfk_n*].
È preferibile dare un nome "parlante" alla foreign key in modo da poterla richiamare quando serve senza dover scoprire il nome assegnato da MySQL.

Se volessimo modificare la FOREIGN KEY dovremmo prima eliminarla:

```
ALTER TABLE libro
DROP FOREIGN KEY fk_libro_editore;
```

e poi ricrearla da capo con le nuove regole:

```
ALTER TABLE libro ADD CONSTRAINT fk_libro_editore
  FOREIGN KEY(editore_id) REFERENCES editore(id)
    ON DELETE NO ACTION ON UPDATE CASCADE;
```

In questo modo abbiamo modificato la chiave esterna e abbiamo modificato le azioni da seguire in caso di eliminazione o aggiornamento di un record nella tabella editore.

```
DELETE FROM editore
WHERE editore_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 editore dalla tabella "editore", prima bisogna eliminare le dipendenze nella tabella libro.

```
DELETE FROM libro
WHERE editore_id=1;
```

Ora posiamo eseguire la query precedente:

```
DELETE FROM editore
WHERE editore_id=1;
```

```
ALTER TABLE libro
DROP FOREIGN KEY fk_libro_editore;
```

nuove regole:

```
ALTER TABLE libro ADD CONSTRAINT fk_libro_editore
  FOREIGN KEY(editore_id) REFERENCES editore(id)
    ON DELETE SET NULL;
```

Nel caso specifico abbiamo stabilito che eliminando un editore dalla tabella “editore”, impostiamo a NULL le dipendenze (campo editore_id) nella tabella libro.

```
DELETE FROM editore
WHERE editore_id=1;
```

Questa query viene eseguita; contemporaneamente vengono aggiornate le righe con il campo editore_id = 1 della tabella libro, impostando il valore del campo editore_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 impiegato dove per ciascun impiegato registro anche l'attributo identificativo dell'impiegato responsabile:

```
CREATE TABLE IF NOT EXISTS impiegato (
  id int auto_increment,
  nome varchar(50),
  cognome varchar(50),
  ruolo varchar(50),
  id_responsabile int,
  stipendio decimal(6,2),
    CONSTRAINT sk_responsabile
    FOREIGN KEY (id_responsabile) REFERENCES impiegato(id)
     ON UPDATE CASCADE
     ON DELETE NO ACTION,
  PRIMARY KEY(id)
);
```

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

Per visualizzare la chiave esterna di una tabella:

```
SHOW CREATE TABLE nome_tabella;
```

Per visualizzare tutte le chiavi esterne del db:

```
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;
```

Per disabilitare* temporaneamente le chiavi esterne:

```
SET FOREIGN_KEY_CHECKS=0;
```

Ricordatevi di ripristinarle dopo un eventuale inserimento massiccio di record

```
SET FOREIGN_KEY_CHECKS=1;
```

* disabilitare/abilitare le chiavi esterne può servire a popolare un db senza tenere conto dell'ordine di inserimento dei dati che altrimenti verrebbe controllato dalle chiavi esterne - FOREIGN_KEY_CHECKS è una variabile di sistema;
es: con chiave esterna attiva non posso caricare record dei libri prima di caricare i record degli editori

CHECK (constraint)

I vincoli forniscono restrizioni sui dati che è possibile aggiungere a una tabella. Ciò consente di applicare il controllo dei dati da MySQL, piuttosto che tramite la logica dell'applicazione.

Quando un'istruzione viola un vincolo, MySQL genera un errore.

Esistono quattro tipi di vincoli di tabella:

- PRIMARY KEY: Imposta la colonna per fare riferimento alle righe. I valori devono essere univoci e non nulli.
- FOREIGN KEY: Imposta la colonna in modo che faccia riferimento alla chiave primaria su un'altra tabella.
- UNIQUE: Richiede valori nella colonna o nelle colonne si verificano solo una volta nella tabella.
- CHECK: Verifica se i dati soddisfano la condizione specificata.

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. *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 vengono valutati nell'ordine in cui sono definiti. 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, comprese le UDF (User Defined Function).

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 libro2 (
  id int AUTO_INCREMENT,
  titolo varchar(255),
  prezzo decimal(6,2) NOT NULL,
  pagine smallint unsigned CHECK (pagine > 0),
  editore_id int,
  PRIMARY KEY (id),
  KEY k_titolo (titolo),
  CONSTRAINT chk_prezzo CHECK (prezzo > 0)
);
```

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

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

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

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

Unire le tabelle: UNION

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;
- il *datatype delle colonne sia corrispondente (o convertibile)*, non essendo possibile unire i valori estratti da colonne aventi tipi di dato disomogenei (ad esempio INT e VARCHAR).

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 dell'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 cognome, data_nascita, 'X' Generazione
FROM studente
WHERE data_nascita < '1980-01-01'

UNION

SELECT cognome, data_nascita, 'Millenials' Generazione
FROM studente
WHERE data_nascita BETWEEN '1980-01-01' AND '1994-12-31'

UNION

SELECT cognome, data_nascita, 'Z' Generazione
FROM studente
WHERE data_nascita > '1994-12-31'

ORDER BY data_nascita;
```

otteniamo l'elenco degli studenti divisi per generazione

Altro 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 di due continenti

Altro esempio:

```
SELECT nome, cognome FROM parente
UNION ALL
SELECT nome, cognome FROM studente
ORDER BY cognome;
```

EQUIJOIN o JOIN semplice

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

La relazione tra le tabelle è data dall'uguaglianza della chiave, la query restituisce solo i valori corrispondenti, se una riga non soddisfa la condizione non verrà mostrata:

```
SELECT libro.id, titolo, editore_id, editore.id "(id) Da tab editore" , nome
FROM libro, editore
WHERE libro.editore_id=editore.id;
```

id	titolo	editore_id	id	nome
1	Harry Potter e la Camera dei Segreti	4	4	Salani
2	Guerra e Pace	1	1	BUR
3	Harry Potter e il Calice di Fuoco	4	4	Salani
4	Ulisse	3	3	Sellerio
5	Marcovaldo	2	2	Mondadori

Unire le tabelle, sintassi: INNER JOIN

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'eguaglianza: *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 titolo dei libri e i relativi autori le tabelle da interrogare sono tre: provate a costruire la query...

INNER JOIN

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

Vediamo la query:

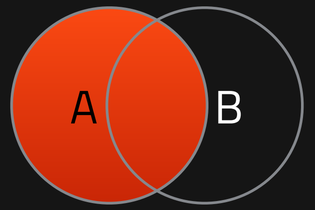
```
SELECT titolo AS Titolo, nome AS Editore, prezzo AS Prezzo
FROM libro
JOIN editore
ON libro.editore_id=editore.id;
```

Vogliamo estrarre l'elenco completo delle informazioni sui libri usando il costrutto INNER JOIN.

```
SELECT l.titolo AS Titolo, CONCAT(a.nome,' ',a.cognome) AS Autore, e.nome AS
'Edito da', l.prezzo AS Prezzo
FROM libro l
JOIN editore e ON l.editore_id=e.id
JOIN autore_libro al ON al.libro_id=l.id
JOIN autore a ON al.autore_id=a.id;
```

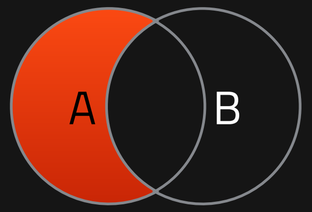
LEFT JOIN

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
```

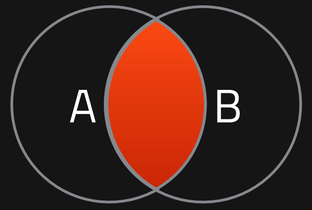
Estrae solo i valori della tabella a sinistra che non hanno corrispondenza nella tabella a destra




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

INNER JOIN

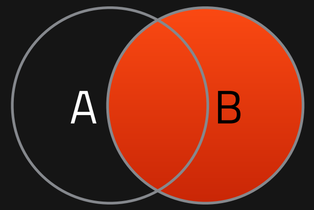
Estrae esclusivamente i valori che hanno una corrispondenza su tutte le tabelle




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

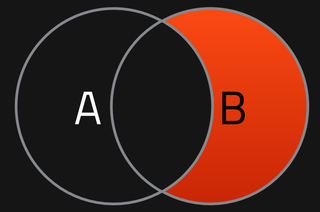
RIGHT JOIN

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
```

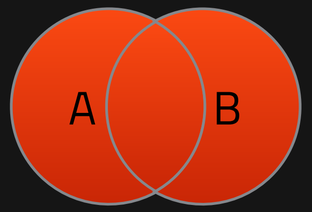
Estrae solo i valori della tabella a destra che non hanno corrispondenza nella tabella a sinistra




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

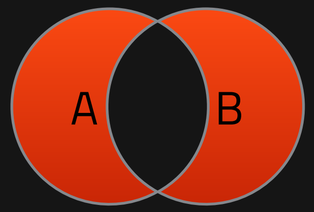
FULL OUTER JOIN

Estrae 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
```

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.

OUTER JOIN

Vogliamo estrarre l'elenco dei libri considerando anche quelli a cui non è stato associato un editore.
Usiamo **LEFT JOIN** per estrarre anche i record dalla tabella di sinistra (libro -la prima che compare nella query) che non hanno corrispondenza con la tabella di destra (editore).

Vediamo la query:

```
SELECT titolo AS Titolo, nome AS Editore, prezzo AS Prezzo
FROM libro l
LEFT JOIN editore e
ON l.editore_id=e.id;
```

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

Vediamo la query:

```
SELECT titolo AS Titolo, nome AS Editore, prezzo AS Prezzo
FROM libro l
RIGHT JOIN editore e
ON l.editore_id=e.id;
```

Vogliamo estrarre solo l'elenco dei libri a cui non è stato associato un editore.

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

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

Vediamo la query:

```
SELECT titolo AS Titolo
FROM libro l
LEFT JOIN editore e
ON l.editore_id=e.id
WHERE e.id IS NULL;
```

Vogliamo estrarre l'elenco degli editori considerando solo quelli a cui non è stato associato alcun libro.

Vediamo la query:

```
SELECT nome AS Editore
FROM libro l
RIGHT JOIN editore e
ON l.editore_id=e.id
WHERE l.editore_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 titolo, nome
FROM libro
LEFT JOIN editore
ON libro.editore_id=editore.id

UNION

SELECT titolo, nome
FROM libro
RIGHT JOIN editore
ON libro.editore_id=editore.id;
```

FULL OUTER JOIN

solo gli esclusi

```
SELECT titolo, nome
FROM libro
LEFT JOIN editore
ON libro.editore_id=editore.id
WHERE editore.id IS NULL

UNION

SELECT titolo, nome
FROM libro
RIGHT JOIN editore
ON libro.editore_id=editore.id
WHERE libro.editore_id IS NULL;
```

SELF JOIN

Immaginiamo di avere una tabella "impiegato".

All'interno della tabella registriamo, tra gli attributi (*nome, cognome, ruolo, id_resp, stipendio, ufficio_id*), 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 impiegato i
JOIN impiegato r
ON i.id_resp = 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 impiegato i
LEFT JOIN impiegato r
ON i.id_resp = r.id
ORDER BY ruolo;
```

Scorciatoia JOIN

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

```
SELECT titolo AS Titolo, nome AS Editore, prezzo AS
Prezzo
FROM libro
JOIN editore
USING (editore_id);
```

[1]La clausola nomina un elenco di colonne che devono esistere in entrambe le tabelle.

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 libro;
```

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:

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

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

```
SELECT COUNT(*)
FROM studente
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 libro;
```

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 magazzino'
FROM libro;
```

FLOOR() e CEILING()

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

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

```
SELECT FLOOR(prezzo), CEILING(prezzo)
FROM libro;
```

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 libro;
```

LENGTH()

La funzione LENGTH() restituisce il numero totale di caratteri in ogni campo. Anche gli spazi tra le parole sono contati

```
SELECT nome, LENGTH(nome)
FROM studente;
```

CONCAT(campo1, campo2, campo3): concatena le stringhe;

```
SELECT CONCAT(nome,' ',cognome)
FROM studente;
```

La funzione CONCAT_WS () concatena due o più espressioni insieme e aggiunge un separatore tra di esse.

```
SELECT CONCAT_WS(' ', nome, cognome)
FROM studente;
```

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 studente;
```

LEFT() / RIGHT():

mostra uno specifico numero di caratteri iniziando da sinistra(LEFT) o destra(RIGHT)

```
SELECT nome, LEFT(nome,1)
FROM studente;
```

```
SELECT CONCAT(LEFT(nome,1),'.',cognome)
FROM studente;
```

Uso di funzioni combinate

Lunghezza media in termini di caratteri considerando la stringa *nome* + *cognome* insieme:

```
SELECT AVG(LENGTH(nome)+LENGTH(cognome))
FROM studente;
```

```
SELECT AVG(LENGTH(CONCAT(nome, cognome))
FROM studente;
```

Elimina i numeri dopo la virgola, FLOOR() arrotonda verso il basso, CEILING() verso l'alto;

```
SELECT FLOOR(AVG(LENGTH(CONCAT(nome, cognome))))
FROM studente;
```

```
SELECT CEILING(AVG(LENGTH(CONCAT(nome, cognome))))
FROM studente;
```

Funzioni informative[1]

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 studente(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:

REPLACE

Vediamo ad esempio come aggiornare le informazioni presenti in un campo modificando solo una parte del suo valore

```
UPDATE studente
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.

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 studente;
```

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['01-01-2001']` e un formato `FORMAT[%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 studente (nome, cognome, email, data_nascita)
VALUES ('marco','allegri','marco.allegri@gmail.com', STR_TO_DATE('05,10,1969','%d,%m,%Y'));
```

```
INSERT INTO studente (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
```

Come si vede, per sommare alla data cinque giorni si possono usare due espressioni diverse, INTERVAL 5 DAY o semplicemente il numero 5 (potete aggiungere mesi o anni usando MOUNTH e YEAR)

```
SELECT ADDDATE('2017-03-01',INTERVAL 5 YEAR); restituisce: 2022-03-06
```

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 successiva alla 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 STUDENTE;
```

Vediamo come sfruttare questo calcolo in fase di UPDATE e di INSERT.

```
UPDATE studente
SET eta = TIMESTAMPDIFF(YEAR, data_nascita, CURDATE());
```

```
INSERT INTO studente
(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.

IMMAGINAZIONE E LAVORO

Giorni al compleanno

```
SELECT nome, cognome, DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita) AS Giorni
FROM studente
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 studente
WHERE (DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita)) BETWEEN 0 AND 31;
```

Se uso WHERE l'alias della colonna non viene intercettata (vedi slide successiva)

```
SELECT nome, cognome, (DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita)) AS Giorni
FROM studente
WHERE Giorni BETWEEN 0 AND 31
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 delle istruzioni in mysql:

SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
LIMIT

```
SELECT nome, cognome, (DAYOFYEAR(CURDATE())-DAYOFYEAR(data_nascita)) AS Giorni
FROM studente
HAVING Giorni BETWEEN 0 AND 31
ORDER BY Giorni;
/* seleziono quelli a cui al compleanno mancano meno di 31 giorni */
```

IMMAGINAZIONE E LAVORO

JSON function

Inserire i dati in formato JSON, per un attributo definito in questo modo, richiede la notazione JSON

```
INSERT INTO articolo(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*.

* 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 articolo(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 articolo(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_ESTRACT(json_doc, path[, path] ...)

Restituisce i dati da un documento JSON, selezionato 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 articolo;
```

Questa funzione ha un alias che ne semplifica la scrittura.

```
SELECT column -> '$.key'
FROM tableName;
```

```
SELECT specifiche -> '$.uscite'
FROM articolo;
```

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 articolo;
```

```
SELECT specifiche -> '$.uscite[2]'
FROM articolo;
```

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 articolo
SET specifiche =
JSON_SET(specifiche,’$.marca','LG','$.uscite',JSON_ARRAY('HDMI
','SCART','S/PDIF'),'$.ingressi',JSON_ARRAY('ETHERNET','USB'))
WHERE id = 1;
```

IMMAGINAZIONE E LAVORO

JSON_INSERT(json_doc, path, val[, path, val] ...)

La funzione JSON_INSERT inserisce i valori senza sostituire i valori esistenti.

```
UPDATE articolo
SET specifiche = JSON_INSERT(specifiche,'$.uscite[2]','RGB')
WHERE id = 1; -- non produce risultato perché la posizione
nell'array è occupata
```

```
UPDATE articolo
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 articolo
SET specifiche = JSON_REPLACE(specifiche,'$.marca','SONY')
WHERE id = 1;
```

```
UPDATE articolo
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 articolo
SET specifiche = JSON_REPLACE(specifiche,'$.uscite[1]','HDMI2')
WHERE id = 1;
```

IMMAGINAZIONE E LAVORO

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 articolo
SET specifiche = JSON_REMOVE(specifiche,'$.profondita')
WHERE id = 1; – non produce risultato perché la proprietà non esiste
```

```
UPDATE articolo
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 , 2 , 3 ); -- 3
```

```
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 studente
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 studente;
```

```
SELECT
  titolo,
  prezzo,
 CASE
    WHEN prezzo < 5 THEN 'economico'
    WHEN prezzo >= 5 AND prezzo <= 10 THEN 'medio'
    WHEN prezzo > 10 THEN 'caro'
 END valore
FROM libro;
```

Esempio

```
SELECT
    cognome,
    CASE WHEN genere = 'f' THEN 'Donna' ELSE 'Uomo' END  Genere
from studente;
```

Esempio

Consideriamo la query della slide relativa alla istruzione UNION, utilizzata per suddividere gli studenti sulla base della generazione di appartenenza (**slide 146**), 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) < '1995' THEN 'millenials'
    WHEN year(data_nascita) >= '1995' THEN 'Z'
    WHEN year(data_nascita) IS NULL THEN 'Manca data nascita'
END Generazione
FROM studente
ORDER BY Generazione;
```

Raggruppamenti

Elenco dei valori distinti

Al fine di visualizzare i risultati senza ripetere i valori possiamo usare l'istruzione **DISTINCT**.

```
SELECT DISTINCT cognome
FROM studente;
```

Possiamo anche usare ORDER BY per ordinare i risultati:

```
SELECT DISTINCT cognome
FROM studente
ORDER BY cognome;
```

Possiamo aggiungere un filtro utilizzando la clausola WHERE:

```
SELECT DISTINCT cognome
FROM studente
WHERE cognome LIKE 'v%'
ORDER BY cognome;
```

GROUP BY

La clausola GROUP BY è un elemento facoltativo che può essere inserito all'interno di una SELECT.

Il suo scopo è quello di effettuare dei raggruppamenti di dati.

Supponiamo, ad esempio, di voler raggruppare i record della nostra tabella "studente" in base al cognome. Per fare ciò utilizzeremo:

```
SELECT cognome
FROM studente
GROUP BY cognome;
```

Il risultato di questa query è il medesimo che avremmo potuto ottenere con SELECT DISTINCT:

```
SELECT DISTINCT cognome
FROM studente;
```

IMMAGINAZIONE E LAVORO

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 libri raggruppati per editore possiamo scrivere:

```
SELECT COUNT(*) quanti, e.nome
FROM libro l, editore e
WHERE e.id=l.editore_id GROUP BY e.nome ORDER BY quanti DESC;
```

Per conoscere l'età media degli studenti raggruppati per genere possiamo scrivere:

```
SELECT genere,
FLOOR(AVG(TIMESTAMPDIFF(YEAR,data_nascita,CURDATE()))) 'età media'
FROM studente
GROUP BY genere;
```

Per conoscere il valore dei libri raggruppati per editore possiamo scrivere:

```
SELECT e.nome, sum(prezzo) valore
FROM libro l, editore e
WHERE e.id=l.editore_id
GROUP BY e.nome
ORDER BY valore;
```

Per conoscere per ciascun editore quanti libri abbiamo a catalogo, il loro valore, il loro prezzo medio, il più economico e il più caro possiamo scrivere:

```
SELECT
  e.nome,
  count(*) quanti,
  sum(prezzo) valore,
  avg(prezzo) 'prezzo medio',
  min(prezzo) 'meno caro',
  max(prezzo) 'più caro'
FROM libro l
JOIN editore e
ON e.id=l.editore_id
GROUP BY e.nome
ORDER BY quanti;
```

- quando è utilizzata insieme ad **HAVING** che aggiunge un filtro su valori raggruppati:

```
SELECT cognome, COUNT(cognome) AS 'numero'
FROM studente
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 studente
GROUP BY provincia, genere
ORDER 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 se aggiunta filtra ulteriormente il gruppo considerando i valori contati maggiori di 1.

```sql
SELECT genere, COUNT(cognome) numero
FROM studente
WHERE provincia = 'to'
GROUP BY genere
HAVING numero > 1 # >= 6
ORDER BY genere DESC;
```

WHERE può essere usato ma prima del GROUP BY,

non potete usare funzioni di gruppo nel WHERE, WHERE filtra le righe; per un filtro ulteriore sul gruppo usate HAVING.

```sql
SELECT provincia, genere, FLOOR(AVG(TIMESTAMPDIFF(YEAR, data_nascita,
CURDATE()))) 'età media', COUNT(*) AS numero
FROM studente
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 studente
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 studente
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.

Esercizio

Database per una applicazione web che gestisce l'iscrizione a corsi.

Disegnare una base dati per la gestione dell'iscrizione ai corsi offerti da una piattaforma web.

Gli studenti possono iscriversi a molti corsi. I corsi sono a pagamento.

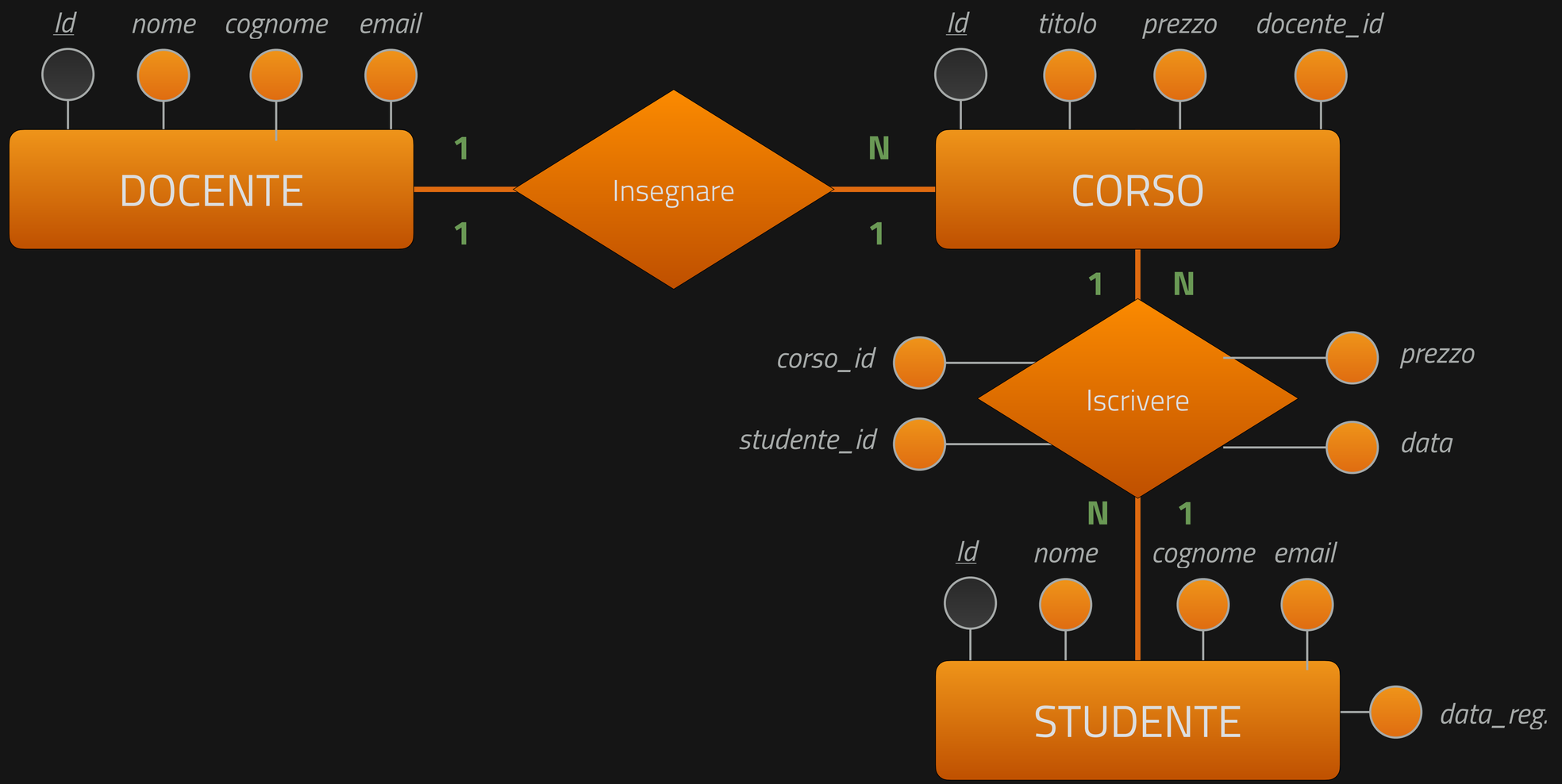
All'atto dell'iscrizione lo studente si deve registrare con nome, cognome ed email. Viene memorizzata anche la data di registrazione.

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.

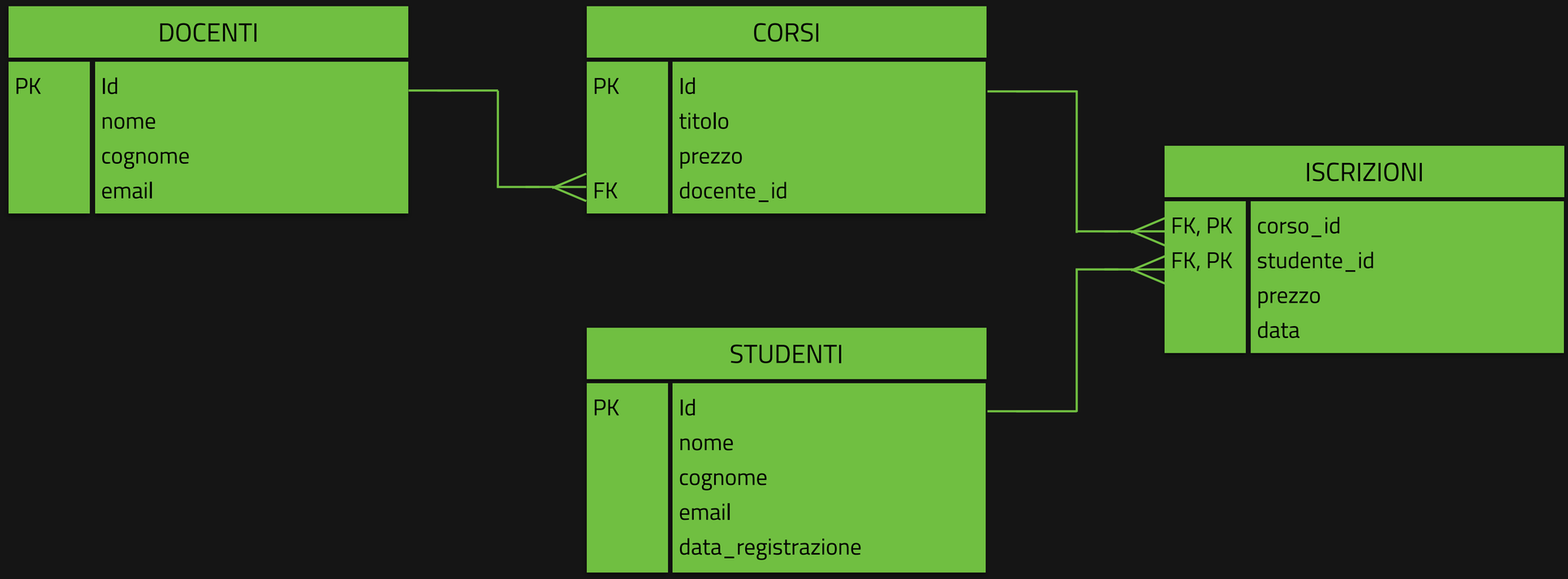
Ogni corso viene tenuto da un docente. Un docente può insegnare in molti corsi.

Diagramma E-R



Modello logico

Simbologia delle relazioni



Esercizio

Vogliamo gestire gli ordini dei clienti di un eventuale shop di hardware e software, tenere traccia degli articoli e gestire il magazzino;

- Di ogni ordine registriamo la *data*, il *cliente* che esegue l'ordine, gli *articoli* richiesti e la *quantità* per ciascun articolo;
- Gli ordini vengono spediti, per cui bisogna considerare lo *stato dell'ordine* e le *informazioni per l'invio*;
- Ogni ordine fa maturare al cliente un credito che definisce il livello del cliente, *ogni € speso corrisponde ad un credito* (1);
- Gli articoli sono categorizzati; per semplificare ogni articolo può appartiene ad una categoria.
  Le categorie sono date: *hardware* e *software* (non è quindi necessaria una tabella ad hoc);
- I dipendenti appartengono a specifici uffici: *amministrazione*, *logistica*, *vendita*, *assistenza*.. ; ogni dipendente ha un ruolo: *tecnico*, *amministrativo*, *venditore*, *magazziniere*... Per cui per ogni area appartengono diversi dipendenti.

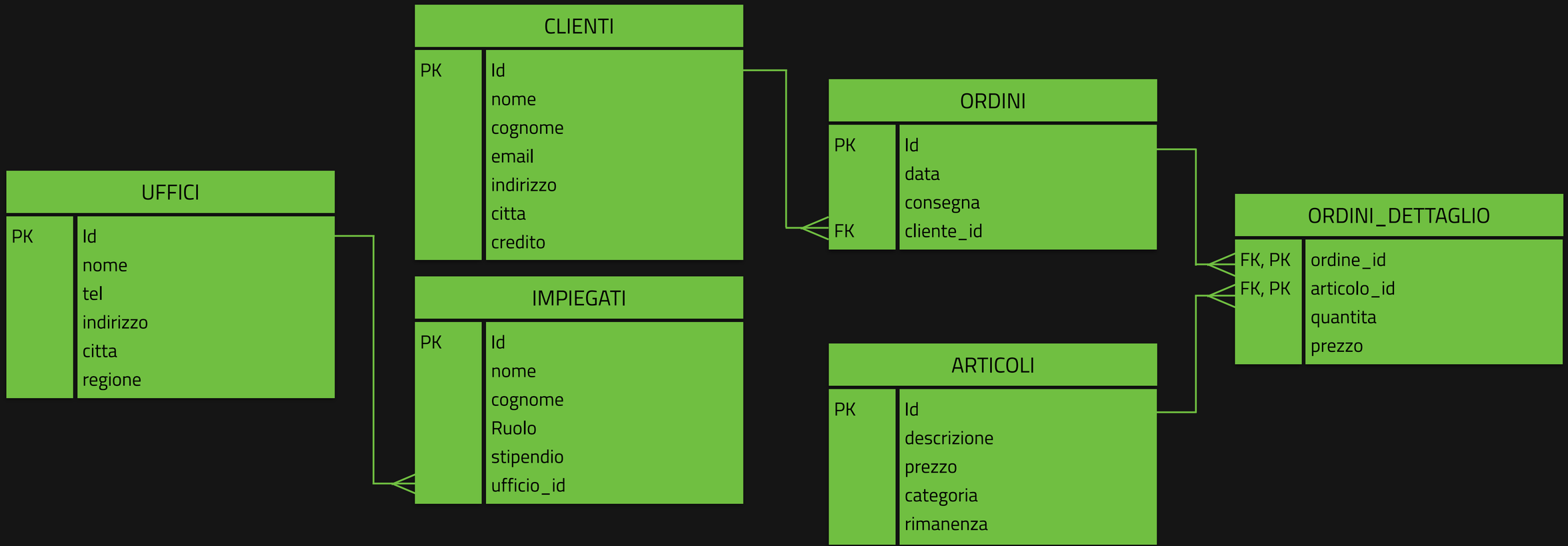
Realizzare il diagramma relativo al *progetto concettuale* (entità/associazioni) e definite lo *schema logico* (tabelle e attributi), quindi:

- 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



IMMAGINAZIONE E LAVORO

Assegnazione accesso a nuovo database all'utente attuale - da shell

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';
```

IMMAGINAZIONE E LAVORO

ESERCIZIO - ESEGUIRE LE QUERY PER OTTENERE LE SEGUENTI INFORMAZIONI

1) selezionate il valore del magazzino;

2) selezionate il valore del magazzino diviso per categoria;

3) selezionate la quantità articoli ordinati divisi per descrizione, ordinati per quantità discendente;

4) selezionate la quantità di articoli ordinati divisi per categoria;

5) selezionate gli articoli ordinati e la quantità relativa dell' ordine con id=7;

6) selezionate il valore degli ordini: totale denaro speso dai clienti;

7) selezionate cognome e email dei clienti che hanno effettuato ordini;

8) selezionate l'ordine, la data dell'ordine e il nome del cliente che ha effettuato ordine;

9) selezionate i clienti e il denaro speso in totale da ciascuno.

Subquery

Una subquery è un'istruzione `SELECT` all'interno di un'altra istruzione SQL(`SELECT` , `INSERT` , `UPDATE` o `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 `>` , `<` , o `=` .
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 *query interna* viene eseguita prima della *query esterna*.

Tipi di subquery:

- subquery scalare
- subquery con operatori di confronto
- subquery con operatori di confronto avanzato `ALL` , `ANY` , `IN` , `NOT IN`
- row subquery
- subquery con `EXISTS` o `NOT EXISTS`
- subquery correlate
- 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" con id 6;

Possiamo ottenere l'elenco in due passaggi:

```
SELECT stipendio FROM impiegato WHERE id = 6; ##[ r:1500.00 ]
```

```
SELECT nome, cognome, stipendio
FROM impiegato
WHERE stipendio > 1500.00
ORDER BY cognome;
```

Oppure unire le due query nidificandone una nell'altra:

```
SELECT nome, cognome, stipendio
FROM impiegato
WHERE stipendio >
(SELECT stipendio FROM impiegato 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;

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 impiegato
WHERE stipendio > (SELECT stipendio FROM impiegato WHERE cognome='Barba')
ORDER BY cognome;
```

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

La subquery scalare.

Una subquery scalare è una subquery che restituisce esattamente un valore di colonna da una riga.
Ad esempio:

```
SELECT e.nome, l.titolo
FROM editore e
JOIN libro l
ON e.id=l.editore_id
WHERE editore_id = (SELECT MAX(editore_id) FROM libro);
```

In questo esempio la subquery (cioè l'istruzione SELECT utilizzata nella clausola WHERE) restituisce un numero corrispondente all'editore_id più grande della tabella "libro" il quale viene utilizzato dalla SELECT principale per filtrare i risultati della tabella "libro".

Se la subquery restituisce 0 righe, allora il valore dell'espressione subquery scalare è NULL;
se la subquery restituisce più di una riga, MySQL restituisce un errore

```
[ ERROR 1242 (21000): Subquery returns more than 1 row ]
```

La subquery scalare.

Vediamo altro esempio:

Selezioniamo dalla tabella libro, i libri il cui prezzo è maggiore del prezzo medio del nostro catalogo.

```
SELECT titolo, prezzo
FROM libro
WHERE prezzo > (SELECT AVG(prezzo) FROM libro)
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 libro WHERE prezzo > AVG(prezzo);
```

```
ERROR 1111 (HY000): Invalid use of group function
```

Subquery con operatori di confronto.

Una Subquery può essere utilizzata prima o dopo 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 altro esempio

```
SELECT nome, cognome, stipendio
FROM impiegato
WHERE stipendio < (SELECT AVG(stipendio) FROM impiegato)
ORDER BY stipendio DESC;
```

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.

`NOT IN` è un alias per `<> ALL` . Quindi, queste due istruzioni sono uguali.

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 impiegato i
JOIN ufficio u
ON u.id = i.ufficio_id
GROUP BY i.ufficio_id
HAVING `Stipendio medio` >= ALL
  ( SELECT AVG( stipendio ) FROM impiegato 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 impiegato
WHERE ufficio_id = ANY
(SELECT id FROM ufficio 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.

IMMAGINAZIONE E LAVORO

Subquery con: ANY(SOME)

Posso contare gli impiegati che lavorano in una data regione

```
SELECT 'Piemonte', COUNT(*)
FROM impiegato
WHERE ufficio_id = ANY
(SELECT id FROM ufficio WHERE regione = 'piemonte');
```

Subquery con: IN, NOT IN

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 clienti che *hanno effettuato* ordini.

```
SELECT cognome, telefono, citta
FROM cliente
WHERE id IN (SELECT cliente_id FROM ordine);
```

di seguito lo stesso esempio con una JOIN.

```
SELECT DISTINCT cognome, telefono, citta
FROM cliente
INNER JOIN ordine ON cliente.id=ordine.cliente_id;
```

Vediamo l'esempio con NOT IN: selezioniamo i clienti che *non hanno effettuato* gli ordini.

```
SELECT cognome, telefono, citta
FROM cliente
WHERE id NOT IN (SELECT cliente_id FROM ordine);
```

```
SELECT cognome, telefono, citta
FROM cliente c
LEFT JOIN ordine o ON c.id=o.cliente_id
WHERE o.cliente_id IS NULL;
```

Row subquery: ROW(field1, field2)

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)`.

```
SELECT colonna1, colonna2 FROM t1
WHERE ROW( colonna1, colonna2 ) =
( SELECT colonna1, colonna2 FROM t2 WHERE campo ='valore' );
```

Vediamo un esempio:

```
SELECT nome, cognome FROM amici
WHERE ROW( nome, cognome ) =
( SELECT nome, cognome FROM studenti WHERE id = 4 );
```

Questa query confronta le righe dalla tabella amico per i campi nome e cognome con la riga estratta nella subquery finché non trova una corrispondenza.

```
SELECT nome, cognome FROM amici
WHERE ROW(nome, cognome) =
( SELECT nome, cognome FROM studenti WHERE cognome='rossi' );
```

Questa query restituisce: ERROR 1242 (21000): Subquery returns more than 1 row

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 articolo a
SET rimanenza = 100 -
(SELECT SUM(quantita) FROM ordine_dettaglio od
WHERE od.articolo_id = a.id
GROUP BY 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 articolo SET rimanenza = 100 WHERE rimanenza IS NULL;
```

Riprendendo l'esempio precedente relativo all'aggiornamento del magazzino, grazie alla funzione IF tutto si può scrivere in una sola query:

IF
expr1
expr2
expr3

```
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

);
```

Subquery per aggiornare il credito di tutti i clienti:

```
UPDATE cliente c SET credito =
IF(
    (SELECT SUM(prezzo*quantita)
    FROM ordine o
    JOIN ordine_dettaglio od
      ON c.id = o.cliente_id
      AND o.id = od.ordine_id
    GROUP BY c.id > 0),
    (SELECT SUM(prezzo*quantita)
    FROM ordine o
    JOIN ordine_dettaglio od
      ON c.id = o.cliente_id
      AND o.id = od.ordine_id
    GROUP BY c.id),
    0
);
```

Subquery per aggiornare il credito di un cliente:

```
UPDATE cliente c SET credito =
IF (
    (SELECT SUM(prezzo*quantita)
    FROM ordine o
    JOIN ordine_dettaglio od
      ON c.id = o.cliente_id
      AND o.id = od.ordine_id > 0),
    (SELECT SUM(prezzo*quantita)
    FROM ordine o
      JOIN ordine_dettaglio od
      ON c.id = o.cliente_id
      AND o.id = od.ordine_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 degli editori che hanno almeno un libro registrato nella tabella libri.

```
SELECT nome
FROM editore e
WHERE EXISTS
 (SELECT editore_id FROM libro WHERE editore_id = e.id);
```

Vediamo la stessa cosa con una JOIN

```
SELECT DISTINCT nome
FROM editore e
INNER JOIN libro l
ON e.id = l.editore_id;
```

Questa query estrae i nomi degli editori che non hanno libri registrato nella tabella libri.

```
SELECT nome
FROM editore e
WHERE NOT EXISTS
 (SELECT editore_id FROM libro
  WHERE editore_id = e.id);
```

Vediamo la stessa cosa con una OUTER JOIN

```
SELECT nome
FROM editore e
LEFT JOIN libro l
ON e.id = l.editore_id
WHERE l.editore_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 libro_v in cui mostriamo il titolo, il prezzo e le pagine dei libri

```
CRERATE OR REPLACE VIEW libro_v
SELECT titolo Titolo, pagine Pagine, prezzo Prezzo
FROM libro;
```

Quando interroghiamo la vista la SELECT è la seguente:

```
SELECT * FROM libro_v;
```

Siccome la vista è una query memorizzata è come se scrivessimo:

```
SELECT * FROM
 ( SELECT titolo Titolo, pagine Pagine, prezzo Prezzo
FROM libro
ORDER BY Titolo ) AS libro_v;
```

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 tabella ordini:

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 ordine_dettaglio
GROUP BY ordine_id
ORDER BY q_articoli DESC) 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.

VIEW (Viste)

Una vista è una tabella logica (query) basata su tabelle fisiche o su un’altra vista.

Una vista viene creata da una o più SELECT su una "base table".

Le viste non contengono dati, ma solo la definizione della query sulle "base table".

È possibile creare un sottoinsieme di dati o una combinazione di dati;

Vantaggi

- la vista può visualizzare una selezione limitata di colonne di una tabella così da avere l’accesso ad un insieme ristretto di dati. Limita l’accesso a dati sensibili.
- Maschera la complessità del DB.
- Riduce l’impatto dei cambiamenti sul database.
- Utile per fare query semplici ottenendo risultati di query complesse; per esempio una vista può essere usata per ottenere informazioni da tabelle multiple senza che l’utente sappia come scrivere il join.

VIEW (Viste)

Vantaggi

- Semplificano le query
- Riducono l’impatto dei cambiamenti
- Limitano accesso ai dati

```sql
CREATE OR REPLACE VIEW nome_vista AS
SELECT nome_campi
FROM nome_tabella
WHERE condizioni;
```

Le viste si comportano come delle tabelle ma non sono tabelle, sono il risultato di una query.

Quando creiamo una vista e come se creassimo una query permanente.

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 nome_vista AS
SELECT nome_campi
FROM nome_tabella
WHERE condizioni;
```

Oppure, in alternativa, è possibile utilizzare il comando ALTER VIEW in questo modo:

```
ALTER VIEW nome_vista AS
SELECT nome_campi
FROM nome_tabella
WHERE condizioni;
```

La differenza tra i due comandi dovrebbe essere intuitiva: ALTER VIEW può essere utilizzato solo per modificare una Vista esistente.

Con CREATE OR REPLACE VIEW possiamo non solo modificare, ma anche creare la vista se questa ancora non esiste.

Per rinominare la view, modificarne solo il nome, potete usare:

```
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, **una vista semplice è sempre aggiornabile**.

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;

VISTA semplice

```
CREATE OR REPLACE VIEW studenti_v AS
SELECT id, nome, cognome, email
FROM studenti
ORDER BY cognome;
```

VISTA complessa

```
CREATE OR REPLACE VIEW libro_tot AS
SELECT l.titolo, e.nome AS Editore, CONCAT(a.cognome,' ', a.nome) AS
Autore, nazionalita 'Nazionalità', ROUND(l.prezzo*1.22,2) AS Prezzo
FROM libro l
JOIN editore e ON l.editore_id=e.id
JOIN autore_libro la ON l.id=la.libro_id
JOIN autore a ON a.id=la.autore_id
ORDER BY titolo;
```

IMMAGINAZIONE E LAVORO

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;
```

Potete interrogare *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;
SHOW CREATE TABLE nome_vista;
```

VIEW con **WITH CHECK OPTION**

```
CREATE OR REPLACE VIEW studente_v AS
SELECT id, nome, cognome, email, provincia
FROM studente
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 la vista, non possono creare o modificare i dati di cui la vista non ha visibilità.

```
UPDATE studente_v
SET provincia = 'cn'
WHERE id = 3;
```

```
INSERT INTO studente_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'
```

È possibile togliere il check option ridefinendo la vista con alter view.

Backup e restoring

Backup di un DB MySQL (da shell)

Amministratore: utente root

L'operazione di backup 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 --databases db_1 db_2 db_3 > nomefile.estensione**
mysqldump -u root -p --all-databases > nomefile.estensione
```

Nel primo caso stiamo esportando tre database chiamati db_1, db_2, db_3;

Nel secondo caso stiamo esportando tutti i database;

L'opzione *--databases* scrive l'istruzione: CREATE DATABASE IF NOT EXIST che crea prima il database appunto e poi il resto, tabelle e dati.

* = 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 e restoring

Backup di un DB MySQL (da shell)

Utente:

L'operazione di backup 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 user[1] –p[2] –no–tablespaces mio_db > mio_db.sql[3]
```

[1] nome dell'utente

[2] per sicurezza la password viene digitata successivamente e non *passata in chiaro*

[3] 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.

[4] se di presentano problemi di privilegi: https://anothercoffee.net/how-to-fix-the-mysqldump-access-denied-process-privilege-error/
soluzione: usare l'opzione --no-tablespaces, -y prima di -u user (obbligatorio a partire da mysql 5.7.31 e 8.0.21)
https://dev.mysql.com/doc/refman/5.6/en/innodb-system-tablespace.html

```
mysqldump –u user[1] –p ––no–tablespaces mio_db nome_tabella > nomefile.sql[2]

mysqldump –u user[3] –p ––no–tablespaces mio_db nome_tabella01 nome_tabella02
nome_tabella03 > nomefile.sql[4]
```

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 user[1] –p ––no–tablespaces –d mio_db > nomefile.estensione[2]
```

Esportazione dei **soli dati del database** (contenuti)

```
 mysqldump –u user[1] –p –no–tablespaces –t mio_db > nomefile.estensione[2]
```

Una volta premuto invio possiamo verificare se il file è stato creato correttamente nella directory indicata.

[1,3] nome dell'utente

[2,4] 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.

IMMAGINAZIONE E LAVORO

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 nella ricerca.

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);
```

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 (**`DESCRIBE nome_tabella`**) l'indice viene indicato con la sigla: *MUL*, che sta per "multipli" 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*, quindi una tabella essa stessa, 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.

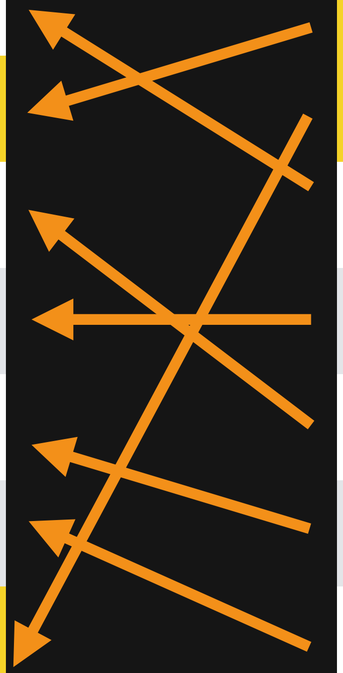
- si evita il Table Scan,
- il recupero dei dati su cui le query devono lavorare avviene in modo più veloce;
- la query stessa è più performante.

L’indice memorizza i valori dell’attributo, ordinati, e i riferimenti con la tabella.

Quando si effettua una ricerca mysql legge solo i record dell’indice e poi recupera i dati dalla tabella.

CLIENTE

Id	cognome	nome	indirizzo	provincia
				MI
				AL
				TO
				MI
				TO
				TO
				AL



INDICE

Provincia
AL
AL
MI
MI
TO
TO
TO

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[1].

Queste strutture dati sono mantenute in memoria, mentre le tabelle sono memorizzate sul disco.

Accedere prima agli indici è più performante e veloce nel recupero dei record piuttosto che leggere tutta la tabella (full table scan).

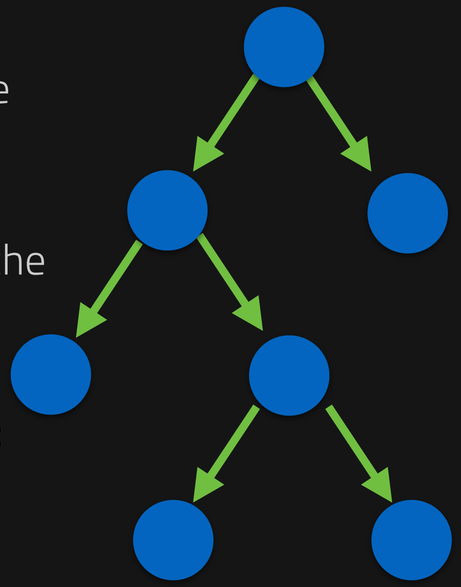
Se gli indici sono fondamentali per le performance delle query, bisogna considerare che:

- 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

IMMAGINAZIONE E LAVORO

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 * FROM cliente
WHERE provincia = "To";
```

Se eseguiamo questa query in assenza di indici la tabella cliente verrà letta record per record fino a trovare le corrispondenze.

Il database inoltre accede al disco per la lettura della tabella.

Verifichiamola con l'istruzione EXPLAIN

```
EXPLAIN SELECT * FROM cliente
WHERE provincia = "To";
```

Eseguita la query con EXPLAIN, mysql restituisce il piano di esecuzione risultante:

```
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | cliente | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    7 |    14.29 | Using where |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
```

Osservate la colonna type, la colonna key e la colonna rows:

In assenza di indici vengono letti tutti i record.

Ora creiamo un indice sulla tabella cliente per l'attributo *provincia*

```
CREATE INDEX k_prov ON
cliente(provincia);
```

id	L'identificatore SELECT
select_type	Il tipo SELECT
table	La tabella per la riga di output
partitions	Le partizioni corrispondenti
type	Il tipo di unione
possible_keys	I possibili indici tra cui scegliere
key	L'indice effettivamente scelto
key_len	La lunghezza della chiave scelta
ref	Le colonne rispetto all'indice
rows	Stima delle righe da esaminare
filtered	Percentuale di righe filtrate in base alla condizione della tabella
Extra	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, il type ci dice che sono stati letti solo i riferimenti, 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 * FROM cliente
WHERE provincia = "To" and credito > 100;
```

In questo caso la query utilizza l'indice.

```
EXPLAIN SELECT * FROM cliente
WHERE provincia = "To" and credito > 100;
```

```
+----+-------------+---------+------------+------+---------------+--------+---------+-------+------+----------+-----------------------------------+
| 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 si può ottimizzare ulteriormente?

Creiamo un idice composto

```
CREATE INDEX k_credito_prov ON cliente(provincia,credito);
```

Eseguiamo la query con EXPLAIN

```
EXPLAIN SELECT * FROM cliente
WHERE provincia = "To" and credito > 100;
```

```
+----+-------------+---------+------------+-------+-----------------------+----------------+---------+------+------+----------+-----------------------+
| 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:

```
SELECT * FROM cliente
WHERE provincia = "To";
```

```
SELECT * FROM cliente
WHERE provincia = "To" and credito > 100;
```

L'indice composto più adatto è

```
CREATE INDEX k_prov_credito ON cliente(provincia,
credito);
```

Indice FULLTEXT

MySQL supporta l'indicizzazione e la ricerca full-text:

Gli indici full-text possono essere utilizzati solo con tabelle InnoDB e MyISAM

possono essere creati solo per colonne CHAR, VARCHAR o TEXT.

L'indice FULLTEXT può essere creato quando viene creata una tabella o aggiunta 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.

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 full-text hanno un design dell'indice invertito. *Gli indici invertiti memorizzano un elenco di parole* e, per ogni parola, un elenco di documenti in cui appare la parola. Per supportare la ricerca di prossimità, vengono memorizzate anche le informazioni sulla posizione di ogni parola

Ricerca

La ricerca full-text viene eseguita utilizzando la sintassi: MATCH() e AGAINST().

MATCH() prende un elenco separato da virgole che denomina 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

```
CREATE FULLTEXT INDEX k_titolo_testo ON posts(titolo, testo);
```

```
SELECT * 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 *, MATCH(titolo, testo) AGAINST('react redux') 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 *, 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 SELECTe 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 *, 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 parole 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".

Se la frase non contiene parole presenti nell'indice, il risultato è vuoto. Ad esempio, se tutte le parole sono stopword[1] o più corte della lunghezza minima delle parole indicizzate, il risultato è vuoto.

1) Una stopword è una parola come " the " o " some " 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 operatori[1] 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.

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.

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."

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.

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.
- Possiamo assegnare la variabile definita dall'utente a tipi di dati limitati come intero, float, decimale, stringa o NULL.
- La variabile definita dall'utente può avere una lunghezza massima di 64 caratteri

IMMAGINAZIONE E LAVORO

Esempio di USER VARIABLE

Il seguente esempio illustra il funzionamento delle variabili definite dall'utente:

```
SET @mediaPrezzo = (SELECT avg(prezzo) FROM libro);
SELECT * FROM libro WHERE prezzo > @mediaPrezzo;
```

L'esempio serve ad illustrarne il funzionamento, attraverso una subquery avremmo potuto semplicemente scrivere:

```
SELECT * FROM libro 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 locale (dichiarate nel corpo dei TRIGGERS delle STORED FUNCTION o STORED PROCEDURES)

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:

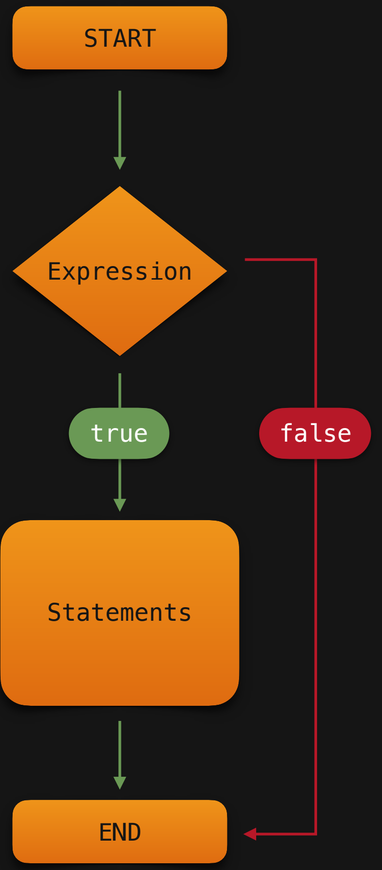
```
SHOW VARIABLES;
SHOW VARIABLES LIKE '%event_scheduler%';
```

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.

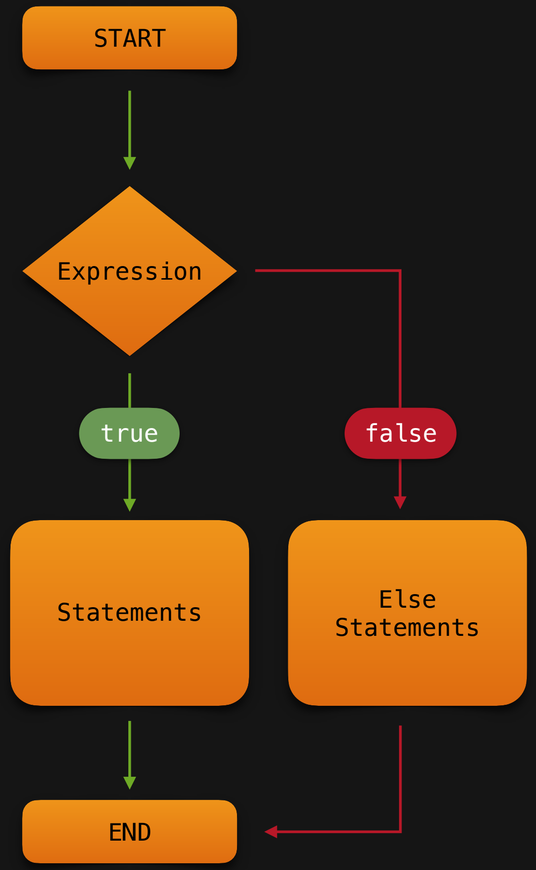



```
IF expression THEN
   statements;

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.

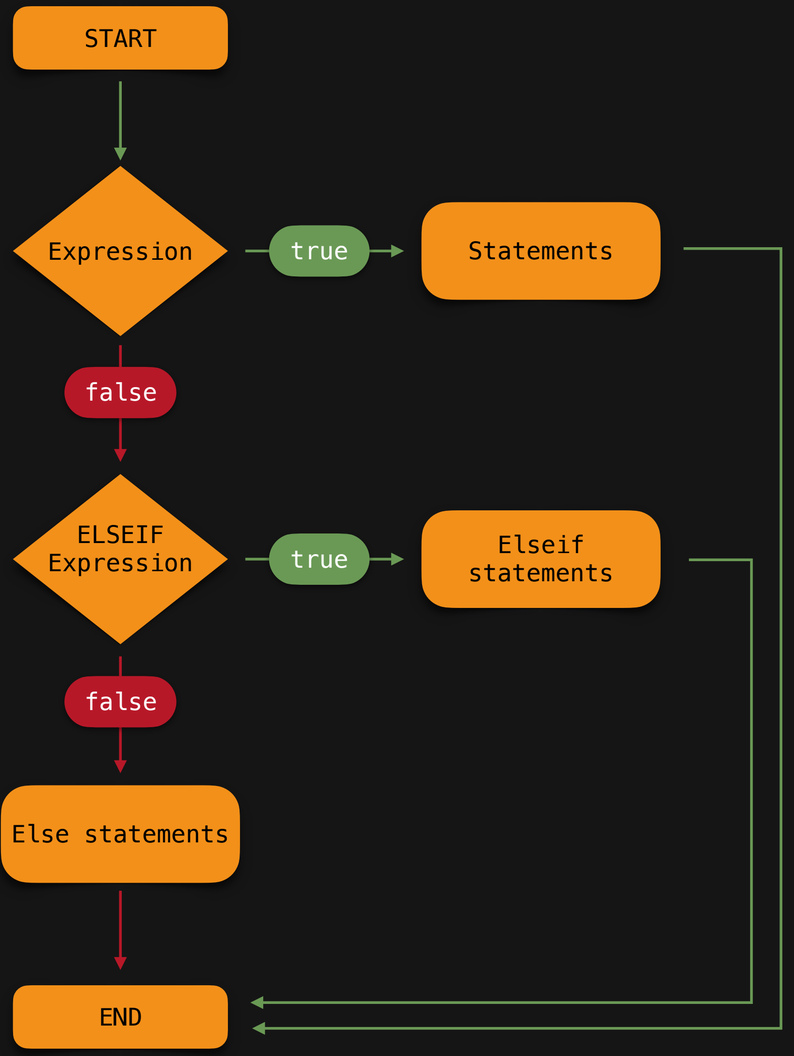
In caso contrario saranno eseguite le istruzioni del blocco ELSE.



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 SET tempo_cons = '1 giorno';
 WHEN 'Lombardia' THEN SET tempo_cons = '2 giorni';
 ELSE
    SET tempo_cons = '5 giorni';
END CASE;
```

istruzione **CASE** *searched*

L'istruzione **CASE** *semplice* 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

SIGNAL[1] 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 SQLSTATE[2], 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

IMMAGINAZIONE E LAVORO

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 nascita[1)]

```
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 studente
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 $$
CREATE FUNCTION clientLevel(p_credit_value SMALLINT) RETURNS VARCHAR(8)
    DETERMINISTIC
BEGIN
    DECLARE client_level varchar(10);
     CASE
       WHEN p_credit_value > 3000 THEN SET client_level = 'PLATINUM';
       WHEN (p_credit_value <= 3000 AND p_credit_value >= 2000) THEN SET client_level = 'GOLD';
       WHEN (p_credit_value <= 2000 AND p_credit_value >= 1000) THEN SET client_level = 'SILVER';
       WHEN (p_credit_value <= 1000 AND p_credit_value >= 1) THEN SET client_level = 'BASIC';
       WHEN p_credit_value < 1 THEN SET client_level = 'NONE';
      END CASE;
    RETURN (client_level);
END$$
DELIMITER ;
```

Richiamo la funzione

```
SELECT cognome Cognome, credito, clientLevel(credito) Livello
FROM cliente
ORDER BY Livello desc;
```

CREAZIONE DI UNA STORED FUNCTION: clienteLevelIf()

Calcolo livello del cliente sulla base del campo credito

```
DELIMITER $$
CREATE FUNCTION clientLevelIf(p_credit_value SMALLINT) RETURNS VARCHAR(8)
    DETERMINISTIC
BEGIN
    DECLARE client_level varchar(10);
    IF p_credit_value > 3000 THEN
     SET client_level = 'PLATINUM';
    ELSEIF (p_credit_value <= 3000 AND p_credit_value >= 2000) THEN
        SET client_level = 'GOLD';
    ELSEIF (p_credit_value <= 2000 AND p_credit_value >= 1000) THEN
        SET client_level = 'SILVER';
    ELSEIF (p_credit_value <= 1000 AND p_credit_value >= 1) THEN
        SET client_level = 'BASIC';
    ELSEIF p_credit_value < 1 THEN
        SET client_level = 'NONE';
    END IF;
 RETURN (client_level);
END$$
DELIMITER ;
```

Richiamo la funzione

```
SELECT cognome Cognome, clientLevelIf(credito) Livello
FROM cliente
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;
```

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.

Memorizza e Organizza l'SQL

- 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.

Esecuzione VELOCE

- 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.

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 studente 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();
```

Scriviamo altra semplice stored procedure denominata `getAllClients()`

`getAllClients()` seleziona tutti i clienti dalla tabella cliente:

```
CREATE PROCEDURE getAllClients()
  SELECT cognome, nome, telefono, email, indirizzo,
citta, provincia, regione, credito
 FROM cliente;
```

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 ordiniCliente 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 cliente c, ordine 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 cliente
     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 restituisce 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 cliente 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 cliente;
    ELSE
        SELECT * FROM cliente 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
```

IMMAGINAZIONE E LAVORO

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 studente;
```

Ottimizzazione della procedure precedente: funzione IFNULL().

```
DROP PROCEDURE IF EXISTS getClientProv;
DELIMITER $$

CREATE PROCEDURE getClientProv(IN p_prov char(2))
BEGIN
  SELECT * FROM cliente
  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
```

Il seguente esempio illustra una procedure con due parametri di IN.

Questa PROCEDURE estrae 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 cliente c
        JOIN ordine 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 cliente c
        JOIN ordine 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
```

Il seguente esempio illustra come utilizzare una PROCEDURE per l'UPDATE dei dati passando due parametri.

Questa procedure aggiornerà il campo quantità della tabella ordine_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 ordine_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 ordine
   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 ordine WHERE cliente_id = p_codice_cliente AND consegna = 'consegnato';

 -- da_consegnare
 SELECT COUNT(*) INTO p_da_consegnare FROM ordine WHERE cliente_id = p_codice_cliente AND consegna = 'da consegnare';

 -- spedito
 SELECT COUNT(*) INTO p_spedito FROM ordine 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;
```

MySQL Stored Procedures con istruzione IF

Il seguente esempio illustra una procedure per ottenere il livello del cliente.

```
DELIMITER $$
CREATE PROCEDURE getPoints(
    IN p_codice_cliente INT,
    OUT p_livello_cliente VARCHAR(10))
BEGIN
    DECLARE credito_lim SMALLINT;

    SELECT credito INTO credito_lim
    FROM cliente
    WHERE id = p_codice_cliente;
     IF credito_lim > 2000 THEN
    SET p_livello_cliente = 'PLATINUM';
    ELSEIF (credito_lim <= 2000 AND credito_lim >= 1000) THEN
        SET p_livello_cliente = 'GOLD';
    ELSEIF credito_lim < 1000 THEN
        SET p_livello_cliente = 'SILVER';
    END IF;
END$$

DELIMITER ;
```

```
CALL getPoints(1,@p_livello_cliente);
SELECT @p_livello_cliente AS 'Livello cliente';
```

La stored procedure GetCredito() accetta due parametri:
IN p_codice_cliente
OUT p_livello_cliente

In primo luogo, si ottiene il credito dalla tabella clienti e si imposta il valore di credito_lim.

Poi, sulla base del credito, si determina il livello del cliente

Il parametro p_livello_cliente memorizza il livello del cliente

L'esempio seguente mostra l'uso dell'istruzione **CASE** searched per trovare il livello del cliente in base al suo credito (esempio già visto con IF):

```
DELIMITER $$

CREATE PROCEDURE getPointsCASE(
 IN  p_codice_cliente INT,
 OUT p_livello_cliente VARCHAR(10))
BEGIN
  DECLARE credito_lim SMALLINT;

  SELECT credito INTO credito_lim
  FROM cliente
  WHERE id = p_codice_cliente;

  CASE
   WHEN credito_lim > 2000 THEN
    SET p_livello_cliente = 'PLATINUM';
   WHEN (credito_lim <= 2000 AND credito_lim >= 1000) THEN
    SET p_livello_cliente = 'GOLD';
   WHEN credito_lim < 1000 THEN
    SET p_livello_cliente = 'SILVER';
  END CASE;
END$$

DELIMITER ;
```

Richiamare la procedure:

```
CALL getPointsCASE(1,@p_livello_cliente);
SELECT @p_livello_cliente AS 'Livello cliente';
```

```
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 cliente
    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 delle codice delle stored procedure:

```
SHOW CREATE PROCEDURE stored_procedure_name;
```

```
SHOW CREATE PROCEDURE getTempoSpedizione;
```

Eliminare una stored procedure:

```
DROP PROCEDURE IF EXISTS nome_procedura;
```

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 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.

[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 parole 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 *studente_audit* e poi il trigger relativamente all'azione BEFORE UPDATE:

```
CREATE TRIGGER before_studente_update
BEFORE UPDATE ON studente
FOR EACH ROW
    INSERT INTO studente_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;
```

Eseguiamo degli aggiornamenti dalla tabella studenti e verifichiamo se vengono tracciate nella tabella *studente_audit*

Ora creiamo il primo trigger relativamente all'azione AFTER DELETE:

```
CREATE TRIGGER after_studenti_delete
AFTER DELETE ON studente
FOR EACH ROW
    INSERT INTO studente_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;
```

Eseguiamo delle cancellazioni dalla tabella studenti e verifichiamo se vengono tracciate nella tabella studente_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 articolo relativamente all'attributo "*rimanenza*" mediante un trigger evocato dalla tabella ordini_dettaglio:

```
CREATE TRIGGER after_od_insert_rimanenza
AFTER INSERT ON ordine_dettaglio
FOR EACH ROW

    UPDATE articolo
    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 ordine_dettaglio
FOR EACH ROW
    UPDATE articolo
     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 ordine_dettaglio
FOR EACH ROW
   UPDATE articolo
   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 *ordine dettaglio*

Ora vediamo l'uso di Trigger per incrementare il credito del cliente quando inserisco un record in ordine_dettaglio.

Creiamo il trigger relativamente all'azione AFTER INSERT

Aggiornamento della tabella **cliente** relativamente all'attributo "*credito*" mediante un trigger evocato dalla tabella ordini_dettaglio:

```
CREATE trigger after_od_insert_credito
AFTER INSERT ON ordine_dettaglio
FOR EACH ROW

    UPDATE cliente c SET credito =
    credito + (NEW.quantita * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordine 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 **cliente** relativamente all'attributo "*credito*" mediante un trigger evocato dalla tabella ordini_dettaglio:

```
CREATE trigger after_od_delete_credito
AFTER DELETE ON ordine_dettaglio
FOR EACH ROW

    UPDATE cliente c SET credito =
    credito - (OLD.quantita * OLD.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordine 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 **cliente** relativamente all'attributo "*credito*" mediante un trigger evocato dalla tabella ordini_dettaglio:

```
CREATE trigger after_od_update_credito
AFTER UPDATE ON ordine_dettaglio
FOR EACH ROW

    UPDATE cliente c SET credito =
    credito + ((NEW.quantita - OLD.quantita) * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordine 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*

IMMAGINAZIONE E LAVORO

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 *articolo* e credito in tabella *cliente*

```
DELIMITER $$

CREATE TRIGGER after_od_insert_rimanenza_credito
AFTER INSERT ON ordine_dettaglio
FOR EACH ROW
BEGIN

    -- aggiorno rimanenza in articolo
    UPDATE articolo
    SET rimanenza = rimanenza - NEW.quantita
    WHERE id = NEW.articolo_id;

    -- aggiorno credito in cliente
    UPDATE cliente c SET credito =
    credito + (NEW.quantita * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordine o
        WHERE NEW.ordine_id = o.id
        );

END$$

DELIMITER ;
```

TRIGGER unico per aggiornare rimanenza in tabella *articolo* e credito in tabella *cliente*

```
DELIMITER $$

CREATE TRIGGER after_od_delete_rimanenza_credito
AFTER DELETE ON ordine_dettaglio
FOR EACH ROW
BEGIN

    -- aggiorno rimanenza in articolo
    UPDATE articolo
    SET rimanenza = rimanenza + OLD.quantita
    WHERE id = OLD.articolo_id;

    -- aggiorno credito in cliente
    UPDATE cliente c SET credito =
    credito - (OLD.quantita * OLD.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordine o
        WHERE OLD.ordine_id = o.id
        );

END$$

DELIMITER ;
```

TRIGGER unico per aggiornare rimanenza in tabella *articolo* e credito in tabella *cliente*

```
DELIMITER $$

CREATE TRIGGER before_od_update_rimanenza_credito
BEFORE UPDATE ON ordine_dettaglio
FOR EACH ROW
BEGIN

    -- aggiorno rimanenza in articolo
    UPDATE articolo
    SET rimanenza = rimanenza-(NEW.quantita - OLD.quantita)
    WHERE id = OLD.articolo_id;

    -- aggiorno credito in cliente
    UPDATE cliente c SET credito =
    credito + ((NEW.quantita - OLD.quantita) * NEW.prezzo)
    WHERE c.id = (
        SELECT o.cliente_id FROM ordine o
        WHERE NEW.ordine_id = o.id
        );

END$$

DELIMITER ;
```

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 ordine_dettaglio
FOR EACH ROW
BEGIN

 DECLARE rimanenza_d TINYINT;
 DECLARE msg VARCHAR(255);
 SET rimanenza_d = (SELECT rimanenza FROM articolo WHERE id = NEW.articolo_id);

   IF NEW.quantita <= rimanenza_d
        THEN
        UPDATE articolo
        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.

BEFORE INSERT: TRIGGER unico per aggiornare rimanenza in *articolo*, aggiornare credito in *cliente e* per tenere conto dello stato dell'ordine in *ordine*

```
DELIMITER $$
CREATE TRIGGER before_od_insert_totale
BEFORE INSERT ON ordine_dettaglio
FOR EACH ROW
BEGIN

DECLARE spedizione enum('consegnato','da spedire','spedito');
DECLARE rimanenza_d tinyint;
DECLARE msg varchar(255);

SET rimanenza_d = (SELECT rimanenza FROM articolo WHERE id = NEW.articolo_id);
SET spedizione = (SELECT consegna FROM ordine WHERE id = NEW.ordine_id);

IF spedizione = 'da spedire'
     THEN
        IF NEW.quantita <= rimanenza_d
           THEN

           -- aggiorno rimanenza in articolo
           UPDATE articolo
           SET rimanenza = rimanenza_d-NEW.quantita
           WHERE id = NEW.articolo_id;

           -- aggiorno credito in cliente
          UPDATE cliente c SET credito =
          credito + (NEW.quantita * NEW.prezzo)
          WHERE c.id = (
              SELECT o.cliente_id FROM ordine o
              WHERE NEW.ordine_id = o.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;
ELSE
     SIGNAL SQLSTATE VALUE '45000'
     SET MESSAGE_TEXT = "Non puoi aggiornare l'ordine, gli articoli sono stati già spediti";
END IF;

END$$
DELIMITER ;
```

BEFORE UPDATE: TRIGGER unico per aggiornare rimanenza in *articolo*, aggiornare credito in *cliente e* per tenere conto dello stato dell'ordine in *ordine*

```
DELIMITER $$
CREATE TRIGGER before_od_update_totale
BEFORE UPDATE ON ordine_dettaglio
FOR EACH ROW
BEGIN

DECLARE spedizione enum('consegnato','da spedire','spedito');
DECLARE rimanenza_d tinyint;
DECLARE msg varchar(255);

SET rimanenza_d = (SELECT rimanenza FROM articolo WHERE id = NEW.articolo_id);
SET spedizione = (SELECT consegna FROM ordine WHERE id = NEW.ordine_id);

IF spedizione = 'da spedire'
     THEN
        IF (NEW.quantita - OLD.quantita) <= rimanenza_d
          THEN

          -- aggiorno rimanenza in articolo
          UPDATE articolo
          SET rimanenza = rimanenza_d-(NEW.quantita - OLD.quantita)
          WHERE id = OLD.articolo_id;

          -- aggiorno credito in cliente
          UPDATE cliente c SET credito =
          credito + ((NEW.quantita - OLD.quantita) * NEW.prezzo)
          WHERE c.id = (
              SELECT o.cliente_id FROM ordine o
              WHERE NEW.ordine_id = o.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;
ELSE
     SIGNAL SQLSTATE VALUE '45000'
     SET MESSAGE_TEXT = "Non puoi aggiornare l'ordine, gli articoli sono stati già spediti";
END IF;

END$$
DELIMITER ;
```

BEFORE DELETE: TRIGGER unico per aggiornare rimanenza in *articolo*, aggiornare credito in *cliente e* per tenere conto dello stato dell'ordine in *ordine*

```
DELIMITER $$
CREATE TRIGGER before_od_delete_totale
BEFORE DELETE ON ordine_dettaglio
FOR EACH ROW
BEGIN

DECLARE spedizione enum('consegnato','da spedire','spedito');
DECLARE msg varchar(255);

SET spedizione = (SELECT consegna FROM ordine WHERE id = OLD.ordine_id);

IF spedizione = 'da spedire'
    THEN

        -- aggiorno rimanenza in articolo
        UPDATE articolo
        SET rimanenza = rimanenza + OLD.quantita
        WHERE id = OLD.articolo_id;

        -- aggiorno credito in cliente
        UPDATE cliente c SET credito =
        credito - (OLD.quantita * OLD.prezzo)
        WHERE c.id = (
            SELECT o.cliente_id FROM ordine o
            WHERE OLD.ordine_id = o.id
            );
    ELSE
     SIGNAL SQLSTATE VALUE '45000'
     SET MESSAGE_TEXT = "Non puoi eliminare gli articoli, gli articoli sono stati già spediti";
END IF;

END$$
DELIMITER ;
```

LIMITAZIONI del TRIGGER: i trigger non vengono attivati da azioni di chiavi esterne

Consideriamo la situazione: TRIGGER + chiave esterna ON DELETE CASCADE in ordine_dettaglio.

TRIGGER:

AFTER DELETE: triggers che aggiorna le quantità del magazzino in caso di eliminazione di un dettaglio in ordine_dettaglio.

```
CREATE TRIGGER after_od_delete_rimanenza
AFTER DELETE ON ordine_dettaglio
FOR EACH ROW

     UPDATE articolo
     SET rimanenza = rimanenza + OLD.quantita
     WHERE id = OLD.articolo_id;

  ...
```

CHAVE ESTERNA:

```
CONSTRAINT fk_od_ordine
FOREIGN KEY(ordine_id) REFERENCES ordine(id)
ON DELETE CASCADE
```

Quando *elimino una riga da ordine_dettaglio*, il trigger verrà attivato.

Quando *elimino una riga da ordine*, anche *le righe correlate in ordine_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 *ordine* che elimina le righe correlate in *ordine_dettaglio*

Quando elimini una riga da ordine, il trigger verrà attivato e cancellerà le righe correlate *"direttamente"* in ordine_dettaglio.

Dal momento che i record vengono eliminati direttamente, il trigger su *ordine_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 ordine_dettaglio
DROP FOREIGN KEY fk_ordine_dettaglio_ordine;

ALTER TABLE ordine_dettaglio ADD CONSTRAINT fk_ordine_dettaglio_ordine
    FOREIGN KEY(ordine_id) REFERENCES ordine(id)
    ON DELETE NO ACTION ON UPDATE CASCADE;
```

l'eliminazione di un ordine sarà impedita se ci sono righe figlie in *ordine_dettaglio*.
Dovrò prima eliminare le righe in *ordine_dettaglio* (attivando così i trigger) e poi la riga in *ordine*.

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 ordine_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 delle stato della spedizione prima di eliminare un ordine.

Chiave esterna su *ordine_dettaglio* riferita a *ordine* impostata a RESTRICT.

```
DELIMITER $$
CREATE TRIGGER before_ordine_delete
BEFORE DELETE ON ordine
FOR EACH ROW
BEGIN

 DECLARE spedizione ENUM('consegnato','da spedire','spedito');
 SET spedizione = (SELECT consegna FROM ordine WHERE id = OLD.id);

  IF spedizione = 'da spedire'
       THEN
       DELETE FROM ordine_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 ordine 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

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 documentazione[1]

[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 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 oridini 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/

per trovare soluzioni a molti problemi di codice: http://stackoverflow.com/